CMU-ITC-078

A Description and Evaluation of PARAGON's Type Hierarchies for Data Abstraction

Mark Sherman

Information Technology Center

Carnegie-Mellon University

Pittsburgh, PA. 15213

[This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.]

1. Goals of Paragon

The goals of Paragon can be grouped into three broad classes: abstract data type specification goals, abstract data type representation goals and automatic-processing goals. These goals are listed below:

Abstract Data Type Specification Goals

- Refinements of specifications of abstract data types may be written.
- Related specifications may be combined in a single module.

Abstract Data Type Representation Goals

- Multiple implementations of an abstract data type may be declared in a program.
- Several implementations of an abstract data type may be used simultaneously in a program (one implementation per variable).

ADVANCED PROGRAMMING METHODOLOGIES
ISBN 0-12-174690-9

- If several implementations of an abstract data type are used for different variables, those variables may interact.
- A single implementation may be written for several separate specifications.

Automatic-Processing Goals

- Static type checking of all variable declarations (object creations) and procedure calls should be supported.
- Compile-time checking should ensure that all representation-selection decisions result in a program that can execute without run-time errors.

The abstract data type specification goals are partially met by the object-oriented language designs in Simula [Dah 68] and Smalltalk [Gol 81, Ing 78, Ing 81, Mor 81, Xer 81], the use of clusters in Enhanced C [Kat 83a, Kat 83b], the Traits additions to Mesa [Cur 82] and the Flavors facility for Lisp [Wei 81]. An extension of Simula proposed by Ingargiola [Ingr 75] allows, in a very restricted way, layers of specifications. The Scratchpad II system provides a nice way to layer specifications [Sut 87]. A similar kind of hierarchy was proposed by Smith and Smith [Smi 77] and in Taxis [Myl 80] for organizing relations, views and objects in a database. Further, the Program Development System [Che 79] uses a refinement hierarchy for writing system modules. But all of these systems use the refinements only as a way to refine objects or system components however, and not as refinements of specifications with the intention of later refining the specifications into implementations.

Both sets of abstract data type goals are partially met by Clu [Lis 81], Alphard [Sha 81], Enhanced C and Ada [Ich 80]. However, all of these languages use two levels of refinements, the upper level being the abstract specification and the lower level being the concrete implementation. The proposed layers of specifications are a departure from most languages that provide data abstraction facilities.

Further, these languages place strong restrictions on how representations of abstract data types must be related to their specifications. These restrictions limit the languages' abilities to define and use multiple implementations of abstract data types, or to let different implementations interact. Paragon provides features that allow a programmer to define and use multiple implementations of an abstract data type in a program. Further, the Paragon design permits different representations to have access to additional details about their parameters and use of their operations. This ability represents a substantial departure from current data abstraction methodology which insists that a representation may be used anywhere the specification is used.

The goal of static type checking is a departure from the procedure-call (dynamic) checking performed in typical object-oriented, hierarchy-based systems, such as for Smalltalk's methods and Simula's virtual procedures. There is a tradeoff in these designs between safety and efficiency, and flexibility. Because the parameter matching for procedure calls can be verified during compilation, static checking is considered safer, and because more is known about the program being checked, a more efficient program should result. Therefore Paragon opts for a safe and efficient language rather than for flexibility.

In applying this philosophy to a language with multiple representations of abstract data types, the design of Paragon enforces compile-time checking of implementations, guaranteeing that all variables have a feasible representation. An implementation for a program's variables and procedures is feasible if appropriate procedure implementations exist for all procedure calls as dictated by the representations of the actual parameters in the call. The design of Paragon ensures that a program's feasibility may be checked at compile time.

Another part of this goal is that no run-time expense should be incurred for making a selection decision. Although Paragon allows multiple procedure implementations to be provided for each procedure specification, the selection of an implementation to use for each procedure call must be made at compile time.

