Multiple Inheritance in Contemporary Programming Languages

by

Daniel Joseph Carnese, Jr.
A.B., University of California (1974)

Submitted in partial fulfillment
of the requirements for the
degree of

Master of Science

at the

Massachusetts Institute of Technology

August, 1984

Signature of Author . . . . . [illegible] . . . .
Department of Electrical Engineering and Computer Science
August 31, 1984

Certified by . . . . . . [illegible] . . . . DAVID Gifford
Thesis Supervisor [illegible]

Accepted by . . . . . [illegible] . . . .
Chairman, Departmental Committee

Multiple Inheritance in Contemporary Programming Languages

by

Daniel Joseph Carnese, Jr.

Submitted to the Department of Electrical Engineering and Computer Science on August 31, 1984 in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science.

Abstract

In one paradigm of abstract type definition, a separate procedure definition is given for each abstract operation. This paradigm can be extended by allowing the operation set of a type to be computed from the operation set of an existing type; this is commonly referred to as "definition by inheritance." When the operation set of a type can be computed from the operation sets of more than one existing type, it is called "definition by multiple inheritance." The key advantage of type definition by multiple inheritance is that it reduces the amount of work required by programmers to design new types and make changes to existing systems of types.

This report examines four designs for type definition by multiple inheritance: Zetalisp "flavors," the "traits" extension to Mesa, and the "classes" of the Borning/Ingalls extension to Smalltalk-80 and the Loops extension to Interlisp. The constructs involved in the designs are described in a common framework. For each construct involved, examples are provided and a brief overview of relevant design issues is given.

Acknowledgments

It is a pleasure to thank the people who helped me in producing this report.

Dave Gifford was my advisor; he was invaluable.

Gary Leavens read the whole document as it was being written. Considering the volume of paper and the uneven quality of the first draft, this was action truly above and beyond the call of academic duty.

Mark Shirley, Kent Pitman, and Greg Faust participated over the course of writing this report in a significant number of technical discussions. They were a great help in getting me to focus on what was truly important.

I also benefitted from discussions with many other people, including: Liz Allen, Robert Ayers, Pavel Curtis, Gary Drescher, Ken Haase, Hal Haig, Walter Hamscher, Richard Ilson, Henry Lieberman, John Mallery, David MacQueen, John Mitchell, Fanya Montalvo, Reid Simmons, Brian Williams, Dan Weld, and Richard Zippel.

In the AI Lab, Randy Davis, Patrick Winston, and Karen Prendergast provided the environment needed to complete this project.

My friends from the (ex-)OA and PT groups, Stan Zdonik, Dennis McLeod, Tim Anderson, Dave Lebling, Chris Reeve, and Sunil Sarin, did a wonderful job in helping me keep the appropriate perspective on life at M.I.T.

Finally, my family was truly a constant source of support and encouragement. This work is dedicated to them.

Table of Contents

List of Figures

1. Overview

1.1 The goal of this report

This report is concerned with type definition in programming languages. It focuses on type construction where properties of new types are synthesized from properties of some number of existing types. A well-known name for this technique is **definition by inheritance.** If more than one existing type is involved, it is known as **definition by multiple inheritance.**

The literature on type definition by multiple inheritance largely consists of descriptions of the languages in which such definition is possible. Motivation for specific constructs and cross-system analysis is either cursory or simply omitted. Furthermore, language descriptions have been generally given in the terminology of "object-oriented programming" (e.g., [Hewitt 77, XeroxLRG 81, Rentsch 82]). This has made it difficult to relate inheritance-based type definition to the mainstream of work in typed programming languages.[1]

As a result, there has been no satisfying account to date of:

- the similarities and differences between inheritance-based and non-inheritance-based type definition;
- the similarities and differences among the proposals for type definition by inheritance; or
- the pragmatic significance of either of the above.

This report ameliorates this situation by examining how type definition via multiple inheritance is realized in four contemporary programming languages.

[1] [Suzuki 81] and [BorningIngalls 82a] are two attempts; there have been few others to date.

1.2 The four subject languages

The four languages which are discussed in this report are as follows.

- **Zetalisp.** Zetalisp (e.g., [Bawden et al. 77, WeinrebMoon 81, Symbolics 84, MoonStallmanWeinreb 84]) is the primary implementation and application language of the family of high-performance personal "Lisp Machines" based on the original design of the MIT Artificial Intelligence Laboratory. It has been subsequently developed by two commercial spinoffs, Symbolics, Inc. and Lisp Machines, Inc. The systems described in [Symbolics 84] and [MoonStallmanWeinreb 84] are slightly different realizations of the same fundamental approach.

  Although flavors were developed after the initial design of the machine, they rapidly became a key aspect of the system architecture, e.g., for multiprocessing, building graphical user interfaces, network i/o, and condition and error handling. The Lisp Machine software is the most complex system built to date in which multiple inheritance has been extensively used.

- **"Star Mesa".** The initial release of the Xerox Star office workstation [Seybold 81] used an extension to Mesa [Mitchell et al. 79] which supports synthetic type definitions in the form of "traits" [Curry et al. 82, Lipkie et al. 82, CurryAyers 83]. Like the Lisp Machine software, the Star software is a substantial application of the multiple-inheritance methodology. However, more recent versions of the software have eliminated the use of type definition by multiple inheritance and rely solely on inheritance from a single type.

- **"Smalltalk-82".** [BorningIngalls 82b] describes an extension to Smalltalk-80 in which classes can have more than one superclass. This extension has been used internally at Xerox PARC, but does not now have an active user community. In this paper we will sometimes use the name "Smalltalk-82" to distinguish it from unextended Smalltalk-80.

- **Loops.** Loops [BobrowStefik 83, Stefik et al. 83a] is an extension of Interlisp-D which runs on the Xerox 1100 computer family. Although its original intention was to implement "object-oriented programming" (e.g., [Rentsch 82] in Interlisp, it evolved into a system which supported the "rule-oriented" and "action-oriented" paradigms as well.[2] It is in active use as a tool for research and development in "knowledge engineering."

These four languages represent much of the state of the art in type definition by multiple inheritance. However, they do not constitute all current proposals and/or implementations of this idea. Other relevant work includes the following.

[2]The former refers to the paradigm of building computer systems through the use of predicate/action rules (e.g., see [WatermanHayes-Roth 78]). The latter term refers to the "constraint system" approach to programming developed at MIT (e.g., [SussmanSteele 80, Steele 80]) and elsewhere (e.g., [Borning 81])

- [Wood 82] and [Allen et al 82] describe how types similar to Zetalisp's flavors were incorporated in Franz Lisp.

- [Novak 82, Novak 83a, Novak 83b] describes an extension to Lisp which incorporates type-like "object descriptions" whose operation sets are synthesized from those of other object descriptions.

- The "capsules" system described in [Zippel 83] constructs new types from one or more existing types, based on descriptions of known properties of the existing types and of the desired properties of the new type.

- The "QLogo" and "QLisp" languages of [Drescher 84] are extensions of Logo and Lisp which embody a type system which allows any object to be a type, and allows properties of such objects to be defined as syntheses of those of any number of existing objects.

1.3 Properties of types defined by inheritance

Using inheritance to defining some property P of an object O involves a set of other objects from which the value of P is computed. We will refer to this set of objects as the "parents" of O. and will use the terms "child," "ancestor," and "descendant" in the obvious way. We will see some realizations of inheritance in which an ordering relation is imposed on the parents, and some in which order is irrelevant.

There are two aspects of programming-language types for which definition by inheritance has been used. The following is a brief description; chapters 6 and 5 provide a more thorough discussion.

1.3.1 The generic operation set of a type

Each of our subject languages supports one or more forms for procedure invocation where the identity of the invoked procedure depends on (a) a symbol derived from the form, and (b) the type of a distinguished parameter of the form. Such expressions are commonly referred to as "generic" procedure invocations. We will refer to the symbol as the "operation name" of the invocation and to the distinguished parameter as the "generic parameter" of the invocation.

Thus, for each type T, we can identify a set of associations between operation names and procedures which the generic invocation form will invoke when the generic parameter has type T. We will refer to this association between operation names and procedures as the "generic operation set" of the type T. The procedure associated with a particular operation name will be called the "method" for that operation name for type T.

In many programming languages, the generic operation set of a type is enumerated by the type definer. For the types which are the focus of this report, generic operation sets are computed rather than enumerated. The computation depends on a directly specified property of a type and its ancestors. This property, referred to as the "local operation set," is also a set of associations between symbols and procedures.

The algorithm used to construct the generic operation set of a type from the local operation sets of the type and its ancestors is one of the key distinguishing characteristics of different approaches to type definition by inheritance. However, all share the following property, which we will refer to as the "consistent method inheritance" property.

> If (a) all parents which define a given generic operation use the identical method and (b) the new type has no local method, then the method of the new type is identical to the method shared by the parents.

The "inheritance" metaphor is relevant because we can view the method as being "passed down" from the parents of the type.

As we will see, the consistent method inheritance property can greatly reduce the work required to define new types.

1.3.2 The instance variables of a type

A common abstraction embodied by instances of all of our subject types is that of a function, in the mathematical sense, from symbols to values. Variants of this abstraction are well-known, e.g., the RECORD types of Algol 68 and Pascal, the "property lists" and "association lists" often used in Lisp programs, and the general concepts of "symbol table" and "dictionary."

Each of our four languages provides procedures for manipulating instances of our subject types in terms of some variant of this abstraction. Specifically, procedures exist for "storage" and "retrieval;" i.e., for associating a value with a given symbol and obtaining the value associated with a given symbol. Since the term "instance variable" is often used to refer to these symbols, we will refer to the storage and retrieval procedures as **instance-variable operations.**

The details of the behavior of the procedures for instance variable storage and retrieval differ in our four languages. For example, associating a value with a symbol for which no value is currently associated is possible in Loops, but not in Zetalisp, Smalltalk, or Star Mesa. However, the behavior of the procedures of the instance-variable operation set always depends on a set of symbols associated with the type. We will refer to these symbols as the instance variable names of the type.

The algorithm for computing the instance variable names of a type involves a second property associated with types, which we will refer to as its "local instance variable names." In each of our four languages, the instance variable names of a type are computed as the union of the local instance variable names of the type and the local instance variables of all ancestors of the type. Thus, the set of instance variables of a type is defined by inheritance.

1.4 The historical context

A thumbnail history of the significant predecessors of type definition by multiple inheritance is as follows.

- The Sketchpad graphics system of [Sutherland 63] embodied primitive mechanisms for generic invocation, the definition of a generic operation set of a type, and the definition of other properties of objects (e.g., the "parts" of an object) using multiple inheritance. But adding new types, generic operations, or other inherited properties required intimate knowledge of the implementation of the system. The Thinglab system described in [Borning 81] was based on many of the same ideas, but allowed new kind of objects and properties to be defined with far less knowledge of implementation detail.

- The CLASS construct of Simula 67 [Dahl 68, Birtwhistle et al. 73] embodied the key concepts of generic invocation, associating a generic operation set with a type, associating a set of instance variables with types, and defining both the generic operation set and the set of instance variables by inheritance.

- The programming methodology of "abstract type definition" (e.g. [Hoare 72, Morris 73, LiskovZilles 74, Reynolds 74, Demers et al. 78, Guttag 80]) is based on the utility of associating two operation sets for each type: an "abstraction" operation set to embody the desired behavior of instances of the type, and a "representation" operation set to implement that functionality. This methodology was a key motivating force in the design of the multiple-inheritance languages. It is natural to treat the generic operations as abstract operations and to have the instance variable operations fill the role of representation operations.

  Despite the common motivation, there are three significant differences between the mainstream of the "abstract type" languages (e.g., [Wulf et al. 76, Tennent 77, GeschkeMorrisSatterthwaite 77, Lampson et al. 77, Liskov et al. 79, Gordon et al. 79, BoehmDemersDonahue 80, Wirth 80] and the "multiple inheritance" languages we are studying here.

  - With few exceptions, type definitions in the abstract-type languages construct neither the abstract operation set nor the representation operation set of a type by inheritance. For example, of the languages referred to above, only Russell's `with` constructor [BoehmDemersDonahue 80] synthesizes the operation set of a type from the operation set of an existing type and a local operation set.[3] The `with` constructor cannot have more than one parent and cannot create types with a different set of underlying primitive operations than its parent. To illustrate the latter, no matter what replaces the . . . in the following:

    ```
    Integer with ...
    ```

    all computations on instances of the resulting type can be expressed in terms of `Integer` operations.

  - The abstract-type languages incorporate syntactic constraints which prevent the representation operations of a type from being used outside a specified lexical context. The multiple inheritance languages allow the use of the instance-variable operation set anywhere in programs.

  - The abstract-type languages incorporate constraints which exclude programs for which a type-checking algorithm fails. The multiple inheritance languages impose no such constraints, but can guarantee that an error will be signalled on any attempt to invoke a non-existent generic operation or to obtain the value of a non-existent instance variable.

- The Smalltalk family of languages (Smalltalk-72 [Shoch 79], Smalltalk-76 [Ingalls 78], and Smalltalk-80 [XeroxLRG 81, GoldbergRobson 83]) extended both the Simula and the abstract-type designs by:

[3] In Russell, types constructed via the "operation modification" constructor:

TypePrimary `with` *withlist*

have operation sets defined by inheritance. *TypePrimary* represents the single parent type and *withlist* enumerates the local operation set.

- o defining the generic operation sets and the instance variables of all types via inheritance from an existing type;
- o having the primitive procedure-invocation form cause generic invocation;
- o realizing types as data structures which can be created and modified during program execution (cf. [Wegbreit 70, Wegbreit 74]);
- o allowing instances of programmer-defined types to be themselves used as types.

1.5 What's the point?

Why define the generic operation set and the instance variables of a type by inheritance rather than by enumeration? Based on the literature which attempts to motivate type definition by inheritance (e.g., [Ingalls 78, WeinrebMoon 81, XeroxLRG 81, Curry et al. 82, Lipkie et al. 82, Rentsch 82, GoldbergRobson 83, BobrowStefik 83, CurryAyers 83, Novak 83b]) we can identify the following advantages.

1.5.1 The use of inheritance for generic operation set definition can reduce the work required to define new types.

If generic operation sets are defined by enumeration, defining a type with N operations requires the definition of the N procedures to be used as methods. But if generic operation sets are defined by single inheritance, the number of procedure definitions needed to construct a type is proportional to the number of methods for which the algorithm for computing the generic operation set does *not* produce the desired procedure. In these situations, less work is required to define new types.

For example, suppose we are given a type `window` with a `redisplay` operation, and wish to add a new type which exhibits the same behavior but keeps statistics on `redisplay` invocations. Definition of the generic operation set by enumeration requires defining a procedure for each of the generic operations of `window`. But if definition by inheritance is available, only the `redisplay` operation need be reimplemented. The number of procedure definitions which can potentially be eliminated increases with the number of generic operations of the type and with the degree of similarity between the operation set of the new type and that of existing types.

The availability of multiple inheritance offers the potential for an even greater reduction in the work required to define new types. This will occur in situations where the generic operation set of the new type is similar to the operation sets of more than one existing type. To illustrate this point, [BorningIngalls 82b] uses the example of defining the type ReadWriteStream given an existing ReadStream and WriteStream. The generic operation set of ReadWriteStream should have all the operations of both ReadStream and WriteStream.

If only single inheritance is available, then either ReadStream or WriteStream must be chosen as the parent. If we choose the former, then a method must be defined for *all* generic operations of WriteStream which are not operations of ReadStream. If we choose the latter, the situation is reversed. But if ReadWriteStream can be defined by multiple inheritance, then methods for both ReadStream and WriteStream operations can be defined by the method computation algorithm rather than by enumeration.

It should be clear that definition by inheritance has no utility if the collection of computed methods does not exhibit the desired abstraction. A common observation is that inheritance is more likely to be useful when the behavior embodies by inherited abstractions is "independent" (e.g., [WeinrebMoon 80, Cannon 82]), but no definition of "independent" has been proposed.

An informal argument for a sufficient condition for the success of an inheritance algorithm is as follows. If the generic operation names of each of the parent types are disjoint, and the instance variables used in the realization of each type are also disjoint, then there cannot possibly be any interaction between the methods. Thus, the generic operation set produced by an inheritance algorithm which embodies the consistent method inheritance property of section 1.3.1 will satisfy the abstractions embodied by each of the parent types.

1.5.2 The use of inheritance for generic operation set definition can make it easier to modify shared abstractions.

Suppose we have a collection of types which are all intended to support the same abstract operation set through their generic operations. Suppose further that it has been deemed desirable to add additional operations to the abstraction, and methods can be defined for these operations which will exhibit the desired behavior for all of these types.

If the generic operation set is defined by enumeration, a definition of the new operation must be made for each type. But if the generic operation set is defined by inheritance *and* there is a common ancestor which corresponds to the abstraction, modifying the local operation set of that ancestor will add the new operation to the operation set of all descendants. This behavior follows from the consistent method inheritance property.

As a concrete example, consider adding an "incremental redisplay" operation to a "window" abstraction realized by the three types `notification-window`, `documentation-window`, and `editing-window`. If the generic operation sets of each of these types were defined by enumeration, a separate definition of `incremental-redisplay` would have to be added for each type. But if these three types were defined by inheritance with a `basic-window` type as an ancestor, defining an `incremental-redisplay` operation for `basic-window` would automatically result in its definition for notification, documentation, and editing windows.

This consequence of generic operation set definition by inheritance has two principal advantages. First, analogous to the new-type-definition case, the number of procedure definitions needed to carry out the modification is reduced. Second, and even more important, *the augmenter of the abstraction does not have to know the identity of the augmented types.* This latter aspect greatly facilitates the design of of large, multi-layered systems, because it reduces the need for communication between the designers of the various layers. The software of the MIT Lisp Machine is an excellent example of such a system.

1.5.3 The rationale for defining instance variable by inheritance

Given that it is desirable to define the generic operation set by inheritance, the desirablilty of defining instance variables by inheritance follows directly. For example, it would be pointless for a type to inherit methods which implement an "list" abstraction in terms of operations on instance variables named "first" and "rest" if no such instance variables were defined for instances of the type.

Thus, if inherited methods are to work at all, all instance variables of all parent types must be instance variables of the new type. Defining instance variables by inheritance is more advantageous than defining them by enumeration because less work is required to define the new type, less work is required to modify an existing type, and a source of programming error (i.e., omitting an instance variable of a parent in the enumeration) is eliminated.

1.6 What's *not* the point

There are a number of advantages of using our four subject languages which do not derive from definition by inheritance. However, the literature concerning these languages does not clearly distinguish between the benefits gained by definition by inheritance and the advantages which flow from other aspects of the language. Thus, we will briefly discuss two other benefits associated with using our four subject languages but which do not derive from inheritance-based type definition.

1.6.1 Modularity is increased by using generic invocations to implement abstract operations

If a given abstract computation is carried out by a generic invocation, alternative algorithms can be defined for specific types without affecting the procedure bodies which implement the operations for other types. This modularity results in two principal benefits.

- It is easier to modify the algorithm for a specific type, since the procedure body to be modified is only applicable for a single type, and can thus be smaller.
- It is less dangerous to experiment with such modifications, since the modified

procedure body will never be used to carry out the computation for instances of unrelated types.

Both of these advantages would still obtain even if the ~~the~~ generic operation set of a type were defined by enumeration (e.g., as in Clu [Liskov et al. 79] or Russell [BoehmDemersDonahue 80]) rather than by inheritance.

1.6.2 Redundancy is reduced by sharing methods among types

If several types can use the same algorithm to implement a given generic operation, definition by inheritance allows the procedure embodying the algorithm to be defined once rather than many times. The advantages of such sharing are well known: less work in initial creation and subsequent modification, increased program clarity, and no opportunity for copying errors.

The consistent method inheritance property often results in sharing methods among types, e.g., the sharing of the method for the `incremental-redisplay` operation described above. But the same sharing could occur even if generic operation sets had to be defined by enumeration of operation names and procedures. Definition by inheritance *reduces the work* required to obtain sharing, but is *not necessary* for such sharing to occur.

1.6.3 Redundancy is reduced by having methods of types invoke methods of ancestor types

In many cases, the methods of a type cannot be *shared* by some descendant, but can be *invoked* as a subroutine by methods of descendants. As an often-used example, it is easy to imagine that the **display** operation for a **bordered-window** type calls the **display** for **window** either before or after drawing the border.

Again, the use of definition by inheritance can reduce the work required to employ this programming methodology, but is not essential for its use. Notice that in this situation, the consistent method inheritance property *is not applicable*, since the new type must have a local method which realizes the additional computation required (e.g., drawing the box). Thus, the critical properties are (a) the procedure produced by the method definition algorithm in the presence of conflicting inheritance, and (b) how methods of types can invoke methods of ancestor types. In these properties, our four languages differ considerably.

1.7 Inheritance in programming languages v. inheritance in knowledge representation languages

Definition by inheritance is used in domains other than the definition of programming language types. The "knowledge representation" languages of artificial intelligence (e.g., [Minsky 74], [BobrowWinograd 77a], [BobrowWinograd 77b], [GoldsteinRoberts 77], [Brachman 78], [Stefik 78], [GreinerLenat 80], [BrachmanFikesLevesque 83], [Touretzky 84], [BrachmanSchmolze 84]), use inheritance to define the computational objects used to represent the real-world objects with which the program is concerned and to represent assertions about those objects.

The key difference between inheritance in programming languages and inheritance in knowledge representation lies in the abstractions which are associated with the objects thus defined. In programming languages, there are two operation sets which are relevant for an object: instance-variable operations and generic operations. But the highest level of abstraction embodied in the current generation of knowledge representation languages are procedures with the semantics of "assert that property P of object O has value V" and "what is the value of property P of object O?" These are direct analogues of the instance-variable operations of programming language types.

Some knowledge representation systems (e.g., [GreinerLenat 80, IntelliGenetics 83, Haase 84]) do allow type-specific methods to be specified for the instance-variable operations. In other words, specialized procedures can be defined for computing a given property of the instances of a given type. However, these procedures are still concerned with instance variables rather than type-specific abstractions.

Thus, the current situation can be summarized by the following aphorism:

In knowledge representation, instance variables are part of the abstraction. In programming languages, they are not.

1.8 Organization of the report

An outline of the remainder of this document is as follows.

- Chapter 2 presents a programming problem to which reference is made throughout the sequel. It shows how to accomplish this task using multiple inheritance in our four subject languages.

- Chapter 3 describes the generic invocation mechanisms applicable to our subject types.

- Chapter 4 describes the most basic aspects of our subject types. It identifies the minimal type-defining forms, the procedures which provide information about the type of an object and the parent/child relation on types.

- Chapter 5 describes how the instance-variable operation set is realized for each of our subject types, and how the collection of instance variables of a type are defined.

- Chapter 6 describes how the generic operation set of our subject types are computed from the local operation sets of the type and its ancestors.

- Chapter 7 describes how local operation sets are associated with our subject types.

- Chapter 8 covers mechanisms for type declarations and type checking.

- Chapter 9 concludes by summarizing the similarities and differences which we will have seen, and the impact of these differences on programmer productivity.

The descriptive material in this report is, in essence, a synthesis of four reference manuals in a common conceptual framework. It contains examples of each construct presented, and provides at least some explanation of the rationale behind the design and the advantages and disadvantages of the chosen alternative. As a result of its comprehensiveness and level of detail, it can be quite difficult to read. The reader who would like to get right to the bottom line can begin with chapter 9 and explore the details in the body of the report as desired.

2. The illustrative example

The explanation of concepts is always easier in the context of an example. We now present a programming task for which type definition by multiple inheritance can be brought to bear, and the way in which this task can be carried out using each of our four multiple-inheritance type constructors.

This example is concerned with defining four types and associated operations. The description will be given at three levels of abstraction. The functional level describes the observable effects of operation invocations. The representational level describes the data structures representing the four types. The implementation level describes how the methods for each generic operation are defined in terms of other methods and procedures which have specified effects on the data structures of the representation.

2.1 Functional description

The simplest of our four types, a **point**, is a formalization of a movable point in one-dimensional space. It is a one-dimensional, mutable variant of the **Point** described in chapter 18 of [GoldbergRobson 83].[4]

There are four relevant abstraction operations for points: **create**, **location**, **move**, and **display**.

- The **create** operation takes an optional numeric parameter, **location**, whose value if unspecified is 0. It returns a point object distinct from all other objects.
- The **location** operation maps a point into a number. The initial **location** of a newly created point is the value is the **location** parameter of the **create** operation.
- The **move** operation takes a point and a number and modifies the point so that subsequent **location** operations will return that number.
- The **display** operation takes a point and a "stream" (a realization of an output device) and prints the string "Point at *location*" on the stream, where **location** is its current location.

[4] Since the **Point** class is central to the design of Smalltalk's graphical user interface, no sane programmer would ever redefine it. We do so here solely for pedagogical reasons.

The second type, **history-point**, is a **point** which is associated with a list containing a record of its initial position and all subsequent moves. Its description is identical to that of **point** with the following addition:

- The **history** operation returns an ordered list representing the sequence of **locations** the point has had since its creation.

The third abstract type is **bounded-point**. Intuitively, bounded points are associated with a **min** and a **max** which define a lower and upper bound for **location**. I.e., for any bounded point p:

min (p) <= location (p) <= max (p)

The description for **bounded-point** can be derived from the description of **point** via four additional operations and three additions to the description of the **point** operations.[5] The new operations are as follows.

- The **min** operation takes a point and returns the current upper bound for the point.
- The **max** operation takes a point and returns the current lower bound for the point.
- The **setmin** operation takes a point and a number. If the number is less than or equal to the current location, the number is made the new lower bound for the point. Otherwise, an error is signalled.
- The **setmax** operation takes a point and a number. If the number is greater than or equal to the current location, the number is made the new upper bound for the point. Otherwise, an error is signalled.

Here are the modifications:

- The **create** operation takes to additional optional parameters, **min** and **max**. It returns a new **bounded-point**, using its parameters as the initial values of **location**, **min**, and **max**. If the **bounded-point** invariant does not hold between these three numbers, it signals an error rather than creating a new object. The default values of **min** and **max** are 0 and 100, respectively.
- The **move** operation will not move a point to a location smaller than **min** or larger than **max**.

[5] As an aside, the syntactic approach of the Larch specification system [GuttagHorning 83] facilitates the process of constructing formal specifications where "text" of new specifications can be described as a manipulation of old ones. This style of specification is well-suited for formal specification for inheritance-based type definition.

- The **display** operation also prints the **min** and **max**.

The final type, **bh-point**, can be viewed as a synthesis of **bounded-point** and **history-point**. It can be described as consisting of adding the description for the **history** operation given above and the description of **bounds-history** operation given below to the description of **bounded-point**.

- The **bounds-history** operation returns a list whose elements have the form

  (min *number history-length*)

  or

  (max *number history-length*)

  The semantics of each entry is that a change to min was made to *number* at when the **history** of the related point was *history-length* long. The bounds-history of a newly created point has one "min" entry and one "max" entry, corresponding to the initial minimum and maximum of the point. Each subsequent **setmax** and **setmin** causes an additional entry to be added to the end of the list.

The motivation for the **bounds-history** operation is that it, together with **history**, allows reconstruction of the sequence of state changes undergone by **bh-points**.

2.2 Representation description

The data structures used to represent instances of our four types are as follows:

- Instances of point are represented as a memory cell containing a number. Create creates a new cell and stores the specified number. Move stores a new number in the cell.

- Instances of history-point have three cells: one for the location, one for the beginning of a list of conses[6] containing the history, and one to the last cons cell of the history list. Create creates a list which contains the initial value as the only member, and stores it in history list and the tail pointer. Move replaces the cdr of the cons stored in the tail pointer cell with a new cons containing the new location. It then changes the tail pointer to contain the new cons. History returns a copy of the list rooted at the cons contained in the history list cell.

- Instances of bounded-point are represented using three memory cells. These cells contain numbers representing the current location, max, and min. Create checks whether the specified number is within the specified bounds, then creates a

[6]Conses are objects with car, cdr, rplaca, and rplacd, with the usual Lisp semantics. A Smalltalk OrderedCollection would provide a more elegant implementation, but would make comparison among the languages more difficult.

cell and stores the three values. **Move**, **setmin**, and **setmax** makes that the requested state change after checking that it would not violate the invariant.

- Instances of **BH-point** have seven cells. Five of these, for **location**, **min**, **max**, the history list, and the tail pointer of the history list, serve the same function as described above. The two new cells hold a pointer to the bounds-history list and a tail pointer to this list. Besides updating the min or max cell, setmin and setmax add a new cons cell to the bounds-history list by replacing the cdr of the tail pointer to that list.

2.3 Implementation level

An important criterion in the design of the modularization is the reduction of redundancy in procedure definition and redundancy in execution. The former refers to duplicated program text; the latter means the unnecessary multiple execution of procedures.

There are two ways in which this motivation is brought to bear on the design. First, methods of types will invoke methods of ancestor types where possible. Second, methods of types will be modularized such that methods of descendant types can call a subset of these procedures.

We will use the implementation of the **display** procedure will as a simple illustration of the rationale and use of this approach to method implementation.

2.3.1 The implementation of *display*

The implementation of the **display** operations is as follows.

- **Point.** The method is a procedure which prints (the name of) the type of the point, followed by the **location** of the point.
- **History-point.** The method invokes two procedures:
    1. The **point** method.
    2. A procedure which prints the location-history list.
- **Bounded-point.** The method invokes two procedures:
    1. The method for **point**.
    2. A procedure which prints the current upper and lower bounds.
- **BH-point.** The method invokes four procedures:

1. The **point** method.

2. Procedure (2) called by the **history-point** method.

3. Procedure (2) called by the **bounded-point** method.

4. A procedure which prints the bounds-history list.

This modularization introduces two auxiliary procedures, the of **bounded-point** and the location-history printer of **history-point.**

Why did history-point and bounded-point introduce extra procedures? Otherwise, the method for bh-point would have had either redundancy in execution or definition.

To see why this is so, suppose that the method for **bounded-point** invoked the method for **point**, then printed the upper bound, then printed lower bound.

```
(lambda (p)
  (point-display p)
  ... code for printing upper and lower bounds ... )
```

Suppose further that the method for **history-point** invoked the method for **point**, then printed the location history:

```
(lambda (p)
  (point-display p)
  ... code for printing location history ... )
```

How are we now to define the method for **BH-point**? Suppose we defined it to call the method of **bounded-point**, then call the method of **history-point**, then print the bounds history:

```
(lambda (p)
  (bounded-point-display p)
  (history-point-display p)
  ... code for printing location history ... )
```

Such a method would not exhibit the desired behavior because the method for point would be called twice: once by the method for **bounded-point**, and once by that of **history-point**. Thus, the result of displaying an instance of **bh-point** would produce something like:

```
bh-point at location: 5
  with bounds: 0, 100
bh-point at location: 5
  with location history: 5 17 32
  with bounds history: <min, 0, 0> <max, 20, 0> <max, 50, 2>
```

rather than:

```
bh-point at location: 5
  with bounds: 0, 100
  with location history: 5 17 32
  with bounds history: <min, 0, 0> <max, 20, 0> <max, 50, 2>
```

The above is an example of execution redundancy: the undesired multiple execution of a procedure. In this case, the redundancy produces observable erroneous behavior.

A second approach would involve copying parts of the **history-point** and **bounded-point** methods:

```
(lambda (p)
  (point-display p)
  ... code for printing upper and lower bounds ...
  ... code for printing location history ...
  ... code for printing bounds history ... )
```

Such a procedure would exhibit the desired behavior, but at the cost of redundancy of procedure definition.

Given that both of the above are undesirable, the rationale for the original implementation should be clear. There is no structural redundancy: the implementation of each of the four printing modules (location, upper/lower bounds, location history, and bounds history) each occur only in a single procedure. And there is no functional redundancy, since none of these procedures is invoked more than once.

2.3.2 The implementation of *create*

The **create** operation is implemented as follows.

- **Point.** The method calls two procedures.
    1. A procedure to allocate the storage for the new object.
    2. A procedure to store the value specified for **location** into the corresponding representation component of the new object.
- **Bounded-point.** The method calls four procedures:
    1. A procedure to verify that the specified location, min, and max are consistent.
    2. Procedure (1) called by the **point** method.
    3. Procedure (2) called by the **point** method.

   4. A procedure to store the values for the specified min and max in the corresponding representation components.

- **History-point.** The method calls three procedures.
   1. Procedure (1) called by the **point** method.
   2. Procedure (2) called by the **point** method.
   3. A procedure to create the first cons cell for the history list, and store it in the **hlist** and **htail** components of the new object.
- **BH-point.** The method calls five procedures:
   1. Procedure (1) called by the **bounded-point** method.
   2. Procedure (1) called by the **point** method.
   3. Procedure (2) called by the **point** method.
   4. Procedure (3) called by the **bounded-point** method.
   5. Prccedure (3) called by the **history-point** method.

A slight difference in the Zetalisp implementation will realize steps (1) and (4) of the **bounded-point** procedure as a single procedure.

2.3.3 The implementation of *move*

The implementation of the **move** operation in the four types is as follows:

- **Point.** The method is a procedure which changes the value of the **location** component of the representation.
- **Bounded-point.** The method invokes two procedures:
   1. A procedure to check the validity of the new location.
   2. The method for **point.**
- **History-point.** The method invokes two procedures:
   1. The method for **point.**
   2. A procedure to create the initial values of the **hlist** and **htail** components.
- **BH-point.** The method invokes three procedures:
   1. Procedure (1) called by the **bounded-point** method.

2. The method for **point**.

3. Procedure (2) called by the **history-point** method.

2.3.4 The implementation of *setmin* and *setmax*

The implementations of **setmin** and **setmax** both satisfy the following description:

- **Bounded-point.** The method is a procedure which first checks the bounds then stores into the relevant representation component.

- **BH-point.** The method invokes two procedures:

  1. The method for **bounded-point.**

  2. A procedure to add a new cons cell to the cell stored in the **bhtail** representation component, then stores the new cell in that component.

2.3.5 The implementations of the remaining operations

The implementations of **location**, **min**, **max**, **history**, and **bounds-history** are completely straightforward. They are all realize by the same procedure in all types for which the operation is meaningful.

- The method for **location**, **min**, and **max** returns the current value of the corresponding components of the representation.

- The method for **history** returns a copy of the list rooted in the **hlist** component.

- The method for **boundshistory** returns a copy of the list rooted in the **bhlist** component.

2.4 Four implementations of the example program

The following figures describe an implementation of the above-described program in each of our for languages. The Zetalisp implementation is in figures 2-1 through 2-4. The Loops implementation is in figures 2-5 through 2-10. The Smalltalk-82 implementation is in figures 2-11 through 2-17.[7] The Star Mesa implementation is in figures 2-18 through 2-27.

[7] Knowledgeable Smalltalk programmers will be surprised to see the familiar `compile:classified:` replaced with `compileAndStore:`. This simpler version was used in order to emphasize the similarity between operation set definition in Smalltalk and the other subject languages.

It should be clear that many different solutions could be constructed in our four languages. Without going into detail now, the solutions were chosen to illustrate the impact of different facilities for operation set definition and invocation. A more complete rationale will be provided later. described in more detail later.

In these programs, the types and operations have been given different names to make our example program consistent with the naming conventions used in the languages. For example, the operation name **display** is realized as `display` in Zetalisp, `Display` in Smalltalk and Star Mesa, and `display:` in Smalltalk.

Additionally, the internal procedures used in the different implementations are given different names. This reflects the different modularization paradigms used in the different approaches. For example, the `(:before :move)` method of the `bounded-point` flavor in Zetalisp, the `InternalBeforeMove` operation of the class `BoundedPoint` in Loops, the `internalMove` operation of the Smalltalk `BoundedPoint` class, and the `InternalMove` local procedure of the `BoundedPoint` trait in Star Mesa.

```
(defflavor point
        ((location 0))
        ()
  :gettable-instance-variables
  :initable-instance-variables)

(defmethod (point :move) (newloc)
  (setq location newloc)
  self)

(defmethod (point :display) (stream)
  (format stream
          "~%~s at location: ~d"
          (typep self)
          (send self :location)))
```

Figure 2-1: The Zetalisp point flavor

```
(defflavor history-point
        (hlist htail)
        (point))

(defmethod (history-point :after :init) (ignore)
  (setq hlist (list (send self :location)))
  (setq htail hlist))

(defmethod (history-point :after :move) (newloc)
  (rplacd htail (list newloc))
  (setq htail (cdr htail)))

(defmethod (history-point :history) ()
  (copylist hlist))

(defmethod (history-point :after :display) (stream)
  (format stream
          "~%  with history:~{ ~d~}"
          (send self :history)))
```

Figure 2-2: The Zetalisp history-point flavor

```
(defflavor bounded-point
        ((min 0)
         (max 100))
        (point)
  :settable-instance-variables
  :initable-instance-variables)

(defmethod (bounded-point :before :init) (unused)
  (if (not (and (<= min (send self :location)) (<= (send self :location) max)))
      (ferror "Location ~d for new point ~s is inconsistent with min ~d and max ~d."
              (send self :location)
              (send self :min)
              (send self :max))))

(defmethod (bounded-point :before :move) (newloc)
  (if (not (and (<= min newloc) (<= newloc max)))
      (ferror "New location ~d for point ~s is not between ~d and ~d."
              newloc self min max)))

(defmethod (bounded-point :before :set-min) (newmin)
  (if (not (<= newmin (send self :location)))
      (ferror "New minimum ~d for point ~s is greater than present location ~d."
              newmin self (send self :location))))

(defmethod (bounded-point :before :set-max) (newmax)
  (if (not (<= (send self :location) newmax))
      (ferror "New maximum ~d for point ~s is less than present location ~d."
              newmax self (send self :location))))

(defmethod (bounded-point :after :display) (stream)
  (format stream
          "~%  with bounds: ~d, ~d"
          (send self :min)
          (send self :max)))
```

Figure 2-3: The Zetalisp bounded-point flavor

```
(defflavor bh-point
        (bhlist bhtail)
        (bounded-point history-point))

(defmethod (bh-point :after :init) (ignore)
  (setq bhlist (list '(min ,min 0) '(max ,max 0)))
  (setq bhtail (cdr bhlist)))

(defmethod (bh-point :after :set-min) (newmin)
  (rplacd bhtail '((min ,newmin ,(length hlist))))
  (setq bhtail (cdr bhtail)))

(defmethod (bh-point :after :set-max) (newmax)
  (rplacd bhtail '((max ,newmax ,(length hlist))))
  (setq bhtail (cdr bhtail)))

(defmethod (bh-point :bounds-history) ()
  (copylist bhlist))

(defmethod (bh-point :after :display) (stream)
  (format stream
          "~%  with bounds history:~{ ~{<~s, ~d, ~d>~}~}"
          (send self :bounds-history)))
```

Figure 2-4: The Zetalisp bh-point flavor

```
(DEFINEQ (AllLocalMethods! (NLAMBDA .arglist.
    "Call the first parameter ''self'' and the second ''opname.''
    AllLocalMethods! invokes all local methods for opname of the class of self
    and all ancestors of that class."
    (PROG ((.args. (MAPCAR .arglist. 'EVAL))
           (self (CAR .args.))
           (opname (CADR .args.))
           (class (Send self Class))
           (ancestors (DREVERSE (Send class List! 'Supers))))
       (MAPC ancestors
             (LAMBDA (c)
                 (PROG ((handler (GetMethodHere c opname)))
                   (IF (NEQ handler NotSetValue)
                       (APPLY handler (CDDR .args.))))))))))

(PUTPROP 'AllLocalMethods
         'MACRO
         '(form '(AllLocalMethods! ,(CAR form) ',(CADR form) . ,(CDDR form))))
```

Figure 2-5: Utilities for the Loops implementation

```
(DefClass PointClass
    (MetaClass MetaClass)
    (Supers Class)
    (Methods (Create PointClass.Create)
             (New PointClass.New)))

(DEFINEQ (PointClass.Create (self plist)
    (PROG ((newobj))
        (SETQ newobj (SendSuper self New)) ;Invokes the New method of $Class.
        (Send newobj Initialize plist)
        newobj)))

(DEFINEQ (PointClass.New (self)
    (Send self Create Nil)))


(DefClass Point
    (MetaClass Class)
    (Supers Object)
    (InstanceVariables (loc 0))
    (Methods (Initialize Point.Initialize)
             (Location Point.Location)
             (Move Point.Move)
             (Display Point.Display)))

(DEFINEQ (Point.Initialize (self plist)
    (AllLocalMethods self PartialInitialize plist)))

(DEFINEQ (Point.PartialInitialize (self plist)
    (PutValue self 'loc (PlistGet plist 'location 0))))

(DEFINEQ (Point.Location (self)
    (GetValue self 'loc)))

(DEFINEQ (Point.Move (self loc)
    (PutValue self 'loc loc)
    self))


(DEFINEQ (Point.Display (self stream)
    (PRIN1 (Send self ClassName) stream)
    (PRIN1 " at location: " stream)
    (PRIN1 (Send self Location) stream)))
```

Figure 2-6: The Loops PointClass and Point classes

```
(DefClass HistoryPoint
    (MetaClass PointClass)
    (Supers Point)
    (InstanceVariables
        (hlist nil)
        (htail nil))
    (Methods
         (Initialize HistoryPoint.Initialize)
         (PartialInitialize HistoryPoint.PartialInitialize)
         (Move HistoryPoint.Move)
         (PartialMove HistoryPoint.PartialMove)
         (History HistoryPoint.History)
         (Display HistoryPoint.Display)
         (PartialDisplay HistoryPoint.PartialDisplay)))

(DEFINEQ (HistoryPoint.PartialInitialize (self plist)
    (PutValue self 'hlist (list (@ :loc)))
    (PutValue self 'htail (@ :hlist))))

(DEFINEQ (HistoryPoint.Move (self loc)
    (SendSuper self Move loc)           ; Invokes Point.Move
    (AllLocalMethods self PartialMove loc)))

(DEFINEQ (HistoryPoint.PartialMove (self loc)
    (RPLACD (@ :htail) (list (@ :loc)))
    (PutValue self 'htail (CDR (@ :htail)))))

(DEFINEQ (HistoryPoint.History (self)
    (COPY (@ :hlist))))

(DEFINEQ (HistoryPoint.Display (self stream)
    (SendSuper self Display stream)     ; Invokes Point.Display
    (AllLocalMethods self PartialDisplay stream)))

(DEFINEQ (HistoryPoint.PartialDisplay (self stream)
    (PRINT " with location history: " stream)
    (PRIN1 (Send self History) stream)))
```

Figure 2-7: The Loops HistoryPoint class

```
(DefClass BoundedPointClass
    (MetaClass MetaClass)
    (Supers PointClass)
    (Methods (Create BoundedPointClass.Create)
             (PartialCreate BoundedPointClass.PartialCreate)))

(DEFINEQ (BoundedPointClass.Create (self plist)
    (PROG ((newobj))
        (Send self PartialCreate plist)
        (SETQ newobj (DoMethod self 'New $Class plist))
                                        ; Invokes the New method of $Class
        (Send newobj Initialize plist)
        newobj)))

(DEFINEQ (BoundedPointClass.PartialCreate (plist)
    (PROG ((loc (PlistGet plist 'loc 0))
           (min (PlistGet plist 'min 0))
           (max (PlistGet plist 'max 100)))
       (COND ((NOT (AND (<= min loc) (<= loc max)))
              (ERROR "Location of new point is inconsistent with min and max."
                     (LIST loc min max))))))
```

Figure 2-8: The Loops BoundedPointClass class

```
(DefClass BoundedPoint
    (MetaClass BoundedPointClass)
    (Supers Point)
    (InstanceVariables
        (min 0)
        (max 100))
    (Methods
        (Initialize BoundedPoint.Initialize)
        (PartialInitialize BoundedPoint.PartialInitialize)
        (Move BoundedPoint.Move)
        (PartialMove BoundedPoint.PartialMove)
        (Min BoundedPoint.Min)
        (Max BoundedPoint.Max)
        (Setmin BoundedPoint.Setmin)
        (Setmax BoundedPoint.Setmax)
        (Display BoundedPoint.Display)
        (PartialDisplay BoundedPoint.PartialDisplay)))

(DEFINEQ (BoundedPoint.PartialInitialize (self plist)
    (PROG ()
       (@← :min (PlistGet plist 'min 0))
       (@← :max (PlistGet plist 'max 100)))))

(DEFINEQ (BoundedPoint.Move (self loc)
    (Send self PartialMove loc)
    (SendSuper self Move loc)))         ; Invokes Point.Move

(DEFINEQ (BoundedPoint.PartialMove (self loc)
    (IF (NOT (AND (<= (@ :min) loc) (<= loc (@ :max))))
        (ERROR "New location for point is not between min and max."
               (LIST loc self (@ :min) (@ :max))))))

(DEFINEQ (BoundedPoint.Max (self)
    (@ :max)))

(DEFINEQ (BoundedPoint.Min (self)
    (@ :min)))

(DEFINEQ (BoundedPoint.Setmin (self newmin)
    (IF (<= newmin (Send self Location))
        (@← :min newmin)
        (ERROR "New minimum for point is greater than present location."
              (LIST newmin self (Send self Location))))))

(DEFINEQ (BoundedPoint.Setmax (self newmax)
    (IF (<= (Send self Location) newmax)
        (@← :max newmax)
        (ERROR "New maximum for point is less than present location."
              (LIST newmax self (Send self Location))))))

(DEFINEQ (BoundedPoint.Display (self stream)
    (SendSuper self Display stream)     ; Invokes Point.Display
    (AllLocalMethods self PartialDisplay stream)))

(DEFINEQ (BoundedPoint.PartialDisplay (self stream)
    (PRINT " with bounds: " stream)
    (PRIN1 (Send self Min) stream)
    (PRIN1 ", " stream)
    (PRIN1 (Send self Max) stream)))
```

Figure 2-9: The Loops BoundedPoint class

```
(DefClass BHPoint
    (MetaClass BoundedPointClass)
    (Supers BoundedPoint HistoryPoint)
    (InstanceVariables
        (bhlist nil)
        (bhtail nil))
    (Methods
        (PartialInitialize BHPoint.PartialInitialize)
        (Move BHPoint.Move)
        (SetMin BHPoint.Setmin)
        (PartialSetMin BHPoint.PartialSetmin)
        (SetMax BHPoint.Setmax)
        (PartialSetMax BHPoint.PartialSetmax)
        (BoundsHistory BHPoint.BoundsHistory)
        (PartialDisplay BHPoint.PartialDisplay)))

(DEFINEQ (BHPoint.PartialInitialize (self plist)
    (@← :bhlist (LIST (LIST 'min (@ :min) 0)
                      (LIST 'min (@ :max) 0)))
    (@← :bhtail (CDR (@ :bhlist)))))

(DEFINEQ (BHPoint.Move (self loc)
    (DoMethod self 'PartialMove $BoundedPoint loc)
    (DoMethod self 'Move $Point loc)
    (DoMethod self 'PartialMove $HistoryPoint loc)))

(DEFINEQ (BHPoint.Setmin (self newmin)
    (SendSuper self Setmin newmin)      ; Invokes BoundedPoint.Move
    (Send self PartialSetmin newmin)))

(DEFINEQ (BHPoint.PartialSetmin (self newmin)
    (@← :bhtail (LIST 'min (Send self Min) (LENGTH (@ :hlist))))
    (@← :bhtail (CDR (@ :bhtail)))))

(DEFINEQ (BHPoint.Setmax (self newmax)
    (SendSuper self Setmax newmax)      ; Invokes BoundedPoint.Move
    (Send self PartialSetmax newmax)))

(DEFINEQ (BHPoint.PartialSetmax (self newmax)
    (@← :bhtail (LIST 'max (Send self Max) (LENGTH (@ :hlist))))
    (@← :bhtail (CDR (@ :bhtail)))))

(DEFINEQ (BHPoint.BoundsHistory (self)
    (COPY (@ :bhlist))))

(DEFINEQ (BHPoint.PartialDisplay (self stream)
    (PRINT " with bounds history: " stream)
    (PRIN1 (Send self BoundsHistory) stream)))
```

Figure 2-10: The Loops BHPoint class

```
Behavior
  addSelector: #compileAndStore
  withMethod:
    Class compile:
      'compileAndStore: sourceCode
        self addSelector: sourceCode extractOperationName
            withMethod: self compile: sourceCode'.

String
  addSelector: #extractOperationName
  withMethod:
    String compile:
      'extractOperationName
        "If self is an acceptable method description, extracts the name
         of the operation from the text.  If not, invokes the error:
          operation on self."
        ... definition omitted ...'.
```

... definition of the ConsCell class and operations (e.g., car, cdr, cons, rplaca, rplacd, copylist) are omitted ...