These goals are met by the Paragon design, which is discussed in the next section. The following section then shows how these features are used to write layers of specifications, multiple implementations of abstract data types and various kinds of shared representations.

2. Introduction to Paragon

2.1. Classes and Inheritance

Classes in Paragon are similar to classes in Simula: they contain declarations, statements and parameters. Unlike Simula classes, Paragon classes may inherit more than one class, and an ancestor may be inherited more than once. Also unlike Simula, classes declared inside of a class may be selected from that class.

Variable declarations in Paragon name a class that specifies the variable. Only the visible declarations in the class or one of its ancestors may be used by the variable. An implementation for that variable is selected from the subclasses of the specificed class. (The Paragon system also contains a representation selection system that makes the actual representation decisions. The discussion of the selection system is beyond the scope of this paper, but is described fully in [She 85].)

2.2. Procedure Specifications and Implementations

Procedures (which include functions and iterators) have separate specifications and implementations. A procedure specification is the signature of the procedure: the procedure's identifier and list of formal parameters. A procedure implementation is a signature followed by local declarations and statements. Any number of implementations may be written for each specification in the class (and in any subclass of the class) that contains the procedure specification. Paragon will select an appropriate procedure implementation given the implementations of the actual parameters for each procedure call.

2.3. Uniform Object Notation

Paragon uses a uniform object notation in all expressions. Expressions are used as formal parameters in class and procedure declarations, as actual parameters in class instantiations and procedure calls, as "types" in variable declarations and as statements. The same interpretation of an expression is used regardless of where it appears. There are three interesting parts of this notation: definite objects, indefinite objects and any objects. A definite object results from the instantiation of a class. This is the same as object generation in Simula, and uses the same notation: new Class ID. An indefinite object can be thought of as a type. Its notation is merely a class name: Class ID. An "any" object is a special, predefined object with the property that any object matches it. Its notation is any.

This uniform object notation allows a programmer to specify a "type" parameter by using an indefinite object as an actual parameter, to specify a constraint by using a definite object as a formal parameter, and to specify a procedure parameter by declaring a procedure in the a class which can then be passed. Thus Paragon can use this single notation to provide for commonly used facilities in other languages.

2.4. Comparing Objects

A relation called matching may exist between an actual object and a formal object. The terms actual and formal are used in the conventional sense. Unlike most languages, this relation is not symmetric. When an actual matches a formal, there is no implication that the formal matches the actual.

As parameters are objects, object matching is used for comparing parameters. At different times, the same parameter may be used as a formal and an actual. The following table summarizes the kinds of comparisons that occur in Paragon.

Actual	*Formal*
Proc. Call Parameter	Proc. Specification Parameter
Proc. Call Parameter	Proc. Implem. Parameter
Class Instantiation	Class Declaration Parameter
Subclass Declaration Parameter	Class Declaration Parameter
Proc. Implem. Parameter	Proc. Specification Parameter

Much of the power of multiple procedure implementations and subclasses comes from the fact that their parameters need not be identical with parameters in the corresponding procedure specification or parent class. This is a radical departure from other data abstraction languages, since this ability implies that an implementation may not necessarily work anywhere that the specification is permitted (because of incompatible implementations of variables and procedures). Instead, Paragon defines a process called elaboration with implementations that ensures that feasible representation selections have been made. This process is similar to execution simulation.

The basis for matching is the comparison of two instances of classes. Intuitively, an actual object matches a formal object if the underlying class of the formal is an ancestor of, or the same as, the underlying class of the actual. To ensure compatibility between definite and indefinite instances, one of the following constraints must also be met:

- The formal is an any instance,
- The formal is an indefinite instance and the underlying class of the formal is a ancestor of, or the same class as, the underlying class of the actual, or
- The formal is a definite instance and the actual is the same definite instance.

Similar rules are recursively applied to parameters of an instance (which are also instances of objects) to ensure that any parameters of the instances match, but these rules are omitted for brevity.