Figure 2-11: Utilities for the Smalltalk implementation

```
Object subclass: #Point
       instanceVariableNames: 'loc'
       classVariableNames: ''
       poolDictionaries: ''
       category: #CanonicalExample.

(Point class) compileAndStore:
  'create: plist
     |newobj|
     newobj ← self super.new.  "Invokes method for 'new' of Behavior".
     newobj initialize: plist.
     ^ newobj.'.

(Point class) compileAndStore:
  'new
     ^ self create: Nil.'.

Point compileAndStore:
  'initialize: plist
      loc ← plist at: #location ifAbsent: [0]'.

Point compileAndStore:
  'location
      ^ loc'.

Point compileAndStore:
  'move: newloc
      loc ← newloc'.

Point compileAndStore:
  'display: stream
      (self class) name printOn: stream.
      '' at location '' printOn: stream.
      (self location) printOn: stream'.
```

Figure 2-12: The Smalltalk Point class

```
Point subclass: #HistoryPoint
      instanceVariableNames: 'hlist htail'
      classVariableNames: ''
      poolDictionaries: ''
      category: #CanonicalExample.

HistoryPoint compileAndStore:
  'initialize: plist
     self super.initialize: plist.
     self partialInitialize: plist'.

HistoryPoint compileAndStore:
  'partialInitialize: unused
     hlist ← (ConsCell car: (self location) cdr: Nil).
     htail ← hlist'.

HistoryPoint compileAndStore:
  'move: newloc
     self super.move newloc.
     self partialMove newloc'.

HistoryPoint compileAndStore:
  'partialMove: newloc
     htail rplacd: (ConsCell car: (self location) cdr: Nil).
     htail ← htail cdr'.

HistoryPoint compileAndStore:
  'history
     ^ hlist copylist'.

HistoryPoint compileAndStore:
  'display: stream
    self super.display: stream.
    self partialDisplay: stream.'

HistoryPoint compileAndStore:
  'partialDisplay: stream
      stream cr.
      '' with location history: '' printOn: stream.
      (self history) printOn: stream'.
```

Figure 2-13: The Smalltalk HistoryPoint class

```
Point subclass: #BoundedPoint
      instanceVariableNames: 'min max'
      classVariableNames: ''
      poolDictionaries: ''
      category: #CanonicalExample.

(BoundedPoint class) compileAndStore:
  'create: plist
     |newobj|
     self partialCreate: plist.
     newobj ← self basicNew.  "Invokes method for 'new' of Behavior".
     newobj initialize: plist.
     ^ newobj.'.

(BoundedPoint class) compileAndStore:
  'partialCreate: plist
     |initmin initmax|
     initmin ← plist at: #location ifAbsent: [0].
     initmin ← plist at: #min ifAbsent: [0].
     initmax ← plist at: #max ifAbsent: [100].
     (initmin <=  location) & (location <= initmax)
        ifFalse:
          [self error:
            ''Location of new point is inconsistent with max and min.''].

BoundedPoint compileAndStore:
  'initialize: plist
     self partialInitialize: plist.
     self super.initialize: plist.'   "Invokes method for 'initialize' of Point."

BoundedPoint compileAndStore:
  'partialInitialize  plist: plist
     min ← plist at: #min ifAbsent: [0].
     max ← plist at: #max ifAbsent: [100]'.
```

Figure 2-14: The Smalltalk BoundedPoint class, part 1

```
BoundedPoint compileAndStore:
  'move: newloc
     self partialMove: newloc.
     self super.move: newloc.'

BoundedPoint compileAndStore:
  'partialMove: newloc
     (min <= newloc) & (newloc <= max) ifFalse:
        [self error: ''New location for point would be out of bounds.'']'.

BoundedPoint compileAndStore:
  'min
    ^ min'.

BoundedPoint compileAndStore:
  'max
    ^ max'.

BoundedPoint compileAndStore:
  'setmin: newmin
    (newmin <= (self location))
      ifTrue: min ← newmin
      ifFalse:
        [self error:
        ''New minimum for point is greater than present value.'']'.

BoundedPoint compileAndStore:
  'setmax: newmax
    ((self location) <= newmax)
      ifTrue: max ← newmax
      ifFalse:
        [self error:
          ''New maximum for point is less than present value.'']'.

BoundedPoint compileAndStore:
  'display: stream
    self super.display: stream.
    self partialDisplay: stream.'

BoundedPoint compileAndStore:
  'partialDisplay: stream
    stream cr.
    '' with bounds: '' printOn: stream.
    min printOn: stream.
    '', '' printOn: stream.
    max printOn: stream'.
```

Figure 2-15: The Smalltalk **BoundedPoint** class, part 2

```
BoundedPoint
     subclass: #BHPoint
     otherSuperclasses: (Array with: HistoryPoint)
     instanceVariableNames: 'bhlist bhtail'
     classVariableNames: ''
     poolDictionaries: ''
     category: #CanonicalExample.

(BHPoint class) compileAndStore:
  'create: plist
     ^ self BoundedPoint.class.create: plist'.


BHPoint compileAndStore:
  'initialize: plist
     "Note that partialInitialize.all would result in redundant execution
     of Point.partialInitialize.  The same is true for the method for display:"
     self Point.partialInitialize: plist.
     self HistoryPoint.partialInitialize: plist.
     self BoundedPoint.partialInitialize: plist.
     self partialInitialize: plist.'

BHPoint compileAndStore:
  'partialInitialize: unused
     bhlist ←
       (ConsCell car: (Array with: #min with: (self min) with: 0)
                cdr: Nil).
     bhlist rplacd:
        (ConsCell car: (Array with: #max with: (self max) with: 0)
                cdr: Nil).
     bhtail ← bhlist cdr'.
```

Figure 2-16: The Smalltalk-82 BHPoint class, part 1

```
BHPoint compileAndStore:
  'move: newloc
     self BoundedPoint.partialMove: newloc.
     self Point.move: newloc.
     self HistoryPoint.partialMove: newloc'.

BHPoint compileAndStore:
  'setmin: newmin
     self super.setmin: newmin.
     self partialSetmin: newmin'.

BHPoint compileAndStore:
  'partialSetmin: newmin
     bhtail rplacd:
       (ConsCell car: (Array with: #min with: newmin with: (hlist length))
                 cdr: Nil).
     bhtail ← bhtail cdr'.

BHPoint compileAndStore:
  'setmax: newmax
     self super.setmax: newmax.
     self partialSetmax: newmax'.

BHPoint compileAndStore:
  'partialSetmax: newmax
     bhtail rplacd:
       (ConsCell car: (Array with: #max with: newmax with: (hlist length))
                 cdr: Nil).
     bhtail ← bhtail cdr'.

BHPoint compileAndStore:
  'boundsHistory
     ^ bhlist copylist'.

BHPoint compileAndStore:
  'display: plist
     self Point.display: plist.
     self all.partialDisplay: plist.
     self partialDisplay: plist.'

BHPoint compileAndStore:
  'partialDisplay: stream
      stream cr.
      '' with bounds history: '' printOn: stream.
      (self boundsHistory) printOn: stream'.
```

Figure 2-17: The Smalltalk-82 BHPoint class, part 2

```
-- Documentation convention: in all trait definitions,
-- (a) the trait component type is TCType, and (b) the instance component
type is ICType.


-- A storage allocator

ConsAllocate: PROC [carSize: NATURAL] RETURNS [POINTER TO UNSPECIFIED] =
  ... Returns a pointer to a newly allocated storage region of size carSize + SIZE [POINTER TO UNSPECIFIED] ...


-- RealConsCell and related operations

RealConsCell: TYPE = PRIVATE POINTER TO RECORD [car: REAL, cdr: POINTER]

RealNil: RealConsCell ← LOOPHOLE [NIL, RealConsCell];

RealCons: PROC [initcar: REAL, initcdr: RealConsCell] RETURNS [RealConsCell] =
   {newcons: RealConsCell ←
             LOOPHOLE [ConsAllocate [SIZE [REAL]],
                       RealConsCell];
    newcons.car ← initcar;
    newcons.cdr ← initcdr;
    RETURN [newcons]};

RealCar: PROC [c: RealConsCell] RETURNS [Real] =
   {IF c = RealNil
    THEN ERROR;
    ELSE RETURN [c.car]}

RealCdr: [c: RealConsCell] RETURNS [RealConsCell] =
   {IF c = RealNil
    THEN ERROR;
    ELSE RETURN [c.cdr]}

RealRplaca: PROC [c: RealConsCell, newcar: REAL] =
   {IF c = RealNil
    THEN ERROR;
    ELSE c.car ← nowcar};

RealRplacd: PROC [c: RealConsCell] =
   {IF c = RealNil
    THEN ERROR;
    ELSE c.cdr ← newcdr};

RealLength: PROC [c: RealConsCell] RETURNS [Natural] =
   {IF c = RealNil
    THEN RETURN [0]
    ELSE RETURN [RealLength [RealCdr [c]] + 1]};

RealCopylist: PROC [c: RealConsCell] RETURNS [RealConsCell] =
   {IF c = RealNil
    THEN RETURN [RealNil]
    ELSE RETURN [RealCons [RealCar [c], RealCdr [c]]]}

RealPrintlist: PROC [c: RealConsCell, s: Stream] =
  ... definition omitted ... ;
```

Figure 2-18: Contents of a "Utilities" module for the Star Mesa implementation, part 1

```
-- Entry, EntryConsCell, and related operations --

Entry: TYPE = RECORD [minormax: STRING, location: REAL, histlength: NATURAL];

EntryConsCell: TYPE =
   PRIVATE POINTER TO RECORD [car: Entry, cdr: POINTER TO EntryConsCell];
```

... The definition of EntryCar, EntryNil, EntryCopylist, etc. are omitted ...

```
-- Procedures for string output

Stream: TYPE = ... definition omitted ...;

Print: PROC [string: String, stream: Stream] =
  ... definition omitted ...;

NewLine: PROC [stream: Stream] =
  ... definition omitted ...;
```

Figure 2-19: Contents of a "Utilities" module for the Star Mesa implementation, part 2

```
Point: TRAIT IMPORTS TM, Utilities = {

-- Introduced generic operation names: Point.Location, Point.Move, Point.Display.

ICHandle: TYPE = POINTER TO ICType;
ICType: TYPE =  RECORD [loc: REAL];

TCType: TYPE =
    RECORD [Location: PROC [p: TM.Object] RETURNS [REAL],
            Move: PROC [p: TM.Object] RETURNS [REAL],
            Display: PROC [p: TM.Object, s: Stream]];

Register: PROC [] RETURNS TM.RegistrationRecord = {
    RETURN [name: "Point",
            parents: [],
            ICSize: SIZE [ICType],
            TCSize: SIZE [TCType],
            classTrait: TRUE]};

Location: PROC [p: TM.Object] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, Point];
    RETURN [TC.Location [p]]};

Move: PROC [p: TM.Object, loc: REAL] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, Point];
    RETURN [TC.Move [p, loc]]};

Display: PROC [p: TM.Object, s: Stream] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, Point];
    RETURN [TC.Display [p, s]]};

Create: PROC [initloc: Real ← 0] RETURNS [TM.Object] =
   {newp: TM.Object ← TM.Alloc [Point];
    PartialInitialize [newp, initloc];
    RETURN [newp]};

PartialInitialize: PROC [newp: TM.Object, initloc: Real] =
    h: ICHandle ← TM.InstComp [newp, Point];
    h.loc ← initloc};
```

Figure 2-20: The Star Mesa **Point** trait, part 1

```
LocalInitializeTrait: PROC [] =
   {LocalInitializeTrait [Point]};

LocalInitializeTrait: PROC [trt: TM.Trait] =
   {PointTC: POINTER TO TCType ←
         TM.TCFromTrait [trt, Point];
    PointTC.Location ← LocationImpl;
    PointTC.Move ← MoveImpl;
    PointTC.Display ← DisplayImpl};

LocationImpl: PROC [p: TM.Object] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, Point];
    RETURN [h.loc]};

MoveImpl: PROC [p: TM.Object, loc: REAL] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, Point];
    h.loc ← loc;
    RETURN [loc]};

DisplayImpl: PROC [p: TM.Object, s: Stream] =
   {NewLine [s];
    Print [TM.TraitName [TM.Type [p]], s];
    Print [" at location: ", s];
    Print [Point.Location [p]]};

}
```

Figure 2-21: The Star Mesa **Point** trait, part 2

```
HistoryPoint: TRAIT IMPORTS TM, Utilities, Point = {

-- Introduced generic operation name: HistoryPoint.History

ICHandle: TYPE = POINTER TO ICType;
ICType: TYPE = RECORD [hlist: RealConsCell, htail: RealConsCell];

TCType: TYPE =
  RECORD [
    History: PROC [p: TM.Object] RETURNS [RealConsCell]];

Register: PROC [] RETURNS TM.RegistrationRecord = {
    RETURN [name: "HistoryPoint",
            parents: ["Point"],
            ICSize: SIZE [ICType],
            TCSize: SIZE [TCType],
            classTrait: TRUE]};

History: PROC [p: TM.Object] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, Point];
    RETURN [TC.History [p]]}.

Create: PROC [initloc: REAL ← 0] RETURNS [TM.Object] =
   {newp: TM.Object ← TM.Alloc [HistoryPoint];
    Point.PartialInitialize [newp, initloc];
    PartialInitialize [newp];
    RETURN [newp]};

PartialInitialize: PROC [newp: TM.Object, initloc: REAL] =
   {h: ICHandle ← TM.InstComp [newp, HistoryPoint];
    h.hlist ← RealCons [initloc, RealNil];
    h.htail ← h.hlist};
```

Figure 2-22: The Star Mesa **HistoryPoint** trait, part 1

```
InitializeTrait: PROC [] =
   {Point.LocalInitializeTrait [HistoryPoint]
    HistoryPoint.LocalInitializeTrait [HistoryPoint]};

LocalInitializeTrait: PROC [trt: TM.Trait] =
   {PointTC: POINTER TO Point.TCType ←
         TM.TCFromTrait [trt, Point];
    HistoryPointTC: POINTER TO TCType ←
         TM.TCFromTrait [trt, HistoryPoint];
    PointTC.Move ← MoveImpl;
    HistoryPointTC.History ← HistoryImpl;
    PointTC.Display ← DisplayImpl};

MoveImpl: PROC [p: TM.Object, loc: REAL] RETURNS [REAL] =
   {Point.MoveImpl [p, loc];
    PartialMove [p, loc];
    RETURN [loc]};

PartialMove: PROC [p: TM.Object, loc: REAL] RETURNS [TM.Object] =
   {h: ICHandle ← TM.InstComp [p, HistoryPoint];
    Rplacd [h.htail, RealCons [loc, RealNil]];
    h.htail ← RealCdr [h.htail]};

HistoryImpl: PROC [p: TM.Object] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, HistoryPoint];
    RETURN [RealCopylist [h.hlist]]};

DisplayImpl: PROC [p: TM.Object, s: Stream] =
   {Point.Display [p, s];
    PartialDisplay [p, s]};

PartialDisplay: PROC [p: TM.Object, s: Stream] =
   {NewLine [s];
    Print [" with location history:", s];
    RealPrintlist [HistoryPoint.History [p], s]};
}
```

Figure 2-23: The Star Mesa HistoryPoint trait, part 2

```
BoundedPoint: TRAIT IMPORTS TM, Utilities, Point  = {

-- Introduced generic operation names:
--    BoundedPoint.Min, BoundedPoint.Max, BoundedPoint.SetMin, BoundedPoint.SetMax

ICHandle: TYPE = POINTER TO ICType;
ICType: TYPE = RECORD [min: REAL, max: REAL];

TCType: TYPE =
  RECORD [
    Min: PROC [p: TM.Object] RETURNS [REAL],
    Max: PROC [p: TM.Object] RETURNS [REAL],
    SetMin: PROC [p: TM.Object, min: [REAL] RETURNS [REAL],
    SetMax: PROC [p: TM.Object, max: [REAL] RETURNS [REAL]];

Register: PROC [] RETURNS TM.RegistrationRecord = {
    RETURN [name: "BoundedPoint",
            parents: ["Point"],
            ICSize: SIZE [ICType],
            TCSize: SIZE [TCType],
            classTrait: TRUE]};

Min: PROC [p: TM.Object] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
        TM.TCFromObject [p, BoundedPoint];
    RETURN [TC.Min [p]]};

Max: PROC [p: TM.Object] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
        TM.TCFromObject [p, BoundedPoint];
    RETURN [TC.Max [p]]};

SetMin: PROC [p: TM.Object, newmin: REAL] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
        TM.TCFromObject [p, BoundedPoint];
    RETURN [TC.SetMin [p, newmin]]};

SetMax: PROC [p: TM.Object, newmax: REAL] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
        TM.TCFromObject [p, BoundedPoint];
    RETURN [TC.SetMax [p, newmax]]};

Create: PROC [initloc: REAL ← 0, initmin: REAL ← 0, initmax: REAL ← 100]
          RETURNS [TM.Object] =
   {newp: TM.Object;
    PartialCreate [initloc, initmin, initmax];
    newp ← TM.Alloc [BoundedPoint];
    Point.PartialInitialize [newp, initloc];
    PartialInitialize [newp, initmin, initmax];
    RETURN [newp]};

PartialCreate: PROC [initloc, initmin, initmax: REAL] =
   {IF NOT (initmin <= initloc & initloc <= initmax)
      THEN ERROR;}

PartialInitialize: PROC [newp: TM.Object, initmin, initmax: REAL] =
   {h: ICHandle ← TM.InstComp [newp, BoundedPoint];
    h.min ← initmin;
    h.max ← initmax};
```

Figure 2-24: The Star Mesa **BoundedPoint** trait, part 1

```
InitializeTrait: PROC [] =
   {Point.LocalInitializeTrait [BoundedPoint];
    BoundedPoint.LocalInitializeTrait [BoundedPoint]};

LocalInitializeTrait: PROC [trt: TM.Trait] =
   {PointTC: POINTER TO Point.TCType ←
         TM.TCFromTrait [trt, BoundedPoint];
    BoundedPointTC: POINTER TO TCType ←
         TM.TCFromTrait [trt, Point];
    PointTC.Move ← MoveImpl;
    BoundedPointTC.Min ← MinImpl;
    BoundedPointTC.Max ← MaxImpl;
    BoundedPointTC.SetMin ← SetMinImpl;
    BoundedPointTC.SetMax ← SetMaxImpl;
    PointTC.Display ← DisplayImpl};

MinImpl: PROC [p: TM.Object] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BoundedPoint];
    RETURN [h.min]};

MaxImpl: PROC [p: TM.Object] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BoundedPoint];
    RETURN [h.max]};

MoveImpl: PROC [p: TM.Object, loc: REAL] RETURNS [TM.Object] =
   {PartialMove [p, loc];
    RETURN [Point.Move [p, loc]]};

PartialMove: PROC [p: TM.Object, loc: REAL] RETURNS [TM.Object] =
   {h: ICHandle ← TM.InstComp [p, BoundedPoint];
    IF NOT (h.min <= loc & loc <= h.max)
      THEN ERROR};

SetMinImpl: PROC [p: TM.Object, min: REAL] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BoundedPoint];
    IF min <= Point.Location [p]
      THEN h.min ← min
      ELSE ERROR;
    RETURN [min]};

SetMaxImpl: PROC [p: TM.Object, max: REAL] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BoundedPoint];
    IF Point.Location [p] <= max
      THEN h.max ← max
      ELSE ERROR;
    RETURN [max]};

DisplayImpl: PROC [p: TM.Object, s: Stream] =
   {Point.Display [p, s];
    PartialDisplay [p, s]};

PartialDisplay: PROC [p: TM.Object, s: Stream] =
   {NewLine [s];
    Print [" with bounds: ", s];
    Print [BoundedPoint.Min [p], s];
    Print [" , ", s];
    Print [BoundedPoint.Max [p], s]};

}
```

Figure 2-25: The Star Mesa **BoundedPoint** trait, part 2

```
BHPoint: TRAIT
   IMPORTS TM, Utilities, Point, BoundedPoint, HistoryPoint = {

-- Introduced generic operation name: BHPoint.BoundsHistory

ICHandle: TYPE = POINTER TO ICType;
ICType: RECORD [bhlist: EntryConsCell, bhtail: EntryConsCell];

TCType: TYPE =
   RECORD [
    BoundsHistory: PROC [p: TM.Object] RETURNS [EntryConsCell]];

Register: PROC [] RETURNS TM.RegistrationRecord = {
    RETURN [name: "BHPoint",
            parents: ["BoundedPoint", "HistoryPoint"],
            ICSize: SIZE [ICType],
            TCSize: SIZE [TCType],
            classTrait: TRUE]};

BoundsHistory: PROC [p: TM.Object] RETURNS [EntryConsCell] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, BHPoint];
    RETURN [TC.BoundsHistory [p]]};

Create: PROC [initloc: REAL ← 0, initmin: REAL ← 0, initmax: REAL ← 100]
          RETURNS [TM.Object] =
   {newp: TM.Object;
    BoundedPoint.PartialCreate [initloc, initmin, initmax];
    newp ← TM.Alloc [BoundedPoint];
    Point.PartialInitialize [newp, initloc];
    BoundedPoint.PartialInitialize [newp, initmin, initmax];
    HistoryPoint.PartialInitialize [newp, initloc];
    PartialInitialize [newp, initmin, initmax];
    RETURN [newp]};

PartialInitialize: PROC [newp: TM.Object, initmin, initmax: REAL] =
   {h: ICHandle ← TM.InstComp [newp, BHPoint];
    entry1: Entry ["min", min, 0];
    entry2: Entry ["max", max, 0];
    h.bhlist ← EntryCons [entry1, EntryCons [entry2, EntryNil]];
    h.bhtail ← EntryCdr [h.bhlist]};
```

Figure 2-26: The Star Mesa BHPoint trait, part 1

```
InitializeTrait: PROC [] =
   {Point.LocalInitializeTrait [BHPoint];
    BoundedPoint.LocalInitializeTrait [BHPoint];
    HistoryPoint.LocalInitializeTrait [BHPoint];
    BHPoint.LocalInitializeTrait [BHPoint]};

LocalInitializeTrait: PROC [trt: TM.Trait] =
   {PointTC: POINTER TO Point.TCType ←
         TM.TCFromTrait [trt, Point];
    BoundedPointTC: POINTER TO Point.TCType ←
         TM.TCFromTrait [trt, BoundedPoint];
    BHPointTC: POINTER TO TCType ←
         TM.TCFromTrait [trt, BHPoint];
    PointTC.Move ← MoveImpl;
    BoundedPointTC.SetMin ← SetMinImpl;
    BoundedPointTC.SetMax ← SetMaxImpl;
    BHPointTC.BoundsHistory ← BoundsHistoryImpl;
    PointTC.Display ← DisplayImpl};

MoveImpl: PROC [p: TM.Object, loc: REAL] RETURNS [REAL] =
   {BoundedPoint.PartialMove [p, loc];
    Point.Move [p, loc]
    HistoryPoint.PartialMove [p, loc];
    RETURN [loc]};

SetMinImpl: PROC [p: TM.Object, min: REAL] RETURNS [REAL] =
   {BoundedPoint.SetMinImpl [p, min];
    PartialSetMin [p, min];
    RETURN [min]};

PartialSetMin: PROC [p: TM.Object, min: REAL] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BHPoint];
    historyh: HistoryPoint.handle ← TM.InstComp [p, HistoryPoint];
    newentry: Entry ←
      Entry ["min", Point.Location [p], RealLength [historyh.bhlist]];
    EntryRplacd [h.bhtail, EntryCons [newentry, EntryNil]];
    h.bhtail ← EntryCdr [h.bhtail]};

SetMaxImpl: PROC [p: TM.Object, max: REAL] RETURNS [REAL] =
   {BoundedPoint.SetMaxImpl [p, max];
    PartialSetMax [p, max]
    RETURN [max]};

PartialSetMax: PROC [p: TM.Object, max: REAL] RETURNS [REAL] =
   {h: ICHandle ← TM.InstComp [p, BHPoint];
    historyh: HistoryPoint.ICHandle ← TM.InstComp [p, HistoryPoint];
    newentry: Entry ←
      Entry ["max", Point.location [p], RealLength [historyh.hlist]];
    EntryRplacd [h.bhtail, EntryCons [newentry, EntryNil]];
    h.bhtail ← EntryCdr [h.bhtail]};

BoundsHistoryImpl: PROC [p: TM.Object] RETURNS [EntryConsCell] =
   {h: ICHandle ← TM.InstComp [p, BHPoint];
    RETURN [EntryCopylist [h.bhlist]]};

DisplayImpl: PROC [p: TM.Object, s: Stream] =
   {Point.Display [p, s];
    BoundedPoint.PartialDisplay [p, s];
    HistoryPoint.PartialDisplay [p, s];
    PartialDisplay [p, s]};

PartialDisplay: PROC [p: TM.Object, s: Stream] =
   {NewLine [s];
    Print [" with bounds history:", s];
    EntryPrintlist [BHPoint.BoundsHistory [p], s]};
}
```

Figure 2-27: The Star Mesa BHPoint trait, part 2

3. Generic invocation

Each of our subject languages supports one or more forms for procedure invocation where the identity of the invoked procedure depends on (a) a name derived from the form, and (b) the type of a distinguished parameter of the form. Such expressions are commonly referred to as **generic** procedure invocations. We will refer to the name as the **operation name** of the invocation and to the distinguished parameter as the **generic parameter** of the invocation.

Unlike the early approaches of PL/1 [IBM 64] and Algol 68 [Branquart et al 71], the procedure invoked by a generic invocation expression depends on a type derived from a *single* parameter of the invocation. Thus, for each type T, we can identify a set of associations between operation names and procedures which the generic invocation form will invoke when the generic parameter has type T. We will refer to this association between operation names and procedures as the **generic operation set** of the type. The procedure associated with a particular operation name will be called the **method** for that operation name.

There is a clear analogy between the generic operation set of a type and the "abstract operations" of "abstract types," since both identify types with a set of name/procedure associations. However, there are two fundamental differences.

- First, for all but a fixed number of procedures, the set of abstract operations provide the only way in which instances of abstract types can be manipulated. The same assertion cannot be made for the methods of the generic operation set of Zetalisp flavors, Smalltalk or Loops classes, or Star Mesa traits.

- Second, not all abstract-type designs involve generic invocation. For example, generic invocation is not possible for the operations of types defined via the **abstype** constructor of ML [Gordon et al. 79], or for procedures used as abstract operations for "private" types defined in Mesa or Euclid modules. In these approaches, the names of the operations of different types must be disjoint.

A second related concept is the popular notion of "object-oriented programming." Although no single definition is universally accepted for this term (see [Rentsch 82] for a discussion of one person's view), it is often the case that a procedure invocation mechanism which uses an operation set associated with a distinguished parameter is present. However, it is not always the case that such a mechanism exists (e.g., it does not in the "actor" paradigm described in section 3.1.4). Even if such a mechanism *is* present, it may be the case that objects do not have types of which their operation set is a function (e.g., the "objects" of the T dialect of Lisp [Rees et al. 84]).

The generic operation sets of our subject types are defined by inheritance. This chapter describes why generic invocation is useful and how it is realized in our four languages. Chapters 6 and 7 contain a detailed discussion of the algorithms used in our four subject languages.

3.1 The semantics of generic invocation

In each of our four languages, generic invocation results in the invocation of a procedure on some parameters, possibly after binding some variables. The following describes how that general concept is realized in each of our four languages.

3.1.1 The Star Mesa and Loops semantics

The most straightforward semantics for generic invocation are those which apply to Star Mesa and Loops.

In Star Mesa, any instance of a Mesa `PROCEDURE` type whose first parameter has type `TM.Object` can be used as a generic method. The generic invocation:

operation-proc [exp_0, exp_1, ... exp_N]

where results in invoking the procedure used as the generic method with parameters:

exp_0, exp_1, ... exp_N

In Loops, any object which the Interlisp interpreter recognizes as a function can be used as a generic method. Twelve different kinds of procedures are recognized, as described by the following excerpt from section 4 of [Teitelman 78]:

In Interlisp, each function may independently:

a. have its arguments evaluated ["lambda" functions] or not evaluated ["nlambda" functions)];

b. have a fixed number of arguments ["spread" functions] or an indefinite number of arguments ["nospread" functions];

c. be defined by an Interlisp expression ["exprs"], by built-in machine code ["subrs"], or by compiled machine code ["cexprs"].

Hence there are twelve function types (2 x 2 x 3).

Recall that generic invocation in Loops is accomplished by an invocation of the `Send!` or `Send` procedures. The method used to carry out the generic invocation:

(`Send!` exp_0 *operation* exp_1 ... exp_N)
(`Send` exp_0 *operation-id* exp_1 ... exp_N)

will be invoked with the parameters:

exp_0 exp_1 ... exp_N

Since each of the exp_i will be evaluated when the `Send!` procedure is invoked, using an nlambda procedure as a method will not prevent evaluation of the method parameters.

3.1.2 The Smalltalk semantics

In Smalltalk, the procedures invoked by generic invocation are usually instances of the Smalltalk class `CompiledMethod`.[8] These objects are most commonly obtained by invoking the Smalltalk compiler on a text string which is a *method* in the syntax of the Smalltalk language ([GoldbergRobson 83], p. 715 ff.). The examples of chapter 2 contain examples of such definitions. E.g., in the following expression from figure 2-12:

```
Point compileAndStore:
  'location
     ^ loc'
```

the string

```
'location
   ^ loc'
```

defines a Smalltalk *method* which will be transformed into an instance of class `CompiledMethod` and used to carry out generic invocations of the `location` operation on instances of class `Point`.

When a method is invoked, the distinguished variable `self` is bound to the generic parameter

[8]The Smalltalk compiler always produces instances of `CompiledMethod`. However, since the Smalltalk interpreter does no type-checking of methods, an instance of any class could, in principle, be used as a generic method.

and the formal parameters of the compiled method are bound to the non-generic parameters. Thus, a method which is to carry out generic invocations with some number of non-generic parameters should have the same number of formal parameters.

Given Smalltalk's unorthodox syntax, we will briefly summarize the syntax relevant to identifying the parameters of a *method*. If the first token of the method definition string is an identifier (e.g., `'location'`), the operation name is the symbol corresponding to the identifier and there are no formal parameters. If the first token is a binary selector (e.g., `'+ ...'`), the corresponding symbol is the operation name and the next token, which must be a simple identifier, is the single formal parameter. Otherwise, the head of the definition must contain one or more occurrences of a keyword (i.e., an identifier with a colon suffix) followed by an identifier. Given that there are N pairs of keywords and identifiers, the operation name is the concatenation of those N keywords, and the N formal parameters are the N identifiers. Thus, in the method definition string:

```
    'sumArg1: x arg2: y  ^ (+ x (+ y self))'
```

the operation name is `sumArg1:Arg2:` and the two formal parameters are x and y."

As further illustration, consider the following examples.

- The compiled method produced from the string:
  ```
      'location ^ loc'
  ```
  has no formal parameters, and can thus be used for invocations with no non-generic parameters, e.g.,
  ```
      P location
  ```
  or
  ```
      P perform: #location
  ```
  The variable `self` would be bound to the value of P during the invocation.

- The compiled method corresponding to:
  ```
      '** x ^ (x * x)'
  ```
  has one formal parameter, x. If used to carry out the invocations:
  ```
      10 ** 20
  ```
  or
  ```
      10 perform: #** with: 20
  ```
  `self` would be bound to 10, x to 20, and the returned value would be 200. (The form ^ *exp* causes the value of *exp* to be returned as the value of the method invocation.)

- The procedure produced from:
```
      'move: newloc
        loc ← newloc'
```
  has one formal parameter, **newloc**, so could be used to carry out:
```
      P move: 10
```
  or
```
      P perform #move: with: 10
```
  The variable **self** would be bound to the value of P and **newloc** would be bound to **10**.

- The procedure produced from:
```
      'sumArg1: x arg2: y  ^ (+ x (+ y self))'
```
  has two formal parameters, x and y. If used as the method for:
```
      10 sumArg1: 20 Arg2: 30
```
  or:
```
      10 perform: #sumArg1:Arg2: with: 20 with: 30
```
  x would be bound to 20, y would be bound to 30, and the result would be 60.

3.1.3 The Zetalisp semantics

Generic invocation in Zetalisp results in an invocation of an ordinary Lisp function. Any object recognized by the Lisp interpreter as a function can be invoked as a generic method; see [Symbolics 84] or [MoonStallmanWeinreb 84] for a complete enumeration. The most common of these are lists whose first element is the symbol **lambda** or **named-lambda**, and the objects produced by the Zetalisp compiler, known as "Function Entry Frames". The **lambda** form is well-known; an example of a named lambda is:
```
    (named-lambda h (x) (f (g x)))
```
The symbol h is an internal name for the function; this name usually corresponds to the global name to which the function is bound. The use of the standard **defun** form produces named-lambdas, as do the **defmethod** forms used in chapter 2.

Surprisingly enough, the parameters passed to the generic method are an implementation-dependent aspect of Zetalisp. However, all implementations of Zetalisp supports the pleasant fiction that no such dependence on implementation details exists. We first describe the fiction, the reality.

3.1.3.1 The *defmethod* abstraction

Functions to be used as Zetalisp methods can be created via **defmethod** forms. Such forms must have the structure:

```
(defmethod (id₁ ... id_N) parameter-list
    exp
    ...
    exp)
```

where id_1 is the name of a flavor and *parameter-list* is as in a conventional **lambda** form. A simple example of such a form is:

```
(defmethod (.point :move) (newloc)
  (setq location newloc)
  self)
```

Subsequent chapters will describe the semantics of **defmethod** forms in detail. For now, we note that such forms will produce a function which, if used as the method for our canonical generic invocation:

```
(send exp₀ operation exp₁ ... exp_N)
```

will result in the following bindings.

- The value of exp_0 will be bound to the variable **self**.
- The value of *operation* will be bound to the variable **operation**.
- The values of exp_1 ... exp_N will be bound to the first through Nth formal parameters specified by *lambda-list*.

To illustrate, suppose the function produced by the **defmethod** form given above were used as the method for the generic invocation:

```
(send p :move 10)
```

The variable **self** would be bound to the value of **p**, **operation** would be bound to the symbol **:move**, and **newloc** would be bound to 10.

3.1.3.2 The reality

Let F be the function which is to be called via the generic invocation mechanism.

- In the system described in [WeinrebMoon 81], the form

  ```
  (send exp₀ exp_operation exp₁ ... exp_N)
  ```

  binds the variable **self** to the value of exp_0, then calls F with parameters:

  $exp_{operation}$ exp_1 ... exp_N

  This implementation is no longer in use.

- In the system described in [Symbolics 84], the form

  `(send` exp_0 $exp_{operation}$ exp_1 `...` exp_N`)`

  calls F with parameters:

  exp_0 *mapping-table* $exp_{operation}$ exp_1 ... exp_N

  The identity of the *mapping-table* object is determined by the flavor of which the generic parameter is an instance and the flavor named by the id_1 of the **defmethod** which created the procedure being invoked.[9] The significance of the mapping table is described in section 5.3.

- In the system described in [MoonStallmanWeinreb 84], the form

  `(send` exp_0 $exp_{operation}$ exp_1 `...` exp_N`)`

  binds **self** to the value of exp_0, binds **self-mapping-table** to an appropriate mapping table, then calls F with parameters:

  $exp_{operation}$ exp_1 ... exp_N

  The significance of the mapping table is the same as above.

The important point to notice is that in the latter two implementations, the ordinary function invocation mechanism cannot always be used to invoke functions used as generic methods. Without an appropriate binding for **self-mapping-table**, the functions can behave incorrectly.

Many Zetalisp programmers are completely unaware of the above. This is due to the following important property:

If a procedure defined via **defmethod** *is invoked through the generic invocation mechanism,* **self-mapping-table** *will always be correct.*

The payoff is that if a program contains no explicit invocations of such procedures (i.e., no **funcalls** of the procedure), the existence of **self-mapping-table** can be considered an implementation detail.

Given that **defmethod** and the generic invocation mechanism can hide the existence of **self-mapping-table**, why can't its existence be ignored entirely? As we will see shortly , knowledge of its existence is necessary to invoke a generic method of one type on an instance of a descendant type.

[9] This is not completely accurate. In reality, the identity of the second flavor is determined by a declaration inserted in the body of procedures created by **defmethod** forms. This declaration is computed from the id_1 of the **defmethod** form.

3.1.4 Generic invocation vs. "message-passing"

Since the concept of "message-passing" is so often informally used, it is worth comparing to the approaches to generic invocation used in our four languages. As embodied in Smalltalk-72 [GoldbergKay 76, Shoch 79] and the "actor" family of languages (e.g., [HewittBishopSteiger 73, Hewitt 77, Lieberman 80, Theriault 83]) message passing is analogous to conventional generic invocation where:

- objects are conceptualized as conventional procedures (i.e., "scripts") with private data,
- the operation name and the non-generic parameters of an invocation are packaged in a "message" object,
- a generic invocation is analogous to a conventional invocation of procedure associated with the generic parameter given the message as its parameter, and
- the possible continuations for each generic invocation are explicitly passed as additional parameters.

This approach was clarified in the work on the Scheme language at M.I.T. [SussmanSteele 78a, SussmanSteele 78b], which showed that a message-passing programming methodology could be realized in languages with conventional procedure invocation and either lexical or dynamic procedure closures. Zetalisp's `entity` datatype, designed before the introduction of flavors, is another realization of the latter idea.

Aside from the impact of continuation passing on the control structure, the most significant difference between generic invocation and message-passing is that the operation set of a type is realized as a monolithic procedure used for every operation rather than a collection of independent procedures for separate invocations. The key advantage of the latter approach is that the structure imposed on the methods of the operation set is that it makes it easier to define algorithms which synthesize operation sets of new types from those of existing ones. For example, it is impossible to determine whether two "actors" will follow the same "script" in processing a given kind of "message." Thus, it is no accident that none of our four languages uses message-passing semantics for generic invocation.

3.2 The utility of generic invocation

Although support for generic procedure invocation is one of the key design aspects of our four subject languages, the benefits of using this technique are sometimes poorly understood. This section presents a careful description of the advantages gained by this approach.

A generic invocation form causes the invocation of a procedure based on the type of a parameter. In our four subject languages, as well as in many others, the effect of a generic invocation of an operation O can be obtained by a conditional expression with one arm for each type whose generic operation set includes a method for O. We will refer to such expressions as "type-conditional" expressions.

For example, consider a Loops program in which the types **$Point**, **$BoundedPoint**, **$HistoryPoint**, and **$BHPoint** were the only types whose generic operation set contained a method for the **Location** operation. Suppose the procedures **Point.Location**, **BoundedPoint.Location**, **HistoryPoint.Location**, and **BHPoint.Location** were the **Location** methods for these four types. Given that the class of x is C,[10] the application of the **Location** method of the generic operation set of C to the single parameter x could be achieved by the type-conditional expression:

```
    (COND ((EQ C $Point) (Point.Location x))
          ((EQ C $BoundedPoint) (BoundedPoint.Location x))
          ((EQ C $HistoryPoint)(HistoryPoint.Location x))
          ((EQ C $BHPoint) (BHPoint.Location x))
          (T (ERROR "No 'Location' operation")))
```

Similar expressions could be written in each of our four languages.

Generic invocation expressions have two fundamental advantages over type-conditional expressions.

- *Generic invocation makes programs more concise.* The size of a type-dispatch expression increases with the number of types that include the operation in their generic operation sets. The size of a generic invocation expression is constant. The

[10]The class of an arbitrary object x in Loops can be obtained by the expression "**(Send x Class)**". As we will see, equivalent procedures are available in our other three languages.

result is less work for programmers, less opportunity to make errors (e.g., misspelling a type name), and increased program clarity.

- *Generic invocation makes type addition easier.* Existing generic invocations need no modification to invoke methods of new types. But if an existing type-enumeration invocations is to invoke a method of a new type, the invocation must be modified to add a clause for the new type. Note that this work cannot be done by the type designer; it must be done by all designers of procedures which use the type.

The fact that an existing invocation is applicable to a new type is of little use if the invocation does not have the appropriate effect when applied to a generic parameter of the new type. But it should be clear that not all properties of all existing invocations will hold when applied to instances of new types. As a result, programs which include such invocations might not have the desired effect after new descendant types are introduced. For example, a program which uses the **move** operation on instances of **point** might well cause an "out of bounds" error when applied to instances of **bounded-point**.

Since the incremental augmentation of programs by the introduction of new types is a principal motivation for the use of type definition by inheritance, knowing the conditions under which the new program will maintain the functionality of the original is of considerable importance. Fortunately, it *is* possible to define a set of conditions which is sufficient to guarantee that the desired functionality of existing programs will be preserved when descendant types are introduced. The following conditions are sufficient to guarantee that a property **P** which holds for a procedure **Proc** will be preserved when a new type is introduced.

1. *Valid specifications are provided for the generic operation sets of each type.* The methods of the generic operation sets of each type should be associated with valid assertions with the effect of their invocation. These are nothing more than conventional "type specifications" (e.g., [LiskovBerzins 79, Guttag 80]) treating the generic operations as "abstract operations". We will refer to these assertions as "generic specifications" of the type.

2. *The generic specifications for a type imply the generic specifications for all parent types.* In other words, everything which can be concluded from the generic specifications of a type about some sequence of generic invocations can also be concluded from the generic specifications of each of its children. As a result, generic specifications for a type are specializations the generic specifications of all ancestor types. In our example scenario, the specifications of **bh-point** would imply the specifications, of **bounded-point**, **history-point**, and **point**.

3. *For each generic invocation expression of each procedure of the program, a type T can be identified such that for all possible computations, the type of the generic parameter will be either T or a descendant of T.* We will refer to this type as the "assumed generic type" of the invocation. Such a demonstration could be accomplished in several ways, e.g., static or dynamic type checking, dataflow analysis, or arbitrary reasoning about the program.

4. *For each type T, the only procedures which invoke the underlying operations of T are the methods of T.* The generic specifications of each type define an abstraction. Unless **Proc** is a method of some type, its definition should be in terms of invocations of such operations, rather than the more basic operations used to realize the abstraction. For example, procedures which are not methods of **history-point** should not directly manipulate the **hlist** and **htail** components of its representation.

This set of conditions can be intuitively summarized as: "generic operations should be treated as abstract operations, and the specification of the abstract operation set of each type should be a specialization of the specifications of the operation sets of all ancestor types." In chapter 6, we will examine the support offered by our four languages in the mechanical detection of the violation of this principle.

For example, consider a procedure which contains exactly one generic invocation expression: an invocation of the **history** operation where generic parameter is *exp*. Suppose that it could be shown that for all possible computations, the type of the value of *exp* would be **history-point** or a descendant. Suppose further that the only assumption about the effect of the invocation was that contained in the specification of **history-point**, i.e., that a list containing the previous locations of the point denoted by *exp* would be returned. Then if the generic specifications of **history-point** were satisfied by the methods of each of its descendants, the introduction of new descendants could not falsify the assumption about the effect of the generic invocation of **history**.

The benefit of designing programs which satisfy the above-stated conditions is that the effect of a given invocation can be relied on given only partial information about the type of the generic parameter, i.e., that it is a descendant of a given type. But the considerable benefit of using such a methodology does extract a price: existing programs might need to be modified when new types are added.

To illustrate, suppose that an earlier stage of the example scenario of chapter 2 contained only one type, **point**, and that the specification for the **move** operation of **point** was that the **location** of the point would become the specified coordinate. Upon the introduction of the **bounded-point** type, it would be observed that the desired functionality of **move** for **bounded-point** (i.e, that some invocations would cause an error rather than a change of location) would not satisfy the specification of **move** for **point**. This would require a choice between three alternatives.

- First, the specification of the **move** operation for **point** could be changed so that it would be satisfied by the **move** of **bounded-point**. An example of such a specification would be "either the coordinate is changed or an error is signalled."

  The problem with this approach is that the changes are "incompatible," that is, the new specifications do not entail the old. As a result, programs which relied on the old specification might no longer satisfy the set of design principles outlined above, or, worse, no longer exhibit a desired property. An example of the latter would be one which was intended to move each point in a list through the use of the **move** operation.

- Second, **bounded-point** could be added as a type which was neither an ancestor nor descendant of **point**, e.g., as a type whose parent was the root of the type hierarchy. The problem with this is that *no* existing procedures which contained invocations for which the assumed generic type is **point** would be applicable to instances of **bounded-point**. Since a number of these procedures might depend only on properties which were in fact shared by **point** and **bounded-point**, the result would be precisely the kind of redundant procedure definition which the use of generic invocation is intended to avoid.

- Third, we could invent an additional type, say **vanilla-point**, whose specification captured the commonalities between **point** and **bounded-point**. Then (a) **bounded-point** would be defined to have **vanilla-point** as a parent, (b) the definition of **point** would be changed to have **vanilla-point** as a parent. and (c) all procedures which depended only on the specifications associated with **vanilla-point** would be changed to accept objects whose types were descendants of **vanilla-point**. Again, changes to existing client programs are required.

The bottom line is that the advantages of using a specification-based programming methodology may well not be worth the overhead of specification and program modification. From a pragmatic viewpoint, allowing some procedures to be inapplicable to new types may be less onerous than making the modifications necessary to prevent it.

3.3 The realization of generic invocation

To describe how generic operation invocation is accomplished in our four languages, we will examine how to accomplish a generic invocation of the operation named by the expression *operation* on generic parameter exp_0 and additional parameters exp_1 ... exp_N. We will also examine how generic invocation can be carried out when the contents of a data structure are used to supply the parameters. The latter is useful in the design of procedures which construct invocations based on their input -- menu-driven user interfaces, for example.

3.3.1 Generic invocation for Zetalisp flavors

A number of different techniques are available for invoking generic operations on instances of Zetalisp flavors. The most intuitive of these is to invoke the **send** procedure designed for that purpose. The first parameter to **send** is used as the generic parameter of the desired generic invocation, the second is used as the operation name, and the remaining parameters (of which there can be any number) are the non-generic parameters. Thus, a general form for generic invocation is:

(send exp_0 *operation* exp_1 ... exp_N)

For example, the following will invoke the :move operation on a generic parameter denoted by p and non-generic parameter 1:[11]

```
(send p :move 1)
```

If the parameters are to be taken from a list, the **lexpr-send** procedure can be used. **Lexpr-send** takes a flavor instance, an operation name, and arbitrary number of parameters, of which the last must be a list. The members of this list are taken as the trailing parameters of the invocation. For example, the expressions:

```
(lexpr-send p :move (list 1))
(lexpr-send p :move 1 nil)
```

will also result in the generic invocation of :move on p and 1.

[11] A recent change to Zetalisp, motivated by compatibility with Common Lisp [Steele et al. 83], is that symbols whose names start with colons (i.e., symbols defined in the **keyword** package) evaluate to themselves. For example, the value of the variable **:move** is identical to the result of evaluating **':move**.

The technique of invoking generic operations of any type through a fixed set of procedures which take the generic parameter and operation name as distinguished parameters is a standard programming technique. However, it is not often embodied in programming languages. One exception is the **sfa-call** procedure used to invoke generic operations on Maclisp's "software file arrays" [Pitman 83].

A second way to invoke generic operations for instances of Zetalisp flavors is by invoking the generic parameter as a procedure with the generic operation as the first parameter. There are a number of Zetalisp primitives for procedure invocation; see [WeinrebMoon 81] for details. As one illustration, our example invocation of **move** could be rendered as:

```
      (funcall p :move 1)
```

Notice that the instance of the flavor is used as the "function" being invoked, *not* the name of the operation.

The technique of treating generic parameters as procedures has its roots in the "message-passing" paradigm described in section 3.1.4.

3.3.2 Generic invocation for Loops classes

Generic invocation in Loops is accomplished through the use of procedures which, like Zetalisp's **send** and **lexpr-send**, has generic invocation as their only function. The basic procedure for generic invocation is **Send!**, which takes an arbitrary number of parameters. **Send!** (alias **←!**) is analogous to Zetalisp's **send**; it uses the first parameter as the generic parameter, the second as the operation name, and the remaining parameters are used as the non-generic parameters. Thus, the canonical form for generic invocation is:

(Send! *exp*$_0$ *operation exp*$_1$ ... *exp*$_N$)

and our example **Move** invocation could be expressed as:

```
      (Send! p 'Move 1)
```

Send (alias **←**) is like **Send!**, has the same functionality as **Send** but does not evaluate its second parameter; e.g.,

```
      (Send p Move 1)
```

To allow the members of a list to be used as parameters of a generic invocation, **Send** can be invoked via the Interlisp **APPLY** procedure. **APPLY** takes two arguments: a procedure-denoting object (i.e., a list which satisfies the syntactic requirements for a procedure definition or a symbol whose "function definition cell" contains such a list), and a list. It invokes the specified procedure using the members of the list as actual parameters. For example, our example invocation could be rendered as:

```
    (APPLY 'Send (LIST p 'Move 1))
```

3.3.3 Generic invocation for Smalltalk classes

Generic invocation of Smalltalk methods is accomplished through either (a) the use of one of three syntactic forms which name operation name is syntactically apparent, or (b) the invocation of one of five primitive procedures, i.e., procedures which could not be defined in the language if not already provided. The latter must be used if the name of the operation or the parameter list must be computed.

3.3.3.1 Syntactic forms for generic invocation

If a syntactic form is to be used for generic invocation of a given operation, the choice of form depends on the name of the generic operation. The three possibilities are as follows.

1. The operation name is a simple identifier. If so, the method can take no non-generic parameters. The appropriate form for invoking such operations is:

   exp operation-id

   For example:

   ```
   p location
   ```

   invokes the **location** operation for generic parameter **p**. As in the **Send** form of Loops, *operation-id* is unevaluated.

2. The operation name is a sequence of identifiers followed by colons, i.e.:

   $id_1:id_2:\ldots id_N:$

   If so, the invocation will take as many non-generic parameters as there are identifier/colon pairs. To invoke the operation named $id_1:id_2:\ldots id_N:$ on generic parameter exp_0 and non-generic parameters exp_1 ... exp_N, the following form is used:

   exp_0 $keyword_1$ exp_1 ... $keyword_N$ exp_N

   Again, none of the $keyword_i$ are evaluated.

   For example,

   ```
   x computeArg1: a Arg2: b Arg3: c
   ```

is an invocation of the **computeArg1:arg2:arg3** operation on generic parameter **x** and non-generic parameters **a**, **b**, and **c**, and

```
      p move: 1
```

is an invocation of the **:move** operation on generic parameter **p** and non-generic parameter **1**.

There are three differences between this technique and the use of parameter-identifying "keywords" in invocation forms (e.g., as available in Mesa ([Mitchell et al. 79], chapter 5), Ada [Ichbiah, J. D., et al. 79], and, through the use of **&key** parameters, in Zetalisp). In the Smalltalk forms, the keywords are (a) required to be present, (b) required to be in a certain order, and (c) derived from the name of the operation.

3. The operation name is a member of a special syntactic class, referred to in Smalltalk as "binary selectors". This class consists of strings of one or two members of a non-alphanumeric subset of the Smalltalk character set. For example, the tokens **+**, **@**, and **~=** are all members of this class.

   Invocations of operations named by binary selectors must take exactly one non-generic parameter. To invoke such an operation on generic parameter exp_0 and non-generic parameter exp_1, the operation name is used as a binary infix operator:

   exp_0 *operation-id* exp_1

   For example,

   ```
         addend + augend
   ```

   is an invocation of the + operation with generic parameter **addend** and non-generic parameter **augend**.

   The point of introducing the binary-selector syntactic class is to allow the use of familiar operator names. For example, if the class of binary selectors were taken as ordinary identifiers, "+" would not be a valid *operation-id*; "+:" would have to be used instead. A more sophisticated approach would allow the set of identifiers that can be used as binary infix operators to be determined by program declaration. This technique has been used in EL/1 [Wegbreit 74] and the "Clisp" extension of Interlisp ([Teitelman 78], section 23).

The use of syntactic forms to indicate generic invocation and the absence of forms for non-generic invocation are two of Smalltalk's distinctive characteristics. The association of generic invocations with a language-defined subset of tokens has a long history -- Fortran arithmetic operators are an early example. Algol 68 [Branquart et al 71] and earlier and later "extensible" languages (e.g., [Standish 67, WellsCornwall 76]) allowed the methods invoked by these forms to be programmer-definable.

3.3.3.2 Primitive procedures for generic invocation

In order to have generic invocations with computed names and parameter lists, a family of **perform** operations is provided. Each such operation expects its first non-generic parameter to be a symbol. This symbol is used to identify the operation to be invoked.

Four of these variants, **perform:**, **perform:with:**, **perform:with:with:**, and **perform:with:with:with:** are suitable for invoking an operation which takes zero, one, two, or three non-generic parameters, respectively. For example, the following pairs of invocations are equivalent:[12]

-
  ```
  p perform: #location
  ```
  and
  ```
  p location
  ```

-
  ```
  p perform: #move: with: 1
  ```
  and
  ```
  p move: 1
  ```

-
  ```
  a perform: #+ with: b
  ```
  and
  ```
  a + b
  ```

-
  ```
  x perform: #computeArg1:arg2:arg3: with: a with: b with: c
  ```
  and
  ```
  x computeArg1: a arg2: b arg3: c
  ```

The reason why more than one **perform** operation is required is that each Smalltalk method takes a fixed number of parameters.