3. Supporting Data Abstraction

In this section, the features of Paragon are use to illustrate how data abstractions may be defined in Paragon. These examples show how Paragon can describe and use shared specifications and representations. These examples are programmed using a style of programming called the object-manager model. This model is described below.

3.1. Object Managers and Nested Classes

Paragon relies on the object-manager model of programming. The object-manager model divides program objects into two categories: managers and individuals. The manager is created first and contains data and procedures that are shared among all individuals. For each manager, there may be any number of individuals created, and each individual has a single manager. Naturally, each individual may have private data and procedures not shared with other individuals.

As an example of this model, consider integers. Each individual integer can be represented as a word in memory. Further, there exists a procedure, Addition, that is shared among all the individual integers, and so this procedure belongs to the manager of all integers. Paragon implements this model by using nested classes. The outer class defines the manager and the inner class defines the individual.

3.2. Classes for Specifications

Classes in Paragon are used to represent two kinds of specifications: generalizations and descriptions. Generalizations attempt to abstract some commonly used properties that are inherited by other specifications. For example, Ada provides a generalization called nonlimited private which specifies that types declared as nonlimited private have the assignment operation defined for them. Clu provides a generalization that provides an object with the ability to be transmitted over a network (see [Her 80]). By properly defining a set of classes for each set of operations that one might want to inherit later, one can provide the same predefined generalization features that other languages do without limiting the choices of operations. For example, a set of declarations that simulate the concept of nonlimited in Ada is shown below:

```
class AssignableManager is
  class Assignable is begin end;
  procedure Assign(Assignable,Assignable);
  procedure Equal(Assignable,Assignable) return Booleans.Bit;
end;
```

A description corresponds to an abstract data type specification in other languages. It too consists of nested classes, and usually inherits generalization classes. An object manager that inherits the AssignableManager class would then define an nonlimited type. Providing the details for the integer example shows this property:

```
class IntegerManager of AssignableManager is
begin
  ! Shared data and procedures go here ;
  procedure Addition(Integer,Integer) return Integer;
  ! And the class definition for individuals ;
  class Integer of Assignable is begin end;
end;
```

One could use these declarations to declare variables

```
var AppleManager ⇒ new IntegerManager;
var Lisa ⇒ AppleManager . new Integer;
var MacIntosh ⇒ AppleManager . new Integer;
```

and perform the operations declared in the specified classes or their ancestors:

```
AppleManager.Assign(Lisa,MacIntosh);
```

The distinction between generalization and description classes is by convention alone. Paragon places no restrictions on how a class is used. There do exist classes that are used both as generalization and description classes.

3.3. Classes for Implementations

Implementations are declared through the use of subclasses. Typically, a subclass that is intended to implement an abstract data type contains the implementations for those procedures specified in its ancestors and contains subclasses for the nested classes. Assuming that a full specification and implementation for computer words exists in a manager called CM, an implementation for the IntegerManager/Integer classes is:

```
class WordIntegerManager of IntegerManager is
begin
 ! ....................;
 procedure Assign(L:WordInteger, R:WordInteger)
  return WordInteger is
 begin
  CM.Assign(L.Rep,R.Rep);
 end;
  !....................;
 procedure Equal(L:WordInteger, R:WordInteger)
  return Booleans.Bit is
 begin
  return CM.Equal(L.Rep,R.Rep);
 end;
 !....................;
 procedure Addition(L:WordInteger, R:WordInteger)
  return WordInteger is
 begin
  return CM.Plus(L.Rep,R.Rep);
 end;
 !....................;
 ! And the class definition for individuals;
 class WordInteger of Integer is
 begin
  var Rep ⇒ CM . new Word;
 end;
end;
```

The conventional methodology for implementing an abstract data type requires that all operations in the specification must be implemented, that a representation for the object must be described and that there is some way to separate the abstract object from the concrete object. Procedure implementations for Assign, Equal and Addition are declared, the class WordInteger defines the representation of Integer and use of the names Integer and WordInteger separates the abstract object from the concrete object. Thus all of the requirements for an abstract data type implementation are met in the example above.