To invoke an operation with a computed parameter list (or to invoke an operation with a computed name but more than three non-generic parameters), the **perform:withArguments:** operation can be used. An invocation of **perform:withArguments:** takes two non-generic

[12] In Smalltalk, the form *#id* denotes the symbol whose name is *id*.

parameters: the name of the operation to be invoked and an array containing the non-generic parameters to be used for that invocation. For example, the effect of the four invocations described in the previous paragraph could be obtained by the following:

- If a1 is an empty array:
```
      p perform: #location withArguments: a1
```
- If a2 is an array containing one element, the value of 1:
```
      p perform: #move: withArguments: a2
```
- If a3 is an array containing one element, the value of b:
```
      a perform: #+ withArguments: a3
```
- If a4 is an array containing three elements, the values of a, b, and c:
```
      x perform: #computeArg1:arg2:arg3: withArguments: a4
```

3.3.4 Generic invocation for Star Mesa traits

In Star Mesa, the invocation of generic operation O is accomplished through the invocation of a procedure associated with O The first parameter of such an invocation is taken as the generic parameter of the invocation, and the subsequent parameters are taken as the non-generic parameters. Thus, the canonical form for generic invocation is:

operation-proc [exp_0, exp_1, ... exp_N]

where *operation-proc* is an expression whose value is a procedure which performs generic invocations of *operation*. An example of this form would be:
```
    Point.Move [p, 1]
```
where **Point.Move** is an expression whose value is a generic-invocation procedure.

In this approach, specifying a generic invocation of a given operation requires knowing which procedure implements a generic operation of a given name. In contrast to our other three languages, the names of the generic operations defined for a trait are neither syntactically apparent or available through examination of a data structure. Instead, they are determined by the first of many Star Mesa programming conventions we will encounter.

> **Star Mesa Convention 1:** Each trait definition designates a set of procedures as generic operation procedures.

To illustrate, consider the trait definitions of figures 2-20 through 2-27. The generic operation procedures are identified by comments contained in the text of each trait definition. Thus, the generic operation procedures of the four traits are as defined in figure 3-1.

Trait	Generic Operation Procedures
Point	Point.Location Point.Move Point.Display
BoundedPoint	BoundedPoint.Min BoundedPoint.Max BoundedPoint.Setmin BoundedPoint.Setmax
HistoryPoint	HistoryPoint.History
BHPoint	BHPoint.BoundsHistory

Figure 3-1: Generic operation procedures in the Star Mesa example

Having generic invocations which use computed parameter lists is not as straightforward as in our three other languages. Unlike our other three languages, Mesa provides no procedure analogous to `lexpr-send`, `apply`, or `perform:withArguments:` for invoking the procedure passed as its first parameter on the members of the data structure passed as the second parameter.

The problem is that each procedure-denoting expression must be typed, and procedure types must specify the types of the formal parameters and the returned values. For example, `PROC [INTEGER, REAL, STRING] RETURNS [REAL]` is the type of all procedures which take three parameters, an `INTEGER`, a `REAL`, and a `STRING`, and return a `REAL`. But no such type would be appropriate for a generalized **apply**, since it should be able to accept a procedure of any type. Thus, even though it would be possible to tell if any particular invocation of a generalized **apply** were type-correct, its parameters could not be declared in such a way for Mesa's type-checker to do so.

The principal reason why this is the case is that Mesa's type-checking algorithm could not guarantee the validity of such invocations. The procedure-typing approach of ML [Milner 78] offers an elegant solution to this problem by allowing the controlled use of variables in type expressions.

In order to have a type-checking apply, Mesa programmers must define a specialized version for each procedure type. This procedure will take a procedure parameter and a record parameter, and invoke the procedure parameter on the components of the record parameter. For example, the following defines a procedure which applies a procedure of the above type to the contents of a record containing an INTEGER, REAL, and STRING.

```
    T1: TYPE = RECORD [a: INTEGER, b: REAL, c: STRING];

    SpecializedApply:
      PROC [p: PROC [a: INTEGER, b: REAL, c: STRING] RETURNS [REAL],
                   args: T1]
                RETURNS [REAL] =
      BEGIN
        RETURN [p [args.a, args.b, args.c]]
      END;
```

T1 is necessary because textually identical RECORD expression denote different Mesa types.

This procedure could be used as follows:

```
    ArgsExample: T1 = [1, 1.0, "foo"];

    ProcExample: PROC [x: INTEGER, y: REAL, z: STRING] RETURNS [REAL] =
      BEGIN
        RETURN [y]
      END;

    SpecializedApply [ProcExample, ArgsExample];
```

The latter invocation would return 1.0.

Star Mesa's approach of associating operation names with procedures for generic invocation is well known. Other examples of its use include:

- the invocation of GENERIC procedures in PL/1
- the invocation of EL/1 procedures with GENERIC expressions as bodies [Wegbreit 74, Holloway et al. 74]
- the invocation of Ada procedures defined in the scope of a GENERIC program unit [Ichbiah, J. D., et al. 79]
- the invocation of operations in T [Rees et al. 84][13]

[13] In the T dialect of Lisp, operation objects rather then symbols are used to name abstract operations. These objects are data structures which contain a procedure which is to carry out the invocation no implementation of the operation is applicable to the generic parameter. In other words, "operations contain their own defaults."

3.3.5 Other forms for generic invocation

It is interesting to note that none of our four subject languages use the generic invocation form of the pioneering inheritance-based programming language, Simula 67 [Birtwhistle et al. 73]. In this approach, instances of types created with the `CLASS` constructor are conceptualized as records which have components that are with procedures which implement generic operations. The key idea is that these procedures are specialized for (i.e., closed over) the containing object, and take as arguments only the non-generic parameters of the invocation. This approach has been widely copied, e.g., to invoke the operations defined by `modules` of Mesa [LauerSatterthwaite 79] and Euclid [Lampson et al. 77].

Thus, the canonical generic invocation form is:

$exp_0.operation\text{-}id\ (exp_1,\ \ldots exp_N)$

Our point-moving example would be:

```
p.move (1)
```

The procedure `p.move` takes one parameter, the location, and will move the specific point `p` to that location.

A long-recognized disadvantage is the unnaturalness of the notation, especially for binary operations, e.g.,

```
x.plus [x]
```

As realized in the above languages, a more subtle problem is the difficulty of invocations with computed operation names. This is because *operation-id* must be an identifier and is not evaluated. A third problem can arise in defining methods for operations which take two or more instances of the same type; this will be discussed in chapter 6.

Note also that the Star Mesa approach to generic invocation is *not* the same as the syntactically similar forms used in Clu [Liskov et al. 79] and Russell [BoehmDemersDonahue 80]:

$id_1\$id_2\ (exp_1,\ exp_2,\ exp_N)$

The difference is that the latter results in the invocation of different procedures only if id_1 is a formal parameter of type **type**.[14] If *id*$_1$ is a type constant, the method used to carry out the invocation will always be the method associated with that type. For example, if Point were a Clu or Russell type, the invocation:

```
    Point$Move (p, 1)
```

could only invoke the **Move** method of type **Point**. But in Star Mesa, the same generic invocation would apply the **Point.Move** method of the type of p. By convention, this type could be Point or any descendant.

On the other hand, if *id*$_1$ is a type parameter, the procedure invoked will be a member of the operation set of the type denoted by *id*$_1$. Thus, according to the definition of generic invocation given at the start of the chapter, the Clu invocation expression given above does not qualify as a generic invocation expression.

This is not simply splitting hairs, because there are significant problems associated with this approach. The problem with the "operation set specified by type parameter" approach is that programs must be modified when a non-generic procedure is transformed into a generic one, e.g., when new types are added for which the operations of an existing type are meaningful. In Clu or Russell, procedures which took parameters of the existing type can be applied to instances of the new type only if they are modified to take additional **type** parameters. For example, the Clu procedure:

```
    f = proc (X: point)
        ...
        print (point$location (X))
        ...
        end
```

would have to be transformed into

[14]Clu procedures can have two kinds of formal parameters, one of which can be bound only to constant expressions or other parameters of the same kind. In order to obtain increased performance, type parameters in Clu are *required* to be of this kind. A principal difference between Clu and Russell is that the latter has only one category of parameter, and this category can be used to pass **type** objects.

```
f = proc [T: type] (X: T)
    where T has location: proctype (T) returns (real)
    ...
    print (T$location (X));
    ...
    end
```

Some of the unfortunate aspects of this technique are familiar: a considerable amount of work is required to add new types, and the work cannot be done by the designer of the type. Even worse, all *callers* of the old procedures must then be modified in order to pass the newly-introduced type parameters. In our example, each invocation of f:

```
f (exp)
```

would have to be transformed into

```
f [point] (exp)
```

Needless to say, this is highly undesirable.

We may note that the disadvantages of caller modification are not an inevitable consequence of strong type checking. For example, the version of Alphard described in [Wulf et al. 76] allowed formal parameter declarations which are satisfied by any actual parameter whose type has a specified set of operations with specified parameter and return types. Operation invocations involving such parameters are generic invocations, and such invocations can be statically type-checked.

3.4 Non-generic invocation of generic methods

It is often useful for generic methods for types to invoke generic methods of other types as subroutines. For example, in the scenario of chapter 2, **display** of **bounded-point** can call **display** of **point** to perform part of its work. Another use of this technique is to implement the technique of "delegation" (e.g., [HewittAttardiLieberman 79]), where a task assigned to one "actor" is to be carried out by another). We will refer to such invocations as "ancestor-generic" invocations.

The key benefit of ancestor-generic invocations is the avoidance of the redundant realization of an algorithm in programs which use that algorithm to carry out *part* of their tasks. Notice that this same form of redundancy avoidance would be useful even if generic invocation were not available *and* the operations of each class had to be enumerated.

Historically, the first use of this technique was realized in Simula-67 through the **qua** expression. To illustrate, suppose an identifier **opname** was defined in two classes **A** and **B**, with **B** as a subclass of **A**. If **b** was an instance of **B**, then **b.opname** would refer to the method for **opname** defined in class **B**, while **b qua A.opname** would refer to the method defined in **A**.

Non-generic invocation of generic methods is realized in our subject languages as follows.

3.4.1 Smalltalk-82

Smalltalk-82 provides three variants of non-generic invocation of generic methods. In order to understand these invocation forms, the concept of the "class" of a method must be introduced. Recall that methods are instances of class **CompiledMethod**, and are produced by the Smalltalk compiler from strings containing method definitions. In fact, the compiler takes an additional parameter, a class. The class used by the compiler in order to produce a **CompiledMethod** will be the class of that **CompiledMethod**.

In each of the examples of chapter 2, the generic parameter of the **compileAndStore:** invocation is used as the class in which the associated method string is compiled. For example, in the expression:

```
    HistoryPoint compileAndStore:
      'display: stream
        self super.display: stream.
        self partialDisplay: stream.'
```

the class of the **CompiledMethod** would be **HistoryPoint**.

3.4.1.1 Type-specific invocation

Just as in Simula-67, it is possible to invoke the generic method for a specific operation of a specific type. This will occur if the operation name of the invocation has a prefix of the form

```
    id.
```

where *id* is the name of an ancestor of the class of the method. For example, in the method produced by:

```
    BHPoint compileAndStore:
      'exampleOp: x
        self HistoryPoint.op2.
        self BoundedPoint.op2'
```

the two invocations will be of the methods for op2 of HistoryPoint and BoundedPoint, not BHPoint. If the specified operation name is not in the generic operation set of the specified type, an runtime error is signalled.

3.4.1.2 Type-relative invocation

In Smalltalk-76 [Ingalls 78] and unextended Smalltalk-80, classes have a unique parent. It is possible to cause the generic method of the parent of the class of the method to be invoked. This is accomplished through the replacement of self with super in any of the invocation forms. For example, if Point was the single superclass of BoundedPoint in a Smalltalk-80 program, the invocation:

```
    super move: newloc
```

in a local method of BoundedPoint would invoke the move: operation of Point.

The Smalltalk-82 analog to this technique is applicable if *all* parents of a class which have *some* method for an operation have the *same* method for that operation. The invocation of this unique method is accomplished by adding a prefix of super. to the operation name of an invocation form. If different parents have different methods, or if no parent has a method, a runtime error is signalled.

For example, if HistoryPoint had an method for op2 but not op3: and BoundedPoint had one for op3: but not op2, then

```
BHPoint compileAndStore:
  'exampleOp: x
    self super.op2.
    self super.op3: x.
```

would result in an invocation of the op2 of HistoryPoint followed by the op3: of BoundedPoint.

Due to the fact that the Smalltalk-82 implementation did not modify the Smalltalk interpreter and associated data structures (i.e., the "virtual machine" of part four of [GoldbergRobson 83]), the use of Smalltalk-80 super invocation will not result in the appropriate behavior. The latter will only examine a single parent for each class. Thus, the unintentional use of Smalltalk-80 super invocation in a Smalltalk-82 program can lead to undetected errors.

3.4.1.3 Sequential parent invocation

The invocation of the methods for an operation defined for *all* parents can be accomplished by adding a prefix of all. to the operation name of an invocation form. For example,

```
BHPoint compileAndStore:
  'display: plist
     self Point.display: plist.
     self all.partialDisplay: plist.
     self partialDisplay: plist.'
```

defines a method which will invoke the display: method of Point, then the partialDisplay: of HistoryPoint, BoundedPoint, and BHPoint.

3.4.1.4 Distinguished method invocation

A final means for non-generic invocation of generic methods allows the generic methods for specific operations of specific classes to be invoked. In particular:

- An invocation of the basicNew and basicNew: operations on an instance of any class is carried out by the new and new: methods of the class Behavior.

- An invocation of the basicAt:, basicAt:put:, and basicSize operations on an instance of any class is carried out by the at:, at:put:, and size methods of the class Object.

The reason why the operation names at:, at:put:, and size of Object were singled out for special treatment is that they were given different semantics for the built-in Smalltalk classes which represent collections of objects ([GoldbergRobson 83], chapters 9 and 10). The names `new` and `new:` are given such treatment because they are commonly reimplemented by class definers; see chapter 5 for a discussion. In both cases, the overridden methods from `Behavior` and `Object` embody system facilities which should be invocable on instances of any class.

These operations are defined in Smalltalk-80 and usable in Smalltalk-82. However, the availability of type-specific invocation means that they are semantically unnecessary. For example, the Smalltalk-80 invocation:

```
    X basicSize
```

could be realized in Smalltalk-82 as:

```
    X Object.size
```

This is especially advantageous because the capability described in this section depends on adherence to a programming convention of not defining methods for the `basic...` operations in any class.

3.4.2 Loops

Type-specific invocation, type-relative invocation, and sequential-parent invocation are all available in Loops.

- Type-specific invocation is implemented by the `DoMethod` and `ApplyMethod` procedures. For example,

  ```
      (DoMethod bhp 'Op3 $HistoryPoint x)
  ```

  or

  ```
      (ApplyMethod bhp 'Op3 (LIST x))
  ```

  would invoke the method for `Op2` in the generic operation set of the class named `HistoryPoint` on the value of x.[15] Type-relative invocation is implemented by the `SendSuper` (alias `←Super`) syntactic form. For example,

  ```
      (SendSuper bhp Op3 x)
  ```

  invokes the unique method for `ExampleOp` defined by all parents of the class of the method in which it is found. An error is signalled if a no method or more than one method is defined by the parents.

- Sequential parent invocation is implemented by the `SendSuperFringe` (alias `←SuperFringe`) syntactic form. For example:

  ```
      (SendSuperFringe bhp Op3 x)
  ```

[15] Chapter 4 describes the $ construct.

In addition, the DoFringeMethods procedure described in [BobrowStefik 83] sometimes results in non-generic invocation of generic methods. Its description is deferred until chapter 7, as it involves concepts defined there.

Finally, we note that the name of the Interlisp function which implements a given operation of a given type is computable by an invocation of the List operation of class Class with the symbol Method as a parameter. For example,

```
    (Send $BHPoint List 'Method 'Op3)
```

computes the name of the Interlisp function which is the method for Op3 in the generic operation set of BHPoint. This function can be invoked using the Interlisp APPLY primitive described above.

3.4.3 Zetalisp

There are no facilities in Zetalisp for type-relative or sequential-parent invocation. In order to achieve the same functionality, Zetalisp programmers must choose a algorithm for generic method computation (via a "method combination type" and "method combination order") which will produce a method with the desired functionality. The facilities for doing so are described in detail in chapter 6.

Type-specific invocation is possible, albeit not through the procedures documented in [Symbolics 84] or [MoonStallmanWeinreb 84]. The method for operation O in the generic operation set of flavor F can be computed by invoking the fdefinition function with a list of the form:

    (:handler *flavor-name operation-name*)

For example, if the variable x is bound to the symbol :display, the expression:

```
    (fdefinition (list :handler 'bh-point x))
```
[16]

will return the function which is the method for display in the generic operation set of the flavor named bh-point. If all elements of the list are constant, the above can be simplified to:

```
    #'(:handler bh-point :display)
```

[16] Knowledgeable Lisp programmers will see that this cries out for a backquote.

However, once the method is obtained, an appropriate mapping table (recall the discussion in the previous section) must be computed and passed as the second parameter of the invocation (in the [Symbolics 84] system) or bound to the variable `self-mapping-table` (in the system described in [MoonStallmanWeinreb 84]). The failure to use an appropriate mapping table can cause the syntactic forms for instance variable manipulation described in section 5.3 to operate on the wrong instance variables with no indication of error.

How can the appropriate mapping table can be computed? Although there are no documented procedures in [Symbolics 84] or [MoonStallmanWeinreb 84] for doing so, it is sufficient to know (a) the flavor of which the generic parameter is an instance and (b) the flavor named in the `defmethod` which created the procedure.[17] Notice that flavor (b) need not be the one specified in the `:handler` list, since the method could have been inherited from some ancestor.

How can flavor (b) be determined? If the flavor name F and operation name O of the `:handler` list are both constants, knowing the method definitions of O for F and its ancestors and the method construction algorithm used (see chapter 6 for a discussion of these concepts) allows flavor (b) to be determined by inspection. Thus, it can be used as a constant in the program. The problem with this approach is that if this set of method definitions changes (i.e., by adding or removing `defmethods` for F and its ancestors), this constant will become incorrect, with no indication of error.

The only general technique for type-specific invocation of a generic method is to extract flavor (b) from the method itself, through the use of undocumented properties of the implementation. For example, if the method is a `lambda` or `named-lambda` its body can be examined for an appropriate declaration. If the method is a compiled function, its "debugging info"[18] can be examined for a specified property. The main problem here is that the data structure which

[17]As we will see in chapter 6, some generic methods are constructed by the type system rather than being specified in any `defmethod`. For such methods, flavor (b) should be the flavor for which the method was initially constructed.

[18]In Zetalisp, each compiled procedure is associated with a property list referred to as its "debugging-info."

represent the procedure to be invoked must be searched on every invocation. A reasonable solution requires some small amount of creativity in data structure and interpreter design.

How could could such a fundamental capability have been omitted? The availability of an extensive repertoire of method-definition algorithms often obviates the need for users to define procedures which contain ancestor-generic invocations. The lack of the ability to perform ancestor-generic invocations has an impact only when none of the system-defined algorithms is appropriate. When this occurs, the usual response of Zetalisp programmers is to redesign their programs so that one of the predefined algorithms *is* appropriate. This can be considerably frustrating, especially for naive users.

Since this information is already computed by the method combination algorithms, it would be straightforward to provide an interface by which it would be accessible to users. Even better, a procedure for invoking an arbitrary generic method of an arbitrary flavor on a generic parameter of an arbitrary type should be provided. The advantage of the latter is that it eliminates the possibility of using an inappropriate mapping table for a given invocation.

3.4.4 Star Mesa

Type-specific invocation is the only form of non-generic invocation available in Star Mesa. Such invocations require knowledge of the format of the "trait component storage" of the trait whose generic method is to be invoked; see chapter 7 for details. What is important here is that the format of this storage is an aspect of the implementation of the trait which is subject to change. Thus, type-specific invocations in Star Mesa require undesirable dependence on the implementation of ancestor types..

3.5 Invocation errors

There are two possible ways in which the generic invocation methods described above can fail. The first kind of failure, which is possible in any of our four subject languages, occurs when the generic operation set of the specified type contains no method for the specified operation. The second, which is possible in Zetalisp, Loops, and Star Mesa, occurs when the type of the generic parameter is not one of those for which the generic invocation mechanism is defined. We consider each in turn.

3.5.1 "Nonexistent operation" errors

What happens when the operation set of the type of the generic parameter of a generic invocation does not contain an association for the specified operation name? We will refer to this situation as a **nonexistent operation** error. Here is the technique used in our four languages in response to a generic invocation where the generic parameter X has type T, the operation name is O, and the generic operation set for T contains no method for O.

- **Star Mesa.** In Star Mesa, operation names are expressions which have procedure values. Thus, the result of a nonexistent-operation error depends on the value of the expression used as the operation name.

    - Undefined: the Mesa compiler or binder will signal an error.

    - Evaluates to a non-type-compatible procedure: the Mesa compiler will signal an error.

    - Evaluates to a type-compatible procedure: the procedure will be applied to the parameters with no indication of error.

  The latter, of course, is highly undesirable.

- **Smalltalk.** A second generic invocation occurs, where the generic parameter is X, the operation name is `doesNotUnderstand:` and the non-generic parameter is an instance of class `Message` from which O and an array containing the original parameters can be extracted. Since such a method is defined for the Smalltalk class named `Object`, and `Object` is an ancestor of all other classes, the algorithm for method definition (the subject of chapter 5) guarantees that the generic operation set of every class contains a `doesNotUnderstand:` method. The method defined for `Object` halts program execution.

  In this approach, different procedures for handling nonexistent operation errors can be invoked when the generic parameter is of different types. This contrasts with Star

Mesa, in which the same procedure is invoked in response to all nonexistent operation errors.

- **Loops.** A second generic invocation occurs, where the generic parameter is X, the operation name is `MethodNotUnderstood`, and O is the single non-generic parameter, i.e.,

  ```
      (Send x MethodNotUnderstood O)
  ```

  If this invocation returns a non-nil symbol S, the original invocation will be retried using S as the operation name rather than O. If not, execution is halted. The `MethodNotUnderstood` method for `Object` uses the spelling correction algorithm of Interlisp's "DWIM" facility ([Teitelman 78], section 17) to try to find an operation name of T which is lexicographically similar to O.

  The difference between this approach and Smalltalk's is that the latter makes it easier for the error handler to (a) examine the non-generic parameters of the original invocation and (b) continue the computation with an arbitrary invocation, rather then an invocation of a different operation on the same parameters. Although the stack-manipulation facilities of Interlisp makes either of these possible, the Smalltalk design makes doing so simpler and less prone to programming errors.

- **Zetalisp.** There are three possible responses to an erroneous-operation error in Zetalisp. The response taken depends on the type T of the generic parameter X.

  1. If the generic operation set of T contains a method for the `:unclaimed-message` operation, then as in Smalltalk and Loops, another generic invocation occurs. The generic parameter is X, the operation is `:unclaimed-message`, and the parameters are O and the parameters of the original invocation. The result is returned as the result of the original invocation.

  2. A "default operation handler" procedure can be defined for a flavor. If such a procedure is defined for T, it is applied to X, O, and the arguments to the erroneous generic invocation. The result is returned as the result of the original invocation.

     A default operation handler can be associated with a given flavor via the `:default-operation-handler` clause in a `defflavor` form for that flavor. For example, given the definition:

     ```
         (defflavor example () ()
           (:default-operation-handler some-proc))
     ```

     the global definition of the symbol `some-proc` will be used as the default operation handler for `example`.

  3. Otherwise, a `sys:unclaimed-message` condition will be signalled. Condition signalling in Zetalisp results in a search of a list of "condition-handling" procedures for a specific procedure declared to be a handler for the type of condition being signalled. A procedure invocation can add new handlers to the front of this list; when the invocation is completed, the associated handlers are removed.

The first such handler found is invoked with the condition object as one parameter; X, 0, and a list containing the original parameters can be extracted from the condition. The handler can either (a) cause the original invocation to be retried using a different operation name, (b) perform a non-local return from some frame on the runtime invocation stack, or (c) inform the condition-handling mechanism that another condition handler should be found.

Thus, the response to a nonexistent operation error for a given type can be made to depend *either* on a fixed procedure associated with the type or on the state of the invocation stack. If an `sys:unclaimed-message` operation is provided for the type or any ancestor or a default operation handler is provided for the type, fixed-procedure behavior will occur; if not, condition-handling behavior will be dynamically determined. The availability of both kinds of error handling is the principal difference between the Zetalisp and Smalltalk approaches.

As an aside, it is interesting to note that the design of Zetalisp's condition-handling system itself embodies a a novel use of the ancestor/descendant relation between types. Conditions are represented as typed objects, condition handlers are associated with sets of types, and a given handler is defined to be applicable to a given condition iff the type of the condition is a descendant of one or more of the types associated with the handler.

For example, if some condition is an instance of the (hypothetical) type `unexpected-closing-of-file-server-connection`, and this type is a descendant of the types `file-system-error` and `network-error`, handlers associated with the latter two types will be applicable to the condition. A similar approach is used for objects returned by condition handlers which indicate how the invocation which signalled the condition should proceed. The details of the design of the condition-handling mechanism is beyond the scope of this report; [Symbolics 84] and [MoonStallmanWeinreb 84] contain a complete description.

The advantage of this approach is quite similar to the advantage of definition by inheritance: less work is required to define new condition types for which the handlers associated with existing types are applicable. For example, if the condition type

unexpected-closing-of-archive-file-server-connection is declared to be a child of **unexpected-closing-of-file-server-connection**, all the procedures defined as handlers for the latter become applicable to the former. In the alternative approach, the applicability of each condition of the latter must be individually declared.

3.5.2 "Inappropriate generic type" errors

One of the distinctive features of the Smalltalk family of languages is that generic invocation is applicable to all Smalltalk objects. But in Zetalisp, Star Mesa, and Loops, the generic invocation techniques described above apply only to instances of **flavor** types, **trait** types, and **class** types, respectively. We will refer to the situation where the type of the generic parameter is not appropriate for the invocation mechanism as an **inappropriate generic type** error.

The following is the result of an inappropriate generic type error where the type of the generic parameter **X** is inappropriate.

- In Zetalisp, a **sys:invalid-function** condition is signalled. **X** can be extracted from the condition object. The default response is to signal a condition from which the result of the failing invocation can be returned.

- In Star Mesa, "inappropriate generic type" errors correspond to invocations of generic operation procedures where the type of the first parameter is not **TM.Object**. Such an invocation would be flagged by the Mesa compiler as a type error.

- In Loops, the method for **NoObjectForMsg** of the class which is the value of the variable **DefaultObject** is invoked with parameters **X** and **O**. If a value is returned, it is used as the result of failing invocation. Note that, in a departure from programming convention, the **NoObjectForMsg** operation of the class bound to **DefaultObject** will never take an instance of that class as a parameter.

4. Basic operations on types

As a first step in describing inheritance-based type definition in our four subject languages, we examine those aspects of types relevant to the parent/child relation involved in inheritance.

The organization of the description is as follows. First, we present technique used to describe the common characteristics of our four languages. Next, the properties which our four languages have in common are described. Following that, the way in which the abstract types and procedures of the specification are realized is described for each language separately. For those readers whose initial goal is to obtain perspective rather than explore details, these latter sections may be skimmed or skipped.

4.1 The methodology for description

In many programming languages, types are meta-linguistic objects associated with syntax and semantics different from that which applies to "ordinary" data structures. In each of our four languages, types defined by inheritance are not treated in this way. Type "declarations" are simply procedure invocations which happen to create objects which can serve as types. Wegbreit's work on the EL/1 language [Wegbreit 70, Wegbreit 74] was an early and influential embodiment of this approach.

Since types are truly data structures in our four languages, an informal variant of an "abstract type specification" (e.g., [Guttag 80]) will be used in the description. The specification will define some properties of the abstract types **type** and **object** in terms of abstract procedures such as **type-of** and **ancestors**. These specifications will hold for our subject types and their instances in each of our four languages.

Types in our subject languages and instances of those types will be characterized as being modelled by one or more named "attributes"-- i.e., as a tuple with named components. Each such component will be described in terms of a mathematical concept, such as a set or an association between two objects. The behavior of the abstract procedures will be described in terms of the set of components which model types and instances, as well as in terms of invocations of other abstract procedures.

In order to facilitate the reader's comprehension of a considerable amount of detail, the set of components used to model types will not be presented all at once. The components which are relevant to a particular aspect of types will be presented at the time that aspect is discussed.

It is important to note that not programs can be written in each of our four languages in which the descriptions given below are not accurate. Such programs can be characterized as using one of two kinds of capabilities:

- "bit-level" operations on data: e.g., Zetalisp "subprimitives" (chapter 14, [WeinrebMoon 81])
- "representation-level" operations on types themselves: e.g., directly modifying the instance variables of the data structures which represent Smalltalk classes.

These descriptions will assume "identity" as a primitive modelling concept, and **identical** as a two-place predicate which succeeds iff the two parameters are "the same object". Although the formalization of this intuitively simple idea is not a simple task, (see, e.g., [GuttagHorning 80], [Rich 81], [Smith 83]) the fact that it is realized as the `EQ` of Interlisp and Zetalisp, the `=` procedure in Mesa, and the == operation in Smalltalk should make the meaning clear.

The descriptions will also use the abstract type **symbol**. This type is realized by instances of the `Symbol` class in Smalltalk (e.g., `#ASymbol`), the `ATOM` type of Interlisp (`'ASymbol`), the `:symbol` type in Zetalisp (`'a-symbol`), and the `STRING` type of Mesa (`"ASymbol"`). When symbols are referred to by name, the quoting characters will be omitted.

Finally, it will be clear that the following "specifications" are by no means a precise characterization of behavior. The purpose of introducing such quasi-formalism is purely pedagogical. It does not seem unreasonable that the direction suggested in the following could be used as the basis of a satisfactory formalization, but it is beyond the scope of the present work to do so.

Given the above, we now begin the actual description.

4.2 The common abstraction

4.2.1 Type introduction

Since our subject types are full-fledged computational objects, we begin the description with how such objects are created. In order to do so, we introduce the abstract type **type** and the three relevant abstract procedures for type introduction are **new-type**, **type-name**, and **type-named**:

- **new-type: {<internal-name: symbol, ...>} => type**
- **type-name: {type} => symbol**
- **type-named: {symbol} => type**

The above notation describes the "signature" (e.g., [Demers et al. 78]) of the procedures; for example, **type-named** is a procedure which takes a type as a parameter and returns a symbol. A parameter list of the form:

<id: *id*, . . .>

uses a "keyword" style to provide a partial specification of its parameters. For example, the use of:

<internal-name: symbol, ...>

in the signature of **new-type** indicates that *one* of its parameters is **internal-name**, and that the value of this parameter must be a **symbol**.

The relevant properties of **new-type**, **type-name**, and **type-name** are as follows.

- Each invocation of **new-type** creates a type with a new identity:
  not (identical (new-type (<...>), new-type (<...>)))

- The name of a newly created type is specified as a parameter to **new-type**:
  identical (type-name (new-type (<internal-name: N, ...>)), N)

- The **type-named** procedure can be used to refer to a newly-created type given its internal name:
  type-named (new-type (<internal-name: N, ...>)) = N

We can summarize the above by saying that types have identity and are associated with symbols which can be used to name them. There are two reasons why types are treated as objects with identity. One will be described here; the other in the following section.

A principal reason why types have identity is that aspects of our subject types can be modified during program execution. The fact that this is possible may or may not be relevant to program semantics,[19] but it is an essential aspect of a usable programming environment. If types were purely "applicative" data structures, common tasks in program development would be much more painful. For example, it would be impossible to replace a generic method of an existing type; a new type would have to be created instead.[20]

The purpose of the name-related procedures is slightly more subtle. The key benefit of this approach is as follows. If the printed representation of a type includes its name, then a reference to the type can be obtained from its printed representation. For example, suppose that the result of printing a type was the string "`Type: `" followed by the printed representation of the name of the type. Then if we knew that the printed representation of some type T was:

```
Type: Point
```

we could obtain a reference to T by the expression:
type-named ('Point)

Without the existence of the **type-named** function, there would be no way to compute T from the above.

This technique is commonly used when data structures are used to model real-world objects

[19]As we will see in chapter 6, it is critical in describing generic operation set computation in Star Mesa.

[20]To be sure, it is possible to design a system where all type-modifying operations are invocable only through a "user interface" and not by programs in the language themselves. However, this technique guarantees that the programs which implement the "interface" cannot themselves must be written in a separate language.

with identity, e.g., as in databases or in knowledge representation systems. In particular, Loops allows such names to be associated with any instance of any class. Thus, **type-named** is actually a special case of a more general **object-named** procedure.

The soundness of this technique depends on the assumption that no two types have the same name. However, that is not the case in any of our four languages. New types can be introduced with the same name as existing ones, or, in Smalltalk and Loops, the name of one type can be changed to be identical to that of another. If such activities are performed, then the naming operations must be used with caution.

For completeness, we will introduce the abstract **set-name** procedure to denote the operation of changing the name of a type.

- **set-name: {type, symbol}**

The abstract representation of a type must then be modified to include a **symbol** component. This component will be initially set by **new-type**, returned by **type-name**, and modified by **set-name**. As alluded to above, a **set-name** procedure is available in Loops and Smalltalk, but not in Zetalisp or Star Mesa.

4.2.2 Types as value-space partitions

In each of our languages, there is a distinguished procedure which maps objects to types. The key aspect of this function is that the mapping is functional, i.e., the same type will always be associated with the same object. Thus, the collection of types can be viewed as partitioning a space of objects. Furthermore, each of our languages includes a primitive operation which can create a new object of a given type, distinguishable from all existing ones.

In order to describe this behavior, we will introduce the abstract type **object** and the abstract procedures **type-of** and **instantiate**:

- **type-of (object) => type**
- **instantiate (type) => object**

These procedures are related by the following properties:

- Each invocation of **instantiate** creates an object distinct from all other objects:
  not (identical (instantiate (T1), instantiate (T2)))

- The type of a newly-created object is a parameter of the instantiation procedure:
  identical (type-of (instantiate (T)), T)

We will henceforth say that an object O is an "instance" of a type T iff
identical (type-of (O), T)

Notice the contrast between this approach and that used in the mainstream of work in programming language semantics. In describing type-related behavior in our four languages, we are associating types with computational objects, i.e., the data structures which result from evaluating expressions. But in the standard approach to semantics, types are ascribed to the expressions of the language, not to their referents. These two uses of the concept of "type" can be a point of considerable confusion.

Given the above, we can describe a second reason why it is useful for different invocations of **new-type** to produce non-identical types. The distinguishability of types allows the type of an object to be used to represent information about the object not recoverable through its operation set.

As a simple example, consider the types **faculty** and **student** which both have the same two abstract operations, **name** and **phone**. Determining whether an object represented a faculty member or a student would be impossible though invocations of the **name** and **phone** operations. But if the **faculty** and **student** types were distinguishable, then the faculty/student information could be recovered by examining its type. This information could either be used to direct further computation (as in common in "databases" which contain typed objects [SmithSmith 77, McLeod 79, BrodieZilles 81]) or for type checking (e.g., [GeschkeMorrisSatterthwaite 77, Demers et al. 78]).

Finally, we note that a significant difference among our four languages is whether instances of the abstract type **type** are also instances of the abstract type **object**. This is the case in Smalltalk and Loops; each class is itself an instance of a class. But Zetalisp flavors are not instances of flavors, and Star Mesa traits are not instances of traits.

4.2.3 The parent/child relation

The final basic aspect of types involves the parent/child relation described in chapter 1. The use of this relation to compute other properties of types is the distinguishing characteristic of type construction algorithms which embody inheritance.

In order to describe this aspect of types, an abstract **set-of** type constructor is used. Completely in line with intuition, instances of the abstract type:

set-of (T)

are sets which contain only instances of abstract type T. These are realized as arrays in Smalltalk and Star Mesa, and lists in Zetalisp and Loops. The basis of the extension is the introduction of a **parents** procedure. and the addition of a **parents** parameter to **new-type**.

- **parents: {type} => set-of (type)**
- **new-type: {<parents: set-of (type), ...>} => type**

We will henceforth say that "A is a parent of B" iff

A ε parents(B)

Given the existence of the **parents** function on types, the following procedures are relevant:

- **ancestors: {type} => set-of (type)**
- **components: {type} => set-of (type)**
- **is-a: {object, type} => boolean**

The semantics of these procedures can be defined in via **parents** and **type-of**.

- Let Parent (T1, T2) stand for the relation

  T1 ε parents (T2)

  and let Ancestor be the transitive closure of Parent. Then the ancestors of a class T2 are those objects T1 which satisfy

  Ancestor(T1, T2)

  Informally, the ancestors of a type are its parents, its parents' parents, etc.

- The components of a type is the union of the type and its ancestors:
  components (T) = ancestors (T) U {T}

- The predicate **is-a(O, T)** is satisfied if the type of O is a component of T:
  is-a (O, T) = type-of (O) ε components (T)

There are two relevant aspects on which our subject types differ.

- In Smalltalk, Zetalisp, and Loops, there is a distinguished type which is an ancestor of all other types. In Star Mesa, no such type exists. In the languages where such a type exists, we will use **root-type** to refer to it. From chapter 6, we will see that the operations associated with this type will be applicable to all instances of each of our subject types, although the methods used can be different. Examples of such operations include those for printing, instance variable manipulation (see chapter 5), and type testing.

- In each of our four languages, the parents of a newly created type are specified as a parameter to **new-type**; i.e.,
  parents (new-type (<parents: P, ...>)) = P
  In Smalltalk, Zetalisp, and Star Mesa, the parents of a type are fixed at the time the type is created. But in Loops, the parents can be modified at any time. The introduction of such a procedure requires a similar change to the abstract representation as that described for **set-name** in section 4.2.1.

4.3 The Smalltalk realization

Figures 4-1 through 4-3 identify the types and procedures of Smalltalk-80 and Smalltalk-82 which realize the above-described abstraction. Specific examples are included as well. For example, the entry for **ancestors** in figure 4-1 states that it is realized by the `allSuperclasses` operation of class `Behavior`, and that

```
    BHPoint allSuperclasses
```

is an example of its use. Starred procedures, such as the `classNamed` operation of `Class` defined in figure 4-1, are not predefined in the language; their definitions are presented as well.

4.3.1 Type introduction operations

Abstract op	*Realized as*	*Example*
type	*Any instance of an instance of* Metaclass	
new-type	*The* ...subclass:... *operations of* Class	
	In Smalltalk-82: The ...subclass:otherSubclasses:... *operations of* Class	
type-named	*(*) The* className *operation of* Symbol	#BHPoint classNamed
type-name	*The* name *operation of* ClassDescription	BHPoint name
set-name	*The* rename: *operation of* ClassDescription	BHPoint rename: #NewName

In the operation set of Symbol:
```
classNamed
  ^ Smalltalk at: self
```

Figure 4-1: Type introduction in Smalltalk-80 and Smalltalk-82

object	*Any Smalltalk object*	
type-of	*The* class *operation of* Object	bhp class
instantiate	*The* new *operation of* Behavior	BHPoint new
	The new: *operation of* Behavior	SomeOtherClass new: 3

Figure 4-2: Partitioning operations in Smalltalk-80 and Smalltalk-82

parent	*In Smalltalk-80: the* superclass *operation of* Behavior	BHPoint superclass
parents	*In Smalltalk-82: the* immediateSuperclasses *operation of* Behavior	BHPoint immediateSuperclasses
ancestors	*The* allSuperclasses *operation of* Behavior	BHPoint allSuperclasses
is-a	*The* isKindOf: *operation of* Object	p isKindOf: Point
root-type	Object	

Figure 4-3: Parent/child operations in Smalltalk-80 and Smalltalk-82

4.3.1.1 The realization of *type*

EL/1 was the one of the first languages in which types were themselves instances of types. In the approach taken in EL/1, the abstract type **type** is realized as the collection of all objects for which the realization of **type-of** returns a distinguished type: **TYPE**. In other words, an object O is a type iff:

type-of (O) = TYPE

The abstract **type** and **type-of** are realized in EL/1 as the concrete MODE and MD [Holloway et al. 74].

In Smalltalk-80 and Smalltalk-82, the concept of "class" is used to refer to types. As in EL/1, the realization of the abstract type **type** is also defined in terms of the object computed by the realization of **type-of**. The realization of **type-of** is the method for **class** in the generic operation set of the class **Object**.

In contrast to the EL/1 model, there is no single Smalltalk class of which all classes are instances. Instead, there is a distinguished class **Metaclass** of which the class of all classes are instances. Thus, O is a Smalltalk type iff:

    type-of (type-of (O)) = Metaclass

In English, an object is a class just in case the type of its type is the distinguished type **Metaclass**.

This definition of **type** in Smalltalk is a consequence of the following properties of classes described in [GoldbergRobson 83, chapter 16, p. 269]:

- The class of a class is called its metaclass.
- Every class is an instance of a metaclass.
- Every metaclass is an instance of **Metaclass**.

The illustrations accompanying the text make it clear that "instance of" is meant to refer to **type-of**, not **is-a**. As we will see, the realization of **type-of** will be the method for the operation **class** of the distinguished type **Object**.

Why aren't all Smalltalk classes instances of the same class? The extra complexity is due to the desire:

1. to have the primitive creation operation be the **new** of **Class**. This should create an uninitialized instance of the generic parameter.
2. to have the **new** operation of the class of each class create an instance of its generic parameter (i.e., of the class), then perform initialization specific for that class. E.g., the **new** of (Point class) should create an uninitialized instance of **Point**, then initialize it. And **(Class class) new** should create a new instance of **Class** and initialize it. Chapter 5 describes why this is desirable.

If this is the case, then **new** of the class of **Class** should perform initialization specific for **Class**. If the class of **Class** is **Class**, then the **new** of **Class** should perform initialization specific to **Class**. But this contradicts the principle that **new** of **Class** creates an uninitialized instance.

4.3.1.2 The classes relevant to *type*

In Smalltalk, the abstract procedures of section 4.2 are realized as methods for generic operations. This is possible because each object which represents an instance of the abstract type **type** also represents an instance of the abstract type **object**. Thus, each such object can itself be used as the generic parameter of generic invocations.

Since the abstract procedures are to be realized as methods for generic operations, each must be defined in the operation set of some class. The following is a summary of the six classes in whose operation sets the realizations of these abstract procedures can be found.

- `Object` contains definitions for the operations which all Smalltalk objects are expected to implement.

- `Behavior`, a child of `Object`, contains definitions for operations assumed by the Smalltalk interpreter to be implemented for any object used as a type. Examples of these operations include those which create an instance of the class and modify the set of generic methods of the type.

- `ClassDescription`, a child of `Behavior`, contains more definitions for operations assumed to be implemented for any object used as a class. These operations are depended upon by Smalltalk's "system methods," (e.g., the top-level user interface) rather than the interpreter per se. Examples of these operations are those which categorize classes and generic operations of classes, and those which take a class and create a sequence of Smalltalk expressions whose evaluation will produce a copy of the class. The latter is used to "store" class definitions on I/O devices.

- `Class`, a child of `ClassDescription`, contains definitions of operations assumed to be implemented for all classes whose instances *are not* themselves classes. It defines operations to create instances of these classes and to manipulate the collection of "class variables" and "pool variables" of a class (see section 5.6.1).

- `Metaclass`, another child of `ClassDescription`, contains the definition of the single operation assumed to be implemented for all classes whose instances *are* classes. This is the operation which creates new classes. All classes whose instances are classes are themselves instances of `Metaclass`.

- `Metaclass class`, [that is, the type of the object `Metaclass`], also contains the definition of exactly one operation: the procedure which creates new classes whose instances are classes. In other words, this is the procedure which creates instances of `Metaclass`.

4.3.1.3 The realization of *new-type*

Given the above, we can turn to the question of the realization of the abstract procedures. The first of these, **new-type**, has different realizations in Smalltalk-80 and Smalltalk-82.

In Smalltalk-80, the **new-type** procedure is realized as four generic operations of the system-defined class `Class`. The most basic of these has the unwieldy name of:

```
subclass:
  instanceVariableNames:
  classVariableNames:
  poolDictionaries:
  category:
```

For the purposes of the abstraction of section 4.2.1, the only relevant parameter is `subclass:`. This parameter is used as the name of the newly created class. As an example, the invocation:

```
Object subclass: #Point
       instanceVariableNames: 'loc'
       classVariableNames: ''
       poolDictionaries: ''
       category: #CanonicalExample
```

creates a new class whose name is the symbol `Point`.

Smalltalk-80 also defines three variants of the `subclass:` operation, `variableSubclass:...`, `variableWordSubclass:...:`, and `variableByteSubclass:...`. These all take the same parameters as the `subclass:...` operation, and all create new classes whose name is the first non-generic parameter. Looking ahead a bit, we will see that instances of classes defined using these latter three operations can contain a different number of references to other objects. This explains the Smalltalk term for these classes: "variable-length classes," in contrast to "fixed-length classes." Variable-length classes will be further described in the following section and section 5.2.

The significance of the `instanceVariableNames:`, `classVariableNames:`, and `poolDictionaries:` parameters in all four of the class-creating operations will also be described subsequently. The `category:` parameter is solely for the benefit of the user interface (see, e.g., [Goldberg 83] or chapter 17 of [GoldbergRobson 83]); it has no effect on the semantics of the class.

In Smalltalk-82, the **new-type** operation is realized through variants of the four **...subclass:...** operations described above. The difference is that the Smalltalk-82 operations take an additional parameter, referred to here as **otherSuperclasses:**. We will shortly see that this additional parameter allows a class to be associated with more than one parent. As an example, the following invocation from figure 2-16 creates a class whose name is the symbol **BHPoint**.

```
    BoundedPoint
       subclass: #BHPoint
       otherSuperclasses: (Array with: HistoryPoint)
       instanceVariableNames: 'bhlist bhtail'
       classVariableNames: ''
       poolDictionaries: ''
       category: #CanonicalExample'
```

Since classes in Smalltalk are instances of other classes, a description of these eight class-creation operations ought to include an indication of the class of the created objects. In each case, the newly created class is an instance of a *second* newly created class. This class of this latter class is the distinguished type **Metaclass**.

The point of creating two different classes is that the operations associated with a given data abstraction be distributed between the class and its metaclass. The class will have abstract operations which can be invoked with instances of the class as the generic parameter. The metaclass will have abstract operations which do not take an instance of the abstraction as a parameter. For example, the "create" operations of many data abstractions fall in the latter category.

4.3.1.4 The other procedures

The name of any class can be obtained through the invocation of the **name** operation of class **ClassDescription**, e.g.,

```
    C name
```

All eight realizations of the **new-type** procedure bind the newly created class to a variable corresponding to its name. Thus, if a class were created whose name was the symbol **Point**, the newly created class would be bound to the global variable **Point**. Since naming environments are implemented as instances of class Dictionary, and the distinguished name Smalltalk refers to the global dictionary, the dictionary-lookup operation at: can be applied to Smalltalk to obtain a newly-created class given its name. This is the technique used to implement the classNamed operation defined in figure 4-1.

As an aside, we note that the classNamed operation illustrates the awkwardness of having all procedure invocations be generic invocations. The abstract **type-named** procedure should be associated with the **type** abstraction, not that of **symbol**. But doing so would require the addition of an extraneous generic parameter, e.g.,

```
    Class classNamed: N
```

The sole utility of this extra parameter is to cause the generic invocation mechanism to look in the right operation set to find the method.

The abstract **set-name** procedure is realized as the rename: operation of class ClassDescription. If the name of a class C is N, then the invocation:

```
    C rename: NewN
```

results in the removal of the binding of N in the global environment and the binding of NewN to C.

Given the above, it is easy to see how the invocation:

```
    classNamed C
```

can fail to produce a class whose name is C. First, more than one class named C might exist, either through different invocations of the class-creation operations or through renaming one class to have the same name as another. Second, the global variable which has the same name as the class may have been rebound to some other value.

4.3.2 Partitioning operations

As described above, the abstract **type-of** procedure is realized as the class operation of Object. The expression

```
    X class
```

computes the class of X.

The abstract **instantiate** procedure is realized by the `new` and `new:` operations of class `Behavior`. The former, used for fixed-size classes, takes no non-generic parameters:

```
     Point new
```

The latter, used for variable-size classes. takes a single integer parameter:

```
     SomeOtherClass new: 10
```

We will see in section 5.2 that the non-generic parameter of `new:` is used to determines the number of "indexed instance variables" associated with the created object.

4.3.3 Parent/child operations

4.3.3.1 The parents of a class

In Smalltalk, the parent/child relation between classes is often referred to as a "superclass/subclass" relationship. Unfortunately, these terms are also commonly used to refer to the ancestor/descendant relation. In this report, the terms "superclass" and "subclass" will always refer to the parents and children of a class.

Invocations of the `...subclass:...` operations produce class which have a single parent. The parent of the class created by an invocation of a `....subclass:...` operation is the generic parameter of the invocation. For example, in the definition of `Point` given above, the parent of `Point` is the generic parameter, `Object`.

In Smalltalk-82, the `...subclass:otherSubclasses:...` operation allows classes with more than one parent to be created. The additional parents are specified through the `otherSuperclasses:` parameter, which must be an array of classes. The newly created class will have as its parents the generic parameter of the invocation *together with* each member of the `otherSuperclasses:` array. For example, the above invocation of `...subclass:otherSubclasses:...` creates a class whose parents are `BoundedPoint` and `HistoryPoint`.

Recall that the class-creation operations also create instances of `Metaclass`. For all eight of the class-creation operations, the parents of the class of the newly-created class are the classes of the parents of the newly-created class. For example, since the parent of `Point` is `Object`, the parent of the class of `Point` is the class of `Object`. And since the parents of `BHPoint` are `HistoryPoint` and `BoundedPoint`, the parents of the class of `BHPoint` are the class of `HistoryPoint` and the class of `BoundedPoint`.

Since `Object` is an ancestor of all system-defined classes (except itself, as it has no parent), and since the class-creation operations create classes which are children of one or more existing classes, `Object` is the **root-type** in both Smalltalk-80 and Smalltalk-82. Furthermore, the class hierarchy dominated by `Object` will be isomorphic to that dominated by the class of `Object`. In other words:

> If (1) A is a parent of B and (2) A and B are non-metaclasses, then the class of A is a parent of the class of B. [21]

Thus, the two parallel class and metaclass hierarchies can be viewed as a single "abstraction" hierarchy.

4.3.3.2 The information-extracting procedures

In Smalltalk-80, the single **parent** of a class can be computed by the `superclass` operation of class `Behavior`. For example,

```
    Point superclass
```

returns the unique parent of `Point`, namely `Object`.

In Smalltalk-82, the abstract **parents** procedure is realized as the `immediateSuperclasses` operation of `Behavior`. This operation computes an array containing the parents of its parameter. For example,

```
    BHPoint immediateSuperclasses
```

returns an array containing two members `BoundedPoint` and `HistoryPoint`. The `superclass` operation is also available in Smalltalk-82, but will return only one of the parents of the class.

[21] This relationship will hold for any program which uses only the standard operation for class introduction described above. If the **subclassOf:** operation of the class of **Metaclass** (which creates a new metaclass), or the **name:environment:...** operation of **Metaclass** (which actually creates the class) are used, this property will no longer necessarily hold. See [GoldbergRobson 83], p. 287.

In both Smalltalk-80 and Smalltalk-82, the **ancestors** procedure is realized as `allSuperclasses` operation of `Behavior`. The realization of the **set-of (type)** returned by this operation is an instance of the Smalltalk class `Array` whose members are the parents. For example, the invocation

```
     Point allSuperclasses
```

returns a single-element array containing `Object`, and

```
     BHPoint allSuperclasses
```

returns a four-element array containing `Object`, `Point`, `BoundedPoint`, and `HistoryPoint`.