The example above also illustrates a feasible implementation for IntegerManager. In WordIntegerManager, procedure implementations are provided for the procedure specifications in all inherited ancestors: here the Assign, Equal and Addition procedures from the IntegerManager and AssignableManager classes. This is not required by Paragon but does guarantee that this subclass may be used as an implementation anywhere the specification is used. If some operation had been missing, and if a program used that operation on abstract integers, then the implementation subclass for the specification could not be used. An attempt to use such an incomplete subclass in this circumstance would render the program infeasible.

The distinction between the abstract use of a object and the concrete use of an object is also illustrated by this example. The example above specifies the class WordInteger in all of the procedures' parameters in the WordIntegerManager class. This implies that only the WordInteger representation of Integer can be used with these procedures and provides a boundary between the abstract and concrete representations. Some languages provide an operation (in Clu called cvt) that is supposed to translate between an abstract object and a concrete one. Within the implementation of the abstract data type, one may restrict the implementation to use only the abstract properties of the object by omitting the special operation. Other languages reverse the convention and allow the programmer access to the representation unless the programmer specifies that only the abstract operations should be allowed. Ada uses still another approach by unconditionally permitting access to the representation of an object within the implementation of the abstract data type. Paragon attempts to strike a balance by using the names in the class declarations. Should only the abstract operations be permitted, then the programmer may specify this by writing the name of the specification class in the parameter. If access to the representation is required, then the name of the class used as a representation should be written in the parameter. Because each procedure specifies that WordInteger objects may be used as parameters, it may use the details of WordInteger objects, such as selecting the Rep field. Had the procedures merely required Integer objects, then access to the Rep field would have been denied, even if an instance of WordInteger had been given to the procedure.

The use of names rather than conventions for the abstract/concrete decision permits a greater flexibility in the definition of implementations. This is more fully explored in the next section where some methods for providing multiple implementations of abstract data types are considered.

3.4. Multiple Implementations

There are times when a programmer may wish to have more than one implementation for an abstract data type. This can be illustrated with the previously specified IntegerManager. Many computers have more than one size of data representation provided by the hardware so it seems reasonable that different integer variables might be able to take advantage of these differences in order to improve a program's performance. Each different sized representation has its own representation class and its own procedure implementations. Most data abstraction languages allow only one representation for each specification. If the one word representation for integers were present in a program, such languages would prohibit the inclusion of a half word integer and a double word integer.

Paragon does not have such a rule. A new representation may be provided by declaring a new set of nested classes. For example, a program might contain the following declarations for integers requiring less than a word of storage:

```
class ShortWordIntegerManager of IntegerManager is
begin
 !...................;
 procedure Assign(L:ShortWordInteger, R:ShortWordInteger)
   return WordInteger is
 begin
   CMSW.Assign(L.Rep,R.Rep);
 end;
 !...................;
 procedure Equal(L:ShortWordInteger, R:ShortWordInteger)
  return Booleans.Bit is
 begin
  return CMSW.Equal(L.Rep,R.Rep);
 end;
 !...................;
 procedure Addition(L:ShortWordInteger, R:ShortWordInteger)
  return ShortWordInteger is
 begin
  return CMSW.Plus(L.Rep,R.Rep);
 end;
 !...................;
 ! And the class definition for individuals;
```

```
    class ShortWordInteger of Integer is
    begin
      var Rep ⇒ CMSW . new ShortWord;
    end;
  end;
```

The ShortWordIntegerManager/ShortWordInteger classes represent another implementation of the integer abstract data type. Two factors are present which allow the second implementation to be declared and included in a program. First, the explicit separation of the specification and implementation of the abstract data type provides a way to bind an implementation to a specification. Second, the ability to name the representation explicitly circumvents a problem of controlling the access to the concrete object.