The abstract **is-a** procedure is also realized identically in both versions of Smalltalk. The relevant operation is `isKindOf:` of class `Object`. For example,

```
     p isKindOf: Point
```

computes whether the class of p is `BHPoint` or some descendant. It is not to be confused with the `isMemberOf:` operation of `Object`, which tests whether the second parameter is the type of the first. E.g.,

```
     p isMemberOf: Point
```

will only succeed if

```
     (p class) == Point
```

Recall that the utility of a program-invocable realization of **is-a** was described in section 4.2.2.

4.4 The Loops realization

4.4.1 Type introduction operations

4.4.1.1 The realization of *type*

The type system defined by Loops classes is embedded in the Interlisp type system. As a result, all instances of Loops classes have both a "Loops type" and an "Interlisp type". In all cases, the "Interlisp type" is a "user-defined datatype," i.e., one whose storage format can be specified by programmers.[22] The unary predicate `Object?` can be used to determine if an arbitrary Interlisp object is also an instance of a Loops class.

[22]For details, see sections 3 and 23.11 of [Teitelman 78]).

Abstract op	*Realized as*	*Example*
type	*Any instance of an instance of* `$Metaclass`	
new-type	*The* `DefineClass` *procedure*	
	The `DefClass` *syntactic form*	
	The `New` *operation of* `$Metaclass`	
type-named	*The* `$!` *procedure*	`($! 'BHPoint)`
	The `$` *syntactic form*	`$BHPoint` *or* `($ BHPoint)`
type-name	*(*) The* `Name` *operation of* `$Class`	`(Send c Name)`
set-name	*The* `SetName` *operation of* `$Class`	`(Send $BHPoint SetName 'NewName)`

```
(DM 'Class 'Name '(self)
   (@ :name))
```

Figure 4-4: Type introduction in Loops

object	*Any Interlisp object for which* `Object?` *holds*	
type-of	*The* `Class` *operation of* `$Object`	`(Send bhp Class)`
instantiate	*(*) The* `Instantiate` *operation of* `$Class`	`(Send $BHPoint Instantiate)`

```
(DM 'Class 'Instantiate '(self)
   (Send self NewWithValues))
```

Figure 4-5: Partitioning operations in Loops

parents	*(*) The* `Parents` *operation of* `$Class`	`(Send $BHPoint Parents)`
ancestors	*(*) The* `Ancestors` *operation of* `$Class`	`(Send $BHPoint Ancestors)`
is-a	*The* `InstOf!` *operation of* `$Class`	`(Send bhp InstOf! $Point)`
root-type	`$Object`	

```
(DM 'Class 'Parents '(self)
   (Send self List 'Super))

(DM 'Class 'Ancestors '(self)
   (Send self List! 'Super))
```

Figure 4-6: Parent/child operations in Loops

The realization of **type** in Loops is closely related to that in Smalltalk. In particular:

- All Loops classes are themselves instances of classes.
- The type of any instance of a class can be computed by the **Class** operation of the distinguished class **Object**.
- An object is a class iff the type of its type is the distinguished class **Metaclass**.

4.4.1.2 The classes relevant to *type*

The realization of our common abstraction in Loops involves generic methods of three classes: **Object**, **Class**, and **Metaclass**.

- **Object** serves the same function in Loops as it does in Smalltalk. It "provides a set of default behaviors and generally available subroutines" ([BobrowStefik 83], p. 118).

- **Class** defines operations which are relevant to all objects which are classes, regardless of whether or not their instances are classes. Thus, it corresponds to **Behavior** and **ClassDescription** in Smalltalk. A distinction between the latter two would be irrelevant for Loops, since there is no "Loops interpreter" apart from the Interlisp interpreter.

- **Metaclass** serves the same purpose as **Metaclass** in Smalltalk. It has one operation: that used to create classes whose instances are classes.

4.4.1.3 The realization of the procedures

The abstract **new-type** procedure is realized in Loops through the Interlisp **DefineClass** function. **DefineClass** takes three parameters, two symbols and a list of symbols. The first symbol is taken as the **internal-name** parameter of **new-type**. The second symbol names the class of which the newly-created class is an instance. The list is taken as the **parents** parameter of **new-type**; it names the classes which are the parents of the newly-created class. For example, the invocation

```
(DefineClass 'BHPoint 'BoundedPointClass '(BoundedPoint
 HistoryPoint))
```

creates a class whose name is the symbol **BHPoint**, whose parents are the classes **BoundedPoint** and **HistoryPoint**, and which is itself an instance of the class **BoundedPointClass**.

The **DefClass** syntactic form and the **New** operation of the distinguished class **Metaclass** are interfaces to this procedure. Thus, the above invocation of **DefineClass** is equivalent to:

```
(Send $BoundedPointClass New
                         'BoundedPoint
                         '(BoundedPoint HistoryPoint))
```

or:

```
(DefClass BHPoint
    (MetaClass BoundedPointClass)
    (Supers BoundedPoint HistoryPoint))
```

Since the name of a class is the value associated with its **name** instance variable, defining a procedure which realizes the abstract **type-name** is straightforward. The definition given in figure 4-4 uses operations described in chapter 5 and 6 to define a **Name** operation for the class `$Class` which returns the value of that instance variable.

The abstract **type-named** procedure is realized using a general facility for associating "Loops names" with objects.[23] The class creation operation associates the newly created class with a "Loops name" corresponding to the name of the class. The object associated with a Loops name N in the current environment can be computed by the procedure **$!**.

For example, a type whose name was the symbol `Point` could be referred to via the expression:

```
($! 'Point)
```

Since the syntactic forms $*id* and ($ *id*) compute the object whose Loops name is the symbol corresponding to *id*, either

```
($ Point)
```

or

```
$Point
```

would refer to the same class.

The **set-name** operation is realized by the **SetName** (alias **Rename**) operation of `$Class`. This procedure changes the value of the **name** instance variable of the class, creates a Loops name which corresponds to the new name, and removes the Loops name which corresponds to the old

[23] "Loops names" are used to identify objects which have existence outside the context of a particular programming session. This aspect of Loops evolved from the earlier work on the PIE system described in, e.g., [GoldsteinBobrow 80]. Chapter 9 of [Stefik et al. 83b] provides a detailed description of of the naming facilities and how they are used in the design of community-wide knowledge bases.

one.[24] From the above, we can see that the Loops realization of **set-name** can fail in the same ways as could Smalltalk's. More than one class with the same name can exist, and the global Loops name could have been rebound. A third situation which will lead to failure is if the **name** instance variable of a class is modified without removing the old Loops name and creating a new one.

4.4.2 Partitioning operations

As described above, the abstract type **object** is realized as any Interlisp object for which the `Object?` procedure succeeds. The abstract **type-of** procedure is the `Class` operation of `$Object`, just as in Smalltalk.

The abstract **instantiate** procedure can be realized by an invocation of the `NewWithValues` operation of `$Class` where the generic parameter is the class to be instantiated and no other parameters are given. This is reflected in the definition of the `Instantiate` operation of `$Class` given in figure 4-5. For example, a new instance of `$Point` could be created by:

```
    (Send $Point Instantiate)
```

A full account of the semantics of `NewWithValues` and a description of the `New` operation of `$Class` is given in chapter 5.

4.4.3 Parent/child operations

The abstract **parents** operation can be realized by an invocation of the `List` operation of `$Class` where the symbol `Supers` is given as the second parameter. Figure 4-6 defines a `Parents` operation of `$Class` in terms of `List`. For example,

```
    (Send $BHPoint Parents)
```

will return a list containing the classes `BoundedPoint` and `HistoryPoint`.

The realization of **ancestors** is analogous. The ancestors of a class can be computed by invoking the `List!` operation with the symbol `Supers` as the second parameter, and this

[24] It also renames the methods of the class; see chapter 6 for a description of what this means.

technique is embodied in the **Ancestors** operation of figure 4-6. Given the definitions of chapter 2, the invocation

```
    (Send $BHPoint Ancestors)
```

returns a list containing the four classes **$BoundedPoint**, **HistoryPoint**, **Point**, and **Object**. The **List** and **List!** operations represent a general information-extracting utility in Loops.

As we will see, the order of the **parents** and **ancestors** lists in which the ancestor types appear in the list is is relevant to the behavior of a number of Loops primitives. Here is how the order is determined.

The **parents** list of a newly created class is ordered according the list parameter of the invocation of **DefineClass** which created the class. For example, the parents list of **BHPoint** as defined above would initially be:

```
    ($BoundedPoint $HistoryPoint)
```

The **ancestors** list for a class C is computed as follows.

1. Let G be a graph whose nodes represent all defined classes and which contains an edge from N1 to N2 iff N2 is a parent of N1.

2. Let C^* be a list of the nodes visited in a depth-first enumeration of the subgraph of the G dominated by C. The order of examination of the parents of each class is the order given in the **parents** list of the class.

3. The ordering of the ancestors of C is the result of removing all but the last occurrence of each class from C^*.

For example, consider the computation of the ordering for **BHPoint**. C^* is the sequence:

```
    [BoundedPoint, Point, Object, HistoryPoint, Point, Object]
```

Removing all but the last occurrence of **Point** yields an ordering of:

```
    [BoundedPoint, HistoryPoint, Point, Object]
```

Figure 4-7 describes the **ancestors** list of each of the classes defined in the example of chapter 2. Notice that the ancestor ordering algorithm guarantees that no class will appear before a descendant class in the list. In more formal terms, the enumeration algorithm produces a "topological sort" ([Knuth 69], p. 69). where the partial order is that defined by the "child-of" relation between types.

Class	Ancestor ordering
PointClass	Class, Object
Point	Object
BoundedPointClass	PointClass, Class, Object
BoundedPoint	Point, Object
HistoryPoint	Point, Object
BHPoint	BoundedPoint, HistoryPoint, Point, Object

Figure 4-7: Ancestor ordering in the Loops example

Finally, the **is-a** operation is realized as the `InstOf!` operation of class `Object`. For example,

```
    (Send X InstOf! $Point)
```

succeeds iff the type of `X` is `Point` or a descendant. As with Smalltalk, there is a separate operation, `InstOf`, which succeeds when one parameter is the type of the other. For example,

```
    (Send X InstOf $Point)
```

is equivalent to

```
    (EQ (Send X Class) $Point)
```

4.5 The Zetalisp realization

4.5.1 Type introduction operations

The **new-type** procedure is realized in Zetalisp as the **defflavor** syntactic form. The syntax associated with this form is as follows:

(defflavor id_{name}
        (*varspec* ... *varspec*))
        (id_{parent} ... id_{parent})
  clause
  ...
  clause)

varspec is either an identifier or the form:

(*id* *exp*)

and *clause* is either an identifier or a list whose first element is an identifier. All clauses are optional; the syntax of particular kinds of clauses will be described as they become relevant.

The id_{name} of the **defflavor** form realizes the abstract **internal-name** parameter of **new-type**. Thus, the following form from figure 2-4:

Abstract op	*Realized as*	*Example*
type	*In the non-flavor type system: instances of* `si:flavor` *or their names*	
new-type	*The* `defflavor` *syntactic form*	
type-named	*(*)* `flavor-named`	`(flavor-named 'bh-point)`
type-name	`si:flavor-name`	`(si:flavor-name f)`
set-name	*Not applicable*	

```
(defun flavor-named (n)
  (check-arg-type n :symbol)
  (let ((fl (get n 'si:flavor)))
    (if (typep fl 'si:flavor)
        fl
        (ferror "~s is not the name of a flavor." n))))
```

Figure 4-8: Type introduction in Zetalisp

object	*In the non-flavor type system: instances of* `:instance.`	
type-of	`typep`	`(typep bhp)`
instantiate	*(*)* `instantiate`	`(instantiate (flavor-named 'bh-point))`

```
(defun instantiate (f)
  (check-arg-type f si:flavor)
  (instantiate-flavor (flavor-name f) '(nil)))
```

Figure 4-9: Partitioning operations in Zetalisp

parents	*(*)* `flavor-parents`	`(flavor-parents (flavor-named 'bh-point))`
ancestors	*(*)* `flavor-ancestors`	`(flavor-ancestors (flavor-named 'bh-point))`
is-a	`typep`	`(typep bhp 'point)`
root-type	`si:vanilla-flavor`	

```
(defun flavor-parents (f)
  (check-arg-type f si:flavor)
  (union (si:flavor-depends-on f)
         (si:flavor-includes f)))

(defun flavor-ancestors (f)
  (check-arg-type f si:flavor)
  (cdr (si:flavor-depends-on-all f)))
```

Figure 4-10: Parent/child operations in Zetalisp

```
(defflavor bh-point
        (bhlist bhtail)
        ()
  (:included-flavors bounded-point history-point))
```

introduces a flavor whose name is **`bh-point`**.

The abstract **type-name** procedure is realized as `si:flavor-name`. Thus, if F is a flavor:

```
(si:flavor-name F)
```

will produce its name.

Introducing a flavor has the side effect of associating the flavor with the property `si:flavor` on the property list of the flavor name. For example, after the flavor definition given above, the property list of the symbol `bh-point` contains an association between the symbol `si:flavor` and the object representing the `bh-point` flavor. Thus, the abstract **type-named** procedure can be realized as the `flavor-named` procedure defined in figure 4-8.

Zetalisp contains no realization of `set-name`. However, since a new flavor can be created with the same name as an existing one, the **type-named** procedure can still fail.

4.5.2 Partitioning operations

The abstract **type-of** procedure is realized using the `typep` procedure. When the latter is applied to any object whose Zetalisp type is `:instance`, it returns the name of the flavor of which the object is an instance. For example, if

```
(typep bhp)
```

is the the symbol `bh-point`, then the type of **bhp** is

```
(flavor-named 'bh-point)
```

This technique is used to define the `type-of` procedure of figure 4-9.

The abstract **instantiate** is realized through Zetalisp's `instantiate-flavor` procedure. The two required parameters of this procedure are the name of a flavor and an object representing a property list. The most fundamental version of object instantiation occurs when the property list is empty, and no other parameters are given. More complex initialization actions which can be defined in terms of the above are described in chapter 5.

4.5.3 Parent/child operations

The abstract **parents** parameter is realized through two components of the form: the (id_{parent} ... id_{parent}) component and the optional `:included-flavors` clause. *varspec* is either an identifier or an expression of the form: The latter has the syntax:

(`:included-flavors` *id* ... *id*)

The parents of the flavor are the flavors named by the remaining *id*s, together with the system-defined flavor named `si:vanilla-flavor`.[25] For example,

```
    (defflavor bh-point
            (bhlist bhtail)
            (bounded-point history-point))
```

introduces a flavor named **bh-point** whose parents are the flavors named by the symbols **bounded-point**, **history-point**, and `si:vanilla-flavor`.

The abstract **ancestors** procedure is realized through the **si:flavor-depends-on-all** function. When this function is applied to a flavor, it produces a list whose car is the name of the flavor and whose cdr is a list containing its ancestors. For example, given the example in figures 2-1 through 2-4, the invocation:

```
    (si:flavor-depends-on-all (flavor-named 'bh-point))
```

returns the list:

```
    (bh-point bounded-point history-point point vanilla-flavor)
```

As with Loops, the order of the list produced by the **ancestors** operation is semantically relevant. In the absence of any **:included-flavors** clauses, the algorithm for producing the ancestor ordering is almost identical to that used in Loops.

1. Let G be a graph whose nodes represent all defined classes but **si:vanilla-flavor** and which contains an edge from N1 to N2 iff N2 is a parent of N1.

2. Let C^* be a list of the nodes visited in a depth-first enumeration of the subgraph of the G dominated by C. The order of examination of the parents of each class is the order given in the **parents** list of the class.

3. The ordering of the ancestors of C is the result of removing all but the *first* occurrence of each class from C^*, and adding **si:vanilla-flavor** at the end.

Recall that the Loops algorithm removed all but the *last* occurrence in step 3.

The computation of the ordering for the **bh-point** flavor in Zetalisp illustrates the effect of the difference between the Zetalisp and Loops ancestor ordering algorithms. In the Zetalisp computation, the C^* sequence is identical (up to renaming) to that of Loops:

[25] The inclusion of **si:vanilla-flavor** can be suppressed via the **:no-vanilla-flavor** clause in the **defflavor** form.

```
bounded-point, point, history-point, point
```

Removing all but the first occurrence of **point** and adding **si:vanilla-flavor** yields an ordering of:

```
bounded-point, point, history-point, si:vanilla-flavor
```

Notice that the Zetalisp algorithm does not share the property of the Loops algorithm that the result will be a topological sort of the ancestor types. In the above example, **point** appears before **history-point** in the constructed enumeration. This difference will prove to have considerable significance for the algorithm for constructing generic operation sets.

To account for **:included-flavors** clauses, the algorithm is modified as follows. First, the graph constructed in step 1 contains an edge from N1 to N2 only if N2 appears is named in the id_{parent} list of N1. Second, the following step is added after step 3:

> If any flavor is named in an **:included-flavor** clause of some member of C^{*} but does not itself appear in C^{*}, it is added after the last member of C^{*} for which it is an included flavor.

The point of the **:included-flavors** clause is that it can sometimes be used to construct an ancestor ordering where each type *does* appear before its parents. For example, if the four flavor definitions of our example were:

```
(defflavor point () ())
(defflavor history-point () () (:included-flavor point))
(defflavor bounded-point () () (:included-flavor point))
(defflavor bh-point () (bounded-point history-point))
```

then the ancestor ordering for **bh-point** would be:

```
bounded-point, history-point, point, si:vanilla-flavor
```

and which is a valid topological sort of the four ancestors of **bh-point**.

4.5.4 "Partially defined" flavors

An aspect of type introduction in Zetalisp which is not shared by our other three languages is that parent types need not exist at the time a type is introduced. All that is required is that at the time an instance of the flavor is created, all ancestors exist. The key benefit of this capability is that type-introducing forms needed for a program can be evaluated in any order, thus relieving the programmer from a small amount of logistical responsibility.[26]

For example, it would be valid to introduce the four types of our example by the sequence:

```
    (defflavor bh-point ...)
    (defflavor bounded-point ...)
    (defflavor history-point ...)
    (defflavor point ...)
```

Such a sequence would be unacceptable in Smalltalk or Loops. This sequence would be possible in the *text* of a Star Mesa program, but all trait definitions must be provided before an environment containing them is created.

This capability is realized by allowing the objects which represent flavors to be in a "partially defined" state. In this state, the names of the intended parent flavors are known, but the flavors corresponding to these names may not yet have introduced or may themselves be partially defined. For example, after the following sequence:

```
    (defflavor bh-point () (bounded-point history-point))
    (defflavor bounded-point () (point))
    (defflavor point () ())
```

it is known that the names of the parents of **bh-point** will be named **bounded-point** and **history-point**. The former is in a partially defined state, and the latter does not yet exist.

In order to fully define a flavor, the **compose-flavor-combination** procedure must be used. This procedure will fully define a flavor and all of its ancestors, so long as all ancestors are either fully or partially defined. Thus, it would succeed in the first of the above two examples, but fail in the second. Since **compose-flavor-combination** is automatically invoked whenever an attempt is made to instantiate a partially defined flavor, explicit invocation of the former is often unnecessary.

As a historical note, the concept of a type being in an "unfinished" type can be traced back at least as far as Wegbreit's thesis [Wegbreit 70]. It was needed there so that recursive types could be introduced, e.g. a **LIST** type whose representation was either **NIL** or a record containing an

[26]In fact, Zetalisp allows flavors to be mutual ancestors. The utility of this capability is far from clear.

instance of LIST. This motivation is not present in Zetalisp, since instance variables are not associated with types.

4.6 The Star Mesa realization

Abstract op	*Realized as*	*Example*
type	*In the Mesa type system:* TM.Trait	
new-type	*The* TRAIT *syntactic form*	
type-named	TM.TraitNamed	TM.TraitNamed ["BHPoint"]
type-name	TM.TraitName	TM.TraitName [T]
set-name	*Not applicable*	

Figure 4-11: Type introduction in Star Mesa

object	*In the Mesa type system:* TM.Object	
type-of	TM.TypeOf	TM.TypeOf [bhp]
instantiate	TM.Allocate	TM.Allocate [TM.TraitNamed ["BHPoint"]]
	The TM.Alloc *syntactic form*	TM.Alloc [BHPoint]

Figure 4-12: Partitioning operations in Star Mesa

type-root	*Not applicable.*	
parents	TM.Parents	TM.Parents [TM.TraitNamed ["BHPoint"]]
components	TM.CarriedTraits	TM.CarriedTraits [TM.TraitNamed ["BHPoint"]]
is-a	TM.Carries	TM.Carries [bhp, TM.TraitNamed "Point"]

Figure 4-13: Parent/child operations in Star Mesa

4.6.1 The mechanics of the extension

In the Star extension of Mesa, traits are introduced using the TRAIT syntactic form. The syntax for such definitions is closely related to that of conventional Mesa PROGRAM modules (e.g., [LauerSatterthwaite 79], chapter 7 of [Mitchell et al. 79]), except that the keyword TRAIT replaces PROGRAM. Such definitions have a number of parts, including:

- a name,
- a list of the names of the other modules used,
- a directory which relates module names to file names,
- a Mesa *block*, i.e., a sequence of identifier declarations followed by executable statements.

For example, figures 2-20 through 2-27 contain four trait definition modules.

In order to be processed by the underlying Mesa system, trait definitions are transformed into Mesa **DEFINITIONS** and **PROGRAM** module with related names. In this report, we will assume that if the name of the trait definition module is *id*, then the name of the associated definitions module will be *id*.[27]

Aside from the transducer which creates module definitions from trait definitions, the Star Mesa extension consists of a single "trait manager" module. This module contains the procedure definitions which appear in trait definitions and which obtain and manipulate trait objects. The identifier **TM** will be used to refer to the trait manager module.

Trait definition forms must contain bindings for two distinguished identifiers.

- The identifier **Register** must be bound to a procedure which takes no parameters and return an object of a distinguished record type, **TM.Registration**.[28] The definition of the latter is:

  ```
  RECORD [name: STRING,
          parents: ARRAY OF STRING,
          TCSize: NATURAL,
          ICSize: NATURAL,
          classTrait: BOOLEAN]
  ```

  We will refer to the **Register** procedure of a trait definition module as the "registration procedure" of the defined trait. For example, figure 2-26 defines the following as the registration procedure of trait **BHPoint**:

  ```
  Register: PROC [] RETURNS [TM.Registration] =
     {RETURN [name: "BHPoint",
              parents: ["BoundedPoint", "HistoryPoint"],
              TCSize: SIZE [TCType],
              ICSize: SIZE [ICType],
              classTrait: TRUE]}
  ```

- The identifier **InitializeTrait** must be bound to a procedure which takes no parameters and returns nothing. We will refer to this procedure as the "initialization procedure" of the defined trait. For example, figure 2-26 defines the initialization procedure of **BHPoint** to be:

[27] The published descriptions of Star Mesa do not include the names used of all relevant data structures and procedures. The names used here do not necessarily correspond to the names used in the actual implementation.

[28] All identifiers are explicitly or implicitly qualified by the module in which they appear. The form $id_1.id_2$ refers to the identifier id_2 declared in module id_1. Module definitions constitute the outermost scope for the resolution of unqualified identifiers.

```
        InitializeTrait: PROC [] =
           {Point.LocalInitializeTrait [TM.TraitNamed["BHPoint"]];
            BoundedPoint.LocalInitializeTrait
    [TM.TraitNamed["BHPoint"]];
            HistoryPoint.LocalInitializeTrait
    [TM.TraitNamed["BHPoint"]];
            BHPoint.LocalInitializeTrait [TM.TraitNamed["BHPoint"]]}
```

The significance of the components of the registration record and of the trait initialization procedure will be described as they become relevant.

After the module definitions resulting from the trait definition forms are individually compiled, a C/Mesa program is created. This program is used to direct the Mesa binder in creating a "binary configuration description" from the collection of modules which constitute the program. The latter can be processed by the Mesa loader to create an executable program. The initial actions taken by this program are to invoke the registration procedure of each trait definition, then invoke each initialization procedure of each definition.

Given the above description of logistics, we can now show how the Star Mesa constructs realize the common abstraction.

4.6.2 The realization per se

The abstract type **type** is realized as a Mesa type, referred to here as **TM.Trait**, which is a **PRIVATE** type of the trait manager module, The abstract type **object** is realized as the private type **TM.Object**. Instances of **TM.Trait** are not also instances of **TM.Object**.

The abstract **new-type** procedure is realized as the **TRAIT** syntactic form. The **internal-name** parameter of the trait definition form is the value of the **name** component of the record returned by the initial invocation of the registration procedure. An error will be signalled if two traits are defined to have the same name.

The **parents** parameter of the trait definition form is the set of traits named by the **parents** returned by the initial invocation of the registration procedure. The named parents must all exist after all registration procedures of all trait definition modules have been invoked.

As an illustration, consider the registration procedure given above. The `name` component of the record it returns will always contain the string `"BHPoint"`. The `parents` component will always contain the array:

```
    ["BoundedPoint", "HistoryPoint"]
```

Thus, the name of the trait defined by the containing module will be `BHPoint`, and the parents will be the traits whose names are `BoundedPoint` and `HistoryPoint`.

The remainder of the procedures of the common abstraction are as listed in figures 4-11 through 4-13. Since there need not be a trait which is an ancestor of every trait but itself, there is no realization of `root-type`.

5. Object creation and the instance variable operations

In each of our languages, it is possible to manipulate instances of our subject types as if they were sets of associations between names and objects. Since the term "instance variable" is often used to refer to these names, we will refer to this collection of procedures as the **instance variable** operations. Barring the use of bit-level manipulations of object representations, any computation on instances of any of our subject types can be described in terms of instance-variable operations.

The first section identifies the procedures which constitute the instance-variable operations and describes some properties of their behavior. The second describes how these procedures are realized in our four languages. The final section describes the related concepts of "class variable" and "properties" of variables used in one or more of our languages.

5.1 The common abstraction

Chapter 4 described a set of abstract procedures which characterized the relationship between types and instances and between ancestor and descendant types. In describing the behavior of those procedures, instances of our subject types were viewed as objects whose only relevant aspects were their identity and their type. But to describe the instance variable operations, a richer model is necessary. The approach taken here is as follows.

A new component used to model objects is a function, in the mathematical sense, from names to values. That is, objects behave as if they are collections of associations from names to values, such that no two associations in a given collection contain the same name. The abstract procedure:

IV-names (object) => set-of (IV-name)

denotes the set of names in the domain of the function modelling an object. If the collection modelling the object O associates a symbol S with a value V:

get-IV-value (O, S)

yields V, and

set-IV-value (O, S, V')

removes the association between S and V and replaces it with one between S and V'.

A new component used to model types is a set of names, computable by the abstract procedure:

```
local-instance-IV-names (type) => set-of (IV-name)
```

The procedure:

```
instance-IV-names (type) => set-of (IV-name)
```

is defined to be the union of the **local-IV-names** of T and each member of **ancestors (T)**. For example, if:

```
local-instance-IV-names (Point) = {location}
local-instance-IV-names (BoundedPoint) = {min, max}
local-instance-IV-names (HistoryPoint) = {hlist, htail}
local-instance-IV-names (BHPoint) = {bhlist, bhtail}
```

then:

```
instance-IV-names (Point) = {location}
instance-IV-names (BoundedPoint) = {min, max, location}
instance-IV-names (HistoryPoint) = {hlist, htail, location}
instance-IV-names (BHPoint) = {bhlist, bhtail, min, max, hlist, htail, location}
```

Then the key assertions we can make about our four languages is the following:

- The set of associations modelling a newly created object contains an association for each **instance-IV-name** of the type of the object:

  IV-names (instantiate (T)) = instance-IV-names (T)

  The values associated with these names are intentionally left unspecified, since they differ in our four languages.

- The behavior of **get-IV-value** and **set-IV-value** when the collection does *not* contain an association for the specified symbol is also left unspecified, since it is also language-specific.

  - In Smalltalk, Zetalisp, and Star Mesa, an error is signalled.
  - In Loops, `set-IV-value` creates a new association, and `get-IV-value` returns the value of a distinguished variable.

  Thus, the most we can say in general about the **IV-names** of objects is the following:

  instance-IV-names (type-of (O)) $\subseteq$ **IV-names (O)**

The remainder of this chapter describes the details of how the above abstraction is realized.

IV-name	`Symbol` *or* `Integer`	`#min` *or* `3`
local-instance-IV-names	`instVarNames` *of* `Behavior`	`BHPoint instVarNames`
instance-IV-names	`allInstVarNames` *of* `Behavior`	`BHPoint allInstVarNames`
IV-names	*(*)* `IVNames` *of* `Object`	`bhp IVNames`
get-IV-value	*In certain contexts, an identifier.*	`min`

	(*) `getIVValue:` *of* `Object`	`bhp getIVValue: #min`
		`someOtherObject getIVValue: 3`

set-IV-value	*In certain contexts, an identifier assignment.*	`min ← 17`

	(*) `setIVValue:to:` *of* `Object`	`bhp setIVValue: #min to: 17`
		`someOtherObject setIVValue: 3 to: "abcdefg"`

Figure 5-1: Instance variable operations in Smalltalk-80 and Smalltalk-82

```
IVNames
  |classInstVars|
  classInstVars ← (self class) allInstVarNames.
  (self Object.size) = 0
    ifTrue: ^ classInstVars
    ifFalse: ^ classInstVars union: (Interval from: 1 to: (self Object.size))

getIVValue: name
  (name class) isMemberOf: Symbol
    ifTrue: [^ self instVarAt: (self instVarOffset: name)]
    ifFalse: (name class) isKindOf: Integer
               ifTrue: [^ self Object.at: name]
               ifFalse: [^ self error 'inappropriate IV name']

setIVValue: name to: value
  (name class) isMemberOf: Symbol
    ifTrue: [^ self instVarAt: (self instVarOffset: name) put: value]
    ifFalse: (name class) isKindOf: Integer
               ifTrue: [^ self Object.at: name put: value]
               ifFalse: [^ self error 'inappropriate IV name']

instVarOffset: name
  |ivArray|
  ivArray ← (self class) instVars.
  ^ ivArray indexOf: name
```

Figure 5-2: Auxiliary operations of class `Object` used in figure 5-1

IV-name	`ATOM`	`loc`
local-instance-IV-names	*Invocation of* `List` *of* `Class` *with* `'IVs` *as the second parameter*	`(Send $BHPoint List 'IVs)`
instance-IV-names	*Invocation of* `Listl` *of* `Class` *with* `'IVs` *as the second parameter*	`(Send $BHPoint Listl 'IVs)`
IV-names	*Invocation of* `Listl` *of* `Object` *with* `'IVs` *as the second parameter*	`(Send bhp Listl 'IVs)`
get-IV-value	`GetValueOnly`	`(GetValueOnly bhp 'min)`
set-IV-value	`PutValueOnly`	`(PutValueOnly bhp 'min)`

Figure 5-3: Instance-variable operations in Loops

IV-name	:symbol	hlist
local-instance-IV-names	si:flavor-local-instance-variables	(si:flavor-local-instance-variables (flavor-named 'bh-point))
instance-IV-names	si:flavor-all-instance-variables	(si:flavor-all-instance-variables (flavor-named 'bh-point))
IV-names	(*) IV-names	(IV-names bhp)
get-IV-value	*In certain contexts, an identifier.*	min
	symeval-in-instance	(symeval-in-instance bhp 'min)
set-IV-value	*In certain contexts, an identifier assignment.*	(setq min 17)
	set-in-instance	(set-in-instance bhp 'min 17)

```
(defun IV-names (obj)
  (check-arg-type obj :instance)
  (si:flavor-all-instance-variables
       (flavor-named (typep obj))))
```

Figure 5-4: Instance variable operations in Zetalisp

IV-name	STRING	"BHPoint"
local-instance-IV-names	*The* local-instance-IV-names *of a trait T is always {T}.*	[TM.TraitNamed ["BHPoint"]]
instance-IV-names	TM.CarriedTraits	TM.CarriedTraits [TM.TraitNamed ["BHPoint"]]
IV-names	(*) CarriedTraitsOfTypeOf	CarriedTraitsOfTypeOf [TM.TraitNamed ["BHPoint"]]
get-IV-value	TM.InstanceComponent	TM.InstanceComponent [bhp, TM.TraitNamed ["Point"]]
	TM.InstComp *expressions*	TM.InstComp [bhp, Point]
set-IV-value	*modifying the referent of* TM.InstanceComponent *invocations*	(TM.InstanceComponent [bhp, TM.TraitNamed ["Point"]]) ^ ← Point.TCType [loc: 17]
	modifying the referent of TM.InstComp *expressions*	(TM.InstComp [bhp, Point]) ^ ← Point.TCType [loc: 17]

```
CarriedTraitsOfTypeOf: PROC [i: Instance] =
    {TM.CarriedTraits [TM.TypeOf [bhp]]}
```

Figure 5-5: Instance-variable operations in Star Mesa

5.2 The Smalltalk-80 and Smalltalk-82 realization

5.2.1 Instance variable names

The **local-instance-IV-names** and **instance-IV-names** of a class are represented by `Arrays` of `Symbols` fixed at the time the class is created. These arrays are computed from the value of the `instanceVariableNames:` parameter of the class-creation operations of class `Metaclass` and `Class`.

For all Smalltalk classes, the array representing the **local-instance-IV-names** of the class is formed by creating an array of symbols from the string given in the creation operation, leaving out any symbols which are **IV-names** of any ancestor. For example, the **local-instance-IV-array** of the four example classes of chapter 2 are:

```
    Point: (#loc)
    HistoryPoint: (#hlist #htail)
    BoundedPoint: (#min #max)
    BHPoint: (#bhlist #bhtail)
```

These arrays can be computed for a class through the `instVarNames:` operation of `Class`, as in figure 5-1.

Recall from chapter 4 that there were four variants of both the single-parent and multiple-parent class creation operations: `subclassOf:...`, `variableSubclass:...`, `variableWordSubclass:...`, and `variableByteSubclass:...`. Instances of classes created via the latter two operations are subject to the restriction that their set of **local-instance-IV-names** must be empty. We will see that such objects represent arrays of small integers; the purpose of this restriction is to achieve higher performance through faster indexing.

The **instance-IV-names** of a class is modelled by an array representing the union of the **local-instance-IV-names** of the class and its ancestors:

```
    Point: (#loc)
    HistoryPoint: (#loc #hlist #htail)
    BoundedPoint: (#loc #min #max)
    BHPoint: (#loc #min #max #bhlist #bhtail #hlist #htail)
```

These arrays are accessible through the instVarNames: operation of Class, also illustrated in figure 5-1.

In chapter 4, we saw that Smalltalk distinguishes between "fixed size" and "variable size" classes. The **IV-names** of an object depends on the kind of class of which it is an instance.

- The **IV-names** of an instance of a fixed-size class is simply the **instance-IV-names** of its class.

- The **IV-names** of an instance of a variable-size class consists of the **instance-IV-names** of the class, together with the integers from 1 to the value of the new: parameter of the creation operation which produced the object. For example, if SomeOtherClass were variable-size class whose parent was Object, the **IV-names** of an object produced by the invocation:
  ```
  SomeOtherClass new: 3
  ```
  would consist of the **instance-IV-names** of SomeOtherClass together with the integers 1, 2, and 3. We note that the value of the new: parameter used to create an instance of a variable-size class can be recovered by the method for size of class Object. Thus, the value of:
  ```
  (SomeOtherClass new: 3) size
  ```
  would be 3.

In light of the above, we can define the IVNames operation of figure 5-2 to compute a collection containing the **IV-names** of any object. Its definition uses the size operation of class Object and the indexOf: operation of class SequenceableCollection. The latter returns the index of the first occurrence of the non-generic parameter in the generic parameter ([GoldbergRobson 83], p. 153). The definition also assumes the existence of a union: operation with the obvious "set union" semantics; such an operation is not predefined in Smalltalk-80 and no definition is given here.

In Smalltalk's terminology, instance variables identified by instances of Symbol are referred to as "named" instance variables and those identified by numbers are known as "indexed" instance variables, This terminology will also be used below.

5.2.2 Storage and retrieval via syntactic forms

To store and retrieve the instance variables of an object, two approaches can be used. One involves a distinguished syntactic form, and the other uses conventional procedure invocation. Each is described in turn.

The first technique can be used to retrieve or store some of the instance variables of the current value of **self**.[29] The basic idea, adapted from Simula-67, is that evaluation of and assignment to certain variables have the semantics of **get-IV-value** and **set-IV-value** applied to **self**. For example, the expression

```
    min
```

would be given the semantics of

```
    get-IV-value (self, #min)
```

and

```
    min ← 17
```

would mean

```
    set-IV-value (self, #min, 17)
```

The set of variable expressions for which this non-standard treatment is given depends on the method in which the expression is found. In Smalltalk, each method is associated with a class, and the **instance-IV-names** of this class define the variables for which the non-standard semantics are used. For example, since the **instance-IV-names** of **HistoryPoint** are **loc**, **hlist**, and **htail**, a method associated with **HistoryPoint** would treat the expression

```
    loc ← 50
```

as an operation to store 50 in the **loc** instance variable of **self** and

```
    hlist = htail
```

as a identity test on the current values of the **hlist** and **htail** instance variables of **self**.

What is the class associated with a given method? The answer to that question is fully described in chapter 6. For now, we simply observe that for each expression of the form:

```
    C compileAndStore: S
```

[29]Recall from chapter 3 that **self** is bound to the generic parameter of the operation invocation.

in figures 2-12 through 2-17 the method described by the string S is associated with C. For example, the method defined by:

```
    Point compileAndStore:
      'location
          ^ loc'
```

is associated with **Point**. Thus, the reference to the variable loc is interpreted as a reference to the instance variable named loc of **self**.

Note that if the syntactic forms are used, the consequence of specifying an incorrect instance variable name (i.e., one which is not an **instance-IV-name** of the class associated with the method) will depend on whether a variable of the same name is a local variable of the method. If so, the erroneous reference will be treated as a reference to the local variable. If not, a "reference to undefined variable" error will be signalled by the compiler. Thus, a limited form of compile-time error checking is provided by these mechanisms.

5.2.3 Storage and retrieval via procedure invocation

The second way in which instance variable operation can be carried out is through invoking operations provided for that purpose. The operations **at:**, **at:put:**, **instVarAt:**, and **instVarAt:put:** of class **Object** allow the storage and retrieval of any instance variable of any Smalltalk object. In the following descriptions of these operations, N must be an instance of a descendant of class **Integer**.

- **O instVarAt: N**
  retrieves the value of the instance variable of O whose name is the N^th element of the array representing the **instance-IV-names** of the class of O. For example, since the **instance-IV-names** array of **BoundedPoint** is:

  ```
  (#loc #min #max)
  ```

  the invocation

  ```
  bp instVarAt: 2
  ```

  would retrieve the value of the instance variable named **min**, provided **bp** was an instance of **BoundedPoint**. If N is not a positive integer less than or equal to the the size of the **instance-IV-names** of the class of O, an runtime error is signalled.

- **O instVarAt: N put: X**
  is analogous to **instVarAt:**, except that X becomes the new value of the instance variable referred to by N. The same conditions for runtime error-signalling apply.

- `O Object.at: N`
  retrieves the value of the N^th^ indexed instance variable of O. Recall that invoking the `Object.at:` operation on an instance of any class is carried out by the `at:` method of `Object`. N must be greater than 1 and less than or equal to O **Object.size** otherwise, an error is signalled.

- `O Object.at: N put: X`
  stores X as the N^th^ indexed instance variable of O. Again, N must be greater than 1 and less than the number of indexed instance variables.

The operations defined in figure 5-2 provide an interface to the above four procedures which is consistent with our **get-IV-value** / `put-IV-value` abstraction.

Unlike the syntactic forms described above, an invocation of these four operations can be used to manipulate any instance variable of any object. If a method is to access the indexed instance variables of an object, or to perform instance variable operations on more than one object, the use of these operations is mandatory. A common programming task where the latter is appropriate is in the implementation of binary (or more generally, N-ary) operations on instances of the same type. For example, a linear-time algorithm can be used to compute the union of two sets represented as ordered lists, if access to both lists is allowed.

The use of these procedures does have its drawbacks, however.

- First, they can be used to violate the principle described in chapter 3 that only the generic operations of a type be used to manipulate the representation of its instances. Their presence in the language makes it the responsibility of the programmer to decide when their use is appropriate.

- Second, the Smalltalk implementation will never perform static error checking on invocations involving `self` and a symbol constant. For example, if the **instance-IV-name** of the class of a method does not include the symbol `wrongName`, the expression
  ```
      self setIVValue: #wrongName to: x
  ```
  will never trigger a compiler error. But the equivalent
  ```
      wrongName ← x
  ```
  will be detected as an error, provided the method does not include a local variable of that name.

Notice that the latter disadvantage could easily be overcome by adding `getIVValue:` and `setIVValue:to:` to the language and modifying the compiler to process them appropriately.

Finally, we note the restriction on the instance variable storage operation for instances of classes created with the `variableWordsSubclass:...` and `variableBytesSubclass:...` operations. Instances of these classes represent arrays of integers, and attempting to store values which are not integers causes a runtime error. Furthermore, they can only hold positive integers less than some upper bound. In the implementation described in [GoldbergRobson 83], the former is 2^8-1 and the latter is 2^{16}-1. It should be clear that the existence of these classes is solely for greater efficiency in processing and storing such values.

5.2.4 Instantiation

Recall from chapter 4 that new instances of a class C are created by invoking the `new` and `new:` operations of class `Behavior` with C as the generic parameter. For instances of fixed-size classes and variable-size classes created with the `variableSubclass:` operation each of the IV-names is associated with `Nil`. For instances of `variableWordsSubclass:...` and `variableBytesSubclass:...` classes, all IV-names are associated with `0`.

Of course, for many data abstractions, these default values are inappropriate. The appropriate response is to create instances of classes through operations which perform the appropriate initialization. To illustrate, consider the example of figure 2-12. The `create` operation of `(Point class)` invokes the basic object creation operation and then invokes the `initialize:` method for `Point` on the newly-created object. The latter performs some computation which associates a value with the `location` instance variable.

However, notice that there is no guarantee that the initialization procedures associated with a class will actually be invoked for all instances of a type. The `Object.new` and `Object.new:` operations described above allow any procedure to create an uninitialized instance of any class. For example,

```
    Point Object.new
```

would create an instance of `Point` whose `loc` instance variable was `Nil`.

5.2.5 Conventions for object initialization

A common Smalltalk-80 programming technique is for the designer of a class to reimplement new and/or new: for the instance of Metaclass which is the class of C. This allows type-specific initialization procedures to be invoked immediately after instances of a class are created.

In Smalltalk-80, a common pattern for class-specific new methods is as follows:

```
|newinstance|
    newinstance ← super new.
    newinstance initialize.
    ^ newinstance
```

If all ancestors of a metaclass define new methods in the same way, then the initialize method of each ancestor will get invoked whenever an instance of the class is created. The nested invocations bottom out at new of Behavior (the relevant branch of the initial class hierarchy is Object class, Class, ClassDescription, Behavior) and the initialization actions are performed in ancestor-first order. A separate initialize: operation is used because operations of a metaclass cannot use the syntactic forms described above to operate on instances of the class.

In Smalltalk-82, this approach is no longer adequate. The problem is that if a class has more than one parent, invoking the new operations for each parent will result in a new instance for each path in the class hierarchy from the class to Behavior. For example, if a new method for the class of BHPoint invoked the new method of BoundedPoint and HistoryPoint, the result would be two instances of BHPoint: one initialized by Point class and BoundedPoint class, and the other by Point class and HistoryPoint class.

In order to avoid the problem of duplicated invocation of the object creation procedure, a new paradigm is needed which will not break down in the presence of multiple parents. The following describes the technique used in all four examples of chapter 2.

- *The creation operations for each class invoke* new *of* Behavior *directly, then invoke an* initialize: *operation on the newly created object.* This ensures that exactly one object is created per invocation, and that all initialization actions are performed on this object.

- *If the initialization method for a class is to invoke the initialization method of an ancestor class, it defines a* `partialInitialize:` *operation whose method carries out the class-specific initialization actions but invokes no other* `initialize:` *or* `partialInitialize:` *operations. The method for* `initialize:` *invokes all* `partialInitialize:` *operations of all ancestor classes.*

To illustrate, here is how this convention was applied in our example. The `initialize:` methods of `HistoryPoint` and `BoundedPoint` invokes the `initialize:` operation of `Point` (to initialize the `loc` instance variable), then their own `partialInitialize:` operation (to initialize their local instance variables). The `initialize:` of `BHPoint` invokes the `partialInitialize:` of all ancestors which define it (i.e., `BoundedPoint` and `HistoryPoint`), in addition to the the `initialize:` operation of `Pcint`. Thus, all initialization actions were performed once, and on the same instance of `BHPoint`.

In our simple scenario, the consequences of redundant invocation of the initialization procedure of `point` are negligible; all that is lost is a small increment of performance. But for more complex data abstractions (e.g., creating a new display window with associated I/O stream) the result may be erroneous behavior or intolerable computational overhead.

5.2.5.1 The general phenomenon of redundant invocation

The possibility of redundant invocation of the methods of ancestor types is not limited to initialization operations. It can arise whenever a generic method for a given operation for a type invokes the generic method for the same operation of more than one parent of the type. If those methods are defined to call the method of a common ancestor, redundant invocation will occur.

Understanding this phenomenon is crucial in designing methods of types which have more than one parent. An immediate consequence is that the Smalltalk-80 approach of having a method for an operation O of a class C invoke the method for O of the superclass of C (i.e., via the `super` syntactic form) *does not generalize* when multiple superclasses are involved. For example, if the `display:` methods of `BoundedPoint`, `HistoryPoint`, and `BHPoint` all invoked the `display:` methods of their parents, displaying a type would result in the erroneous behavior such as:

```
  Point at location: ...
   with bounds: ....
 Point at location: ...
   with history: ...
   with bounds history: ...
```

Thus, the same technique which was used for **initialize:** is also used for **move:** and **display:** of each of our four classes, and for **setmin:** and **setmax:** of **BHPoint**.

A significant consequence of the above analysis is that **all.***op* invocation of Smalltalk-82 and **SendSuperFringe** of Loops is often useless in the presence of multiple parents. As we will see in chapter 6, both of these forms result in invocations of all methods for a given operation defined for all parents. Notice that this is "parents," not "ancestors"

To illustrate the inapplicability of these forms, imagine a modification of our example where **Point** was defined in terms of a more basic class, and also had a **partialInitialize:** method. Now consider the problem of writing the **initialize:** method for **BHPoint** so that the **partialInitialize:** methods of **Point**, **BoundedPoint**, and **HistoryPoint** would all be invoked. An invocation of **all.initialize:** would not be appropriate, since it would cause redundant invocation of the **initialize:** method of **Point**. And an invocation of **all.partialInitialize:** would not be adequate, since it would *not* invoke the **partialInitialize:** method of **Point**.

Again, this demonstration of inapplicability is not limited to initialization, but generalizes to other operations, e.g., **display**, We will see in chapter 7 that it is straightforward to define a new Loops procedure with semantics appropriate for the invocation of ancestor methods.

5.3 The Zetalisp realization

5.3.1 Instance variable names

The **local-instance-IV-names** of a flavor is derived from one of the components of the form are derived in a straightforward way from the most recent **defflavor** invocation which involving the flavor. Recall that the syntax for **defflavor** is:

```
(defflavor $id_{name}$
        (varspec ... varspec))
        ($id_{parent}$ ... $id_{parent}$)
  clause
  ...
  clause)
```

where *varspec* is either an identifier or the form:

```
(id exp)
```

The **local-instance-IV-names** are simply the *id*s of the *varspecs*. For example, in the definition of **bh-point**:

```
(defflavor bh-point
        (bhlist bhtail)
        (bounded-point history-point))
```

the **local-instance-IV-names** are **bhlist** and **bhtail**. In the definition of **bounded-point**:

```
(defflavor bounded-point
        ((min 0)
         (max 100))
        (point)
  :settable-instance-variables
  :initable-instance-variables)
```

they are **min** and **max**.

The **instance-IV-names** of a flavor are simply the union of the **local-instance-IV-names** of the flavor and all ancestors. And the **IV-names** of any object are the **instance-IV-names** of its type. Thus, Zetalisp has no analog to the ordered instance variables of Smalltalk.[30]

---

[30]However, Zetalisp's realizations of arrays with "array leaders" [WeinrebMoon 81] provides an operation set similar to that of Smalltalk's variable-size classes. They are not further considered here because there is no associated facility for generic operation invocation.

5.3.2 Storage and retrieval via syntactic forms

As in Smalltalk, there are two principal means for modifying the instance variables of an object, syntactic and procedural. Also as in Smalltalk, syntactic forms for variable evaluation and assignment are interpreted as instance variable operations on the value of `self`. In particular:

```
(setq id exp)
```

sometimes has the semantics of:

set-IV-value (self, S, exp)

and

```
id
```

can mean:

get-IV-value (self, S)

where S is the symbol corresponding to the identifier *id*. For example,

```
min
```

would return the value of the `min` instance variable of the current value of `self` and

```
(setq min 17)
```

would replace that value with 17.

For an expression to be interpreted in this way, it must occur in the body of a function defined by a **defmethod** or **defun-method** syntactic forms.[31]

(defmethod ($id_{flavor\text{-}name}$ ...) *lambda-list*
    body)

(defun-method $id_{function\text{-}name}$ $id_{flavor\text{-}name}$ *lambda-list*
    body)

In each of these forms, the instance variable names of the flavor named $id_{flavor\text{-}name}$ are accessible via the variable assignment and evaluation forms.

However, in order to guarantee that the *correct* instance variable operation is accessed or modified, an additional condition is necessary. The value of the variable **self-mapping-table** must be as described in section 3.4.3, i.e., appropriate for (a) the flavor of which the generic

[31] This is not strictly accurate. Interpretation of the evaluation and assignment of a variable V as invocations of instance variable operations will occur in any function which contains a **sys:instance-variables** declaration which includes the name V. Functions defined by **defmethod** and **defun-method** have such declarations invisibly inserted, as do methods created by the method definition algorithms described in chapter 6.

parameter is an instance and (b) a flavor associated with the procedure. For functions defined by **defmethod**, a sufficient condition is that they are invoked through the generic invocation mechanism; i.e., via `send` or `funcall` with an instance of a flavor as the first parameter. For functions defined by **defun-method**, **self** must be an instance of the flavor named by $id_{flavor\text{-}name}$ or a descendant; if not, an error is signalled.[32]

The consequences of "invalid IV name" errors is somewhat different than in Smalltalk. The Zetalisp interpreter will search the runtime stack for a binding of a variable with the erroneous name, which may or may not cause an "undefined variable" error. The Zetalisp compiler will issue a warning for any use of variable which is neither (a)declared as an instance variable, (b) declared as a "special" variable (i.e., one declared to have been introduced by a dynamically enclosing context), or (c) introduced locally. It will actually be compiled as a reference to a "special" variable.

5.3.3 Storage and retrieval via procedure invocation

The abstract **get-IV-value** and **set-IV-value** procedures are directly realized as `symeval-in-instance` and `set-in-instance`. For example,

```
     (symeval-in-instance p 'location)
```

computes the value associated with `'location` in the set of associations modelling p, and

```
     (set-in-instance p 'location 17)
```

associates a new value with p's `location` variable. If the specified symbol is not an `IV-name` of the object, a runtime error is signalled.

[32] What's the point of using **defun-method** procedures rather than ordinary procedures which invoke the procedural versions of the instance variable operations described below? In the current implementation, (a) the former is more efficient than the latter, and (b) the former causes static error checking to be performed, but the latter does not. One reason why efficiency is gained is that, in the current implementation, the appropriate mapping table is computed once per invocation of the **defun-method** rather than once per invocation of **symeval-in-instance** and **set-in-instance** procedures described below. Another is that an optimization can sometimes be performed so that the mapping table can be obtained in constant time rather than in time proportional to the number of ancestors of the type of **self**.

Notice that both error checking and optimization could be performed on invocations of the procedural versions of instance variable operations if the type of **self** and the identity of the instance variable name were manifest. The current implementation of the Zetalisp compiler performs no such optimizations.

5.3.4 Instantiation

The abstract **instantiate** procedure is realized in Zetalisp by the `instantiate-flavor` procedure. Specifically, if `f` is a flavor and `plist` is an object representing a property list,

```
(instantiate-flavor f plist)
```

can create a new instance of *flavor*. The set of associations which model the newly created object is determined by a non-trivial algorithm, which we next describe. The elaborateness of this algorithm is one of the distinctive characteristics of the Zetalisp realization of the instance-variable operation set.

1. A subset of the **instance-IV-names** of a flavor can be specified as "initializable" by an `:initable-instance-variable` clause of the `defflavor` form. The definitions of `point` and `bounded-point` contain such definitions. If an **IV-name** V of is present as a property on `plist`, its associated value is used as the initial value of V for the newly created object.

2. If an **IV-name** V has a "default initial value expression" specified for F or any ancestor, the first such expression found in a search of F's component list is used to provide a value for V. Such defaults can be specified in one of two ways:

   - through a *varspec* in the `defflavor` which defines F. If a *varspec* has the form *(id exp)*, *exp* is the default for the variable *id*. For example, in the following definition of `point`:
     ```
     (defflavor point
             ((location 0))
             ()
        :gettable-instance-variables
        :initable-instance-variables)
     ```
     the variable `location` is given a default initial value expression of `0`.

   - through a `:default-init-plist` clause of `defflavor`, e.g.,
     ```
     (:default-init-plist
        :location 10 :min (compute-min) :max (compute-max))
     ```

   In either case, the associated expression is eval·· ted and the result is used as the initial value for V.[33]

Any **IV-names** not given initial values by the above algorithm are associated with a value denoting an "unbound variable".

There are two special kinds of flavors which are treated specially by `instantiate-flavor`. These are described next.

[33] If both are specified for a given flavor, the *varspec* definition is used.

5.3.4.1 Abstract flavors

If the flavor F is declared to be an "abstract flavor," an error is signalled. Non-instantiatable flavors are useful because they can be a representation-free realization of some abstraction. For example, an "abstract-queue" flavor could have an operation set which contained operations on queues which depended only on generic invocation of abstract queue operations.

Declaring a flavor which embodies such an abstraction as an abstract flavor has two advantages. First, it allows the error of mistakenly creating instances of such flavors to be detected at the time of their creation. Second, since no instances of the type will ever exist, the representation of its generic operation set need not be constructed. The latter reduces the overhead needed for system creation, but of course is immaterial to the program semantics.

5.3.4.2 Mixture flavors

If the flavor F is declared to be a "mixture," a computation is performed on `Plist` to compute a second flavor F2 which will then be instantiated. The algorithm for performing this computation is specified declaratively in a form not described here,[34] but is constrained to compute either F or some descendant. Thus, mixtures allow the `Plist` of an initialization to determine which specialization of a flavor to create.

Notice that such a mechanism would be conceptually superfluous in Smalltalk. This is because the creation operations of metaclasses can perform arbitrary computation to determine the class to be instantiated. For example, if the metaclass of a class F had the following method for **create:**

```
    create: plist
      ^ exp create: plist
```

then the expression:

```
    F create: Nil
```

would create an instance of whatever class was computed by *exp*.

[34]See p. 39 of the "Flavors" section of **Symbolics84**.

There are two other advantages of using mixture flavors in Zetalisp.

- First, all possible flavors which can be instantiated via a mixture flavor F are automatically constructed when F is constructed. This reduces the number of flavor-introduction forms and guarantees that no flavor construction need occur during program execution.

- Second, the descendant flavors do not have to be referred to by name in order to instantiate them. This reduces the number of names which must be known by the programmer.

The Smalltalk approach would share the advantage of eliminating flavor-construction forms and reducing the number of class names which must be known. But classes would have to be created during program execution.

5.3.4.3 Verification of "required" components of flavors

Before creating a new instance of a flavor, `instantiate-flavor` verifies that the flavor has a number of properties. Through additional `defflavor` clauses, flavors can be associated with "required flavors," "required methods," and "required instance variables." For example:

```
(defflavor abstract-monitored-list () ()
  (:required-flavors monitored-object)
  (:required-methods :car :cdr :rplaca :rplacd)
  (:required-instance-variables first last))
```

Before a flavor F is fully defined, it is verified that the following properties hold.

- Each flavor named in a `:required-flavors` clause of F or any ancestor is an ancestor of F.

- Each operation named in a `:required-methods` clause of F or any ancestor is a generic operation of F.

- Each instance variable named in a `:required-methods` clause of F or any ancestor is an instance variable of F.

The latter two conditions are fully explained in the following chapters.

Required methods are useful because it provides a simple form of pre-execution error checking. The collection of required methods allows detection of missing realizations of abstract operations at "compile-time," i.e., before any attempt is made to invoke them. Thus, serves a similar function as a Clu "cluster heading" ([Liskov et al. 79], p. 7).

Required instance variables are useful in the same way as required methods, but at a lower level of abstraction. For example, methods associated with the `abstract-monitored-list` flavor described above can safely assume that the list will be represented using a `first` and `last` instance variable. Of course, such an assumption is useless unless one also knows how the variables are being used, e.g., through an "abstraction function" associated with the type ([Hoare 72, London et al. 78]). Notice that exactly the same limitation applies to any use of type-checking for detecting errors in programs.

The utility of required flavors involves an entirely separate issue. The point of declaring that flavor A is a required flavor of flavor B is to guarantee that all descendants of B have some ancestor which explicitly declared that A should be a parent. This property does not hold if A were declared as a parent of B or as an included-flavor of B. The result is that designers of descendants of B are forced to be more aware of the consequences of including B.

5.3.5 Conventions for initialization

5.3.5.1 *Instantiate-flavor make-instance*, and the *:init* method

As described above, initialization based on a property list parameter is useful. We have already seen one way in which Zetalisp supports this paradigm. The realization of **instantiate** takes a property list and, if any properties match names of initializable instance variables, the associated values are used as initial values for the instance variables.

In order to perform more complex initialization actions, the `:init` operation is conventionally used. The usual pattern of object initialization is for the `:init` operation to be invoked after the newly-created object has undergone the initialization process described above. This style is made more convenient in two ways. First, an optional parameter to `instantiate-flavor` will cause the `:init` operation to be invoked after the object is created. Second, the "easy to call" interface to `instantiate-flavor`, `make-instance`, always invokes the `:init` operation if one is defined.[35] A significant contrast between this approach and that used in Smalltalk is that there is no way for type-specific creation procedures to be invoked before the object is created. For example, it would be impossible to prevent a `bounded-point` object from being created if an inconsistent `min` and `max` values were specified, as was done in figure 2-14 of the Smalltalk example. Thus, the Smalltalk approach of having type-specific procedures *encapsulate* object creation seems more desirable.

5.3.5.2 Verification of the initialization property list

In the discussion of Smalltalk initialization, we mentioned one of the potential disadvantages of initialization via property lists: less powerful error checking. Zetalisp addresses this problem by allowing the designers of types to specify which keywords are *required* to be on the initialization property list and which are *allowed* to be there. These are specified by two more optional clauses of the `defflavor` form: `:allowable-init-keywords` and `:required-init-keywords`.

Before `instantiate-flavor` either returns the newly-created instance or calls the `:init` methods, it verifies that (a) all required keywords of the flavor and its ancestors are present on the property list, and (b) all properties were specified as valid keywords by the flavor or some ancestor. Missing or extra keyword errors can be detected by the Zetalisp compiler, so long as the flavor name and property names are constants.

5.3.6 Other techniques for instance variable operations

Although much less rarely used those described above, there are three other ways in which instance variable operations can be carried out.

Some subset of the `instance-IV-names` of a flavor can be declared as "special instance

[35]`make-instance` is easy to call because it allows the initialization property list to be specified as a `&rest` parameter. E.g.,

```
      (make-instance 'f :a x :b y :c (g z))
```

is identical to

```
      (instantiate-flavor 'f '(nil :a ,x :b ,y :c ,(g z)) t)
```

variables" This is commonly done through the `:special-instance-variables` clause in the defflavor form. For each such name N associated with a flavor F, generic invocation on an instance of F or any descendant will set up the binding stack in a particular way. Specifically:

- the name N will be visible in the current dynamically naming context, and
- references to N will be treated as references to the instance variable named N of the value of `self`.

This approach allows the `set` and `eval` primitives to be used to carry out instance variable operations on `self`. It also allows procedures invoked by methods to manipulate the instance variables of `self` via `setq`, `set`, or `eval`. The only rationale for using special instance variables rather than `set-in-instance` and `symeval-in-instance` is to obtain efficiency. This is because the work necessary to resolve an instance variable to a memory location is done once per generic invocation, rather than once per instance variable reference.

A final technique for performing instance variable operations is through the `:outside-accessible-instance-variables` clause of defflavor. Given alone, it defines procedures which store and retrieve the instance variable with a given name of an instance of any class. For example: the clause:

```
    (:outside-accessible-instance-variables a)
```

defines procedures with the semantics of:

```
    (lambda (x) (symeval-in-instance s 'a))
    (lambda (x) (set-in-instance s 'a))
```

However, in conjunction with the `:ordered-instance-variables` options an additional declaration, the compiler will assume that (a) the procedures are being applied only to instances of the flavor in which the declaration appears, and (b) the relative offsets of the storage containing the given variables which holds at the time of compilation will hold at invocation. Thus, accessing a given instance variable can be done in constant time, at the cost of having to recompile code if the representation of instances of flavors changes. Since the Zetalisp system provides no support for determining when this has occurred, the use of this option is commonly reserved for only the most performance-critical applications.

Finally, the implementation described in [MoonStallmanWeinreb 84] can also cause all instance variables of the current value of self to be bound as special variables via the special form:

```
(with-self-variables-bound body)
```

This allows the benefits of special-variable access to instance variables which are *always* accessed in that way.

5.4 The Star Mesa realization

Instances of traits are allocated regions of storage, and the procedures for accessing this storage use POINTER values to refer to this storage. We first describe some programming conventions needed for this storage to be treated as objects, in the sense of Smalltalk, Zetalisp, and Interlisp.

5.4.1 The relevant programming conventions

Star Mesa Convention 2: Each trait definition designates a record type as its **instance component type.** If R is the instance component type of trait T, all expressions which compute an instance component defined by T should be used only in contexts which require the type R.

In the example of chapter 2, a comment of figure 2-18 states that the value of the identifier ICType in each trait definition module is the instance component type of the trait. For example, the instance component type of trait BoundedPoint is the value of the identifier ICType defined in module BoundedPoint:

```
ICType: TYPE = RECORD [min: REAL, max: REAL];
```

Star Mesa Convention 3: The amount of storage for instances of the instance component type is the ICSize field of the record returned by the initial invocation of the registration procedure of the trait.

For example, records returned by invocations of the registration procedure of module BoundedPoint contain the value of the expression:

```
SIZE [ICType],
```

in their ICSize field. The Mesa pseudo-procedure SIZE takes a Mesa type and returns the amount of storage used to represent instances of that type.

5.4.2 Instance variable names

A summary of the realization in Star Mesa is as follows:

- In Star Mesa, the **local-instance-IV-names** of each trait T consists of the set whose sole member is the name of T.
- The **instance-IV-names** of a trait are, as usual, the union of the **local-instance-IV-names** of the trait and its ancestors.
- The **IV-names** of an object are the **instance-IV-names** of its type.
- There are no operations which change any of the above.

Thus, the principal difference from the Smalltalk and Zetalisp approach is that there is always exactly one local instance variable per type, whose name is the same as the name of the type.

How can more than one named entity be used to represent that aspect of an object modelled by a trait? The approach commonly taken is to use an instance of a Mesa `RECORD` type to represent the value associated with the variable. For example, to allow the trait `BoundedPoint` to contribute entities named `min` and `max` to the representation, it can use a value of the type:

```
    RECORD [min, max: REAL]
```

as the value of its single instance variable.

5.4.3 Storage and retrieval via procedure invocation

As with Smalltalk and Zetalisp, procedures are used far less often than syntactic forms to perform instance variable operations. However, an explanation of the instance variable operations is clearer if the procedural versions are presented first.

The abstract **get-IV-value** is realized as a procedure which we will refer to as `TM.InstanceComponent`. This procedure takes a instance of a trait and a trait (i.e., objects of Mesa type `TM.Object` and `TM.Trait`). It returns an untyped pointer to the Mesa object which is the value of the instance variable. By convention 2, this pointer should be used in a context which is typed as the appropriate instance component type.

For example, the following expression computes the Mesa object which is the value of the **BoundedPoint** instance component of **p**

```
       TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]]
```

If **p** is not an instance of **BoundedPoint** or an ancestor, an error would be signalled. To obtain the **min** component of that object, the following could be used:

```
     bpPtr: BoundedPoint.ICType
       ← TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]]
     bpPtr^.min
```

The abstract **put-IV-value** is represented as an assignment to the object pointed to by the result of a **TM.InstanceComponent** invocation. For example, to replace the **BoundedPoint** instance variable of an object **bhp**, the following would be used:

```
     (TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]])^
       ← BoundedPoint.IC [min: 20, max: 30]
```

To change only the **min** component:

```
     bpPtr: BoundedPoint.ICType
       ← TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]]
     bpPtr^.min
```

Notice that Mesa requires that the result of **TM.InstanceComponent** be assigned to a variable with a record type for the record accessor to be valid. Thus, the expression

```
     (TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]])^.min
```

would not be legal Mesa. Unfortunately, since **TM.InstanceComponent** is declared to return an untyped pointer, it can be assigned to a variable of *any* Mesa type. Thus, the anomalous expression:

```
     bpPtr: RECORD [wrong1: STRING, wrong2: BOOLEAN, wrong3: ARRAY [17]
       OF REAL]
       ← TM.InstanceComponent [bhp, TM.TraitNamed["BoundedPoint"]]
     bpPtr^.min
```

would be perfectly legal and would go undetected.

5.4.4 Storage and retrieval via syntactic forms

Star Mesa includes a syntactic form[36] which is semantically equivalent to invocations of **TM.InstanceComponent**. The form:

```
    TM.InstComp [exp, id]
```

is equivalent to:

```
    TM.InstanceComponent [exp, TM.TraitNamed ["id"]]
```

For example,

```
    TM.InstComp [bhp, BoundedPoint]
```

is equivalent to:

```
    TM.InstanceComponent [bhp, TM.TraitNamed ["BoundedPoint"]]
```

It is easy to see how this form can be applied to the examples of the previous section. E.g.,

```
    bpPtr: BoundedPoint.ICType
      ← TM.InstComp [bhp, BoundedPoint]
    bpPtr^.min
```

All the examples of chapter 2 use this latter technique.

There are two principal benefits of the use of the syntactic forms. The first advantage is obvious: significantly shorter expressions. The second derives from the fact that the identity of the trait used to name the instance variable is syntactically apparent. This can be used to advantage in compiling such forms.

The key idea is that a subset of the traits present in a program are designated as "fixed-offset" traits. If a trait T is a fixed-offset trait, all instances of T and its descendants will contain the storage for the instance variable named by T at the same offset in the storage allocated for the object as a whole. Thus, the invocation of

```
    TM.InstanceComponent [bhp, BoundedPoint]
```

could be replaced by the expression:

```
    bhp + offset
```

[36] In Star Mesa, syntactic forms (i.e., invocation forms whose meaning is a function of the data structure which represents the form rather than the result of evaluating the components of the form) are represented as the **INLINE** procedures. The capability of **INLINE** procedures to make arbitrary transformations of their parameters was added after the publication of [Mitchell et al. 79].

for some constant *offset*. In Star Mesa, this will in fact be done when "production mode" compilation is indicated through specification of a compiler option.

The reason why this optimization is relevant to the semantics of Star Mesa is that it is not always correct. Recall that instances of all traits are represented as Mesa objects of type **TM.Object**. The problem with the transformation is that it is only valid for those instances of **TM.Object** which represent instances of the trait named by *id*. However, all that is known about *exp* is that it has the Mesa type **TM.Object**, and is thus an instance of *some* trait. Since the optimized expression does not check that the trait of *exp* is *id* or some descendant, there is nothing to prevent its application to instances of a trait which does not carry the trait named by *id*. The consequences of such an occurrence are entirely unpredictable.

For example, let us reexamine the code fragment:

```
    bpPtr: BoundedPoint.ICType
      ← TM.InstComp [bhp, BoundedPoint]
    bpPtr^.min
```

under the assumption that **bhp** is bound to an instance of **HistoryPoint**. Suppose further that the object at the offset computed by

```
    TM.InstanceComponent [bhp, BoundedPoint]
```

was the pointer to the **hlist** cons cell used in the representation of **HistoryPoint** objects. Storing a numeric value in that component would cause havoc. But again, such an errors would go undetected by either the Mesa type-checker or the procedures of the Traits extension.

5.4.5 Instantiation

As listed in figure 4-12, the abstract **instantiate** operation is realized by a Mesa procedure which we are referring to as **TM.Allocate**. It takes a trait as its sole parameter and returns an object of type **TM.Object**. This object points to a newly allocated region of storage of the appropriate size for instances of the trait. The size of this storage is the sum of the **ICSize** components returned by the registration procedures of the trait and all ancestors. Thus, there is a sufficient amount of space for the instance components defined by the trait and all ancestors.

The TM.Alloc syntactic form also can be used to allocate an instance of a trait. It shares the same two advantages over the TM.Allocate procedure as those of TM.InstComp over TM.InstanceComponent. Unlike the latter, the use of TM.Alloc can never lead to undetected errors. The appropriate representation for the specified trait will always be created.

5.4.6 Conventions for object initialization

The usual convention for creating an object with appropriate initial values for its instance variables is as follows. First, the TM.Alloc form is used to create the new object. Subsequently, the PartialInitialize procedures associated with the trait and all ancestors are invoked. The order of this invocation is such that the PartialInitialize of each trait is invoked after the PartialInitialize of all ancestors.

To illustrate, here is the Create procedure for TM.TraitNamed ["HistoryPoint"] defined in figure 2-22

```
Create: PROC [initloc: REAL ← 0] RETURNS [Object] =
   {newp: Object ← TM.Alloc [HistoryPoint];
    Point.PartialInitialize [newp, initloc];
    PartialInitialize [newp];
    RETURN [newp]};
```

The Create procedures defined in figures 2-24 and 2-26 use a simple extension of this technique to invoke procedures *before* the object is created, thus allowing errors to be detected before allocation.

Under this initialization convention, each Create and PartialInitialize takes its own set of parameters. As a result, the definer of a trait must know the initialization interface for *each* ancestor trait. The use of a property list for initialization, as built into Zetalisp and as programmable in Smalltalk and Loops, enables this inter-type dependency to be avoided.

5.5 The Loops realization

One of the distinctive aspects of Loops its elaborate treatment of instance variables of objects. As one illustration of the sophistication of the mechanism, the table on page 24 of [BobrowStefik 83] lists twenty-six different operations for storing and retrieving the values of variables associated with objects.

In large part, this is due to the previous work of its designers in knowledge representation systems (e.g., [BobrowWinograd 77a, Stefik 78]). As discussed in chapter 1, instance variables rather than type-specific operations has historically been the dominant abstraction implicit in "frame-based" representation systems.

5.5.1 The primitive instance-variable operations

The conceptualization of **object** must also be made more sophisticated. In Smalltalk, Zetalisp, and Star Mesa, objects were modelled as a function from the **instance-IV-names** of their type to a set of values. The **get-IV-value** and **set-IV-value** operations were defined in terms of this association alone. All manipulations on objects in the language could be defined in terms of these basic operations.

In Loops, the situation is fundamentally different. In order to describe the behavior of the procedures described in [BobrowStefik 83], objects in Loops must be modelled as an arbitrary set of associations between names and values. We will refer to this set of associations as the "object property list" of an object. The five primitive operations on objects and their Loops realizations are listed in figure 5-6.

Thus, the primitive instance variable operations on objects have nothing whatsoever to do with their type.

Create a new object of a specified type with an empty object property list	(Send *class* NewWithValues)]
	`(Send $BHPoint NewWithValues)]`
Extract the value associated with a name	(GetValueHere *object name*)
	(GetValueHere bhp 'bhlist)
Add a new association or replace a value of an existing association	(PutValueOnly *object name newvalue*)
	(PutValueOnly bhp 'bhlist '(InitList)
Remove an existing association	(Send *object* DeleteIV *name*)
	(Send bhp DeleteIV 'bhlist)
Obtain the current list of names	(Send *object* List 'IVs)
	(Send bhp List 'IVs)

AddIV, DeleteIV, and **List** are operations of class Object.

NewWithValues is an operation of class **Class**.

Figure 5-6: Basic instance variable operations in Loops

5.5.2 The common abstraction can still be realized

Despite the above, Loops provides a second set of operations through which it can be made to appear that the IV-names of objects are a *superset* of the **instance-IV-names** of their type. Thus, our common abstraction can still be used to describe instance variable operations in Loops.

This second set of operations involves a new aspect of classes, which we will refer to as their "class-instance property list." Like that of objects, it is an arbitrary set of associations between names and values. The primitive operation for manipulating the class-instance property lists of classes are listed in figure 5-7.

When an class is initially created, its class-instance property list is empty. The **InstanceVariables** clause of the **DefClass** form can be used to associate a desired class-instance property list with a newly-created class. For example, given the following example from 2-16:

Extract the value associated with a name	`(GetClassIV class name)`
	`(GetClassIV $BHPoint 'bhlist)`

Replace the value associated with a name	`(PutClassIV class name newvalue)`
	`(PutClassIV $BHPoint 'bhlist '(InitEntry)`

Add a new association	`(Send class Add 'IV name value)`
	`(Send $BHPoint`
	`     Add 'IV 'bhlist (list 'ALabel)`

Remove an association	`(Send class Delete 'IV name)`
	`(Send $BHPoint Delete 'IV 'bhlist)`

Obtain the current list of names	`(Send class List 'IVs)`
	`(Send $BHPoint List 'IVs)`

Add, Delete, and List are operations of class Class.

Figure 5-7: Class-instance property list operations in Loops

```
(DefClass BHPoint
    ...
    (InstanceVariables
        (bhlist nil)
        (bhtail nil))
    ...)
```

the class-instance property list associated with BHPoint will be:

```
((bhlist nil) (bhtail nil))
```

Using a "defining form" to initialize the class-instance property list makes sense in terms of our common abstraction, since none of the operations which realize the abstraction involve modifying class-instance property lists.

Here is how the class-instance property list can be used to realize our common abstraction. The key idea is to have the realization of **get-IV-value** for an object search the class property lists of the type of the object and all ancestors. Thus, the names and values of the class-instance property lists can appear to be associated with the objects themselves.

Figure 5-3 lists the Loops operations which use this approach. The **local-instance-IV-names** of a type are realized as the names which appear on its class-instance property list.

- The realization of **instantiate** is an invocation of the NewWithValues operation of Class. The second parameter of the invocation defines the object property list of the newly created object.[37] The realization of **local-instance-IV-names** is an invocation of the `List` operation of class `Class` with the symbol **IVs** as a parameter. It computes a list containing the names found on its class-instance property list.

- The realization of **instance-IV-names** is an invocation of the `List!` operation of class `Class` with the symbol **IVs** as a parameter. It computes a list containing the names found on its class-instance property list of the class and all ancestors.

- The **IV-names** of an object is realized as the `List!` operation of class `Class`. It computes a list which represents the union of (a) the **instance-IV-names** of its type and (b) the names found on its object property list.

- The realization of **get-IV-value** is the Loops procedure `GetValueOnly`. If the specified name is not found on the object property list, it uses the ancestor ordering specified in chapter 4 to search the class property lists. If the name is found on one of these lists, the associated value is returned. If not, the current value of the variable `NotSetValue` is returned.

- The realization of **set-IV-value** is the Loops procedure `PutValueOnly`. It modifies the object property list of the specified object, by either adding a new association or replacing the value of an old one.

For example, suppose the ancestor ordering of a class **D** was:

```
(C B A)
```

and the class-instance property lists of each these types were:

```
A: ((x 1) (y 1))
B: ((x 2) (z 2))
C: ((x 3) (w 3))
D: ((y 4) (v 4))
```

Suppose further that the object property list of an object O of type **D** was:

```
((v 5))
```

Then the associations between names and values computed by the realization of **get-IV-value** would be:

```
((v 5) (w 3) (x 3) (y 4) (z 2))
```

After **set-IV-value** (O, 'X, 6) and **set-IV-value** (O, 'V, 6), the property list of O would be:

```
((v 6) (x 6)
```

and the associations computed by `get-IV-value` would be:

```
((v 6) (w 3) (x 6) (y 4) (z 2))
```

[37] There is actually a more primitive version of `NewWithValues` which does not assign a unique identifier to the created object within the current Loops naming environment. The point of using this more primitive version is only to avoid overhead for short-lived objects.

The above realization of the abstraction will be accurate if both the parents list and the class-instance property lists of all ancestors do not change. If they are modified, the values associated with instances can appear to change "behind one's back."

5.5.3 The "active values" abstraction

We have now seen two levels of abstraction at which instances of Loops classes can be described. We now describe a third level, the one which is most commonly used in Loops programs.

The key new capability offered by active values is the ability to execute programmer-specified procedures as part of the realization of **get-IV-value** and **put-IV-value**. A simple example of the utility of this capability is the ability to maintain a graphical display of the current values of selected instance variables. More sophisticated uses of this capability have proven to be of considerable value in building AI systems; [Stefik et al. 83b] contains a highly readable discussion of this point.

The new concept introduced to obtain this third level is that of an "active value." An active value is a data structure which contains three components, referred to in Loops terminology as its "get function," "put function," and "local state." The operations involved in this level are augmentations of the `GetValueOnly` and `PutValueOnly` which take special action if an active value is present in object or class-instance property lists.

The syntactic form:

$\#(exp_{local\text{-}state}\ exp_{get\text{-}fn}\ exp_{put\text{-}fn})$

denotes an active value whose local state, get function, and put function are the values of the corresponding expressions. Active values are instances of Interlisp datatypes, and are not themselves instances of Loops classes.

5.5.3.1 Storage and retrieval using the active value abstraction

The analog of `get-IV-value` is `GetValue`. It behaves identically to `GetValueOnly` up to the point where an object is returned. If this object is an active value, then the object returned depends on its get function. If the get function is nil, then the local state is returned. If not, then the value returned is the result of an invocation of the get function. The parameters of this invocation are:

- the parameters of the invocation of `GetValue`,
- the local state of the active value, and
- the active value itself.[38]

The analog of `put-IV-value` is `PutValue`. The behavior of `(PutValue Obj Var Val)` is as follows:

- If the object property list of `Obj` contains an association for `Var` that is not an active value, an association between `Var` and `val` is added. This is identical to the behavior of `PutValueOnly`.
- If an association is found which is an active value whose put function is `NIL`, the local state of the active value is modified to contain `Val`.
- If an association is found which is an active value with a non-`NIL` put function, `PutValue` terminates with an invocation of the put function of the active value. The parameters are used for this invocation:
  - o the parameters of the invocation of `PutValue`
  - o the local state of the active value, and
  - o the active value itself.
- Otherwise, the class-instance property lists of the class of `Obj` and all ancestors are searched in canonical order for an association for `Var`. If such an association is found *and* its value is an active value AV, then two actions are taken.
  1. An association between `Var` and a copy of AV is added to the object property list of `Obj`. This copy is a different active-value object with identical local

[38] Actually, one additional parameter is passed. This parameter tells whether the active value is associated with a class variable, method, or class property list. These other concepts are described subsequently.

state, get function, and put function.[39]

2. The put function of the newly created active value is called with the same parameters as above.

An example which illustrates the above-described mechanism will be given shortly.

5.5.3.2 Nested active values

Active values allow procedures to be invoked when storing into or retrieving from instance variables. It may sometimes be desirable to invoke more than one procedure when an instance variable is accessed. As a paradigm example, it may be desired to invoke a display-updating procedure whenever the value of a variable changes, and a breakpoint-causing procedure whenever values which satisfy a certain predicate are stored in the variable.

The standard Loops programming technique for achieving this behavior is to store active values as the local state of other active values; i.e., to have "nested" active values. Then the accessing functions of the active value can test if the local state is a second active value. If it is, then the access functions of the first active value terminate with an invocation of the access functions of the second. This process can be repeated to invoke the access functions of active values at arbitrary depth.

Two new procedures are used to support this behavior, `GetLocalState` and `PutLocalState`. Invocations of `GetLocalState` and `PutLocalState` take the same parameters as the get and put functions of active values. If the local state contains an active value **AV**, the get or put function of **AV** is called with the standard parameters. If not, `GetLocalState` simply returns the local state and `PutLocalState` replaces the old local state with the new value.

By convention, get functions and put functions should always use these procedures to access

[39] An exception: if the local state of the active value is the atom **Shared**, then the copying is suppressed. In that case, **AV** itself is placed in the object property list.

local state. If this convention is followed, then storing active values in local state will always result in the associated accessor procedures being invoked on storage and retrieval.

5.5.3.3 Syntactic forms for active value operations

For completeness, we note that the `GetValue` and `PutValue` functions can be invoked with a number of syntactic forms.

- `(@` *exp* `:`*id*`)` means `(GetValue` *exp* `'`*id*`)`
- `(@←` *exp* `:`*id* exp_2`)` means `(PutValue` *exp* `'`*id* exp_2`)`
- `(@ :`*id*`)` means `(GetValue self '`*id*`)`
- `(@← :`*id* exp_2`)` means `(PutValue self'`*id* exp_2`)`

Thus, using Loops forms such as `(@ :min)` and `(@← :max 100)` to invoke instance variable operations is the analogue to using variable assignment and evaluation to invoke such operations in Smalltalk and Zetalisp. The principal difference in the two approaches is that the Loops form is solely for instance variable access and is never used for manipulating variables of procedures.

5.5.3.4 An example of using active values

Here is a simple example to illustrate how active values are used. Suppose we wish to maintain a count of all accesses and modifications of the location of a particular point. Furthermore, before the location is changed, we want a separate procedure to be invoked with the object, the instance variable name, the old value, and the new value. Then the active value associated with the `loc` instance variable of the point would have the following structure:

```
#( #( value
      NIL
      NoticeUpdate)
   CountAccess
   CountUpdate )
```

where *value* is the number which represents the location.

The function for monitoring updates could be:

```
(DEFINEQ
    (NoticeUpdate (self varName newValue unused activeValue)
      (PROG (
  (PutLocalState activeValue newValue self varName))

  )
```

The two functions for maintaining the access and update counts could be defined as follows:

```
  (DEFINEQ

    (CountAccess (self varName localState unused activeValue)
      (@← :accessCount (ADD1 (@ :accessCount)))
      (GetLocalState activeValue self varName))

    (CountUpdate (self varName newValue unused activeValue)
      (@← :updateCount (ADD1 (@ :updateCount)))
      (PutLocalState activeValue newValue self varName))

  )
```

For the above code to be meaningful the initial value of the `accessCount` and `updateCount` variables would have to be set to a numeric value (presumably zero) at the time the active value is installed.

To illustrate the effect of active values on the class property list, let us modify the above scenario by assuming that the structure:

```
  #( #( 0
        NIL
        NoticeUpdate)
     CountAccess
     CountUpdate )
```

is associated with `loc` on the class-instance property list of `Point`. As a result, the first time the invocation:

```
  (PutValue p 'loc value)
```

is executed for a given point p, the above active value structure would be copied into the object property list of p.

5.5.3.5 Instantiation using the active values abstraction

A non-primitive procedure for creating new objects is available which supports the use of active values. This procedure, invocable as the `New` operation of `Class`, has one required parameter, a class. After creating the new object, it searches the class-instance property lists of all ancestors of the class for some name **N** which is associated with an active values. If the get function of any of these is the atom `AtCreation`, non-standard action is taken.

- If the local state is an atom, it is interpreted as a function to call with the newly created object and the name **N** as parameters. The result of this invocation is associated with **N** in the object property list of the newly created class.
- If the local state is a list, it is evaluated in an environment in which the variables **self**, **varName** are bound to the newly created object and **N**. The result of the evaluation is associated with **N** in the object property list of the newly created class.

Here is an example taken from [BobrowStefik 83]. Suppose the active value:

```
    (# (Date) AtCreation NIL)
```

was associated with the name **date** in the class-instance property list of `Point`. Then the invocation

```
    (Send $Point New)
```

would cause an association between **date** and the current date to be placed in the object property list of the newly created instance of `$Point`.

A related facility is provided by the `GetValue` procedure. If it finds an active value whose get function is `AtCreation` or `FirstFetch` the same actions are taken as described above.

Finally, we note that Loops predefines a `Template` metaclass which facilitates the definition of instances of types whose creation involves the creation of a collection of mutually referential objects. The paradigm domain is "device" which contain "components" which are other "devices." See chapter 8 of [BobrowStefik 83] for details.

5.5.4 An assessment of active values

5.5.4.1 Active values and the common abstraction

How does the use of active values relate to the abstraction used to describe our other three languages? [BobrowStefik 83] states that:

> [The] idea of functional composition for nested active values is most appropriate when the order of composition does not matter.... [A]ctive values work most simply when they interface between independent processes using simple functional composition. Any more sophisticated control is seen as overloading the active value

mechanism.[40]

If the principle of order-independence holds, then it must be the case that get functions always return the value obtained from their local state and put functions will always store the new value passed to it. If this design principle is followed, then the use of active values still satisfies the abstraction specified above.

5.5.4.2 Two shortcomings of active values and how to surmount them

One obvious problem with the use of active values is that the programmer must be aware of the representation used to implement the abstraction represented by the class of the object. Furthermore, changing the representation can cause the intended functionality of active values to not be realized. For example, consider an active value which monitors changes to the `htail` and `bhtail` instance variables of instances of `BoundedPoint`. If the implementation of the history-keeping operations was changed to sometimes access the last cell through the `hlist` and `bhlist` variables, the appropriate monitoring behavior would not be realized. This representation-dependence is clearly undesirable at the "interface between independent processes."

A second problem is that it is awkward to use active values to track changes to objects associated with method invocations rather than instance variable operations. For example, suppose our scenario included two-dimensional points, represented using **x** and **y** instance variables, and a `Move` operation which specified new x and y coordinates. Suppose further that the list returned by the `History` operation should contain one entry for each invocation of `Move`. If we used active values to monitor x and y, it would be awkward to arrange that a single entry be made on the history list for the separate modifications to x and y entailed by the `Move`. For more complex operations on more complex kinds of objects (e.g., updating the "state of the world" resulting from a simulated action of a problem solver), the low level of granularity can well be intolerable.

[40] [BobrowStefik 83], p. 26

It is easy to visualize an enhancement to Loops which would solve both of these problems. The key idea is to associate each object with an "object operation set" in addition to its "object property list." Then the generic invocation mechanism can be modified to search the object operation set before examining the operation set of the object's type. For example, to cause the location of a particular point p to be monitored, a definition for Move would be given in the object operation set of p.

This enhancement allows objects to override the methods defined by their types. What is still needed is a means for these object-specific methods to be nested, so more than one process can monitor invocations of the same operation.

The idea here is to associate each object-specific method with a "local continuation," in the same way that each active value has a "local state." Then, by convention, each object-specific method should either invoke the method found in its local continuation, or, if the local continuation is empty, invoke the method associated with the object's type.

Of course, it is not the case that object-specific methods can replace active values. A single active values allows all changes to an instance variable to be monitored, regardless of the number of methods which can potentially cause such changes. Active values and object-specific methods are complementary facilities for monitoring program execution.

5.6 Two concepts related to instance variables

5.6.1 Class variables

The concept of a "class variable" is used in two of our four languages. All classes in Smalltalk and Loops can be viewed as being associated with a "class property list", which associates "class variables" with values.[41] Both languages have procedures which allow associations to be

[41]Furthermore, Smalltalk-80 and Smalltalk-82 allows any number of "dictionaries" to be associated with a type. The poolDictionaries: parameter of the ...subclass:... operations defines the initial collection of these other "property lists".

added and removed, for values to be replaced, and for values to be extracted. Syntactic forms similar to those used for instance variable operations can be used as well.

Both Smalltalk and Loops have different augmentations of the above-described approach. Smalltalk also allows a number of other "property list" objects (realized as instances of class `Dictionary` ([GoldbergRobson 83], p. 148)) to be associated with a class. Loops provides procedures which treat active values in class variables in the same way as active values in instance variables.

The principal use to which class variables have been put is to hold information which is shared by the procedures which implement the abstraction represented by a class. A simple example would be a count of the number of instances created for a class used to create unique identifiers. As such, they perform the same function as the "private" or "own" variables of module-based languages (e.g., [Lampson et al. 77, LauerSatterthwaite 79, Liskov et al. 79]). Thus, they would be realized as private variables of Star Mesa trait modules.

In Zetalisp, the functions served by class variables are often realized through the property list associated with the flavor name. This is less desirable from the perspective of encapsulation, since naming conventions must be introduced to prevent conflicting uses of the same property names.

5.6.2 Properties of variables

In Loops, a property list is associated with each instance variable of each object, each class, each name on the class-instance property list of the class, each class variable of a class, and each local method of a class (see chapter 7). All the variants of `GetValue` and `PutValue` described above can be applied to the values associated with names in these property list. For example,

```
    (GetValue bhp 'bhlist 'timestamp)
```

either returns the `timestamp` property associated with the `bhlist` instance variable of `bhp` or, if it is an active value, invokes its get function. Notice that this object can be different from that computed by:

```
    (GetValue bhp2 'bhlist 'timestamp)
```

so long as **bhp1** and **bhp2** are not coreferential.

There have been two principal uses of these properties. First, they provide a convenient means for self-documentation of classes and distinguished system objects. Second, for applications in which the instance variables are the subject of inference (recall the discussion of representation systems in chapter 1), the properties can be used to hold information relevant to the inference mechanism.

6. Algorithms for generic method computation

6.1 The assumption of a local operation set

Chapter 3 described the mechanism for generic procedure invocation. We now address the question of how the generic operation set is constructed for our subject types.

In describing the algorithm for generic operation set computation, we will use the concept of the **local operation set** of a type. The local operation set, like the generic operation set, is a set of associations between names and procedures. In each of our four languages, the generic operation set of a type is derived from the local operation sets of the type and its ancestors.

In this chapter, we will see that the algorithms for generic operation set computation differ considerably in our four languages. In chapter 7, we will see that details of how a local operation set is associated with a type are also quite different. The point of introducing the "local operation set" abstraction is that *these two sets of differences are independent.* In other words, any of the techniques used to associate a type with a local operation set could be used with any of the algorithms for generic operation set computation. This is an important conclusion, since the differences in the latter can easily obscure comparisons of the former.

The key conclusion we will make in this chapter is that there is a significant difference in power between Zetalisp's approach to generic method computation and that used in the other three languages. The two basic differences are as follows:

- In Smalltalk, Loops, and Star Mesa, the methods for generic operations are always *selected* from the collection of local methods of the type and its ancestors. But in Zetalisp, there is a second possibility: generic methods can be *synthesized* from the local methods of the type and its ancestors. We will see that this simple kind of "automatic programming" can significantly reduce the work required to define new types.

- In Smalltalk, Loops, Star Mesa, the same algorithm is used for computing every the generic method for every operation of every type. But in Zetalisp, a number of algorithms are available, and different algorithms can be used for different operations.

The organization of this chapter is as follows. Each section will describe generic operation set computation in one of our four languages. The examples of chapter 2 are used illustrate the discussion. The local operation sets will be identified in the discussion; chapter 7 describes the general rules for associating local operation sets with types.

6.2 The Smalltalk-80 and Smalltalk-82 algorithms

6.2.1 The local operation set of the example

The example scenario of figures 2-11 through 2-17 uses the `compileAndStore:` operation of class `Class`. This operation, not defined in the Smalltalk system, allows a single expression to create a compiled method and store it into the method dictionary of a class. Its invocation on a class C and a string S does three things:

1. It obtains an instance of class `CompiledMethod` corresponding to S.
2. It extracts the operation name from S.
3. It stores an association between the operation name and the compiled method in the method dictionary of C.

For example,

```
    Point compileAndStore: 'location ^ loc'.
```

creates an instance of class `CompiledMethod` from the string `'location ^ loc'` and installs this procedure as the definition of `location` in the local operation set of `Point`. Thus, this procedure will subsequently be the result of:

```
    Point compiledMethodAt: #location
```

and used as the method for generic invocations on instances of `Point` such as:

```
    p location
```

or

```
    p perform: #location
```

An implementation of the `compileAndStore` operation is given in figure 2-11. The local operation names of the classes of the scenario of figures 2-12 to 2-17 are listed in figure 6-1. We note here that Smalltalk provides a `compile:classified:` method applicable to all classes; that is described in chapter 7.

Class	Local Operation Name

The class of `Point`	`create:`
	`new`

`Point`	`initialize:`
	`location`
	`move:`
	`display:`

The class of `HistoryPoint`	*The local oepration set is empty.*

`HistoryPoint`	`initialize:`
	`partialInitialize:`
	`move:`
	`partialMove:`
	`history`
	`display:`
	`partialDisplay:`

The class of `BoundedPoint`	`create:`
	`partialCreate:`

`BoundedPoint`	`initialize:`
	`partialInitialize:`
	`move:`
	`partialMove:`
	`min`
	`max`
	`setmin:`
	`setmax:`
	`display:`
	`partialDisplay:`

The class of `BHPoint`	`create:`

`BHPoint`	`initialize:`
	`partialInitialize:`
	`move:`
	`setmin:`
	`partialSetmin:`
	`setmax:`
	`partialSetmax:`
	`boundsHistory`
	`display:`
	`partialDisplay:`

Figure 6-1: Local operation names in the Smalltalk example

Notice that the local operation set of the class of `HistoryPoint` is empty. This existence of this class is due to Smalltalk's principle, discussed in chapter 4, of having a one-one relationship between the class `Object` and its descendants and the descendants of class `Class`. Our example illustrates that this generality can result in metaclasses with no associated functionality.

6.2.2 The algorithm for generic method computation

Given the above, we can now describe the algorithm for computing the generic operation set of any class in Smalltalk-80 or Smalltalk-82. This algorithm is, in the strictest sense, a function of the local operation sets of the class and its ancestors.

- The names of the generic operations of a class C are the union of the local operation names of C and its ancestors.
- The method for a particular operation name O is computed by the following procedure:
    1. Is O a local operation name of C? If so, then the local method for O is used as the method for O in the generic operation set of C.
    2. Let C^* be the parents[42] of C whose generic operation sets contain a method for O.
    3. Does the generic operation set of each member of C^* define the same method for O? If so, than this method is taken as the method for O in the generic operation set of C.
    4. Otherwise, the method for O in the generic operation set of C is a procedure which invokes the `error:` operation on the generic parameter.

The result of this algorithm for method definition may be summarized as follows.

- The application of the algorithm to classes with a single parent parent results in either direct method specification or method inheritance. If a method for the operation name is defined in the local operation set of a class, that method is used. Otherwise, the generic method of the parent class is used.
- The application of the algorithm classes with more than one parent results in either direct method specification, method inheritance, or an error-signalling method. If a method for the operation name is defined in the local operation set of a class, that method is used. Otherwise, if the generic operation sets of all parent classes define the same method for the operation, that method is used. If neither of these conditions hold, an error-signalling method is used.

Notice that all Smalltalk-80 classes fall into the first category.

To illustrate, figure 6-2 outlines the generic operation set of the eight classes used in the scenario of chapter 2, omitting the generic operations inherited from `Object` and `Class`. For

[42] Recall that we are using the term "parent" to refer to "direct ancestors", and "ancestor" to refer to the transitive closure of the "parent" relation.

each class, the names of the generic operations are given. If the method is inherited from some other class, that class is listed. If an error-signalling method is constructed, that is indicated as well.

6.3 The Loops algorithm

6.3.1 The local operation sets of the example

Figures 2-6 through 2-10 uses `Methods` clauses of `DefClass` and the `DEFINEQ` define the desired local operation sets. Figure 6-3 lists the local method names which are defined for each class in the scenario. In the interest of clarity, the local method symbols for each local operation name of each class (i.e., the symbols which appear in the `Method` clause of the `DefClass` invocations) were synthesized from the name of the class and the name of the operation. For example, the local method symbol for operation name `LocalInitialize` of class `$Point` is `Point.LocalInitialize`.

In comparison with the Smalltalk local operation set, notice that there is no `BHPointClass` and that the the local operation set of `BHPoint` contains no method for `Display`. The reason why these operation are not needed to obtain the desired functionality is described shortly.

6.3.2 The algorithm for generic method computation

A description of generic operation set computation for Loops classes uses the component-type ordering defined in chapter 4.

The generic operation set of a Loops class is computed as follows:

- As in Smalltalk, the names of the generic operations of a class C are the union of the local operation names of C and its ancestors.
- The method computation algorithm for operation name O of class C is as follows:
    1. Let C' be the first member of the component ordering of C whose local operation set contains a method for O. Note that such a class must exist, since the operation identification algorithm guarantees that O is a member of the union of the local operation names of the component types.

Class	Operation	Inherited from:
The class of Point	create:	
	new	
Point	initialize:	
	location	
	move:	
	display:	
The class of HistoryPoint	create:	*The class of* Point
	new	*The class of* Point
HistoryPoint	initialize:	
	partialInitialize:	
	location	Point
	move:	
	partialMove:	
	history	
	display:	
	partialDisplay:	
The class of BoundedPoint	create:	
	partialCreate:	
	new	*The class of* Point
BoundedPoint	initialize:	
	partialInitialize:	
	location	Point
	move:	
	partialMove:	
	min	
	max	
	setmin:	
	setmax:	
	display:	
	partialDisplay:	
The class of BHPoint	create:	
	partialCreate:	*The class of* BoundedPoint
	new	*The class of* Point
BHPoint	initialize:	
	partialInitialize:	
	location	Point
	move:	
	partialMove:	*
	display:	
	partialDisplay:	BoundedPoint
	min	BoundedPoint
	max	BoundedPoint
	setmin:	
	partialSetmin:	
	setmax:	
	partialSetmax:	
	history	HistoryPoint
	boundsHistory	

* Error signaller constructed, due to conflict between HistoryPoint and BoundedPoint.

Figure 6-2: The generic operation set of the Smalltalk example

Class	Local Operation Name
PointClass	Create New
Point	Initialize Location Move Display
HistoryPoint	Initialize PartialInitialize Move PartialMove History Display PartialDisplay
BoundedPointClass	Create PartialCreate
BoundedPoint	Initialize PartialInitialize Move PartialMove Min Max Setmin Setmax Display PartialDisplay
BHPoint	Initialize PartialInitialize Move Setmin PartialSetmin Setmax PartialSetmax BoundsHistory PartialDisplay

Figure 6-3: The local operation set of the Loops example

2. The method for O in the generic operation set of C is the method for O in the local operation set of C'.

To illustrate, consider the `Display` operation of `BHPoint`. Recall from chapter 4 that the component ordering for `$BHPoint` is:

```
$BHPoint, $BoundedPoint, $HistoryPoint, $Point, $Object
```

The first class in this list whose local operation set contains contains a method for `Display` is `BoundedPoint`. Thus, the generic method for `Display` of `BHPoint` is `BoundedPoint.Display`.

This algorithm for method computation gives results identical to Smalltalk's under two conditions:

- the local operation set contains a method for the operation, or
- the local operation set contains no relevant method but all parents for which the operation is defined have the same generic method.

Otherwise, the Loops algorithm will result in method inheritance from some ancestor type, while the Smalltalk algorithm will "fail," i.e., use an error-signalling procedure as the method. In the example of chapter 2, the only operation of `BHPoint` which fails in this category is the `Display` operation whose derivation is described above.

Furthermore, when inheritance does occur, the construction of the component ordering list of chapter 4 guarantees that a "maximally specific" method is always chosen. More precisely, if a unique ancestor exists which both (a) defines a method for the operation, and (b) is a descendant of all other ancestors which define a method for the operation, then the method of that unique ancestor will be chosen for the type.

Why is it useful for a method computation algorithm to have this property? Suppose we assume that the program is designed using the principle that the specifications of each operation of a type is a specialization of the specification of the corresponding operation of all ancestors of the type. If so, then if an ancestor which satisfies (a) and (b) exists, its associated method is guaranteed to satisfy the specifications of all other ancestors for which the operation is defined.

As a corollary, if a method computation algorithm selects a method defined by an ancestor which is *not* a descendant of all other ancestors which define a method, there is no reason to believe that the selected method will satisfy the behavioral specifications associated with all other ancestors. For example, since (a) both **bounded-point** and **history-point** defined a method for **move**, and (b) neither is a descendant of the other, there is no reason to believe that the **move** methods of either of those would be appropriate for **bh-point**. It seems difficult to argue with the proposition that such choices should, at the very least, be brought to the programmer's attention. However, none of the algorithms in our four languages singles out this particular kind of conflicting inheritance.

Figure 6-4 summarizes the generic operation sets of the classes defined in figures 2-5 through 2-10. Aside from Smalltalk's extra metaclasses, the principal difference between this collection of operation sets and those of the Smalltalk implementation is in the Display operation of class BHPoint.

In the Smalltalk example, the display: operation of BHPoint couldn't be inherited for BHPoint, because the method for HistoryPoint was a different CompiledMethod than that of BoundedPoint. Thus, leaving it out of the local operation set would have resulted in an error-signalling method for display: of BHPoint. On the other hand, unresolved conflict in Loops always results in one of the alternatives being chosen. Since BoundedPoint is a subclass of Point, the local method of the former is chosen over that of the latter to implement the Display operation of BHPoint.

Which of the alternatives is "right?" On one hand, an algorithm which resolves conflicting inheritance cannot make any fewer correct choices than an algorithm which does not. But it cannot make any fewer incorrect choices, either. A reasonable compromise would be for the algorithm to make the choice, but notify the programmer of the action taken. A refinement of this approach would separate the conflicts where a "maximally specific" method was chosen from those in which the choice of a method was based on the local ordering of the parents of a type.

Class	Operation	Inherited from
PointClass	Create	
	New	
Point	Initialize	
	Location	
	Move	
	Display	
HistoryPoint	Initialize	
	PartialInitialize	
	Location	Point
	Move	
	PartialMove	
	History	
	Display	
	PartialDisplay	
BoundedPointClass	Create	
	PartialCreate	
	New	PointClass
BoundedPoint	Initialize	
	PartialInitialize	
	Location	Point
	Move	
	PartialMove	
	Min	
	Max	
	Setmin	
	Setmax	
	Display	
	PartialDisplay	
BHPoint	Initialize	
	PartialInitialize	
	Location	Point
	Move	
	PartialMove	BoundedPoint
	Min	BoundedPoint
	Max	BoundedPoint
	Setmin	
	PartialSetmin	
	Setmax	
	PartialSetmax	
	History	HistoryPoint
	BoundsHistory	
	Display	BoundedPoint
	PartialDisplay	BoundedPoint

Figure 6-4: The generic operation set of the Loops example

6.4 The Star Mesa algorithm

6.4.1 The local operation set of the example

Trait	Local Operation Name	Local Method
Point	Point.Location	Point.LocationImpl
	Point.Move	Point.MoveImpl
	Point.Display	Point.DisplayImpl
HistoryPoint	Point.Move	HistoryPoint.MoveImpl
	HistoryPoint.History	HistoryPoint.HistoryImpl
	Point.Display	HistoryPoint.DisplayImpl
BoundedPoint	Point.Move	BoundedPoint.MoveImpl
	BoundedPoint.Min	BoundedPoint.MinImpl
	BoundedPoint.Max	BoundedPoint.MaxImpl
	BoundedPoint.Setmin	BoundedPoint.SetminImpl
	BoundedPoint.Setmax	BoundedPoint.SetmaxImpl
	Point.Display	BoundedPoint.DisplayImpl
BHPoint	Point.Move	BHPoint.MoveImpl
	BoundedPoint.Setmin	BHPoint.SetminImpl
	BoundedPoint.Setmax	BHPoint.SetmaxImpl
	BHPoint.BoundsHistory	BHPoint.BoundsHistoryImpl
	Point.Display	BHPoint.DisplayImpl

Figure 6-5: The local operation sets of the Star Mesa example

The local operation sets of our four example traits are as described in figure 6-5. In contrast with the Smalltalk and Loops local operation sets, no `partial...` operations are defined. In the Star Mesa implementation, the "partial" procedures are defined as non-generic procedures of trait definition modules. The rationale for using this approach is that these procedures are not actually invoked generically. In the Smalltalk implementation, local operation sets were the sole means of procedure encapsulation, and these procedures had "nowhere else to go." In the Loops example, it is sometimes possible to take advantage of their presence in the local operation set by using the `AllLocalMethods` procedure defined in chapter 7 to eliminate enumeration of ancestor types.

6.4.2 Programming conventions for trait initialization procedures

Recall from chapter 4 that each trait was associated with a trait initialization procedure. The following conventions involving this procedure is relevant to the computation of the generic operation set of a trait.

Star Mesa Convention 4: Each trait definition designates a procedure as its "local initialization procedure". This procedure should take a single parameter, a trait.

In the scenario of chapter 2, the comment in figure 2-18 states that the procedure bound to the identifier **LocalInitializeTrait** in each trait definition module is designated as the local initialization procedure of the defined trait.

Star Mesa Convention 5: The trait initialization procedure of a trait T should consist solely of a sequence of invocations of the local trait initialization procedures of the trait and all ancestor traits. The order of this invocation should be such that if T_1 is an ancestor of T_2, the local initialization procedure of the former will be called before that of the latter.

For example, consider the initialization procedure for **BHPoint** given in figure 2-26:

```
InitializeTrait: PROC [] = PRIVATE
   {Point.LocalInitializeTrait [TM.TraitNamed ["BHPoint"]];
    BoundedPoint.LocalInitializeTrait [TM.TraitNamed ["BHPoint"]];
    HistoryPoint.LocalInitializeTrait [TM.TraitNamed ["BHPoint"]];
    BHPoint.LocalInitializeTrait [TM.TraitNamed ["BHPoint"]]};
```

This calls the local trait initialization procedures of **Point**, **BoundedPoint**, **HistoryPoint**, and **BHPoint**, in that order. Note that the ordering constraint would still be satisfied if the invocation of **HistoryPoint.LocalInitializeTrait** preceded that of **BoundedPoint.LocalInitializeTrait**, but that no other ordering is valid. Note also that as a consequence of this and the previous convention, the local initialization of a trait T will be invoked by the initialization procedures of T and all descendants.

6.4.3 The algorithm for generic method computation

The generic operation set of a Star Mesa trait T can now be defined as follows:

- The names of the generic operations of a T are the union of the generic operation procedures designated by the module defining T and the modules defining all ancestors of T.

- The method associated with generic operation O is the procedure invoked by O when the first parameter is of type T. If all the conventions defined above have been followed, method computation for operation O of trait T produces results identical to the following algorithm:

   1. Let T^* be an ordering of T and all ancestors such that T_1 precedes T_2 iff the invocation of the local trait initialization procedure of T_1 precedes the

invocation of the local trait initialization procedure of T_2 in the trait initialization procedure of T.

2. Let T' be the *last* member of T^* whose local operation set contains a method for O.

3. The method for O in the generic operation set of T is the method for O in the local operation set of T'.

For example, consider the application of the algorithm to the `Point.Display` operation of `BHPoint`. The trait initialization procedure of `BHPoint` invokes local trait initialization procedures in the following order:

```
    Point, BoundedPoint, HistoryPoint, BHPoint
```

From figure 6-5, we can see that each of these traits has a local method for `Point.Display`. Since the last such trait is `BHPoint` itself, the local method of `BHPoint` is chosen.

The result of this algorithm is similar to that of Loops. The principal difference is the way in which the ordered list defined by step (2) is constructed. In Loops, the order is fixed by the parent/child relation between classes and the order of the lists defining the parents of each class. In Star Mesa, the order depends on the definition of the local trait initialization procedure. By convention 5, any order which satisfies the parent/child relation between traits is acceptable. Thus, the property of choosing the "maximally specific method" holds for inheritance in the traits paradigm.

The generic operation sets of each of the four types are summarized in figure 6-6. In comparison with the Smalltalk algorithm, method computation in Star Mesa resolves conflicting inheritance rather than producing error-signalling methods. In the view of the Star Mesa designers, it is a "design error" for a trait to have no local methods for an operation if different parents have different generic methods (e.g., [Curry et al. 82], p. 6). However, no indication was given to the programmer if this principle of programming methodology was violated.[43]

[43] In personal communication, Curry reports that an experimental version of Star Mesa did incorporate a check for unresolved conflicting inheritance. The technique used was to pass a copy of the trait storage to each local trait initialization procedure. The procedure's effects, and hence the local operation set of the trait, could be determined by comparing the data structure passed to the procedure with the data structure from which the copy was made.

Class	Operation	Inherited from:
Point	Point.Location	
	Point.Move	
	Point.Display	
HistoryPoint	Point.Location	Point
	Point.Move	
	HistoryPoint.History	
	Point.Display	
BoundedPoint	Point.Location	Point
	Point.Move	
	BoundedPoint.Min	
	BoundedPoint.Max	
	BoundedPoint.Setmin	
	BoundedPoint.Setmax	
	Point.Display	
BHPoint	Point.Location	Point
	Point.Move	
	BoundedPoint.Min	BoundedPoint
	BoundedPoint.Max	BoundedPoint
	BoundedPoint.Setmin	
	BoundedPoint.Setmax	
	HistoryPoint.History	HistoryPoint
	BHPoint.BoundsHistory	BoundedPoint
	Point.Display	

Figure 6-6: The generic operation sets of the Star Mesa example

Finally, all of the above is predicated on a large number of conventions being satisfied. In the absence of mechanical assistance for detecting their violation, a considerable burden is placed on Star Mesa programmers. As alluded to in [CurryAyers 83], this burden may well be heavy enough to negate the advantages offered by the method definition algorithm itself.

6.5 The Zetalisp algorithms

Method computation in Zetalisp is significantly different from that of our three other languages. Like the other languages, the generic operation names of a type are the union of the local operation names of the type and its ancestors. But the method for a given generic operation is computed by a user-specified procedure, rather than an algorithm built into the type system. As a result, there is very little which can be said about method computation in general. In particular, it is not necessary that an algorithm have the consistent inheritance property or that it involve the local operation sets of the flavor and its ancestors.

Let us emphasize the contrast between Zetalisp and the other three languages. In Smalltalk, Loops, and Star Mesa, the algorithms for generic method computation have the following property:

> The generic method for operation O of type T is always the local method of O for T or an ancestor.

For example, the generic method for an arbitrary operation op of `BHPoint` must be the local method for op for `Point`, `BoundedPoint`, `HistoryPoint`, or `BHPoint`.

In Zetalisp, the generic method for operation O of type T is the procedure produced by the invocation of a user-specified method computation procedure to two parameters: the flavor and the name of the operation.[44] The method returned by this invocation need not be a local method of T or any ancestor. In fact, it is common for these algorithms to construct entirely new procedures to be used as methods.

However, in the experience with Zetalisp to date, method combination is almost always accomplished via system-defined method combination algorithms rather than those created by users. A principal reason for this is that the definition of new method combination procedures requires information not contained in [Symbolics 84] or [MoonStallmanWeinreb 84]; users are advised to "see the code." It is much easier for users to structure their programs to fit the existing algorithms than to learn the necessary details to create their own algorithms.

The organization of this discussion is as follows. After identifying the local operation sets of our example, we describe Zetalisp's default method computation algorithm: "daemon" method combination. The following sections describe the other twelve method combination types documented in [Symbolics 84].

[44] In the actual implementation, the second parameter is the set of local operation names which are relevant to the name of the operation.

6.5.1 Terminological preliminaries

In describing the Zetalisp method computation algorithms, we will often refer to the "components list" of a flavor. The components list of a flavor F is a list whose `car` is F and whose `cdr` is a list of the ancestors of F ordered as described in chapter 4. For example, the components list of `bh-point` would be:

```
    bh-point, bounded-point, point, history-point, si:vanilla-flavor
```

Unlike Smalltalk, Loops, or Star Mesa, local operation names in Zetalisp are *lists* of symbols rather than symbols. For examples, `(:move)` and `(:move :after)` are valid local operation names in Zetalisp, but `:move` is not.

Each local operation name will be associated with a "relevant generic operation name". For single-element local names, the relevant generic operation name is the sole element of the name. For multiple-element names, the relevant generic operation name is the second element. For example, the relevant generic operation name for both `(:move)` and `(:after :move)` is `:move`.

Each local operation name will also be associated with a "method type." The method type is result of *removing* the relevant generic operation name from the local operation name. For example, the method type of `(:after :move)` is `(:after)` and the method type of `(:move)` is `nil`.

Local operation names whose method type is `nil` are commonly referred to as "untyped" methods; that terminology will be used below. Furthermore, for operations whose type is a single element list, we will usually drop the parentheses when referring to the method type. Thus, the method type of `(:after :move)` may be referred to as `:move`.

6.5.2 The local operation set of the example

In the examples of chapter 2, each **defmethod** form defines a local method of some type.

Definitions of the form:

```
(defmethod (id_type id ... id) lambda-list
  body)
```

associates a method for the operation

```
(id ... id)
```

in the local operation set of the flavor named id_{type}. The procedures defined by such a form can be computed by the syntactic form:

```
#'(:method id_type id ... id)
```

or the equivalent expression:

```
(fdefinition (list :method exp_type exp ... exp))
```

For example, the following definition from figure 2-1:

```
(defmethod (point :move) (newloc)
  (setq location newloc)
  self)
```

defines a method for `(:move)` in the local operation set of `point`. As another example,

```
(defmethod (bounded-point :before :move) (newloc)
  (if (not (and (<= min newloc) (<= newloc max)))
      (ferror "New location ~d for point ~s is not between ~d and
 ~d."
              newloc self min max))))
```

defines a method for `(:before :move)` in the local operation set of `bounded-point`. The procedures defined by these forms can be referred to by the two expressions:

```
#'(:method point :move)
#'(:method point :before :move)
```

A further means for defining local operation sets in our example is through the :settable-instance-variables and :gettable-instance-variables clauses which can optionally appear in **defflavor** forms. In the example of figure 2-1, the **:gettable-instance-variables** clause has the same effect as the following method definition:

```
(defmethod (point :location) ()
  location)
```

In the example of figure 2-3, the effect of the **:settable-instance-variables** clause is:

```
(defmethod (bounded-point :set-min) (.newvalue.)
  (setq min .newvalue.))
(defmethod (bounded-point :set-max) (.newvalue.)
  (setq max .newvalue.))
```

A summary of the local operation names of our four example types is presented in figure 6-7.

- The **point** flavor:
    - **(:location)**
    - **(:move)**
    - **(:display)**
- The **bounded-point** flavor:
    - **(:min)**
    - **(:max)**
    - **(:before :init)**
    - **(:before :move)**
    - **(:set-min)**
    - **(:before :set-min)**
    - **(:set-max)**
    - **(:before :set-max)**
    - **(:display)**
- The **history-point** flavor:
    - **(:after :init)**
    - **(:after :move)**
    - **(:history)**
- The **bh-point** flavor:
    - **(:after :init)**
    - **(:after :set-min)**
    - **(:after :set-max)**
    - **(:bounds-history)**

Figure 6-7: Local operation names in the Zetalisp example

The above has described the names of local operations; what about the methods themselves? The **defmethod** form creates Lisp functions based on the *lambda-list* and *body* specified in the form. Three additional parameters are added to the beginning: `self`, `self-mapping-table`, and `operation`.[45] For example, the definition of the `(:before set-min)` method of `bounded-point` given above:

```
(defflavor (bounded-point :before :set-min)
           (newloc)
   ...)
```

results in the definition of the following function:[46]

```
(named-lambda (:method bounded-point :before :set-min)
              (self self-mapping-table operation newloc)
   ...)
```

Note that if programmers only use the generic invocation mechanism to invoke such methods, they can be unaware of the existence of the three added parameters. For example, the generic invocation:

```
(send p :set-min 7)
```

can be viewed as an invocation of a function of the form:

```
(lambda (newloc) ...)
```

on the parameter 7, in an environment with `self` bound to p.

6.5.3 The default algorithm for generic method computation

In the absence of any declarations to the contrary, the algorithm used for generic method computation is the following. In Zetalisp terminology, it is referred to as "daemon" method combination. The result of this algorithm can be most simply described in terms of three cases.

6.5.3.1 The relevant local operations are all untyped

If all relevant operations have method type `nil`, then the method computation algorithm selects one of the relevant untyped methods as the generic method. The algorithm for choosing this untyped method is similar to that used in Loops and Star Mesa.

> If an untyped method is defined for the flavor whose method is being constructed, that method is used. If not, then the ancestors of the flavor are examined in the canonical order described in chapter 4. Let F' be the first such flavor whose local operation set defines an untyped method for O. The untyped method for O in the local operation set of F' is used as the generic method for O of F.

[45] Recall that the first two are not present in the implementation described in [MoonStallmanWeinreb 84].

[46] The `named-lambda` construct was described n section 3.1.3.

In the example of chapter 2, the relevant local operations for **:location**, **:min**, and **:max** of **bh-point** are all untyped. In each case, there is only one such local operation (i.e., **(:location)** of **point** and **(:min)** and **(:max)** of **bounded-point**), so the methods for these local operations are chosen as the generic method for **bh-point**. The latter are simply the functions constructed by the relevant **defflavor** invocation.

6.5.3.2 The introduction of *:before* and *:after* local operations

The procedure constructed by this algorithm will perform the following actions when invoked.

1. Each relevant **:before** method of the local operation set of the type and its ancestors is invoked. The order of invocation is that in which the flavors appear in the component flavor list.

2. A single relevant untyped method is invoked, and the returned values are cached. The untyped method invoked is chosen as in section 6.5.3.1 above.[47]

3. Each relevant **:after** method of the local operation set of the type and its ancestors is invoked. The order of invocation is the *inverse* of the order in which the flavors appear in the component flavor list.

The parameters passed to of each invocation are the parameters passed to the method itself.

The result of the above algorithm is realized as a **named-lambda** whose internal name is **(:method** *flavor-name* **:combined** *operation-name***)**. For example, the method for **:move** and **:display** of **bh-point** are given in figure 6-8.[48]

6.5.3.3 The introduction of *:whopper* local operations

In the system described in [Symbolics 84], local **:whopper** methods are usually defined through the **defwhopper** special form.[49] Analogously to **defmethod**, the purpose of **defwhopper** is to hide additional parameters needed for the implementation of the underlying

[47] If neither F nor any ancestor defines an untyped local operation relevant to O, the invocation is omitted and the constructed method always returns **nil**.

[48] **Multiple-value-prog1** is a form which evaluates its parameters in order, then returns the values returned by the first evaluation.

[49] In [MoonStallmanWeinreb 84], whoppers are realized as **:around** methods. Since this implementation does not need to hide **self** or **self-mapping-table** parameters, the same **defmethod** forms as for other local operations can be used.

```
(named-lambda (:method bh-point :combined :move)
              (&rest args)
    (multiple-value-prog1
        (apply #'(:method bounded-point :before :move)
               args)
        (apply #'(:method point :move)
               args)
        (apply #'(:method history-point :after :move)
               args)))

(named-lambda (:method bh-point :combined :display)
              (&rest args)
    (multiple-value-prog1
        (apply #'(:method point :display)
                       args)
        (apply #'(:method history-point :after :display)
                       args)
        (apply #'(:method bounded-point :after :display)
                       args)
        (apply #'(:method bh-point :after :display)
                       args)))
```

Figure 6-8: Two examples of Zetalisp daemon invocation

mechanism. The **defwhopper** form is needed because **:whopper** methods take more implementation-relevant parameters than methods for other local operations. In the case of **defwhopper** the hidden parameters include the same three as **defmethod**: **self**, **self-mapping-table**, and **operation**. Figure 6-9 contains four **defwhopper** invocations which add **:whopper** operations to the local operation set of our four example types.

```
(defwhopper (point :display) (p s)
  (print 'point-before)
  (prog1
    (continue-whopper (p s))
    (print 'point-after)))

(defwhopper (history-point :display) (p s)
  (print 'history-point-before)
  (prog1
    (continue-whopper (p s))
    (print 'history-point-after)))

(defwhopper (bounded-point :display) (p s)
  (print 'bounded-point-before)
  (prog1
    (continue-whopper (p s))
    (print 'bounded-point-after)))

(defwhopper (bh-point :display) (p s)
  (print 'bh-point-before)
  (prog1
    (continue-whopper (p s))
    (print 'bh-point-after)))
```

Figure 6-9: Example whopper definitions in Zetalisp

6.5.3.3.1 Some relevant terminology

In order to understand the use of `:whopper` operations, three concepts will prove useful. Let us assume that operation 0 of flavor F has at least one relevant local method of type `:whopper`.

The "encapsulation list" is a list of the `:whopper` methods of F and its ancestors which are relevant to 0. This list is ordered in the same way as the components list of F. For example, if we added the definitions of figure 6-9 to the example of figures 2-1 through 2-4, the encapsulation list would be:

```
( #'(:whopper bh-point :display)
  #'(:whopper bounded-point :display)
  #'(:whopper point :display)
  #'(:whopper history-point :display) )
```

The "kernel method" is the procedure computed by applying the algorithm of section 6.5.3.1 (if there are no relevant `:before` or `:after` methods), or the algorithm of section 6.5.3.2 (if there *are* any such methods). To simplify the exposition, we will refer to kernel methods by the notation:[50]

#'(:method *flavor-name* :kernel *operation-name*)

For example, the kernel method for `:display` of `bh-point` is identical to the second procedure defined in 6-8[51] and will be referred to by the expression

```
#'(:method bh-point :kernel display)
```

Finally, the "methods list" is the result of adding the kernel method to the end of the encapsulations list. In our example, the methods list would be:

```
( #'(:method :whopper bh-point :display)
  #'(:method :whopper bounded-point :display)
  #'(:method :whopper point :display)
  #'(:method :whopper history-point :display)
  #'(:method :kernel bh-point :display) )
```

[50] Local operations of type `:kernel` are not part of the Zetalisp implementation.

[51] In the actual implementation, the `named-lambda` would have a different internal name.

6.5.3.3.2 The algorithm itself

We now have enough information to define the method for operation O of flavor F. Let M be the first method on the encapsulation list for O of F. The method for O of F is:

```
(named-lambda (:method F :combined O)
              (self self-mapping-table successor-methods &rest args)
    (lexpr-funcall M
                   self self-mapping-table successor-methods args))
```

What is the role of the *successor-methods* parameter? As a slight simplification of the actual situation, it can be viewed as a list containing the **cdr** of the methods list. In our **point** example, *successor-methods* would be the list:

```
( #'(:method :whopper bounded-point :display)
  #'(:method :whopper point :display)
  #'(:method :whopper history-point :display)
  #'(:method :kernel bh-point :display) )
```

The utility of this list is most clearly understood in the context of the "partial method" style of method definition described previously. Let us assume the following:

- Each **:whopper** method and each component of the **:daemon** method are "partial" methods which should be invoked as part of **:display** of **bh-point**.

- Each **:whopper** method M performs some computation, invokes the method M' which is the first member of its *successor-methods* list, then performs more computation and returns. If M' is itself a **:whopper**, the *successor-methods* of the invocation of M' is the **cdr** of the *successor-methods* passed to M.

Under these assumptions, *all partial methods will be executed once, without enumerating ancestor types.*

This situation is reflected in the example of figure 6-9. The special form:

```
(continue-whopper arglist)
```

has the effect of calling the first member of the *successor-methods* list with **self**, **self-mapping-table**, the **cdr** of the *successor-methods* list, the **operation** of the invocation, and the remaining parameters specified in *arglist*. Thus, the nesting of the invocations would be as follows:

```
ENTER #'(:method :whopper bh-point :display)
   ENTER #'(:method :whopper bounded-point :display)
      ENTER#'(:method :whopper point :display)
         ENTER #'(:method :whopper history-point :display)
            ENTER #'(:method :kernel bh-point :display)
            EXIT #'(:method :kernel bh-point :display)
         EXIT #'(:method :whopper history-point :display)
      EXIT #'(:method :whopper point :display)
   EXIT #'(:method :whopper bounded-point :display)
EXIT #'(:method :whopper bh-point :display)
```

The execution of the **:display** method for **bh-point** would produce results such as the following:

```
BH-POINT-BEFORE
BOUNDED-POINT-BEFORE
POINT-BEFORE
HISTORY-POINT-BEFORE
BH-POINT at location: ...
   with bounds: ...
   with history: ...
   with bounds history: ...
HISTORY-POINT-AFTER
POINT-AFTER
BOUNDED-POINT-AFTER
BH-POINT-AFTER
```

6.5.3.3.3 Whoppers vs. before/after daemons

The example of figure 6-9 illustrates that **:whopper** methods can be used to guarantee that computation is performed either before or after the execution of the untyped method. Several advantages derive from the use of whoppers rather than **:before** and **:after** methods.

- First, the computation associated with a given type can be defined in a single procedure. For example, suppose a **:before** method computes some result which an **:after** method is to use. To communicate the result between the two procedures requires such undesirable techniques as the use of global variables, the use of the instance variables of the generic parameter (note the havoc that recursion would induce), or the use of special "communication" parameters of the operation. In contrast, the use of **:whopper** methods allows the "before" computation to be done before calling the next method on the *successor-methods* list, and the "after" computation to be done after the successor returns.

- Second, the use of **:whopper** methods enables a dynamic binding environment to be set up within which all partial methods will be executed. For example, if an **unwind-protect**[52] is defined in a **:whopper** method for operation O a flavor F,

[52] The **unwind-protect** special form allows computation to be performed if an abnormal return occurs. A simple example is when the user aborts back to the top level program when an error has left him in a recursive read-eval-print loop.

then every relevant local method of F and its ancestors will be executed within its scope.

- Third, the use of whoppers allows methods can be exited without all components being invoked. For example, if the `#'(:method point :whopper :display)` operation returns without executing the `continue-whopper` invocation, none of the `:before` or `:after` methods will ever get invoked. In contrast, the only way for for `:before` or `:after` methods to prevent any other `:before` or `:after` methods from being invoked is to cause a non-local return by "unwinding the stack" to a specified point. This is because the constructed generic method unconditionally invokes all relevant local methods.

However, the above analysis does not imply that `:before` and `:after` methods could be eliminated from Zetalisp with no loss in programming convenience. This is because the availability of *both* before/after and whopper methods provides four separate ordering "regions":

1. before the invocation of any `:before` methods (via computation in a `:whopper` method before the successor method is invoked)
2. after the invocation of all `:whopper` methods, but before the invocation of the untyped methods (via computation in a `:before` method)
3. after the invocation of the untyped method but within the dynamic scope of all `:whopper` methods (via computation in an `:after` method)
4. after the invocation of all `:after` methods (via computation in a `:whopper` method after the successor method is invoked)

It is interesting to note that the benefits of this inter-method ordering paradigm would obtain *even in a single-inheritance type system.*

The following simple augmentation to Zetalisp would allow `:before` and `:after` methods to be eliminated while obtaining even more sophisticated ordering capabilities than those described above. The idea is to allow `(:whopper` *N*`)` methods, where *N* is a non-negative integer constant. The algorithm for constructing the encapsulations list can then guarantee that for all `i` > `j`, all `(:whopper i)` methods will occur before `(:whopper j)` methods on the successors list.

To obtain the four ordering regions described above, the current `:whopper` methods can be treated as `(:whopper 1)` methods and the `:before` and `:after` methods can be transformed into `(:whopper 0)` methods. I.e.,

```
    (defmethod (f :before :op) (parameters) body)
```

would be transformed into:

```
    (defwhopper (f :op 0) (parameters)
      body
      (continue-whopper parameters))
```

and

```
    (defmethod (f :after :op) (parameters) body)
```

would become:

```
    (defwhopper (f :op 0) (parameters)
      (multiple-value-prog1
        (continue-whopper parameters)
        body))
```

Furthermore, an arbitrary number of additional levels of ordering can be obtained by using larger values for *N*.

We will see that the addition of `(:whopper N)` methods allows a number of other method combination types to be eliminated, with no loss in programming power.

6.5.3.4 Additional details concerning *:daemon* invocation

There are four aspects of the above account which have been simplified in order to facilitate the presentation.

- First, the *successor-methods* parameter referred to in the above is actually implemented as a function referred to as the *continuation*. The `continue-whopper` procedure actually invokes the continuation, which in turn invokes the next method on the *successor-method* list. The continuation of for that next method is an internal method of the continuation of the previous one. The kernel method is an internal function of the last continuation. Thus, rather than a simple list of methods, a nested list of functions is used. The continuation functions also arrange that the appropriate `self-mapping-table` is provided for successor methods; this involves the use of another hidden parameter.

  Thus, a pragmatic disadvantage of using `:whopper` methods is that some measure of performance is lost. This is because the invocation of the methods on the encapsulation list is done indirectly, through the continuation functions. A secondary disadvantage is that the continuation functions must be constructed in the first place, which makes programs slightly larger (the real impact is likely to be on the working set rather than memory consumption per se) and increases the real-time delay perceived by the programmer when new continuations must be constructed.

- Second, we have ignored the presence of local operations whose method type is `:wrapper`. Wrappers are a `macro` analog to whoppers; rather than invoking `continue-whopper`, a macro argument is used to indicate when the invocation of the successor method should be done. Thus, the use of wrappers leads to no increase in functionality over whoppers, but results in a slightly more efficient program. For more details, see [WeinrebMoon 81].

- Third, we have not discussed the `:inverse-around` methods which are available in the system described in [MoonStallmanWeinreb 84]. The `:inverse-around` methods have their own encapsulation list, sorted in the reverse order from the encapsulation list of around methods. The method list is defined to be the concatenation of the `:inverse-around` encapsulation list, the `:around` encapsulation list, and the kernel method. The result is that if a flavor F has no ancestors which define `:inverse-around` methods for an operation O, then an `:inverse-around` for F is guaranteed to be the method for O for F or any descendant. The example given in [MoonStallmanWeinreb 84] uses this facility to define an `:init` method for a window which calls the continuation, then invokes the initial operations to start the associated process and expose the window.

  Of course, the use of `:inverse-around` methods is a two-edged sword. The problem is that it is *impossible* for descendant types to define local methods which will be executed "around" the `:inverse-around` method. Notice that the use of `(:whopper 2)` methods (as proposed in the previous section) provides the same "encapsulation of all normal whoppers" property as `:inverse-around` methods. The advantage is that descendant types can still define methods which encapsulate the `(:whopper 2)` methods, e.g., via `(:whopper 3)` methods.

- Finally, we have ignored the possibility of local methods whose method type is `:default`. Such methods are discussed in section 6.5.5.5.

6.5.4 Evaluation of the default method construction algorithm

There are two important differences between the default Zetalisp algorithm and the ones we have seen above. One of these represents the key advantage of the Zetalisp approach to that of the other three languages. The other represents a small but significant shortcoming in the current Zetalisp algorithm.

6.5.4.1 The Zetalisp algorithm reduces the amount of programming required

As described in section 5.2.5, avoiding redundant invocation often requires the definition of two procedures per operation. In Smalltalk, Loops, and Star Mesa, both of these procedures had to be defined by programmers. But in Zetalisp, one of these procedures can be constructed by the type system itself. For example, the method which is constructed for `:move` of `bh-point` in the scenario of chapter 2 corresponds directly to the methods for **move** explicitly programmed in Smalltalk, Loops, and Star Mesa.

6.5.4.2 The Zetalisp algorithm does a poor job of ordering component flavors

Recall that the Loops algorithm for method selection guaranteed that a most specific method would be chosen if one existed. The Zetalisp algorithm for selecting untyped methods does not have this property. This is because Zetalisp's algorithm for constructing the ancestors list of a flavor does not have the property types appear before all ancestors.

To illustrate, suppose we added the following three local method definitions to the example of chapter 2.

```
    (defmethod (point :some-op) ...)
    (defmethod (history-point :some-op) ...)
```

Since the ancestor ordering for **bh-point** is:

```
    (bounded-point point history-point si:vanilla-flavor)
```

the method for **:some-op** of **bh-point** will be the one defined for **:point**, *not* the more specific one defined for **history-point**.

There is no simple answer to this problem available to the Zetalisp programmer. For example, reordering the parents list in flavor definitions can help in some situations, but is not a general solution. In the above example, using the following definition of **bh-point**:

```
    (defflavor bh-point
               (bhlist bhtail)
               (history-point bounded-point))
```

would cause the ancestors list to be:

```
    (history-point point bounded-point si:vanilla-flavor)
```

so that the method for **history-point** would indeed be chosen. But the result would be that methods for **point** would then occlude those for **bounded-point**.

Furthermore, the fact that the order of the components list is not consistent with the ordering imposed by the "parent-of" relation between types causes problems whenever the order of ancestor flavors has a programmer-observable impact. For example, the order in which

:before, **:after**, the order of the *successor-methods* list for whopper invocations, and the computations used for default instance variable initialization (described in chapter 5) also depend on this ordering. If it is desired that the computation associated with a type take place either before or after the computations associated with descendants, it may well be impossible to obtain the desired result. This can be a source of considerable frustration to Zetalisp programmers.

Fortunately, modifying the Zetalisp type system to use the Loops algorithm for constructing the ancestor list would be quite straightforward. However, the impact on existing programs would be substantial, since their correctness might well depend on the existing ordering algorithm. This suggests that the appropriate response would be to provide some means for the old and new methods to coexist. For example, a syntactically different form could be used to specify parent types when the new algorithm is to be used to construct the components list. Then a definition of **bh-point** which uses the new algorithm might look something like:

```
(define-flavor bh-point
  (:instvars bhlist bhtail)
  (:parents bounded-point history-point))
```

6.5.5 The other algorithms for generic method computation

The system-defined algorithms for generic method construction in Zetalisp differ from that used in daemon combination only in their definition of the "kernel" method. Thus, the heart of this discussion will be the description of the different kinds of kernel methods which are constructed. These will fall into three categories: chaining methods, order-associated methods, and parameter-manipulation methods. We will see that the rationale for many of these algorithms is eliminated if the **(:whopper** *N***)** methods as described in section 6.5.3.3.3 were added to Zetalisp.

6.5.5.1 Determining the method combination type and order

Before entering the main part of the discussion, we describe two additional relevant aspects of flavors: the method combination type and method combination order of its generic operations.

In Zetalisp, each generic operation name of each type is associated with a symbols known as the "method combination type" and "method combination order." The combination type associated with a given operation name determines the method construction algorithm used for the that operation. The combination order is passed as a parameter to the procedure embodying this algorithm.

As with the methods themselves, the combination type associated with a given generic operation of a given flavor is a function of "local" combination types associated with the flavor and all ancestors. It is intended that these local method combination types be specified by **:method-combination** clauses in **defflavor** forms. Such clauses have the syntax:

```
(:method-combination
    (combination-type_1 combination-order_1
         operation-name_1,1 ... operation-name_1,m)
    ...
    (combination-type_n combination-order_n
         operation-name_n,1 ... operation-name_n,m))
```

where the *type*$_i$, *order*$_i$, and the *operation-name*$_{i,j}$ are all identifiers. Such clauses result in the association of local method combination type *combination-type*$_i$ with each of the *operation-name*$_{i,j}$.

For example, consider the flavor definition form:

```
(defflavor f
   (...)
   (...)
   ...
   (:method-combination
       (:progn :base-flavor-first :op-1 :op-2)
       (:list :base-flavor-last :op-3 :op-4))
   ...)
```

This form gives the generic operation names **:op-1** and **:op**$_2$ a local method combination type of **:progn** and a local method combination order of **:base-flavor-first**. It also associates the operation names `:op-3` and `:op-4` with the local method combination type `:list` and the method combination order `:base-flavor-last`.

For any generic operation names which do not appear in this list, the local combination type and local combination order are both defined to be `nil`. Thus, the local combination type and order of all the operation names of all the flavors of figures 2-1 through 2-4 are `nil`. If *combination-type* is not a method combination type known to the system, an error is signalled.

Given the above, the method combination type and combination order of operation O of flavor F is determined as follows:

- If any member of the component-flavor list of F has a non-nil local combination type for O, the combination type and combination order of O for F is the local combination type and order of the *last* such member.[53]

- Otherwise, the method combination type and order are both defined to be `nil`.

A warning is issued if different members have different non-nil local combination types or orders.

For example, if the flavor **f** is defined as above, and the flavor **g** is defined by:

```
(defflavor g
   (...)
   (f)
   ...
   (:method-combination
       (:daemon nil :op-2)
       (:progn :base-flavor-last :op-3)
       (:progn :base-flavor-first :op-4))
   ...)
```

the combination types and orders of `:op-1`, `:op-2`, `:op-3`, and `:op-4` of g will be as follows:

Operation	Combination type	Combination order
:op-1	:progn	:base-flavor-first
:op-2	:daemon	nil
:op-3	:list	:base-flavor-last
:op-4	:list	:base-flavor-last

A warning will be issued for `:op-2` and `:op-4`.

[53] The rationale for choosing the last member rather than the first is not clear.

6.5.5.2 N-ary invocation algorithms

Suppose we are computing the method for generic operation `:op` of flavor F. The first category of method combination algorithms produces kernel methods whose bodies consist of a single invocation form with the following properties:

- The function name of this invocation is determined by the combination type.
- There is one parameter expression for each member of the component-type list of F which defines an untyped method for `:op`.

We will refer to these algorithms as **N-ary invocation** algorithms, since they produce methods which consist of invocations of N-ary operations.

For N-ary invocation algorithms, the ordering of the parameter expressions is a function of the method combination order, as follows:

- If the method combination order is `:base-flavor-first`, then the invocation of a local method of flavor F_1 will precede one of F_2 iff F_1 precedes F_2 on the components list of F.
- If the method combination order is `:base-flavor-last`, then an invocation of an F_1 method will precede one of F_2 iff F_2 precedes F_1 on the components list of F.
- If the method combination order is anything else, an error is signalled.

To illustrate this method combination algorithm, we can use `:progn` combination as an example. Recall that the components list of `bh-point` is:

```
    (bh-point bounded-point point history-point vanilla-flavor)
```

Suppose that we added the following method definitions to the scenario of 2-1 through 2-4:

```
    (defmethod (point :op) (x y) (fn-1 x y))
    (defmethod (bounded-point :op) (a b) (fn-2 b))
    (defmethod (history-point :op) (n1 &optional (n2 0)) (fn-3 n2 n1))
```

If the method for `:op` of `bh-point` were constructed by the `:progn` algorithm with order `:base-flavor-first`, it would have the form:[54]

[54] In this and following sections, the `self` and `self-mapping-table` parameters are ignored. In reality, a different `self-mapping-table` must be passed to the different methods.

```
(lambda (&rest args)
  (progn
    (apply #'(:method history-point :op) args)
    (apply #'(:method point :op) args)
    (apply #'(:method bounded-point :op) args)))
```

If the order were **:base-flavor-last**, the constructed method would be:

```
(lambda (&rest args)
  (progn
    (apply #'(:method bounded-point :op) args)
    (apply #'(:method point :op) args)
    (apply #'(:method history-point :op) args)))
```

Notice that specifying a combination order of **:base-flavor-first** or **:base-flavor-last** does not have the connoted meaning. In this example, the base flavor is point, but the local method for point is neither first nor last in either of the above. This is another manifestation of Zetalisp's unfortunate choice of an ancestor enumeration algorithm.

The method combination algorithm is slightly more complex than that described above. In fact, the combined method will include invocations of typed methods where the type of the method identical to the method combination type. E.g., if the method definitions above were replaced by:

```
(defmethod (point :progn :op) (x y) (fn-1 x y))
(defmethod (bounded-point :progn :op) (a b) (fn-2 b))
(defmethod (history-point :progn :op) (n1 &optional (n2 0)) (fn-3 n2
 n1))
```

If the method combination type and order were :progn and :base-flavor-first the result would be:[55]

```
(lambda (&rest args)
  (progn
    (apply #'(:method history-point :progn :op) args)
    (apply #'(:method point :progn :op) args)
    (apply #'(:method bounded-point :progn :op) args)))
```

In the versions of Zetalisp described in [Symbolics 84] and [MoonStallmanWeinreb 84], the :progn, :list, :and, :or, :append, and :nconc method combination types cause the N-ary invocation algorithm to be used. For example, if the method combination type for :op of

[55] If both typed and untyped methods are defined for an operation, all the invocation of typed methods precede all the invocations of untyped methods. The method combination order determines the order of invocation within both categories.

bh-point were :or, and the order were :base-flavor-first, the constructed method would be:

```
(lambda (&rest args)
  (and
    (apply #'(:method history-point :op) args)
    (apply #'(:method point :op) args)
    (apply #'(:method bounded-point :op) args)))
```

Some simple uses of these method computation algorithms are as follows.

- The :progn algorithm allows the most straightforward kind of local method composition; the relevant local methods of all ancestors is invoked. Note that the constructed method could *not* be expressed using the all.*Op* invocation form of Smalltalk-82 or the SendSuperFringe procedure of Loops. Methods constructed by the latter will only invoke local methods of *some* of the ancestors.

- The :and and :or algorithms allow methods to be constructed which consist of a sequence of"heuristics." Local methods signal "success" by returning a non-nil value (for the :or algorithm) or nil (for :and). Local methods will be executed until one of them succeeds.

- The :list, :append, and :nconc algorithms are useful when the specifications for the operations of a type are defined in terms of the ancestors of the type. For example, suppose types can be associated with local methods which embody "evaluation functions" which can be applied to instances. The use of these algorithms allows each evaluation function of each ancestor to be computed, without enumeration of the relevant ancestors.

Finally, notice that for any N-ary procedure which can be expressed in terms of a binary procedure, :whopper methods and the default :daemon combination type can be used to obtain an equivalent result. For example, any :list method:

```
(defmethod (f :list :op) (parameters) body)
```

could be transformed into a method involving cons:

```
(defwhopper (f :op) (parameters)
  (cons body (continue-whopper parameters)))
```

6.5.5.3 Other forms of daemon combination

The system described in [Symbolics 84] includes four variants of the daemon combination algorithm described above. These are specified by a method combination type of :daemon-with-or, :daemon-with-and, :daemon-with-override, and :case. We will see that all but the latter would be rendered superfluous if the (:whopper *N*) methods as described in section 6.5.3.3.3 were available.

6.5.5.3.1 The *:daemon-with-and* and *:daemon-with-or* algorithms

The difference between these two and simple daemon combination is that the invocation of a single untyped method is replaced by an invocation of *all* untyped methods of the components of F within an **and** or **or** special form. As with the N-ary invocation algorithms, the order in which these appear is either the order of the components list or its inverse, depending on whether the method combination order is `:base-flavor-first` or `:base-flavor-last`. In the following we will describe the use of `:daemon-with-or`; `:daemon-with-and` is entirely analogous.

To illustrate, suppose the following were added to the example of chapter 2:

```
(defmethod (point :op) ...)
(defmethod (point :before :op) ...)
(defmethod (history-point :op) ...)
(defmethod (bounded-point :op) ...)
(defmethod (bounded-point :after :op) ...)
(defmethod (bh-point :op) ...)
```

If `daemon-with-or`, `base-flavor-first` construction were used, the kernel method would be:

```
(lambda (&rest args)
 (apply #'(:method point :before :op) args)
 (multiple-value-prog1
   (or (apply #'(:method bh-point :op) args)
       (apply #'(:method bounded-point :op) args)
       (apply #'(:method point :op) args)
       (apply #'(:method history-point :op) args))
   (apply #'(:method bounded-point :after :op) args)))
```

As a further elaboration, local methods of type `:or` can also be defined. If any are present, they are invoked within the **or** before any untyped methods. The order of invocation of the `:or` methods is determined by the method combination order as usual.

For example, suppose we add the following to the example of chapter 2:

```
(defmethod (point :op) ...)
(defmethod (point :before :op) ...)
(defmethod (history-point :op) ...)
(defmethod (history-point :or :op) ...)
(defmethod (bounded-point :op) ...)
(defmethod (bounded-point :after :op) ...)
(defmethod (bh-point :op) ...)
(defmethod (bh-point :or :op) ...)
```

and declare that :daemon-with-or, :base-flavor-last combination should be used. The constructed kernel method will then be:

```
(lambda (&rest args)
 (apply #'(:method point :before :op) args)
 (multiple-value-prog1
   (or (apply #'(:method bh-point :or :op) args)
       (apply #'(:method history-point :or :op) args)
       (apply #'(:method bh-point :op) args)
       (apply #'(:method bounded-point :op) args)
       (apply #'(:method point :op) args)
       (apply #'(:method history-point :op) args))
   (apply #'(:method bounded-point :after :op) args)))
```

Thus, the :and and :or method combination types described above are simply a degenerate case of :daemon-with-and and :daemon-with-or, where no :before or :after methods are allowed.

Notice that the introduction of (:whopper *N*) methods as described in section 6.5.3.3.3 would make :daemon-with-and and :daemon-with-or superfluous. The untyped methods would be (:whopper 0) methods, the :and/:or methods would be (:whopper 1)s, and the :before/:afters would become (:whopper 2) methods.

6.5.5.3.2 The *:daemon-with-override* algorithm

When :daemon, :daemon-with-and, or :daemon-with-or invocation is used, the before and after methods are always invoked. The :daemon-with-override method combination tvpe allows the invocation of these methods to be suppressed.

The kernel method consists of a single or form. There is one parameter expression for each member of the component-type list of F which defines an :or method for :op. These invocations are ordered :base-flavor-first or :base-flavor-last, depending on the method combination order for :op of F. The last invocation of this form is the multiple-value-prog1 which would have been constructed if :daemon combination had been used. If any of the invocations of :or methods returns a non-nil value, that value is returned. Otherwise, the result is that produced by the before/main/after ensemble.

To illustrate, suppose the following local method definitions were added to the example of chapter 2 and `:daemon-with-override`, `:base-flavor-last` combination was used.

```
(defmethod (point :op) ...)
(defmethod (point :before :op) ...)
(defmethod (history-point :op) ...)
(defmethod (history-point :or :op) ...)
(defmethod (bounded-point :op) ...)
(defmethod (bounded-point :after :op) ...)
(defmethod (bh-point :op) ...)
(defmethod (bh-point :or :op) ...)
```

(These definitions are identical to those in the previous section, with the `:or` methods replaced by `:or` methods.) The resulting kernel method would be:

```
(lambda (&rest args)
 (or (apply #'(:method bh-point :or :op) args)
     (apply #'(:method history-point :or :op) args)
     (progn
       (apply #'(:method point :before :op) args)
       (multiple-value-prog1
            (apply #'(:method bh-point :op) args))
         (apply #'(:method bounded-point :after :op) args))))
```

Thus, if either of the `:or` invocations returns a non-nil value, none of the `:before`, `:after`, or untyped methods will be invoked.

If the `(:whopper N)` methods of section 6.5.3.3.3 were added to Zetalisp, `:daemon-with-override` would not be useful. The same effect could be achieved by transforming the `:or` methods into `(:whopper 2)` methods. Rather than returning `t` to signify that no further local methods are to be executed, the `(:whopper 2)` methods would simply refrain from invoking the next procedure on the successor method list.

6.5.5.3.3 The *:case* algorithm

In this variant of daemon method construction, the invocation of the single untyped method is replaced by an invocation of a procedure which consists of a single conditional expression. The conditions associated with each arm are that the first parameter of the constructed method is identical to some constant.

When `:case` combination is used to construct the method for `:op` of flavor F, there is one arm for each method type T for which a local operation name of the form `(:case :op T)` is defined.

The condition associated with that arm is that the first parameter is identical to T. The expression associated with the arm is an invocation of a method for `(:case :op T)` in the local operation set of flavor F', where F' is the first member of the component-types list of F whose local operation set contains a method for `(:case :op T)`. The parameter list for each such invocation will be the cdr of the parameter list of the invocation of the constructed method.

What happens if none of the specified conditions is satisfied? If the local operation set of F or any ancestor contains a method for the local operation `(:case :op :otherwise)`, then the "all conditions failed" arm of the conditional will invoke the local operation of `(:case :op :otherwise)` for the first flavor on the component-type list of F which defines such a method. If not, then the "all conditions failed" arm of the conditional will invoke a standard error-signalling procedure. The parameters of the constructed method are used as the parameters of this invocation.

For example, suppose the following definitions are added to the example of figures 2-1 through 2-4:

```
    (defmethod (point :case :op :sub1) ...)
    (defmethod (point :case :op :sub2) ...)
    (defmethod (point :otherwise) ...)
    (defmethod (bounded-point :case :op :sub1) ...)
    (defmethod (history-point :case :op :sub2) ...)
    (defmethod (history-point :case :op :sub4) ...)
    (defmethod (bh-point :case :op :sub3) ...)
```

The use of case combination to produce the method for `:op` of `bh-point` would result in the construction of a method which behaved like the following:

```
    (lambda (&rest args)
      (selectq (first args)
        (:sub1 (apply #'(:method bounded-point :case :op :sub1) (cdr
     args)))
        (:sub2 (apply #'(:method point :case :op :sub2) (cdr args)))
        (:sub3 (apply #'(:method bh-point :case :op :sub3) (cdr args)))
        (:sub4 (apply #'(:method history-point :case :op :sub4) (cdr
     args)))
        (otherwise (apply #'(:method point :otherwise :op) args))))
```

The `selectq` form is Zetalisp's equivalent of a case statement where the value of the first subexpression (here `(first args)`) is tested for equality with the implicitly quoted symbol which is the first element of the all but the last subexpression. If successful, then the subsequent forms of the matching arm are evaluated. The subsequent forms of the `(otherwise ...)` clause are evaluated if none of the previous clauses match.[56]

Thus, if p is an instance of `bh-point`, the generic invocation:

```
    (send p :op :sub1 100)
```

would result in the invocation of the method for `(:case :op :sub1)` in the local operation set of `bounded-point`. The single parameter of this invocation would be `100`. The invocation:

```
    (send p :op :sub69 100)
```

would invoke the method for `(:otherwise :op)` in the local operation set of `point`. The parameters of this invocation would be `:sub69` and `100`.

The advantage of `:case` combination over the use of an untyped method whose body is the equivalent conditional is that the type system constructs the conditional, The benefits are that (a) work is saved in the initial definition (since only the different alternatives need to be specified), (b) work is saved when alternatives for ancestor types are added and removed (since the type system will automatically construct new methods), and (c) the possibility of errors in both of the above is eliminated (since the type system will always copy alternatives correctly and will never overlook a relevant alternative).

What is the point of using the `:case` algorithm rather than defining each of the cases (`(:op :sub1)`, `(:op :sub2)`, etc.) as a separate abstract operation? The key benefit is that a single definition of encapsulation methods (i.e., `:before`, `:after`, and `:whopper` methods) and an `:otherwise` clause can be applicable to the entire collection of operations.

Notice that the `:case` algorithm also offers an advantage over defining the method for a given type is a conditional containing arms only for the local alternatives specified for the type, and

[56] In the system described in [MoonStallmanWeinreb 84], untyped methods will be used if no `:otherwise` methods are present. Furthermore, all the methods of local operations (:or :or T) of F or any ancestors are invoked, base flavor last, before the conditional computation. If any of these returns a non-nil value, that value is returned as the value of the combined method.

whose `:otherwise` clause invokes a similarly-defined method of an ancestor type. The problem here is that a separate "complaint department" (e.g., [HewittBishopSteiger 73]) parameter must be passed to indicate the action to be taken when *none* of the ancestors defines a relevant alternative. With `:case` combination, there is no need for such a parameter.

6.5.5.4 Parameter-manipulating combinations

The last two predefined method combination types are `:pass-on` and `:inverse-list`. Both of these differ from the previous combination types in that the local methods are invoked with parameters other than the parameters of the generic invocation.

6.5.5.4.1 The *:inverse-list* algorithm

In methods constructed by the `:inverse-list` algorithm, the relevant local methods are the untyped methods. The parameter must be a list with exactly as many members as there are relevant local methods. Each method is called with one member of the list. The ordering of invocations is determined by the method combination order, which must be either `:base-flavor-first` or `:base-flavor-last`. The ith method is called with the N - i + i^{st} list element. For example, in the situation:

```
(defflavor a () ()
    (:method-combination (:inverse-list :base-flavor-last :foo)))

(defmethod (a :foo) (x) (format t "~%x is ~d" x))

(defflavor b () ())

(defmethod (b :foo) (y) (format t "~%y is ~d" y))

(defflavor c () (a b))

(defmethod (c :foo) (z) (format t "~%z is ~d" z))

(send x :foo '(1 2 3))
```

if x is an instance of c, the invocation:

```
(send x :foo '(1 2 3))
```

will print:

```
z is 1
x is 2
y is 3
```

This algorithm is useful to process lists returned by methods constructed using the `:list` algorithm. To illustrate, suppose that operation `:op-1` of a flavor F was constructed using `:list, :base-flavor-last` combination, and operation `:op-2` of F used `:inverse-list, :base-flavor-last`. Suppose further that the set of ancestors which defined a `:list` local method for operation `:op-1` was identical to those which define an `:inverse-list` method for `:op-2`. Finally, suppose we pass a list constructed by an invocation of `:op-1` on an instance of F as a parameter to an invocation of `:op-2` on an instance of F. The result will be that each value produced by the `:op-1` local method of each ancestor will be given as a parameter to the `:op-2` local method of the same ancestor.

6.5.5.4.2 The *:pass-on* algorithm

In `:pass-on` invocation, the relevant local methods are also the untyped methods, and they are all invoked sequentially in either `:base-flavor-first` or `:base-flavor-last` order. The method combination order is a list whose `car` is `:base-flavor-first` or `:base-flavor-last`, and whose `cdr` is the parameter list of the generic method.

The parameters of the invocation of the first local method are the parameters of the generic invocation. The parameters of the remaining invocations are the values returned by the previous invocation. For example, given:

```
    (defflavor f1 () ()
        (:method-combination
         (:pass-on (:base-flavor-last arg) :bar)))

    (defmethod (f1 :bar) (x)
      (format t "~%Parameter of local method of f1: ~s" x)
      'f1-value)

    (defflavor f2 () (f1))

    (defmethod (f2 :bar) (x)
      (format t "~%Parameter of local method of f2: ~s" x)
      'f2-value)

    (defflavor f3 () (f2))

    (defmethod (f3 :bar) (x)
      (format t "~%Parameter of local method for f3: ~s" x)
      'f3-value)
```

the invocations:

```
    (send f1-instance :bar 1)
    (send f2-instance :bar 1)
    (send f3-instance :bar 1)
```

produce:

```
    Parameter of local method of f1: 1.
    F1-VALUE

    Parameter of local method of f2: 1.
    Parameter of local method of f1: F2-VALUE
    F1-VALUE

    Parameter of local method for f3: 1.
    Parameter of local method of f2: F3-VALUE
    Parameter of local method of f1: F2-VALUE
    F1-VALUE
```

6.5.5.5 The default-method transformation on the local operation set

All of the standard method combination algorithms were defined in terms of the "untyped methods" of the local operation set of the type and its ancestors. These were defined as the local operation names which were singleton lists.

However, that account was a simplification of the actual situation. The account is not accurate in the case where, in computing the method for operation O of flavor F, no component flavors define a local operation **(O)**. In that case, all methods for local operation names of the form **(O :default)** are treated as if they were methods for local operation **(O)**.

For example, suppose that flavor **point** is an ancestor of flavor **bounded-point**, and that neither **point**, **bounded-point**, or any ancestor defines any local operations named **(:inspect)**. Then in the computation of the method for **:inspect** in the operation set of **bounded-point**, the methods defined by:

```
    (defmethod (point :default :inspect) () ...)
```

and

```
    (defmethod (bounded-point :default :inspect) () ...)
```

would be treated as if they were methods for **(:inspect)** in the local operation sets of **point** and **bounded-point**, respectively.

There are two kinds of programming situations in which **:default** methods have been used. First, for the method combination types which specify that all untyped methods will be invoked (i.e., the N-ary invocation combination types), the use of **:default** methods allow methods to be defined which will which will not be invoked if any descendant flavor defines an untyped method. For example, suppose that **h** is a descendant of **g**, **g** is a descendant of **f** and **:progn**, **:base-flavor-first** method combination is used to compute the method for **:op** of **h**. Given the definitions:

```
(defmethod (f :op) () (fn-1)
(defmethod (g :op) () (fn-2))
```

the constructed method will consist of an invocation of **fn-1** followed by one of **fn-2**. But given the definitions:

```
(defmethod (f :default :op) () (fn-1)
(defmethod (g :op) () (fn-2))
```

the constructed method will simply invoke **fn-2**.

Second, consider the method combination types which use a single untyped method from the local operation set of the first flavor on the ancestor list which defines such a method. If a **:default** method is used for a flavor **f**, then it can be guaranteed that if any descendant of **f** defines an untyped method, it can be guaranteed that the method for **f** will not be selected.

To illustrate, consider the same scenario as above, but using **:daemon** rather than **:progn** definition. Suppose we have a flavor **h** which is a descendant of **g**, and suppose that the only ancestors of **h** which define methods for **:op** are **f** and **g**. Given the first pair of method definitions:

```
(defmethod (f :op) () (fn-1)
(defmethod (g :op) () (fn-2))
```

the choice of the method for **:op** of **h** depends on the components list of **h**, which in turn depends on the parents list of **h** and *each* of its ancestors. For example, given:

```
(defflavor f (...) ())
(defflavor g (...) (f))
(defflavor g2 (...) (f))
(defflavor h (...) (g g2))
```

the components list for **h** would be

```
     (h g f g2 si:vanilla-flavor)
```

and the method for **:op** of **h** would be **fn-2**. But if the parents list of **h** were reversed:

```
     (defflavor h (...) (g2 g))
```

the components list for **h** would be

```
     (h g2 f g si:vanilla-flavor)
```

so the method for **:op** of **h** would be **fn-1**. Note that it may be important that g2 precede **g** in the components list, e.g., if there is an untyped method of g2 which should override one of **g**.

The indeterminacy of this example can be eliminated if we change the example to use a **:default** method for **f**. Given the method definitions:

```
     (defmethod (f :default :op) () (fn-1)
     (defmethod (g :op) () (fn-2))
```

either of the above definitions of the flavor **h** would result in **:op** of **h** being an invocation of **fn-1**.

Thus, **:default** methods given the programmer a means to allow more specific methods to override less specific ones; i.e., to allow methods of types T2 to override those of T1 if T2 is a descendant of T1. Recall that this was a key property of the Loops method definition algorithm which was attained by using an algorithm to construct the component-type list which guaranteed that the list was sorted according to the "parent-of" relation.

There are several reasons why Zetalisp approach is not as desirable as that of Loops.

- First, the type definer must notice that the "choose the most specific method" property has not been obtained. For flavors which have many generic operations and/or are deeply nested, the discovery that something has gone wrong may not be trivial.

- Second, there are situations where the use of **:default** methods cannot be used to have the "most specific" method selected for an operation. For example, suppose **h** is a descendant of **g** and, unlike the above examples, **h** defines a local method for **:op**. Then neither:

  ```
       (defmethod (f :default :op) () (fn-1)
       (defmethod (g :default :op) () (fn-2))
  ```

  nor

  ```
       (defmethod (f :default :op) () (fn-1)
       (defmethod (g :op) () (fn-2))
  ```

guarantees that both (1) all descendants of g which do not define a method for `:op` will obtain the one defined by g, and (2) all descendants of h which do not define a method for `:op` will obtain the one defined by h. The first pair of definitions does not satisfy (1), and the second pair does not satisfy (2).

- Third, `:default` methods do not address the problem of a *typed* method of a less specific type being used in place of one of a more specific type. An example of this behavior is the choice of the `:sub2` alternative in the illustration of `:case` method construction given above. In that example, both `point` and `history-point` defined a method for local operation `(:case :op :sub2)`, and the method of `point` was used in the constructed methods.

7. Defining the local operation set

7.1 The common abstraction

Syntactic analysis is commonly used to associate operation sets with abstract types. For example, in [Liskov et al. 79], the generic operation names of types defined by the syntactic construct:

```
idn = cluster [ parms ] is idn, ... [ where ] ;
          cluster_body
          end idn;
```

are defined to be the identifiers of

```
idn ...
```

clause of the construct.

This approach cannot be used in our four subject languages. This is because the generic operation set associated with a given type is dependent on computation rather than syntactic structure. As one example, if C is a Smalltalk class, the expression:

```
C addSelector: O withMethod: M
```

will result in M being used for subsequent generic invocations of O for instances of C. Note however that the inapplicability of the syntactic approach is not due to the fact that inheritance is used for type definition. For example, syntactic analysis is perfectly appropriate for Russell's `with`.[57] The paradigm which is appropriate can be intuitively expressed by the statement that "types are objects with state." One of the earliest embodiments of this approach was Wegbreit's pioneering work on EL/1 [Wegbreit 70, Wegbreit 74]. The principal effect of this approach is that it is possible to write type-manipulating programs, thus allowing easier production of software such as "programming environments," debuggers, and integrated editors. There are two principal disadvantages:

1. Either more expensive runtime support (i.e., operation invocation indirecting through a dispatch table associated with the type) or more elaborate compilation facilities (e.g., to allow tighter binding at the cost of operation-redefinition overhead) is needed.

[57] This constructor was discussed in section 1.4.

2. A workable approach to defining the semantics of such types is far less clear.

Despite the fact that all four of our languages allow state-changing operations on types to affect their operation sets, the use of operation-set definition via state-changes on types is independent of the algorithms used for operation set definition. We will see that any of the algorithms for constructing the generic operation set of a type from the local operation sets of its ancestors could just as well be realized in a purely applicative language where local operation sets were fixed at the time the type is defined.

Thus, it is appropriate to treat the generic operation set of a type as a data abstraction. In order to make the description of that abstraction easier to understand, an auxiliary abstraction, the "local operation set" will be used. Intuitively, the local operation set is directly associated with a type, while the generic operation set is derived from the local operation set of the type and its ancestors.

Like the instance variables of objects, the local operation set of a type can be modelled as a functional set of associations between names and values. We will refer to the names of the associations modelling the local operation set of a type as the "local operation names" of the type, and the set of values as its "local methods." If the association <N, M> is in the local operation set of T, we will say that M is the local method of N for T.

We will use two different approaches to characterize the local operation set of a type.

- In Smalltalk, Zetalisp, and Loops, the local operation set will be described in terms of a conventional "data abstraction", with procedures in the language corresponding to abstract operations.

- In Star Mesa, there are no procedures which realize abstract operations on the local operation set. However, if a program which uses the programming conventions described in [Curry et al. 82] and [CurryAyers 83] a local operation set can be defined for its traits through analysis of the procedures used for trait definition. We will use this approach to describe a technique for associating local operation sets with Star Mesa traits.

The abstract procedures we will use to characterize local operation sets in Smalltalk, Loops, and Zetalisp are as follows. Two additional types are added to those of chapter 4, **local-opname** and **local-method**. The relevant procedures are:

- **get-local-method: {type, local-opname} => local-method**
- **local-opnames: {type} => set-of (local-opname)**
- **put-local-method: {type, local-opname, local-method}**
- **remove-local-method: {type, local-opname} => type U error**

The behavior of these procedures is as follows:

- The set of associations modelling a newly created type is empty:
  local-opnames(new-type(...)) = {}

- The invocation
  put-local-method(T, N, M)
  adds a association between N and M, after removing any prior association for N.

- The invocation
  get-local-method (T, N)
  returns the value associated with N. If no such association exists, an error is signalled.

- The abstract invocation
  local-opnames (T)
  computes a set containing the names of the associations modelling the local operation set of T.

- The invocation
  remove-local-method(T, N)
  removes the association containing N, if one exists. If not, an error is signalled.

Each section of the remainder of this chapter describes how generic operation sets can be associated with types in each of our four languages. The first section of each description shows how the "local operation set" abstraction is realized. The second section describes how the generic operation set for each of our four languages is defined in terms of the local operation sets of the type and its ancestors. This is accomplished by first describing how the generic operation names are determined, then describing how the method associated with a particular generic operation name is computed.

7.2 The realization in Smalltalk-80 and Smalltalk-82

In both Smalltalk-80 and Smalltalk-82, the abstract **local-operation-name** and **local-method** types are realized instances of class `Symbol` and `CompiledMethod`. As the previous discussion indicated, there are no facilities for a given `CompiledMethod` object to be invoked with specified parameters.

The four abstract operations on the local operation set of a class are realized as follows:

- The abstract **get-local-method** operation is realized by the **`compiledMethodAt:`** generic operation of class `Class`. Invocations of this operation take a class and a symbol as parameters, and returns an object. Thus, the abstract invocation:

  get-local-method (C, O)

  is realized as:

  ```
  C compiledMethodAt: O
  ```

  For example,

  ```
  Point compiledMethodAt: #location
  ```

  returns the object most recently defined as the method for the symbol `location` in the local operation set of the class **`Point`**.

- The abstract **local-opnames** operation is realized by the **`selectors`** generic operation of class `Class`. Invocations of this operation take a class as the only parameter, and return an instance of the Smalltalk class **`Set`** whose members are instances of class `Symbol`. The abstract invocation:

  local-opname (C)

  is realized as:

  ```
  C selectors
  ```

  E.g.,

  ```
  Point selectors
  ```

  returns a **`Set`** containing the `Symbols` representing the local operation names of the class **`Point`**.

- The abstract **put-local-method** operation is realized as the **`addSelector:withMethod:`** generic operation of class `Class`. Invocations of this operation take a class, a symbol, and the object to be used as the method, and return the class. An abstract invocation:

  put-local-method (C, O, P)

  is realized as

  ```
  C addSelector: O withMethod: P
  ```

  For example,

  ```
  Point addSelector: #location withMethod: M
  ```

  defines `M` as the method for the symbol **`#location`** in the local operation set of **`Point`**.

- The abstract **remove-local-method** operation is realized by the `removeSelector:` generic operation of class `Class`. Invocations of this operation take a class and a symbol and return the class. An abstract invocation:

  remove-local-method (C, O)

  is realized as:

  ```
  C removeSelector: O
  ```

  For example, after

  ```
  Point removeSelector: #location
  ```

  the symbol `location` would no longer be a local operation name for `Point`. If the specified symbol is not already a local operation name of the class, an error is signalled.

Since any object can be used as an operation name in the above, the abstract **local-operation-name** type does not correspond to any specific Smalltalk class. However, since the operation names of all generic invocations which do not use one of the `perform:...` pseudo-operations are instances of class `Symbol`, operation names which are instances of other classes have limited utility.

Before going on to the computation of the generic operation set, there are two additional points of interest concerning local operation sets in Smalltalk. First, chapter 16 of [GoldbergRobson 83] describes a number of other operations on classes which, among other things, affect the local operation set of a class. All but one can be defined in terms of the operations we have already seen (i.e., the four "local operation set" operations, the superclass/subclass operations of chapter 4, and the `compile:` operation of chapter 3), in conjunction with the following operations of class `Class`:[58]

- operations which allow a class to be associated with a set containing its subclasses (i.e., `addSubclass:`, `removeSubclass:`, `subclasses`);

- operations which allow a method definition string to be associated with a given operation name, and for the string associated with an operation name to be retrieved

[58] Specifically, each of the following operations of class `Class` can be defined in terms of the listed operations: `compile:notifying:`, `recompile:`, `compileAll`, `compileAllSubclasses`, `removeCategory:`, `copy:from:`, `copy:from:classified:`, `copyAll:from:`, `copyAll:from:classified:`, `copyAllCategoriesFrom`, `copyCategory:from:`, `copyCategory:from:classified:`, `compile:classified:`, and `compile:classified:notifying:`.

(sourceCodeAt:);[59] and

- operations which allow the association of a symbolic "category" with operation names (category:) and the retrieval of a category associated with a given name (whichCategoryIncludesSelector:).

The methodDictionary: operation of class Class is the only operation described in chapter 16 of [GoldbergRobson 83] whose invocation cannot be described in terms of the operations listed above. It allows the wholesale replacement of the data structure which represents the local operation set. In Smalltalk terminology, this data structure is referred to as the "method dictionary."

Second, it is interesting to note that Smalltalk programmers can be unaware of the details of the operations which affect the local operation set of a class. This is because Smalltalk's user interface allows the invocations of such operations to be accomplished through menu-based interaction. For example, programmers do not need to know the names and parameter requirements for the operations which create compiled methods and install them in local operation sets, Instead, the programmer can select the Accept option of a menu associated with a text-editing "browser" program. See chapter 17 of [GoldbergRobson 83] for a simple scenario which demonstrates this capability.

7.2.1 Aside: operation classification in Smalltalk

The simplest operation described in [GoldbergRobson 83] which both compiles a method and stored it in a method dictionary is compile:classified:. This operation takes three parameters: a class, a string containing a method definition, and a second string whose contents are arbitrary. The second string is used to "categorize" the operation but has no significance for generic invocation.

A brief description of operation categorization is as follows. Smalltalk classes can be thought

[59] The name of the operation which associates local operation names with strings returned by the **sourceCodeAt:** operation is not identified in [GoldbergRobson 83].

of as being associated with a "classification" dictionary as well as a method dictionary. The classification dictionary associates a programmer-specified subset of the keys of the method dictionary with strings. For example,

```
      Point compile: 'location ^ loc' classified: 'Accessing'.
```

associates the symbol #location with the string 'Accessing' in the classification dictionary of Point.

The classification dictionary is used by a number of operations which allow modification of a class' method dictionary. As a representative example, the copyCategory:from: operation takes a destination class, a string, and a source class. It modifies the method dictionary of the destination class to include all pairs of keys and values of the source class whose keys are classified under the specified string. For example,

```
      HistoryPoint copyCategory: 'Accessing' from: BoundedPoint
```

would create entries in the method dictionary of HistoryPoint for the Accessing operations of BoundedPoint, i.e., min and max.

If classes can have only a single ancestor, as in unextended Smalltalk-80, this kind of method dictionary manipulation is necessary whenever (a) the generic operation set of a class is to include methods defined in the local operation set of more than one class, or (b) when operations were to be moved from the method dictionaries of one class to another. We will a principal benefit of type definition via multiple inheritance is the reduction in the number of situations in which copying is needed to support sharing of methods between generic operation sets.

7.3 The realization in Loops

In Loops, as in Smalltalk, the abstract **local-operation-name** and **local-method** types are realized Interlisp ATOMs and Interlisp procedure objects.

The abstract operations on the local operation set of a Loops class are realized as follows.

- The abstract **get-local-method** operation is realized as invocations of the GetMethod and GETD procedures. The abstract invocation:

get-local-method (C, O)

is realized as:

```
(GETD (GetMethod C O))
```

For example,

```
(GETD (GetMethod $Point 'Location))
```

returns the object most recently defined as the method for the symbol `Location` in the local operation set of the class `Point`.

- The abstract **local-opnames** operation is realized by the `List` generic operation of class `$Class`. The abstract invocation:

  local-opname (C)

  is realized as:

  ```
  Send C List 'Selectors
  ```

  Such invocations return a list containing the atoms which are the local operation names of the class. For example,

  ```
  (Send $Point List 'Selectors)
  ```

  obtains the local operation names of the class `$Point`.

- The abstract **put-local-method** operation is realized by a composition of the `PutMethod` and `PUTD` procedures. Specifically, if `C` is a class, `O` is an operation name, and `P` is a procedure, the abstract

  put-local-method (C, O, P)

  is realized as

  ```
  (PutMethod C O O')
  (PUTD O' P)
  ```

  where `O'` is a symbol which satisfies a property described shortly. For example,

  ```
  (PutMethod $Point 'Location 'Point.Location)
  (PUTD 'Point.Location '(LAMBDA (self) (GetValue self 'loc)))
  ```

  defines `(LAMBDA (self) (GetValue self 'loc))` as the method for the symbol `Location` in the local operation set of `$Point`.

- The abstract **remove-local-method** operation is realized by an invocation of the `PutMethod` procedure. If `C` is a class and `O` is an operation name, the abstract

  remove-local-method (C, O, P)

  is realized as

  ```
  (PutMethod C O Nil)
  ```

  For example, after

  ```
  (PutMethod $Point 'Location Nil)
  ```

  `Location` would no longer be a local operation name of `$Point`. If `O` were not a local operation name of T, the local operation set would not be modified and no error would be signalled.

In the above, the abstract **operation-name** type is realized as the Interlisp datatype `ATOM`. Atoms are Interlisp's realization of what is commonly referred to as a "symbol," i.e., an structure whose members have character-string names and one or more memory cells which can refer to values.

The principal difference between this realization and that used in Smalltalk is that two different kinds of associations are used to define the local operation set of a class. One kind, realized through the GetMethod and PutMethod procedures, associates a class and an operation name with a symbol. The other, realized via the GETD and PUTD procedures, associates symbols with procedures. In contrast, Smalltalk classes have a single association between local operation names and procedures.

An explanation of this approach requires a slightly more detailed description of the Interlisp datatype ATOM. Each Interlisp atom is associated with a "function definition cell:" a storage location separate from the conventional "value cell." While the contents the latter are used by lambda-binding and variable evaluation, the contents of the former are stored and retrieved only by PUTD ("PUT Description") and GETD ("GET Description") procedures.[60] PUTD takes a symbol and an object, and makes the function definition cell of the symbol contain the object, and GETD takes a symbol and returns the object currently in its function definition cell. For example,

```
    (PUTD 'Point.Location
          '(LAMBDA (self)(GetValue self 'loc)))
```

makes the function definition cell of the symbol Location contain the (LAMBDA ...) list, and

```
    (GETD 'Point.Location)
```

returns the contents of the function definition cell of the symbol Location.

Thus, the computation of the local method of an operation of a class (i.e., the abstract

[60] A brief explanation of the rationale for the function definition cells is as follows. The value stored in the function definition cell of a symbol is used as the procedure when the symbol appears in the first position of a procedure-invocation form. For example, to evaluate the form:

```
    (+ 2 2)
```

the value of the function definition cell of + is used to obtain the procedure invoked. The name "function definition cell" is motivated by this usage. Most of the progeny of Lisp 1.5 support a similar distinction between the "local" and "global" values of a symbol, and use the latter to evaluate function-invocation forms.

Why is it helpful to have variable values used for procedure invocation not affected by lambda binding? One benefit is that useful names can have different references in different contexts, e.g., point can be both a function for creating points and the name of a local variable bound to points. A second advantage is that the introduction of local variables in lexically enclosing procedures or dynamically enclosing invocations will not affect the procedure used for such invocations. The result is the elimination of one pathway of interaction between procedure definitions.

get-local-method operation) has two stages: using GetMethod to computing an atom from the class and the local operation name, then using PUTD to extracting the method from the function definition cell of that atom. We will refer to this intermediate atom as the "local method symbol" for the given operation name and class. The rationale for introducing the local method symbol will be described shortly.

The indirection through the local method symbol means that there are two ways to affect the local operation set of a class. The first approach is to change the local method symbol associated with a class and operation name via the PutMethod operation. The second is to store a new object in the function definition cell of an existing local method symbol, via PUTD. The combination of the two can be used to realize the abstract put-local-method operation.

A second implication of the use of programmer-designated local method symbols is that it introduces the possibility of a new kind of programming error: the unintentional use of the same local method symbol for two different operations. For example, if the atom Location were used as the local operation symbol for the Location operation of both the $Point and $BoundedPoint classes, it would not be possible to modify $Point's local method for Location without modifying $BoundedPoint's as well.

This problem is addressed, but not solved, by the introduction of an informal naming convention for local method symbols. In all the examples of [Stefik et al. 83b], the local method symbol for operation O of class whose name is C is C.O, e.g., Point.Location[61] As long as no two classes have the same name, this convention will guarantee that the local method symbols of all classes will be disjoint. And if atoms with embedded periods are used only for local method symbols, naming conflicts will not arise at all. However, the use of this convention is not a completely satisfactory solution, because incorporating programs which do not follow the convention (e.g., procedures written by other programmers for other purposes) requires prior modification of the programs to be included.

[61] Periods have no special significance in Interlisp symbol names.

7.3.1 The procedures used in the example

[Stefik et al. 83b] describe a number of procedures which affect the associations between operations and local method symbols.[62] All of these can be expressed as invocations of the **PutMethod** procedure.[63] invoke active A particularly useful procedure which modifies the local operation set of a class as a side effect is the **DefClass** procedure used to create new classes. If an invocation of **DefClass** contains a **Methods** clause of an appropriate form, invocations of **DefMethod** are generated whose parameters are extracted from the text of that clause. The required form for the clause is:

```
(Methods (key_1 value_1)
         ...
         (key_n value_n))
```

where each of the key_i and $value_i$ are identifiers. If there are n such pairs in a definition of class C, then there will be n invocations of **PutMethod** of the obvious form:

```
(PutMethod C 'key_i 'value_i)
```

For example, the class definition:

```
(DefClass Point
    ....
    (Methods (LocalInitialize Point.LocalInitialize)
             (Location Point.Location)
             (Move Point.Move)
             (Display Point.Display))
    ...)
```

would generate:

```
(PutMethod $Point 'LocalInitialize 'Point.LocalInitialize)
(PutMethod $Point 'Location 'Point.Location)
(PutMethod $Point 'Move 'Point.Move)
(PutMethod $Point 'Display 'Point.Display)
```

The **DEFINEQ** procedure is useful for defining a function and storing it in the function definition

[62] Some of the other procedures which affect the local operation set involve: interfacing with the Interlisp editor (via the **DefMethod** and **EditMethod**, alias **DM** and **EM**, procedures), or the **DefMethod** or **Edit** operation on classes), invoking the "store function" of an active value of the current method definition symbol (the **PutMethod** procedure), changing the local operation names of a class (the **Rename** operation on classes), copying and moving local operation definitions from one class to another (the **CopyMethod** and **MoveMethod** operations on classes), and as a side-effect of changing the name of the class, changing all the local method symbols to be homologous symbols derived from the new class name (the **Rename** and **SetName** operations on classes).

[63] For the record, **PutMethod** and **GetMethod** treat active values in the same way as the **GetValue** and **PutValue** instance variable procedures described in chapter 5. Thus, the abstraction of this chapter will only be satisfied if the get and put functions behave appropriately.

cell of a symbol. **DEFINEQ** takes an unevaluated list of items as its single parameter, extracts a symbol and a function definition from each member of the list, and stores the extracted function in the function definition cell of the extracted symbol. To illustrate its operation, the expression:

```
(DEFINEQ
  (F (LAMBDA (self plist)
       (PutValue self 'loc (or (PlistGet plist 'location) 0))))
  (G (self plist)
       (PutValue self 'loc (or (PlistGet plist 'location) 0)))
  (H (LAMBDA arglist (GetValue (ARG arglist 1)))))
```

stores copies of the same **LAMBDA** expression in the function definition cells of F and G and a list denoting a lambda/nospread/expr procedure in the function definition cell of **H**.

The principal advantage of using local method symbols rather than the direct association of local operation names and procedures is that local method symbols can be the subject of **DEFINEQ** statements. This is because a number of Interlisp facilities are easier to use if **DEFINEQ** is used to associate symbols with procedure definitions. Such facilities include the editor, the compiler, and the "files" package for managing incremental changes made during the course of a programming section. Thus, the use of local method symbols to hold local methods allows the full power of the Interlisp programming environment to be brought to bear on Loops programs.

Finally, figure 2-5 defines **AllLocalMethods!** and a macro cognate, **AllLocalMethods**. **AllLocalMethods** takes an instance I of a class C, an unevaluated atom A, and an arbitrary number of other parameters. It collects all the methods for A in the local operation set of C and all ancestors, and applies those procedures to I and the remaining parameters. The order of application is the reverse of the canonical ordering defined in chapter 4. For example, if **bhp** is an instance of **BHPoint**, the invocation:

```
(AllLocalMethods bhp PartialInitialize plist
```

would invoke the local **PartialInitialize** methods of each of the **HistoryPoint**, **BoundedPoint**, and **BHPoint** types, in that order.

AllLocalMethods supports the method definition technique described in section 5.2.5 by allowing all relevant "partial" procedures to be invoked without having to enumerate the classes of which they are local methods. The benefits are that:

- less work is required to initially define methods which invoke partial procedures of more than one ancestor,
- adding or removing partial procedures does not require changes to methods of descendant types, and
- generic methods of a class C which call partial procedures can be inherited in class C' even if C' has additional ancestors which define partial procedures.

To illustrate the latter point, observe that the `Initialize` of `Point` can be inherited by the other three types. In the Smalltalk and Star Mesa implementations, separate `Initialize` methods are required, since a different set of partial procedures must be invoked.

7.4 The realization in Star Mesa

Star Mesa provides no operations which realize the abstract procedures we have been using to characterize the local operation set of a type. Nevertheless, if Star Mesa programs satisfy the conventions specified in in [Curry et al. 82] and [CurryAyers 83], a local operation set can be associated with each trait definition.

This description is organized as follows. We will first describe a new aspect of traits in terms of which the relevant programming conventions are expressed, then describe the conventions, and finally show how the local operation set is derived.

7.4.1 Traits and trait components

In chapter 5, we saw that instances of a trait are associated with a collection of objects which represented its instance variables. For an instance of trait T, there is one such object for T and all ancestors. For example, instances of `BHPoint` have four instance variables: one for `Point`, `BoundedPoint`, `HistoryPoint`, and `BHPoint`.

In fact, Star Mesa traits themselves are associated with a collection of objects. Each trait T has one trait component for T itself and one for each of its ancestors. Thus, the object of type `TM.Trait` representing `BHPoint` trait has four components: one for `Point`, `BoundedPoint`, `HistoryPoint`, and `BHPoint`. We will refer to to these objects as the "trait components" of the trait.

The primitive means for obtaining references to trait components is through the **TraitComponent** procedure of the trait manager module. The Mesa type of this procedure is:

```
PROC [TM.Trait, TM.Trait] RETURNS [POINTER TO UNSPECIFIED]
```

It returns the memory location of the component of the first trait which is defined by the second trait. For example,

```
TM.TCFromTrait [TM.TraitNamed ["BoundedPoint"],
                TM.TraitNamed ["Point"]]
```

returns the trait component of the trait named **BoundedPoint** which is defined by the trait named **Point**. Since the Mesa type of any invocation of this procedure is an untyped pointer, the returned value can be used in any context where any pointer type is required. For example, the statement:

```
X: POINTER TO T ←
    TM.TCFromTrait [TM.TraitNamed ["BoundedPoint"],
                    TM.TraitNamed ["Point"]]
```

is legal for any type **T**.

The examples of chapter 2 use two special syntactic forms, **TM.TCFromTrait** and **TM.TCFromObject**, which generate invocations of **TM.TraitComponent**.

- The form:

  TM.TCFromTrait [id_1, id_2]

  is equivalent to:

  TM.TraitComponent [TM.TraitNamed ["id_1"],
  TM.TraitNamed ["id_2"]]

  For example,

  ```
  TM.TCFromTrait [Point, Point]
  ```

  generates

  ```
  TM.TraitComponent [TM.TraitNamed ["Point"],
                     TM.TraitNamed ["Point"]]
  ```

  The latter returns the trait component of **Point** which is defined by **Point**.

- The form:

  TM.TCFromObject [*exp*, *id*]

  is equivalent to:

  TM.TraitComponent [TM.Type [*exp*], TM.TraitNamed ["*id*"]]

  For example, if the expression **X** evaluates to an instance of trait **Point**, the form:

  ```
  TM.TCFromObject [X, Point]
  ```

  also computes the component of trait **Point** defined by trait **Point**.

7.4.2 Programming conventions for defining trait components

Trait components are allocated regions of storage, but the procedures for accessing this storage return addresses. The following two conventions relate the addresses with the regions.

> **Star Mesa Convention 6:** Each trait definition designates a record type as its **trait component type.** If R is the trait component type of trait T, all expressions which compute a trait component defined by T should be used only in contexts which require the type R.

In the example of chapter 2, a comment of figure 2-18 states that the value of the identifier **TCType** in each trait definition module is the trait component type of the trait. For example, the trait component type of trait **Point** is the value of the identifier **TCType** defined in module **Point**:

```
RECORD [Location: PROC [p: Object] RETURNS [REAL],
        Move: PROC [p: Object, r: REAL] RETURNS [REAL],
        Display: PROC [p: Object, s: Stream]]
```

Thus, all references to the trait components obtained by the expressions:

```
TM.TCFromTrait [Point, Point]
TM.TCFromTrait [HistoryPoint, Point]
TM.TCFromTrait [BoundedPoint, Point]
TM.TCFromTrait [BHPoint, Point]
```

should appear only in contexts where **Point.TCType** is required, e.g.,

```
PointTC: POINTER TO Point.TCType
          ← TM.TCFromTrait [BoundedPoint, Point]
```

Note that since **TM.TraitComponent** is declared to return untyped pointers, the Mesa compiler cannot detect the violation of this convention. As with the inappropriate use of **TM.InstanceComponent**, the consequences of this occurrence are unpredictable.

> **Star Mesa Convention 7:** The amount of storage for instances of the trait component type is the **TCSize** field of the record returned by the initial invocation of the registration procedure of the trait.

For example, records returned by invocations of the registration procedure of module **BHPoint** contain the value of the expression:

```
SIZE [RECORD [BoundsHistory: PROC [P: Object] RETURNS
   [EntryConsCell]]]
```

in their TCSize field. The Mesa pseudo-procedure SIZE takes a Mesa type and returns the amount of storage used to represent instances of that type.

7.4.3 Programming conventions for modifying trait components

In order to arrange that a trait T has a given generic operation set, the trait components of T must be modified. The following are the conventions for performing such state changes.

> **Star Mesa Convention 8:** The local initialization procedure of each trait should take a single parameter, a trait. The actions of the procedure should consist solely of assignments to fields of trait components of its parameter.

For example, consider the local initialization procedure of the trait HistoryPoint:

```
LocalInitializeTrait: PROC [trt: TM.Trait] =
   {PointTC: POINTER TO Point.TCType ←
        TM.TCFromTrait [trt, Point];
    HistoryPointTC: POINTER TO TCType ←
        TM.TCFromTrait [trt, HistoryPoint];
    PointTC.Move ← MoveImpl;
    HistoryPointTC.History ← HistoryImpl;
    PointTC.Display ← DisplayImpl};
```

This procedure modifies trait components defined by Point, via:

```
PointTC.Move ← MoveImpl
```

and

```
PointTC.Display ← DisplayImpl
```

as well as trait components defined by HistoryPoint:

```
HistoryPointTC.History ← HistoryImpl
```

Notice that the local initialization procedure of a trait modifies trait components defined by other traits.

> **Star Mesa Convention 9:** The only procedures which store into trait components defined by a trait T are the local trait initialization procedures of T and its parameters. The only procedures which invoke local trait initialization procedures are the trait initialization procedures. The trait initialization procedures are never explicitly invoked.

This convention guarantees that the trait components of all traits will be constant during the execution of the program. Although it is not stated explicitly in [Curry et al. 82] or [CurryAyers 83], it is necessary for the generic operation sets of traits to be defined as they describe.

7.4.4 Programming conventions for generic operation procedures

Recall that generic operation names in Star Mesa are procedure objects, not the symbols used in Smalltalk, Zetalisp, and Loops. By convention, these are stylized procedures which access the trait components of the trait of the generic parameter in a particular way.

> **Star Mesa Convention 10:** Each generic operation procedure invokes some other procedure on its original parameters. If P is a generic operation procedure defined in the module defining trait T, the procedure it invokes is chosen as follows. Let T_2 be the Mesa type of the first parameter of P.
>
> 1. Obtain the component of trait T_2 defined by trait T. Call this value C.
>
> 2. For some field identifier `I` of the trait component type, use the value of the `I` field of of `C` (i.e., `C.I`) as the invoked procedure.
>
> We will refer to the field `I` used by a given generic operation procedure as the "method field" of the generic operation.

For example, consider the definition of the generic operation procedure `Point.Location`:

```
Location: PROC [p: Object] RETURNS [REAL] =
   {TC: POINTER TO TCType ←
         TM.TCFromObject [p, Point];
   RETURN [TC.Location [p]]}
```

It obtains the component of the type of p which is defined by `Point`:

```
TC: POINTER TO TCType ←
      TM.TCFromObject [p, Point]
```

then extracts the `Location` field of this component:

```
TC.Location
```

then applies the resulting procedure to the original parameters:

```
TC.Location [p]
```

Thus, the method field associated with the generic operation `Point.Location` is the `Location` field of type `Point.TCType`.

7.4.5 The definition of the local operation set

Given the above, we can (finally!) define the local operation set of a trait as follows.

- The operation names of the local operation set of a trait T are the set of procedures whose method fields are modified by the local trait initialization procedure of T.

- The local method of an operation O is the value stored into the field by the local trait initialization procedure of T.

Thus, the local operation sets of our four example traits are as described in figure 6-5. To emphasize the relationship between local operation sets and local trait initialization procedures, figure 7-1 reproduces the four local trait initialization procedures of the example. Note the correspondence between these procedures and the local operation sets of figure 7.2.

From the `Point` *module:*

```
    LocalInitializeTrait: PROC [trt: TM.Trait] =
       {PointTC: POINTER TO TCType ←
             TM.TCFromTrait [trt, Point];
        PointTC.Location ← LocationImpl;
        PointTC.Move ← MoveImpl;
        PointTC.Display ← DisplayImpl};
```

From the `HistoryPoint` *module:*

```
    LocalInitializeTrait: PROC [trt: TM.Trait] =
       {PointTC: POINTER TO Point.TCType ←
             TM.TCFromTrait [trt, Point];
        HistoryPointTC: POINTER TO TCType ←
             TM.TCFromTrait [trt, HistoryPoint];
        PointTC.Move ← MoveImpl;
        HistoryPointTC.History ← HistoryImpl;
        PointTC.Display ← DisplayImpl};
```

From the `BoundedPoint` *module:*

```
    LocalInitializeTrait: PROC [trt: TM.Trait] =
       {PointTC: POINTER TO Point.TCType ←
             TM.TCFromTrait [trt, BoundedPoint];
        BoundedPointTC: POINTER TO TCType ←
             TM.TCFromTrait [trt, Point];
        PointTC.Move ← MoveImpl;
        BoundedPointTC.Min ← MinImpl;
        BoundedPointTC.Max ← MaxImpl;
        BoundedPointTC.SetMin ← SetMinImpl;
        BoundedPointTC.SetMax ← SetMaxImpl;
        PointTC.Display ← DisplayImpl};
```

From the `BHPoint` *module:*

```
    LocalInitializeTrait: PROC [trt: TM.Trait] =
       {PointTC: POINTER TO Point.TCType ←
             TM.TCFromTrait [trt, Point];
        BoundedPointTC: POINTER TO Point.TCType ←
             TM.TCFromTrait [trt, BoundedPoint];
        BHPointTC: POINTER TO TCType ←
             TM.TCFromTrait [trt, BHPoint];
        PointTC.Move ← MoveImpl;
        BoundedPointTC.SetMin ← SetMinImpl;
        BoundedPointTC.SetMax ← SetMaxImpl;
        BHPointTC.BoundsHistory ← BoundsHistoryImpl;
        PointTC.Display ← DisplayImpl};
```

Figure 7-1: The local trait initialization procedures of the Star Mesa example

7.5 The realization in Zetalisp

In Zetalisp, operations on the local operation set are integrated into the general "function definition" facility ([WeinrebMoon 81], p. 149). As a result, the standard Zetalisp procedures for defining a name as a function, testing if a function name is defined, and obtaining the current definition of a function name can be applied to local operation names and methods. The relationship between function names and the local operation set is as follows.

Function names which are lists of three or more elements and whose first element is the symbol **:method** denote elements of the local operation sets of flavors. Specifically, a list of the form

(:method id_1 id_2 ... id_N)

refers to the method for

(id_2 ... id_N)

in the local operation set of id_1. For example, the definition of the function name:

```
(:method bounded-point :set-min)
```

is the method for **(:set-min)** in the local operation set of **bounded-point**. As another example, the function definition of the function name:

```
(:method bounded-point :after :move)
```

is the method for **(:after :move)** in the local operation set of **bounded-point.**

The use of a general function-defining facility for operating on the local operation set of a type is one difference between Zetalisp and our other subject languages. A second difference is that the actual functions being used as methods are rarely seen by users. As described in chapter 5, the use of the **defmethod** and **defwhopper** macros conceal the presence of required parameters and declarations. Thus, the realization of **put-local-method** is usually invoked only through the detail-hiding **defmethod** and **defwhopper** macros. Furthermore, the realization of **get-local-method** is of limited utility, given the difficulty (discussed in section 3.4.3) of determining the appropriate mapping table for a given invocation.

Nevertheless, all manipulations on the local operation set of a flavor can be expressed in terms of the following.

- The abstract **get-local-method** operation is realized by the **fdefinition** procedure previously described. The abstract invocation:

  get-local-method (C, O)

  is realized as:

  ```
  (fdefinition (:method C . O))
  ```

  or, if C and each member of O is constant, the equivalent:

  ```
  #'(:method C . O)
  ```

  The latter is the form commonly seen when interacting with the Zetalisp system. For example,

  ```
  (fdefinition '(:method point :location))
  ```

  and

  ```
  #'(:method point :location)
  ```

  return the object most recently defined as the method for the symbol **:location** in the local operation set of the class named **point**.

- The abstract **local-opnames** operation cannot be expressed in terms of the procedures described in chapter 20 of [WeinrebMoon 81] or [MoonStallmanWeinreb 84]. However, there are several user-interface procedures (e.g., the **describe-flavor** procedure, the **List Combined Methods** editor command, the "Flavor Inspector" program) which can provide this information to the programmer.[64] The abstract **put-local-method** operation is realized through the **fdefine** procedure. The abstract invocation

  put-local-method (C, O, P)

  is realized as:

  ```
  (fdefine '(:method ,C  ,@O) P)
  ```

  For example,

  ```
  put-local-method (point,
                    (:move),
                    #'(lambda (...) ...  (setq location newloc)
  self)
  ```

  would be realized as:

  ```
  (fdefine '(:method point :move)
           (lambda (...) ... (setq location newloc) self))
  ```

- The abstract **remove-local-method** operation is realized by the **undefmethod** syntactic form. The abstract invocation:

  remove-local-method (C, O)

  is realized as:

  ```
  (undefun (C. O)
  ```

  For example, after

  ```
  (undefun '(point :move))
  ```

  the list `(:move)` would no longer be a local operation name for the flavor named **point**. If the specified list is not already a local operation name of the class, an error is signalled.

---

[64] Since flavors can be operated on in the same way as any data structure, defining such a procedure can be done straightforwardly. In languages where types are not simply objects, such an addition would require modifying the language-implementing software (e.g., compiler).

The use of four optional clauses of the **defflavor** form offers another interface to the abstract **put-local-method** operation. The local methods added through this technique allow the local instance variables of the flavor being defined to be accessed or modified. Here are the four kinds of clauses, together with a description of their effect in a definition of a flavor named *F*:

- **:gettable-instance-variables** For each local instance variable *v* of F, a local method definition corresponding to the following is made:

  ```
  (defmethod (F :v) ()
     v)
  ```

  For example, the **defflavor** of figure 2-1 results in the following:

  ```
  (defflavor point
     ((location 0))
     ...
     :gettable-instance-variables
     ...)
  ```

  results in the equivalent of:

  ```
  (defmethod (point :location) ()
    location)
  ```

- **(:gettable-instance-variables** v_1 ... v_n**)** For each of the v_i which is a local instance variable of F, a local method definition of the form:

  ```
  (defmethod (F :v) ()
     v)
  ```

  is generated. This allows instance-variable-accessing methods to be constructed for a subset of the local instance variables of a flavor.

- **:settable-instance-variables** For each local instance variable *v* of F, a local method definition corresponding to the following is made:

  ```
  (defmethod (F :set-v) (.newvalue.)
      (setq v .newvalue.)
  ```

  For example, the **defflavor** of figure 2-3:

  ```
  (defflavor bounded-point
     ((min 0) (max 100))
     ...
     :settable-instance-variables
     ...)
  ```

  results in the equivalent of:

  ```
  (defmethod (bounded-point :set-min) (.newvalue.)
    (setq min .newvalue.))
  (defmethod (bounded-point :set-max) (.newvalue.)
    (setq max .newvalue.))
  ```

- **(:settable-instance-variables** v_1 ... v_n**)** For each of the v_i which is a local instance variable of F, a local method definition of the form:

  ```
  (defmethod (F :set-v) ()
     (setq v .newvalue.))
  ```

  is generated. This allows methods which store into a particular instance variable to be constructed for a subset of the local instance variables of a flavor.

Furthermore, any variable for which a **:set-*v*** method is created through a **:settable-instance-variable** clause will also have a **:*v*** method generated as a result of the **defflavor** invocation. For example, the equivalent of:

```
     (defmethod (F :min) ()
        min)
     (defmethod (F :max) ()
        max)
```

are defined as a result of the **defflavor** of figure 2-3.

8. Type declaration and type checking

The principal characteristic which distinguishes our four subject type constructors from conventional approaches to abstract type definition is that they embody definition by multiple inheritance. However, the languages in which they are embedded have considerably different approaches to type declaration and type checking.

This chapter shows that these differences are, in fact, inessential. We will see that the same kinds of type declarations can be used in both conventional and multiple-inheritance languages, and that inheritance-based type definition can be used with our without the "strong type checking" algorithm used in conventional languages which support abstract type definition. Furthermore, we will see that the inter-type relationships used for definition by inheritance can be the basis of a new kind of type checking algorithm -- one which requires fewer assumptions than the conventional approach, but still guarantees the detection of all type errors.

8.1 Type declarations

8.1.1 The conventional approach

In conventional abstract type systems, three forms of type declarations are found:

- **Type identity.** In a type-identity declaration, the type of the object must be identical to a specified type. This is the most straightforward approach to type declaration, and has appeared in languages dating from Algol 60 [Naur 63].

- **Operation support.** An operation support declaration is defined by specified set of abstract operations, possibly with type restrictions on parameters and returned values. Such declarations are satisfied by any object whose type has the specified operations. An example of such a declaration would be one that is be satisfied by any object whose type includes abstract operation for `equal` and `less-than`.

  The version of Alphard described in [Wulf et al. 76], [Shaw et al. 76], and [London et al. 76] contains such declarations. Clu and Russell [Demers et al. 78, BoehmDemersDonahue 80] allow operation-support declarations for parameters of type `type`; such types can then be used to construct the desired declarations.[65]

- **Parametric variation.** Parametric variation declarations are satisfied by objects whose types have been produced by a particular abstract type constructor. For example, given a `list` constructor which takes a type parameter, it is possible to have a declaration which is satisfied by any object whose type was produced by that constructor, e.g., "`list` of `integer`," "`list` of `symbol`."

  This kind of declaration is common among languages which allow the definition of new abstract type constructors. For example, Alphard and ML [Milner 78, Gordon et al. 79] allow any variable to have such a declaration. As with operation-support declarations, Clu and Russell require the introduction of auxiliary type parameters for parametric-variation declarations.

In each case, if a variable is associated with a declaration, the rules of the language which define legal programs guarantee that the declaration holds for the lifetime of the variable. This is accomplished by disallowing any program in which a variable of one type is assigned an expression of some other.[66]

8.1.2 The multiple-inheritance languages

In our four subject languages, procedures analogous to these declaration forms are either directly available or can be programmed.

- **Type identity.** These are available in each of our four type systems. To check that an object *Obj* is an instance of a type *T*:

  - Zetalisp: `(eq (typep` *Obj*`) Tname)`, where `Tname` is the symbol which names the type.

  - Loops: `(<-` *Obj* `InstOf` *T*`)`

  - Smalltalk-82: *Obj* `isMemberOf` *T*

  - Star Mesa: `TM.TypeOf [`*Obj*`] =` *T*

[65] Specifically, operation support declarations can be associated with the type parameters, which can then be used in further declarations. This indirection would serve no purpose were it not for the fact that the type parameters must be bound to type constants or other type parameters. As a result, a strongly typed procedure with no type parameters can be derived from each invocation of a procedure which does take such parameters. The topic of strong typing is discussed in the next section.

[66] This is a simplification. It is common for languages to include a number of rules for coercing expressions from one type to another. This set of rules is usually fixed by the language. EL/1 is a notable exception, as it allows a different coercion function to be associated with each type.

- **Parametric variation.** Our four subject type constructors always produce types, never parameterizable constructors. The effect of parametric variation constructors can be programmed by having the parameters of type constructors reconceptualized as attributes of objects. E.g., rather than having a type "array of integer", one would define the abstract type "restricted-array" with a `domain-type` operation that associated a type with each instance. Then any operation which resulted in adding new members to the structure would perform runtime testing to verify that each new member is an instances of the `domain-type`.

- **Operation support.** In Zetalisp, Loops, and Smalltalk-82, operation support declarations can be realized by predicates which succeed iff the type of a given object has an abstract operation of a given name. Testing if an object *Obj* has operation *Op* could be done as follows:

    - Zetalisp: `(send` *Obj* `:operation-handled-p` *Op*`)`

    - Loops: `(<-` *Obj* `Understands` *Op*`)`

    - Smalltalk-82: *Obj* `respondsTo:` *Op*

  Such declarations are not possible in Star Mesa, because the generic operation names of a given trait are determined by convention rather than syntax or data structure.

Each of our four type systems also supports a new kind of declaration, which we will refer to as a **ancestor** declaration. Ancestor declarations are specified by one parameter, a type. An object O of type T satisfies a ancestor declaration of type T' iff the type of O is T or an ancestor.

Here is how ancestor declarations are realized in our four languages:

- Zetalisp: `(typep` *Obj Tname*`)`

- Loops: `(<-` *Obj* `InstOf!` *T*`)`

- Smalltalk-82: *Obj* `isKindOf` *T*

- Star Mesa: `TM.Carries [`*Obj*`,` *Tid*`]`

Ancestor declarations make sense only if types are identified with sets of properties of instances (i.e., "type specifications" (e.g., [LiskovBerzins 79, GuttagHorowitzMusser 78, GuttagHorning 78, Guttag 80, GuttagHorning 80]) such that if a type T1 is a parent of type T2 then the specification of T1 also holds for T2. If this is the case, then programs which assume a given variable satisfies the specification associated with a given type can associate that variable with an ancestor declaration of that type. Of course, since this technique assumes that (a) the specification of the ancestor is actually satisfied by the subtype and (b) the program which uses the ancestor declaration only depends on the properties of the ancestor's specification, the absence of declaration-violation errors in no way guarantees program correctness.

A final difference between type declarations as described above and those in our four subject languages is that they are dynamic rather than static. In other words, it is not valid to assume that if an object denoted by a given variable satisfies a given declaration, then the same declaration will hold for the lifetime of the variable. This is because there are no legality rules which prohibit programs for which this assumption does not hold.

In order to verify that a variable will always satisfy a particular declarations, declaration testing must be performed after each statement which might have modified the variable. Unfortunately, in languages which allow access to environments as data structures, the variables which might be changed by a given statement is quite large. For example, the Lisp statement

```
        (set (f x) (g x))
```

might affect any variable in the present environment.[67] If constructs such as `set` and `eval` are avoided, a reasonable degree of protection is obtainable, albeit at the cost of runtime computational overhead.

8.2 Type checking

8.2.1 Our subject languages are not strongly typed

The most fundamental difference between type checking in conventional abstract-type systems and type checking in our subject languages is that *expressions are not strongly typed.* In other words, assuming that the free variables of an expression are bound to objects of particular types does not make it possible to associate a type with the expression such that all evaluations will

---

[67] Of course, it is likely that only a subset these variables were intended to be affected by a given statement. However, if the program does not behave as intended, this fact may be of little use. Unfortunately, the diagnosis of misbehaving programs is one of the principal reasons for performing type checking.

produce instances of the type. As a simple example, knowing that the type of X is "cons cell" allows nothing to be inferred about the type of the objects produced by `(car X)`.

Strong typing is desirable for two basic reasons:[68] [69]

1. Strong typing makes it possible to guarantee through program analysis that no procedure invocation will result in the type of an actual parameter being incompatible with a declaration for the associated formal. The result is faster and more reliable detection of type errors, with no associated computational overhead.

2. Strong typing allows the optimization of tight binding of abstract operations to procedures. The result is the elimination of the computational overhead of type testing for operation invocation.

The price of strong typing is that it is impossible to create "heterogeneous" data structures: data structures for which different invocations of the same accessor with identically-typed parameters can produce objects of different types. For example, if an array A could contain instances of `symbol` and `number`, different invocations of the of the accessor `nth` could return instances of different types. As a result, expressions of the form `(nth A i)` could not be associated with a unique type.

To be sure, the strong typing discipline *does* allow structures which contain instances of a single "discriminated union" type (e.g., Clu's `any` and `oneof`, Russell's `union`, ML's disjoint sum, the variant records of Euclid and Mesa) from which objects of different types can then be extracted. The pragmatic difficulty with using this technique of "simulated heterogenity" is that

---

[68] In systems which do not maintain the type of an object as part of the representation, static type checking is the only way in which type errors can be detected. Since such errors can compromise the integrity of the language implementation itself, (e.g., the "taking the cdr of an atom" scenario of [Milner 78]), strong type checking is especially important.

However, given an implementation where types *are* part of the representation, safety can be guaranteed by runtime type-checking, implemented in some combination of hardware and software. The type ssytems of EL/1 [Wegbreit 70, Wegbreit 74] and MDL [GalleyPfister 79] were early realizations of this latter approach; Bishop's concept of "self-evident data" [Bishop 77] is also relevant. For all of our subject types, representations of objects include their type.

[69] A further advantage of strong typing occurs in langauges such as Mesa where a pointer can be constructed to the memory location holding the result of an arbitrary expression. Strong typing of pointer expressions guarantees that manipulation of pointers is limited to the procedures declared to take parameters of the appropriate pointer types. If the set of these procedures is limited to those defined by the type definer, allowing users to have or create pointers to data structures cannot result in their being able to perform representation-level manipulation. Thus, no "protection" hardware or software is needed to ensure isolation between the definer and users of an abstraction.

allowing instances of a new type to be added to the structure requires the modification of all procedures which discriminate on the underlying type. This is undesirable in small systems and untenable in large ones.

8.2.2 Multiple inheritance is compatible with strong typing

Is strong type checking incompatible with those of type definition by inheritance? Although the details are not presented here, it is possible to define type constructors which use inheritance but whose operations are strongly typed. The simplest technique for doing so is to use the Star Mesa approach of a single record component per ancestor type. For each abstract operation defined by inheritance a new method is constructed. This method will invoke the appropriate abstract operations of a given ancestor on the representation component which contains an instance of the ancestor type.

To illustrate the idea, consider the definition of the **bh-point** of our example scenario. The representation for **bh-point** would have four instance variables: one for **point**, one for **bounded-point**, one for **history-point**, and one for a record containing **bhlist** and **bhtail**. The synthesized creation operations would ensure that the instance of **point** used in the representation of a **bh-point** BP was identical to the one used in the instances of **bounded-point** and **history-point** used in the representation of BP.

To see how a method would be synthesized, consider the definition of the **move** method for **bh-point** as sequential invocation of the **move** of **bounded-point** and **history-point**. The constructed method would first invoke the **move** of **bounded-point** on the **history-point** component, then invoke the **move** of **history-point** on the **history-point** component. Since new methods are constructed for each abstract operation of the new type, and since these methods only invoke operations of ancestor types on instances of the same type, the requirements for strong typing are satisfiable.

The conclusion is that the benefits of multiple inheritance for type design (i.e., the advantages described in section 2.2) can still be obtained in a strongly typed language. However, the disadvantage of not allowing heterogeneous structures still remains.

8.2.3 Inheritance-based type definition suggests a new kind of type checking algorithm.

In the design described above, the fact that a type was defined by inheritance is invisible to the type checking algorithm. An even more interesting idea is to liberalize this algorithm so that instances of a type satisfy declarations of any ancestor type. For example, instances of **bounded-point, history-point** and **bh-point** would all satisfy a declaration of `point`. The effect is that all type declarations are analogous to the ancestor declarations described above.[70]

As a result of this extension, the association of a type with an expression by the typing algorithm allows the conclusion that all evaluations of the expression will produce instances of a descendant type. Since all descendant types have at least the abstract operations of the parent, the invocation of any of these cannot cause a declaration-failure error For example, **create, location**, and **move** are all applicable to any descendant of **point**.

Thus, as in the strong typing approach, static type checking can guarantee that no declaration errors can occur during program execution. The real impact of our extension is that heterogeneous structures can now be used, so long as (a) the type of each element of the structure is a descendant of a given type, and (b) only operations of this type are applied to elements obtained through the structure's accessors.

For example, one could have an object of type "`array` of `point`," which could contain instances of `point`, `bounded-point`, `history-point` and `bh-point`. The only valid operations on objects obtained from such an array would be the operations associated with `point`. These operations would be guaranteed to be type-correct.

[70]This is certainly not an original idea. For example, personal communication with G. Curry indicates that a design for a modification of Mesa's type-checking algorithm to use ancestor declarations was created but never implemented.

Thus, it is possible to design a statically type-checkable language which includes inheritance-based type definition *and either* conventional strong typing or the liberalized version described above. The tradeoff faced by the language designer is between the increased performance of the former (because operation invocations can be tightly bound to procedures), and the increased flexibility of the latter (due to the ability to program with heterogeneous structures).

However, a significant weakness of this approach is that there is no obvious relation between types such as "`array` of `object`" and "`array` of `point`." Although it may seem that arrays of points are simply specializations of arrays of objects, it is not always appropriate to use instances of the former in contexts declared to take instances of the latter. In one example (due to P. Curtis), it would be inappropriate to pass an array of points as the first parameter of the procedure:

```
procedure (p: array-of [object], o: object)
 p[1] := o
```

Typing algorithms which avoid such pitfalls are an active area of research. A key idea is for "declarations" to be given in terms of the types of operations required to be defined for the type, rather than in terms of a type itself.

9. Summary, Conclusions, and Future Work

We have now covered a considerable amount of technical ground. But what does it all mean? The following is an attempt to sum up the important points of what we have seen.

Section 9.1 outlines the major similarities and differences in the realizations of inheritance-based type definition among our four languages. Section 9.2 describes the "bottom line" impact of the differences on the amount of work required to create new types. Finally, section 9.3 describes how the work described here can be extended.

9.1 How definition by inheritance is realized

9.1.1 Algorithms for generic operation set computation

9.1.1.1 The common properties

The generic operation sets of Zetalisp flavors, Loops and Smalltalk-82 classes, and Star Mesa traits are commonly characterized as being defined using "inheritance". We described three properties of operation set construction which are shared by each of these four kinds of types. These properties can serve as a reasonable characterization of the concept of "defining types by inheritance" in programming languages.

1. *Definition through parents and local operation sets.* The generic operation set of a type is defined by an algorithm which involves two fundamental properties of the type: its **parents** (a set of types) and a **local operation set** (an association between names and procedures).

2. *Operation set synthesis.* The names of the generic operations of a type are a superset of the names of the generic operations of all ancestors of the type. Some method will be computed for each of these "inherited" operations.

3. *Consistent method inheritance.* If (a) all parents which define a given generic operation use the identical method and (b) the new type has no local method, then the new type inherits the parents' method.

A key observation is that *inheritance is a property of an algorithm, not an algorithm itself.*

9.1.1.2 The differences

The most fundamental difference among our four languages is whether the algorithm for computing generic operation sets from local operation sets is fixed by the language or programmer-specifiable. In Smalltalk, Loops, and Star Mesa, a single algorithm is used to compute the method for all operations of all types. In Zetalisp, programmers can specify an arbitrary procedure to be used to compute the method for each operation of each type. A number of such procedures are predefined in the language; the subsequent discussion will be limited to the algorithms embodied in those procedures.

A second important difference among our four languages is whether the algorithms used can *construct* new methods as well as *select* methods from local operation sets. The Smalltalk, Loops, and Star Mesa algorithms always produce a local method of the type or an ancestor. But the Zetalisp algorithms can create new methods which invoke selected local operations of the type and its ancestors. If the method-computation algorithm produces an acceptable method, the result is the reduction of programming effort. This was demonstrated in the example of chapter 2, where the generic methods for `:move` and `:init` of `history-point`, `bounded-point`, and `bh-point` were constructed, as were the `:set-min` and `:set-max` methods for `bh-point`.

A third difference is the relationship of the local operation names to the generic operation names. In Smalltalk, Loops, and Star Mesa, each local operation of a type is also a generic operation. The problem with this approach is that defining multiple local operations which are relevant to a single generic operation requires the use of extraneous generic operations, e.g., the "partial" procedures used in chapter 2 and discussed in section 5.2.5. In Zetalisp, the names of the local operations and the generic operations are disjoint, and defining a new local operations need not result in the availability of a new generic operation. Thus, all generic operations in the Zetalisp realization of the `point` scenario correspond to the abstraction-relevant operations.

A final difference is whether programming conventions must be followed in order to have the specified algorithm carried out. This is the case in Star Mesa; failure of the designer of a type to follow the specified conventions can result cause the generic operation set of the specified type *and its descendants* to be computed in an unpredictable way.

9.1.2 Generic method invocation

9.1.2.1 The common properties

In contrast with more conventional "abstract type" approaches, neither the type of an object nor a partial specification of its generic operation set is required for generic invocation to occur. The advantage is that less work is required to create and modify programs, since no type declarations need be initially given or subsequently changed. This disadvantage is that there is no guarantee that the type of the generic parameter will always include an method for the specified operation; the fact that this will never occur must be determined by the same means as other arbitrary properties of programs. A promising line of research involves extending the ML technique of "type inference" to languages with less stringent requirements for free variable typing (recall the discussion of "heterogeneous" data structures in chapter 8) and generic invocation.[71]

A second common property is that the generic method for a given operation of a given type can be invoked on instances of descendant types. Again, this contrasts with conventional abstract-type paradigm, since procedures declared to accept instances of a specific type cannot be passed instances of any other type. The advantage is that avoiding redundancy in method definition is straightforward; in the conventional approach, it is possible only through considerable circumlocution, if at all.

[71]In personal communication, Pavel Curtis of Xerox PARC reports substantial progress on this front.

9.1.2.2 The differences

An important difference between generic invocation in Star Mesa and our other three languages is the treatment of nonexistent-operation errors. In Zetalisp, Smalltalk, and Loops, a missing operation will be detected by the generic invocation mechanism; language-specific "error handling" action ensues. But in Star Mesa, nonexistent-operation errors will not be detected when "production mode" compilation is indicated. Instead, an unpredictable component of the storage used to represent the trait will be treated as a procedure and executed. Needless to say, debugging becomes considerably more difficult under these circumstances.

The second important difference involves the way "ancestor-generic invocation", i.e., an invocation of a generic method of a type when the generic parameter is an instance of a descendant type. Here is how ancestor-generic invocation is realized in our four languages.

- In Zetalisp, ancestor-generic invocations require (a) computing the method to be invoked, (b) computing the flavor of which that method was a local operation, and (c) computing a mapping table object appropriate for the type of the generic parameter and the flavor computed by step (b). Step (a) is trivial, via the `:handler` function specs described in chapter 3. Step (c) is straightforward, but requires the use of presently undocumented procedures. Step (b) is the most problematic, since it can require a search through the representation of the procedure object. A significant improvement to Zetalisp would be the provision of a documented, efficient procedure for ancestor-generic invocation.

- In Star Mesa, ancestor-generic invocation involves obtaining a procedure from the trait storage of the ancestor type, then applying the procedure to the desired parameters. It is subject to the same kinds of undetected errors as in standard generic invocation.

- In Smalltalk-82, the two problems of Zetalisp and Star Mesa are eliminated. Ancestor-generic invocation is possible through the use of *class.op* operation names. Furthermore, if the specified operation is not defined for the type, an error is signalled. For example, a generic invocation using `Point.display` as the operation name will execute the `display` operation of the class `Point`. The ability to support such invocations is a key capability of Smalltalk-82 not embodied in Smalltalk-80.

  Two additional means for ancestor-generic invocation are also available. One of these, available via operation names of the form `super`.*op*, invokes the unique generic method of all parent types, if any such method exists. The second, available through `all`.*op* operation names, invokes all generic methods for a given operation defined in any parent. The point of using these additional invocation forms is to enable operations to be redistributed among ancestors of a type without requiring the modification of existing invocations. The latter form has the additional advantage of making the enumeration of the relevant types unnecessary, thus (a) reducing the work required to initially define the method, (b) eliminating the need to track the addition and removal of relevant methods from ancestor types, and (c) allowing the method to be appropriate for descendant types that have additional ancestors which define methods for the operation.

- All of the capabilities in Smalltalk-82 are also available in Loops. Ancestor-generic invocation is embodied in the **DoMethod** procedure. The two alternate forms for ancestor invocation are available through the **SendSuper** and **SendSuperFringe** procedures.

9.1.3 Specification and invocation of the local operation set

9.1.3.1 The common properties

With respect to specifying the local operation set, the only characteristic our four languages have in common is that they are determined by the side effects of program execution rather than through the syntax of the language. This characteristic, which represents a difference from conventional "abstract type" designs, means that type checking algorithms based on syntactic analysis cannot be done.

Notice that there is nothing in the paradigm of type definition by inheritance that prevents the use of syntactic rules to specify the local operation set. Furthermore, the ability to determine the current local operation set of a type computationally means that mechanical analysis is still possible. Type checking in the context of dynamic modification requires that (a) "program state" rather than program text be examined, and (b) modifications to the definitions of types, procedures, and global variables be followed by incremental analysis of the modifications. See [Cheatham 81] for a discussion of an operational system which performs such analysis.

9.1.3.2 The differences

The most important difference in local operation set specification is between Star Mesa and our other three languages. In Star Mesa, it is impossible to add a new local operation to a trait. Instead, a new Mesa "binary configuration description" which represents the entire application program must be created. The adverse impact on program development time is clear.

A second difference between Star Mesa and our other three languages is that the local operation set of a type cannot be computed once the type has been defined. A significant disadvantage is that it is impossible to have programming tools which allow local operation sets to be examined and/or manipulated. For example, it is impossible to write a program which can answer the question "which types define local methods for operation O?"

A third difference is in whether the local operation set of a type can be determined by the examination of a single piece of program text associated with the type. This property holds in Star Mesa: if the specified conventions are followed, the local operation set of a trait can be determined by examining its local trait initialization procedure. It is not the case in our other three languages, where local operations can be added and removed by invocations which can appear anywhere in the program. As described above in regard to type checking. the availability of computational support ameliorates the increased complexity of local operation set determination.

9.1.4 The instance variable operations

9.1.4.1 The common properties

As described in chapter 5, the primitive operations on instances of each of our subject types are the familiar "property list" operations: associating a value with an instance variable name and retrieving the value associated with an instance variable name. These operations are relevant to the type system because (a) types are associated with instance variable names, and (b) it can be made to appear that all instances of a type have values associated with all instance variable names of the type.

The key aspect relevant to type definition via inheritance is that the instance variables names which are meaningful for a type are defined to be the union of a set of local instance variable names associated with the type. As a result, if a method M in the local operation set of a type is invoked on an object O which is an instance of a descendant type, all instance variable operations used in M will be meaningful when applied to O.

In addition to the preceding semantic properties, there are also some commonalities regarding how the instance variable operations are invoked. Any procedure can apply the instance variable operations to an instance of any type. This contrasts with the mainstream of the "abstract type" languages, where no access to the representation of a type is allowed outside a given syntactic scope.

Any assessment of this capability is bound to be controversial. All would agree that it is clearly desirable to limit as much as possible the use of instance variable operations on instances of a type outside the local methods of the type. The obvious benefit is that the the representation of the type can be changed with far less impact on its users.

Furthermore, it is even desirable for the *methods* of a type to eschew the use of instance-variable operations, if possible. The benefit here is the increased generality of methods thus defined. For example, the realization of the **display** operation for each of the four kinds of points would be usable regardless of the way in which the generic operations were implemented.

However, the significance of the lack of an mechanism for *enforcing* adherence to any principle of programming methodology is not clear. This is especially true because in each of our four languages, the avoidance of a specific set of syntactic forms and/or procedures will guarantee that this principle will not be violated.[72] Thus, the principal effect of such a procedure would be to prevent programmers from doing something they explicitly want to do.

[72] In Star Mesa, one must use instance variable operations only on the generic parameter of local method definitions.

9.1.4.2 The differences

Of the four realizations of instance variable operations which we have seen, Star Mesa is clearly the simplest. Its most significant shortcoming is that, in "production" compilation mode, the use of an instance variable name not associated with the object will not be detected as an error. Instead, an unpredictable component of the storage used to represent the object will be returned. Furthermore, even if a correct instance variable name is given, references to instance variable values are represented as untyped Mesa pointers, and can thus be coerced into any other type. Given Mesa's reliance on compiler-based type checking, coercion into an inappropriate type can result in unpredictable modifications to unpredictable memory locations.

A less significant disadvantage derives from the fact that precisely one instance variable is associated with each type. This requires two levels of indexing if more than one object is required for the abstract representation associated with a type (e.g., the `min` and `max` components of the `BoundedPoint` definition in figure 2-24). The negative result is the increased verbosity of programs.

The realization in Smalltalk-82 is an improvement over Star Mesa, since any number of instance variables can be associated with a type. However, it does have the disadvantage of not allowing instance variable names of parents to intersect. Smalltalk is also unique among our four languages in that it allows instances of classes to have numeric instance variable names specified when the instance is created. This is one way to introduce the concept of an array in a language where any object can have instance variables with symbolic names.

The key distinguishing aspect of Zetalisp's realization of the instance-variable operations is its considerable support for the property-list style of object initialization described in chapter 5. The fact that initialization property lists are automatically checked for missing or extraneous keywords allows significant error detection to be achieved with little work by programmers. In contrast, checking for extraneous keywords in Smalltalk and Loops requires a separate procedure for each type. In Star Mesa, the use of keyword parameters allows extraneous keywords to be statically detected, so long as a different interface to the initialization procedure for each type is defined. This in turn requires that creation procedures for each type invoke the initialization procedures of each ancestor type explicitly, and with a different set of parameters. The result is increased verbosity and increased inter-type dependence.

Instance variable operations in Loops have two distinguishing attributes. First, objects can have more instance variable names than those defined for their type. The primary significance of this approach is that the instance variable mechanism can be used for attributes not meaningful for all instances of a type. Thus, it allows the instance variable operations to be used to associate a conventional "free-form" property list with each object (e.g., the `accessCount` and `updateCount` variables used in the active-value example of section 5.5). It is also sometimes helpful for program development, in the case where an object should have been an instance of a type with additional instance variables. However, since the same procedure is used to and to replace a value for an existing property,

However, the instance-variable procedures documented in [BobrowStefik 83] which allow objects to appear to have a superset of the instance variable names of their types (i.e., `GetValue` and `PutValue`) will *never* detect "inappropriate instance variable name" errors. Instead, they add new object-specific property/value associations or return a default value. It is trivial to define procedures which signal an error if the instance variable were not already defined; this would allow instance variable operations in Loops to perform the same kind of error checking as in Smalltalk and Zetalisp.

A second major difference between Loops and our other three languages is that instance variable operations can invoke programmer-specified procedures. The principal significance of this capability is that independently-defined procedures can monitor changes to the values of specific instance variables of specific objects. This is useful for ad hoc tracking of selected aspects of program state (e.g., for debugging or performance monitoring) as well as being used as a fundamental principle of program architecture (i.e., the "access-oriented programming" of [Stefik et al. 83a]). In chapter 5, we saw how to extend Loops so that more flexible object-specific computation can be accomplished.

9.2 The pragmatic impact on type construction

The principal motivation for the use of inheritance-based type definition is the reduction of the work required to add new types or modify systems of existing types. How do the algorithms provided in our four languages compare with respect to the enhancement of programmer productivity?

This is a difficult question, and one which is not answered definitively by this report. However, it is possible to examine some common cases of method definition and identify the impact of the different approaches to operation set computation which we have seen. We first describe the situation when all parents for which an operation is relevant have the same method for it. We then examine the situation when different parents define different methods for an operation, but one or more of these methods can be used to define the method of the child type. Finally, we consider the situation in which a composition of the *local* methods of the type and its ancestors is appropriate.

9.2.1 The pragmatic impact in the absence of parent conflict

9.2.1.1 ... when the parents' method can be inherited

The multiple-inheritance type constructors in each of our subject languages satisfy the consistent method inheritance property. Thus, in the situation where (a) all parents of a type T which define a method for an operation O use the same method, and (b) this method is adequate for T, the definer of T need provide no method-definition information for O. This is the most fundamental benefit of type definition by inheritance, and is obtained in each of our subject languages.

As a corollary of the above, if the generic operation names of a the parents of a type are disjoint, all methods of all parents can be inherited by the new type. As a result, if the methods of the parents are appropriate for the new type, its definition can be accomplished by simply enumerating the parent types. Thus, the definition of types which combine "independent aspects of behavior" of ancestor types is facilitated in all four languages.

As a further advantage of the consistent method inheritance property, adding a new operation to the generic operation set of a type will be automatically reflected in the operation sets of descendant types. Thus, when consistent extensions to the operation set of a type are made, the work required to incorporate the extension in descendant types is reduced, as is the need for inter-programmer communication. This latter aspect is increasingly significant as the size of programs and the number of programmers increases.

However, it is important to realize that consistent extensions to a type are not necessarily consistent extensions to all descendants. For example, adding a new operation O to a type will not result in a consistent extension of a descendant if that descendant already has an operation O. Furthermore, extending the functionality of a given operation may not be consistent with functionality extensions made by descendants. As a simple example, consider a descendant of a conventional **array** type whose indexing operations perform "modulo the length of the array" arithmetic.[73] If **array** type were subsequently modified so that out-of-bounds indexing returned a "default" value (cf. the use of **NotSetValue** in the instance-variable operations of Loops), the modification would be a consistent extension for **array**, but not for the descendant.

Yet another kind of problem arises from making consistent extensions to the representation invariant of a type (e.g., [Hoare 72, London et al. 78]) can result in inconsistencies in the representation invariant of descendant types. For example, suppose the type **collection** is represented as a list in which order does not matter, and the type **specific-collection** is a

[73]E.g., for a three-element, zero-origin array A containing 0, 1, and 2, A[4] = A[1] = 1.

descendant in which the member-addition operation guarantees that the members of the list are maintained in order. If the designer of **collection** decides to refine his implementation by maintaining the list in a different order, the extension is inconsistent for **specific-collection.**

Why is it appropriate for the local methods of a type to depend on representation-relevant properties of ancestors? The answer is simple: *sharing implementation-dependent code reduces redundancy in programs.* As the complexity of types and their implementations increases (the Lisp Machine window system unavoidably comes to mind), the benefits of sharing implementation-dependent procedures of other types may well outweigh the cost of increased inter-module dependence and inter-programmer communication.

9.2.1.2 ... when the parents' method can be invoked

In many cases, the computations performed by methods of a type are a subset of the computations which must be performed by methods of descendant types. For example, the desired behavior of the **display**, **move**, **set-min**, and **set-max** operations of **bounded-point** is an augmentation of the behavior of the corresponding operations of **point**. Thus, it is possible for methods of types to invoke the corresponding methods of ancestor types, thereby avoiding program redundancy.

However, as illustrated in section 5.2.5, the possibility that types can have more than one parent leads to the following key observation:

> *If the methods of two ancestor types contain invocations of the method of a common ancestor, the invocation of both methods will cause redundant invocation of the common ancestor's method.*

For example, since the methods for **display** of **bounded-point** and **history-point** both invoke the **display** method of **point**, they cannot be both be invoked by the **display** method for **bh-point**. The problem is that redundant invocation of ancestor methods (in our example, `display` of `point`) can occur. Thus, the preceding observation leads to the following design principle:

> *Code which may be useful for methods of descendant types should not be defined in procedures which invoke methods of ancestor types.*

The consequence of this design principle is as follows. When it is desired for a method of a type to invoke the corresponding method of an ancestor type, the appropriate technique is to define auxiliary "augmentation" procedures which perform the additional computation not performed by the ancestor. The method itself can then invoke the ancestor's method in conjunction with the augmentation procedures. The payoff is that the augmentations can be invoked by any descendant, with no possibility of redundant invocation of the ancestor's method.

The methods for **display** of **history-point** and **bounded-point** illustrate the use of this principle. The former invokes a separate procedure to perform the "augmentation" of printing the history list. The latter does the same for printing the minimum and maximum. As a result, the **display** method for **bh-point** can invoke both of the augmentation procedures without causing the redundant invocation of the **display** method of **point**.

How well do our four languages support this programming technique? In Star Mesa, Smalltalk-82, and Loops, both the augmentation procedure and the method itself must be defined by programmers as separate procedures. The method of the new type will invoke the augmentation procedure before or after it invokes the parents' method. Smalltalk-82 and Loops offer an advantage over Star Mesa in the definition of the latter invocation, since the identity of the parent type need not appear in the invocation form. This is accomplished through *super.op* invocation in Smalltalk-82 and the **SendSuper** procedure of Loops. The advantage is that changing the name of the parent (or the parent itself) need not result in the need to modify these invocation forms. This is especially useful because the error of neglecting to update an invocation would otherwise go undetected. Unfortunately, such errors *do* go undetected in Star Mesa.

In Zetalisp, methods which invoke methods of parents and local augmentation procedures can be automatically constructed by the flavor system, e.g., by defining the augmentation procedures as **:whopper** methods. The result is that less work is required to define the new type in such a way that descendant types can invoke the augmentation procedure. This is the principal advantage of Zetalisp's realization of definition by inheritance over that of our other subject languages.

9.2.2 The pragmatic impact when parent conflict is resolved via choice

9.2.2.1 ... when a parent's method can be invoked

Suppose that different parents have different methods for an operation, but a method of one of them can be used for the new type. The ideal behavior of an inheritance mechanism would be to choose the method desired by the programmer. Given the infeasibility of mind-reading, we can examine how well the mechanisms satisfy two more specific criteria.

- If a unique ancestor exists which is a descendant of all other ancestors for which a method is defined, the method of that ancestor is chosen. As discussed in section 6.3.2, this "most specific method" property increases the likelihood that the method chosen will be appropriate.

- If the algorithm selects a method which is not the one desired by the programmer, it is possible to override the choice knowing only the identity of a parent which uses that method. This "overridability" property is important because it provides programmers with a straightforward response whenever the "mind-reading" of the method computation algorithm fails.

The most-specific-method property is satisfied by the Loops algorithm and, if the appropriate conventions are followed, for Star Mesa trait definitions. It is never satisfied for Smalltalk-82, since algorithm for operation set computation always creates an error-signalling method when parent methods conflict. In Zetalisp, the property may or may not be satisfied, depending on the order in which parents are listed in the flavor introduction form and the use of `:default` methods.

The overridability property is satisfied by Star Mesa and Loops. Overriding method choices in Star Mesa is accomplished by having the local trait initialization procedure of the type extract a procedure from the trait storage of the chosen parent, then assign it to the appropriate component of its own trait storage; see section 7.4 for details. Method choices can be overridden in Loops by (a) using the `GetMethod` procedure on the parent type to obtain a procedure name, and (b) using the `PutMethod` procedure on the new type to define this name as a local method symbol. See section 7.3 for a description of these procedures and the concept of a "local method symbol".

However, the overridability property does not hold for Smalltalk-82 or Zetalisp. It is not possible in either language to state that the method for a given operation of a given type should be identical to the method for any operation of any other type (or, for that matter, of the same type). Instead, it is necessary to define a new procedure which invokes the method of the desired parent; the issues involved in doing so are discussed immediately below. The principal disadvantage of this approach is the overhead in space and time of defining and invoking an extra procedure. The amount of work required is not significantly different than that required for overriding in Star Mesa and Loops.

9 2.2.2 ... when a parent's method can be invoked

If the method of the new type should consist of an invocation of the methods of some number of parents, the following property is relevant:

- It is possible to invoke the method for an operation of a parent knowing only the operation name and the identity of the parent.

This property is satisfied by Smalltalk-82 (via *class.op* invocation), Loops (via the `DoMethod` procedure), and Star Mesa (by invoking a procedure extracted from the trait storage of the chosen parent). As described in section 3.4.3 and summarized in section 9.1.2.2, this property can be achieved in Zetalisp in an inefficient way through the use of undocumented features of the implementation. As we have already noted, the obvious solution is the addition of a supported, efficient procedure for such invocations.

However, notice that each of these approaches shares the unfortunate characteristic of embedding the names of types in methods of child types. As described in the previous section, this characteristic increases the amount of work required to modify the parents of a type, as well as making the result of such modification more likely to cause an error not detected by the language implementation.

9.2.2.3 ... when all parents' methods can be invoked

It is sometimes useful for the method for an operation of a type to invoke the methods for that operation of *all* parents of the type. This kind of scenario captures another sense of "independent" type definitions. The same abstract operations are relevant for the parent types, but their methods can be composed without destructive interaction.

When this technique is relevant, the important property is the following:

- The invocation of the methods of all the parents of a type can be accomplished without enumerating the parents.

The utility of this property was described in section 9.1.2.2.

Again, Smalltalk-82 and Loops have this property, (via `all.`*op* invocation of the former and the `SendSuperFringe` procedure of the latter), while Star Mesa and Zetalisp do not (as in the previous situation, enumeration of the parent types is required). And again, it would be straightforward to add a procedure to Zetalisp which would carry out such invocations. But before moving on to the final case, two additional points are with noticing.

First, as discussed in section 5.2.5 and 9.2.1.2, procedures which contain such invocations will often not themselves be invocable by methods of descendant types. We will shortly examine the utility of "all parents" invocation tc a programming methodology which facilitates such sharing.

Second, although the methods constructed by the standard Zetalisp algorithms never invoke methods of ancestor types, they often have the effect of "interleaving" execution of ancestor methods. For example, suppose two types have methods defined via `:daemon` combination on `:whopper` and untyped local methods. If a child of both types uses `:progn` combination to construct the method, all four local methods will be invoked. Thus, if the methods of the two parents are "independent" in even a stronger sense than the two described above, the resulting method will behave as if the methods were sequentially invoked.

9.2.3 The pragmatic impact when all local methods of parents can be invoked

Sections 5.2.5 and 9.2.1.2 described the most useful way to define a method which invokes procedures used in the invocation of methods of ancestor types. The key idea was to encapsulate the computation not performed by the ancestor in an augmentation procedure which does not call the methods of any ancestor. The actual method calls the augmentation procedures of the type and all ancestors, in addition to the method of the first type which defined a method for the operation. In the example of chapter 2, this technique was used to define the methods for initialization, **move**, **set-min**, **set-max**, and **display**.

The following is an assessment of how our subject languages support this style of method design.

- In Star Mesa, augmentation procedures are realized as procedures defined in a `TRAIT` module but not used as a method for any operation. Invoking the augmentation procedures of a type and all ancestors requires the enumeration of all ancestor types for which such procedures are defined. The disadvantages described above still apply.

- In Smalltalk-82, the augmentation procedures are realized as methods for some operation not germane to the abstraction, e.g., `partialSetmin:`. As in Star Mesa, invoking the augmentation procedures of a type and all ancestors requires enumerating each of the relevant types. It should be clear that `all.`*op* cannot be used to invoke all the augmentation methods, since it invokes the methods of all *parents*, not all *ancestors*. For example, if local methods for some operation O were defined for `Point`, `BoundedPoint`, and `HistoryPoint`, an invocation of `all.O` in a local method of `BHPoint` would not invoke the local method for O of `Point`.

  The above analysis indicates that an `allLocals.`*Op* invocation form with the obvious semantics would be a useful addition to Smalltalk-82. Adding this new form requires knowledge of how the multiple-superclass extension was implemented. [BorningIngalls 82b] contains a brief overview of the technique used; the key idea is to use the handler for "nonexistent operation" errors to construct a method with the appropriate semantics, then compile and install it in the appropriate class. The reason why adding a new invocation form is not trivial is that the addition and removal of methods of a class must trigger the recompilation of appropriate methods of descendant classes. A considerable amount of bookkeeping is required to do so.

- In Loops, the augmentation procedures are also realized as methods for abstraction-irrelevant operations. The key difference from Smalltalk-82 is that it is straightforward to define a procedure which invokes all local methods of the ancestors of a type; such a procedure is defined in figure 2-5. Thus, ancestor enumeration is no longer required. However, the need to define abstraction-irrelevant methods is still present,[74] and without a more sophisticated design than given here, the search through the type hierarchy to compute the relevant local methods will be expensive.

- In Zetalisp, the augmentation procedures can be defined in a number of different ways. As we saw in section 6.5, the most general approach is to define them as `:whopper` methods. But this is simply one alternative; for example, the scenario of chapter 2 uses `:before` and `:after` methods as well. Whatever the approach, the key point is that the method definition algorithms can construct the methods which invoke all augmentation procedures of ancestors.

  This approach does not involve the definition of pseudo-private generic operations. It also does not require an additional definition for the first method which requires sharing. The problem is that the algorithm does not necessarily allow top-down invocation, due to the algorithm used to compute the components list. Section 6.5.4.2 described a straightforward modification to Zetalisp which solves this problem.

9.3 Future work

The work reported here can be extended in a number of ways.

9.3.1 Immediate projects

- The most obvious first step would be to implement the suggested changes in the subject languages and attempt to evaluate the results. Especially interesting additions are adding object-specific methods to Loops (see section 5.5.4) and `(:whopper N)` methods (described in section 6.5.3.3.3 to Zetalisp.

- Second, make the operation set and instance variable abstractions complete and precise.

- Third, the analysis in this paper has focused solely on the impact of the four languages on the programming process. But it is unrealistic to ignore the computational resources needed for execution of the programs thus constructed. Description and analysis of how the type systems described here are implemented would be worthwhile.

- Fourth, it would be useful to extend the example of chapter 2 to include alternative realizations of the same abstraction and "mixin" flavors which can be applied to any of them. For example, we could have two-dimensional points implemented by either x/y or rho/theta coordinates, and have "history," "bounded," and "bounds-history" mixins which operate on all of them.

[74]Notice that if the augmentation procedures were simply defined as ordinary procedures, then it would be necessary to enumerate the procedure names in order to invoke all of them.

9.3.2 Long-term goals

- The most important contribution which could be made to the use of inheritance-based type definition is the ability to have static type checking without the need for declarations (see section 9.1.2) for a discussion.

- Another important goal would be to attempt to measure the improvements in programmer productivity supposedly deriving from the use of inheritance-based type definition. The methodological barriers which must be overcome are considerable, since the skill of the programmers, their familiarity with the language and the programming style, their acceptance of the language and style, and the inherent complexity of the programming problem must all be controlled.

- Finally, a satisfying theoretical account of inheritance-based type definition would be invaluable, since the considerable arsenal of mathematical analysis would then be applicable. In all likelihood, this will require a number of years in which unrealistically simple languages are studied and results are gradually pieced together. In the long run, however, the insights eventually obtained through this approach will almost certainly be well worth the wait.

References

[Allen et al 82] Allen, E. M., Trigg, R. M., and Wood, R. J.
The Maryland Artificial Intelligence Group Franz Lisp Environment.
Technical Report TR-1226, Computer Science Dept., Univ. of Maryland, October, 1982.

[Bawden et al. 77] Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., Weinreb, D.
Lisp Machine progress report.
Technical Report AIM-444, M.I.T. Artificial Intelligence Laboratory, August, 1977.

[Birtwhistle et al. 73]
Birtwhistle, G. M., Dahl, O.-J., Myhraug, B., and Nygaard, K.
Simula Begin.
Auerbach Press, 1973.

[Bishop 77] Bishop, P.
Computer systems with a very large address space and garbage collection.
Technical Report TR-178, MIT Laboratory for Computer Science, May, 1977.

[BobrowStefik 83] Bobrow, D., and Stefik, M.
The Loops manual.
Technical Report, Xerox PARC, December, 1983.

[BobrowWinograd 77a]
Bobrow, D. and Winograd, T.
An overview of KRL, a knowledge representation language.
Cognitive Science 1(1):3-46, 1977.

[BobrowWinograd 77b]
Bobrow, D. and Winograd, T.
Experience with KRL-0: one cycle of a knowledge representation language.
In *5IJCAI*, pages 213-222. International Joint Conferences on Artificial Intelligence, 1977.

[BoehmDemersDonahue 80]
Boehm, H., Demers, A., and Donahue, J.
An informal description of Russell.
Technical Report 80-430, Department of Computer Science, Cornell University, October, 1980.

[Borning 81] Borning, A.
The programming language aspects of Thinglab, a constraint-oriented simulation laboratory.
ACM Transactions on Programming Languages and Systems :353-387, October, 1981.

[BorningIngalls 82a]
Borning, A. and Ingalls, D.
A Type Declaration and Inference System for Smalltalk.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 133-141. ACM, January, 1982.

[BorningIngalls 82b]
Borning, A. H., and Ingalls, D. H. H.
Multiple inheritance in Smalltalk-80.
In *AAAI-82*, pages 234-237. American Association for Artificial Intelligence, 1982.

[Brachman 78] Brachman, R.
A structural paradigm for representing knowledge.
Technical Report 3605, BBN, May, 1978.

[BrachmanFikesLevesque 83]
Brachman, R., Fikes, R., and Levesque, H.
Krypton: a functional approach to knowledge representation.
Technical Report Technical Report 16, Fairchild Laboratory for Artificial Intelligence Research, May, 1983.
An abridged version appears in *IEEE Computer*, September, 1983.

[BrachmanSchmolze 84]
Brachman, R., and Schmolze, J.
An overview of the KL-ONE knowledge representation system.
Cognitive Science , Fall, 1984.

[Branquart et al 71]
Branquart, P., Lewi, J., Sintzoff, M., and Wodon, P.L.
The composition of semantics in Algol 68.
Communications of the ACM 14(11):697-708, November, 1971.

[BrodieZilles 81] Brodie, Michael L., and Zilles, Stephen N. (editor).
Proceedings of the workshop on data abstraction, databases, and conceptual modelling.
ACM, 1981.
Published as the January 1981 *SIGART Newsletter* and *SIGPLAN Notices*, and the February 1981 *SIGMOD Record.*

[Cannon 82] Cannon, H.
Flavors: a non-hierarchical approach to object-oriented programming.
Technical Report, Symbolics, Inc. internal memo, 1982.

[Cheatham 81] Cheatham, T. E.
An overvew of the Harvard program development system.
In Hunke, H. (editor), *Software Engineering Environments*, . North-Holland, 1981.

[Curry et al. 82] Curry, G., Baer, L., Lipkie, D., and Lee, B.
Traits: an approach to multiple-inheritance subclassing.
In *SIGOA Conference on Office Information Systems*, pages 1-9. SIGOA, 1982.

[CurryAyers 83] Curry, G.A., and Ayers, R.M.
Experience with traits in the Xerox Star workstation.
In *Workshop on reusability in programming, Newport, R.I.*, pages 83-96. Advance papers, September, 1983.

[Dahl 68] Dahl, O.-J.
Simula 67 common base language.
Publication S-22, Norwegian Computing Center, Oslo, 1968.

[Demers et al. 78] Demers, A., Donahue, J., and Skinner, G.
Data types as values: polymorphism, type-checking, and encapsulation.
In *Fifth ACM symposium on the principles of programming languages*, pages 23-30. Association of Computing Machinery, 1978.

[Drescher 84] Drescher, G.
ObjectLisp.
Technical Report in preparation, MIT Artificial Intelligence Laboratory, 1984.

[GalleyPfister 79] Galley, S. W., and Pfister, G.
The MDL programming language
1979.

[GeschkeMorrisSatterthwaite 77]
Geschke, C. M., Morris, J. H., Jr, and Satterthwaite, E.
Early experience with Mesa.
Communications of the ACM 20(8):540-553, August, 1977.

[Goldberg 83] Goldberg, A.
The Smalltalk-80 interactive programming environment.
Addison Wesley, 1983.

[GoldbergKay 76] Goldberg, A., and Kay, A. (eds.).
Smalltalk-72 instruction manual.
Technical Report SSL 76-6, Xerox PARC, March, 1976.

[GoldbergRobson 83]
Goldberg, A., and Robson, D.
Smalltalk-80: the language and its implementation.
Addison Wesley, 1983.

[GoldsteinBobrow 80]
Goldstein, I., and Bobrow, D.
A layered approach of software design.
Technical Report CSL-80-5, Xerox PARC, December, 1980.

[GoldsteinRoberts 77]
Goldstein, I., and Roberts, R.
Nudge: a knowledge-based scheduling program.
In *5IJCAI*, pages 257-263. International Joint Conferences on Artificial Intelligence, 1977.

[Gordon et al. 79] Gordon, M., Milner, A., and Wadsworth, C.
Edinburgh LCS -- a mechanised logic of computation.
In (editor), *Lecture notes in compter science 78*, . Springer-Verlag, 1979.

[GreinerLenat 80] Greiner, R., and Lenat, D.
A representation language language.
In *AAAI-1*, pages 165-169. American Association for Artificial Intelligence, 1980.

[Guttag 80] Guttag, J.
Notes on type abstraction (version 2).
IEEE Transactions on Software Engineering SE-6(1):13-23, January, 1980.

[GuttagHorning 78]
Guttag, J., and Horning, J. J.
The algebraic specification of abstract data types.
Acta Informatica 10(1):27-52, 1978.

[GuttagHorning 80]
Guttag, J., and Horning, J.J.
Formal specification as a design tool.
In *Seventh ACM symposium on the principles of programming languages*, pages 251-261. Association of Computing Machinery, 1980.

[GuttagHorning 83]
Guttag, J., and Horning, J. J.
Preliminary report on the Larch shared language.
Technical Report, Massachusetts Institute of Technology Laboratory for Computer Science, August, 1983.

[GuttagHorowitzMusser 78]
Guttag, J., Horowitz, E., and Musser, D. R.
Abstract data types and software validation.
Communications of the ACM 21(12), December, 1978.

[Haase 84] Haase, K.
ARLO: the implementation of a language for describing representation languages.
M.I.T. S.B. thesis, June, 1984.

[Hewitt 77] Hewitt, C.
Viewing control structures as patterns of passing messages.
AI Journal 8(3):323-364, 1977.

[HewittAttardiLieberman 79]
Hewitt, C., Attardi, G., and Lieberman, H.
Security and Modularity in Message Passing.
In *First International Conference on Distributed Computing, Huntsville, Ala.*. 1979.

[HewittBishopSteiger 73]
Hewitt, C., Bishop, P., and Steiger, R.
A universal modular actor formalism for artificial intelligence.
In *1973 International Joint Conference on Artificial Intelligence*, pages 235-245. , 1973.

[Hoare 72] Hoare, C. A. R.
Proof of Correctness of Data Representations.
Acta Informatica 1:271-281, 1972.

[Holloway et al. 74]
Holloway, G., Townley, J., Spitzen, J., and Wegbreit, B.
ECL programmer's manual.
Technical Report 23-74, Center for Research in Computing Technology, Harvard University, December, 1974.

[IBM 64] International Business Machines, Inc.
PL/1 language reference manual
1964.

[Ichbiah, J. D., et al. 79]
Ichbiah, J. D., et al.
Preliminary Ada reference manual.
SIGPLAN Notices 14(6A), June, 1979.

[Ingalls 78] Ingalls, D. H.
The Smalltalk-76 programming system: design and implementation.
In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 9-16. ACM, 1978.

[IntelliGenetics 83]
IntelliGenetics, Inc.
KEE User's Manual [Preliminary Edition]
1983.

[Knuth 69] Knuth, D.
The Art of computer programming: fundamental algorithms.
AddisonWesley, 1969.

[Lampson et al. 77]
Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. L.
Report on the programming language Euclid.
SIGPLAN Notices 12(2), February, 1977.

[LauerSatterthwaite 79]
Lauer, H. C., and Satterthwaite, E. H.
The impact of Mesa on system design.
In *Fourth International Conference on Software Engineering*, pages 174-181. IEEE, 1979.

[Lieberman 80] Lieberman, H.
A preview of Act 1.
Technical Report AIM-728, M.I.T. Artificial Intelligence Laboratory, April, 1980.

[Lipkie et al. 82] Lipkie, D. E., Evans, S. R., Newlin, J. K., and Weissman, R. I.
Star graphics: an object-oriented implementation.
Computer Graphics 16(3):115-124, July, 1982.

[Liskov et al. 79] Liskov, B. H., Moss, J. E. B., Schaffert, J. C., Schiefler, R. W., and Snyder, A.
Clu reference manual
1979.

[LiskovBerzins 79]
Liskov, B. H. and Berzins, V.
An appraisal of software specifications.
In Wegner, P. (editor), *Research directions in software technology*, . MIT Press, 1979.

[LiskovZilles 74] Liskov, B. H., and Zilles, S. N.
Programming with abstract data types.
In *SIGPLAN symposium on very high level languages*, pages 50-59. SIGPLAN Notices, 1974.
Published in *SIGPLAN Notices*, April 1974.

[London et al. 76] London, R. L., Shaw, M., and Wulf, W. A.
Abstraction and verification in Alphard: a symbol table example.
Technical Report, Information Sciences Institute, USC, December, 1976.

[London et al. 78] London, R.L., Guttag, J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G., and Popek, G.J.
Proof rules for the programming language Euclid.
Acta Informatica 10(1):1-26, 1978.

[McLeod 79] McLeod, D.
A semantic data model and its associated structured user interface.
Technical Report TR-214, MIT Laboratory for Computer Science, March, 1979.

[Milner 78] Milner, R.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences 17:348-375, 1978.

[Minsky 74] Minsky, M.
A framework for representing knowledge.
Technical Report TR-306, M.I.T. Artificial Intelligence Laboratory, June, 1974.

[Mitchell et al. 79] Mitchell, J. G., Maybury, W., Sweet, R.
Mesa language manual
1979.

[MoonStallmanWeinreb 84]
Moon, D., Stallman, R., and Weinreb, D.
Lisp machine manual
1984.

[Morris 73] Morris, J. H., Jr.
Types are not sets.
In *First ACM symposium on the principles of programming languages*, pages 120-124. ACM, 1973.

[Naur 63] Naur, P. (ed.).
Revised report on the algorithmic language Algol 60.
Communications of the ACM 6(1):1-17, January, 1963.

[Novak 82] Novak, G.
Glisp: a high-level language for AI Programming.
In *AAAI 82*, pages 238-241. American Association for Artifical Intelligence, 1982.

[Novak 83a] Novak, G.
Glisp user's manual
1983.

[Novak 83b] Novak, G.
Glisp.
AI Magazine 4(3):37 ff., Fall, 1983.

[Pitman 83] Pitman, K.
Revised Maclisp reference manual.
Technical Report TR-295, MIT Laboratory for Computer Science, May, 1983.

[Rees et al. 84] Rees, J., Adams, N., and Meehan, J.
The T reference manual (fourth edition)
1984.

[Rentsch 82] Rentsch, Tim.
Object-oriented programming.
SIGPLAN Notices 17(9):51-57, Sept, 1982.

[Reynolds 74] Reynolds, J. C.
Towards a theory of type structure.
In *Lecture Notes in Computer Sciences 19*, pages 408-425. Springer-Verlag, 1974.

[Rich 81] Rich, C.
Inspection methods in programming.
Technical Report TR-604, MIT Artificial Intelligence Laboratory, June, 1981.

[Seybold 81] Seybold, J.
The Xerox Star: a "professional" workstation.
Seybold Report 4(5), May, 1981.

[Shaw et al. 76]	Shaw, M., Wulf, W. A., and London, R. L. *Abstraction and verification in Alphard: iteration and generators.* Technical Report, Information Sciences Institute, USC, August, 1976.
[Shoch 79]	Shoch, J. An overview of the programming language Smalltalk-72. *SIGPLAN Notices* 14(9):64-73, September, 1979.
[Smith 83]	Smith, B. C. *Reflection and semantics in a programming language.* Technical Report TR-272, MIT Laboratory for Computer Science, 1983.
[SmithSmith 77]	Smith, J. M., and Smith, D. C. P. Database abstractions: aggregation. *Communications of the ACM* 20(6), June, 1977.
[Standish 67]	Standish, T. A. *A data definition facility for programming languages.* PhD thesis, Carnegie Institute of Technology, May, 1967.
[Steele 80]	Steele, G. L., Jr. *The definition and implementation of a computer programming language based on constraints .* Technical Report TR-595, Massachusetts Institute of Technology Artificial Intelligence Laboratory, August, 1980.
[Steele et al. 83]	Steele, G., et al. *Common Lisp reference manual.* Technical Report (unnumbered), Spice Project, Department of Computer Science, Carnegie-Mellon University, November, 1983.
[Stefik 78]	Stefik, M. *An examination of a frame-structured representation system.* Technical Report HPP-78-13, Stanford Heuristic Programming Project, Sept, 1978.
[Stefik et al. 83a]	Stefik, M., Bobrow, D., Mittal, S., Conway, L. Knowledge programming in Loops. *AI Magazine* 4(3):3-13, Fall, 1983.
[Stefik et al. 83b]	Stefik. M., Bobrow, D., Mittal, S., and Conway, L. Knowledge progrmming in Loops. *AI Magazine* 4(3):3-13, Fall, 1983.
[SussmanSteele 78a]	Sussman, G., and Steele, G. *The revised report on Scheme, a dialect of Lisp.* Technical Report 452, M.I.T. Artificial Intelligence Laboratory, January, 1978.
[SussmanSteele 78b]	Sussman, G., and Steele, G. *The art of the interpreter, or, The modularity complex (parts zero, one and two).* Technical Report 453, M.I.T. Artificial Intelligence Laboratory, May, 1978.

[SussmanSteele 80]
Sussman, G., and Steele, G.
Constraints: a language for expressing almost-hierarchical descriptions.
Artificial Intelligence 14:1-39, 1980.

[Sutherland 63] Sutherland, I.
Sketchpad: a man-machine graphical communication system.
PhD thesis, Masachusetts Institute of Technology, 1963.

[Suzuki 81] Suzuki, N.
Inferring Types in Smalltalk.
In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 187-199. ACM, January, 1981.

[Symbolics 84] Symbolics, Inc.
Lisp Machine Manual
Cambridge, Mass., 1984.
Eight volumes.

[Teitelman 78] Teitelman, W.
Interlisp reference manual
1978.

[Tennent 77] Tennent, R. D.
On a new approach to representation independent data classes.
Acta Informatica 8:315-324, 1977.

[Theriault 83] Theriault, D.
Issues in the design and implementation of Act 2.
Technical Report TR-728, M.I.T. Artificial Intelligence Laboratory, June, 1983.

[Touretzky 84] Touretzky, D.
The mathematics of inheritance systems.
PhD thesis, Carnegie-Mellon University, 1984.

[WatermanHayes-Roth 78]
Waterman, D., and Hayes-Roth, F. (eds.).
Pattern-directed inference systems.
Academic Press, 1978.

[Wegbreit 70] Wegbreit, B.
Studies in extensible programming languages.
PhD thesis, Harvard University, 1970.

[Wegbreit 74] Wegbreit, B.
The treatment of data types in EL1.
Communications of the ACM 17(5):251-264, May, 1974.

[WeinrebMoon 80]
Weinreb, D. and Moon, D.
Flavors: message passing in the Lisp Machine.
Technical Report AIM-602, M.I.T. Artificial Intelligence Laboratory, November, 1980.

[WeinrebMoon 81]
Weinreb, D., and Moon, D.
Lisp machine manual, fourth edition
1981.

[WellsCornwall 76]
Wells, M. B., and Cornwall, F. L.
A data type encapsulation scheme utilizing base language operators.
In *Proceedings of: Conference on data: abstraction, definition, and structure*, pages 170-178. ACM SIGPLAN/SIGMOD, 1976.
Published as a special 1976 issue of *SIGPLAN Notices*.

[Wirth 80] Wirth, N.
Modula-2.
Technical Report, Institut fur Informatik, ETH Zurich, March, 1980.

[Wood 82] Wood, R. J.
Franz flavors: an implementation of abstract data types in an applicative language.
Technical Report TR-1174, Department of Computer Science, University of Maryland, June, 1982.

[Wulf et ai. 76] Wulf, W. A., London, R. L., Shaw, M.
Abstraction and verification in Alphard: introduction to language and methodology.
Technical Report, Information Sciences Institute, USC, June, 1976.

[XeroxLRG 81] Learning Research Group, Xerox PARC.
The Smalltalk-80 system.
Byte 6(8), August, 1981.

[Zippel 83] Zippel, R.
Capsules.
In *SIGPLAN 83*. SIGPLAN, 1983.