The ability to name explicitly the representations or specifications in parameters permits multiple representations to be used in a more common setting: differing type parameters. Frequently cited examples are set implementations where alternative representations of the set is caused by different element types [Joh 76, Low 74, Sch 77, Wul 81]. A typical (partial) specification for sets in Paragon appears below:

```
  class SetManager(T:any) is
  begin
    class Set is begin end;
    !...................;
    procedure Union(Set,Set) return Set;
    !...................;
  end;
```

The element type of the set may be any class. However, certain classes have special properties that an implementation may wish to exploit.

For example, if the element type were totally ordered, a B-Tree or discrimination net may be an appropriate implementation. If it can be hashed, a hash table may prove efficient. Sets of a small number of enumerated values are usually represented as a bit vector. Thus one wants the implementation to be able to take advantage of knowledge of the element type.

Other languages, such as Clu and Alphard, do not permit this exploitation in an implementation, or more precisely, they insist that such requirements appear in the specification of the abstract data type. Paragon permits the specification to be as broad as required and the implementation to be as narrow as required by allowing the parameters in subclasses merely to match the parameters in the parent class, and not to be identical. A discrimination-net implementation of the previous SetManager could look like the following:

```
class DiscriminationSetManager(OrderedManager.T:Ordered)
  of SetManager is
begin
  !....................;
  class DiscSet of Set is
  begin
  end;
  !....................;
  procedure Union(DiscSet,DiscSet) return DiscSet is
  begin
    ! Impl of Union operation;
  end;
  !....................;
end;
```

The DiscriminationSetManager class may only be used as an implementation for Setmanager when the element type of the set is ordered. However, all available information about ordered objects (as expressed in the specification for OrderedManager) may be used inside DiscriminationSetManager in its manipulation of the set's element type. This use of a subclass in the parameter of the implementation class also eliminates the need for procedure parameters since the composed data type and its operations are combined in a class declaration. Therefore the user can use the abstract data type without needing to consider the constraints required by any particular implementation. Such considerations are automatically processed by the translation system.

3.5. Explicitly Shared Implementations

The examples given in the previous sections for integers and sets bring up another topic: the sharing of representations. Because the class mechanism does not restrict the way in which specifications and representations may be combined, several arrangements of classes prove useful in selective sharing between the specifications of abstract data types, between the representations of abstract data types, and between the specifications and the representations of abstract data types. Each of these kinds of sharing is considered in turn.

3.5.1. Shared Implementations via Shared Specifications

Selective sharing of specifications is quite common in practice and supported in some languages, such as Ada. This usually takes the form of a single manager being used for several different kinds of individuals. One example is a computer memory, as illustrated below:

```
class MemoryManager is
begin
```

```
    class Byte is begin end;
    class Word is begin end;
    procedure Read(Byte);
    procedure Write(Byte,IM.Integer);
    procedure LeftByte(Word) return Byte;
    procedure RightByte(Word) return Byte;
  end;
```

The single manager MemoryManager provides the shared declarations for two related individuals, Byte and Word. Words and bytes are closely coupled in a memory and should be considered connected in some way. Some languages, such as Clu, have no provisions for this selective sharing. Paragon permits multiple inner classes that are declared in an outer class to denote different kinds of individuals for the same manager.

The implementation of MemoryManager could contain further subclasses for Byte and Word and implementations for Read, Write, LeftByte and RightByte, each of which could access the concrete representation for both bytes and words.

3.5.2. Shared Implementations via Previous Implementations

Another way of combining classes gives the programmer the ability to write procedure implementations that can access multiple representations. Like the MemoryManager example above where one can write a single subclass of the specification class that has access to representations of multiple kinds of objects, one can provide a subclass of implementation subclasses that permits access to multiple, concrete representations of the same abstract object. This can be illustrated by extending the IntegerManager implementations given in Section 3.4). To include a procedure that can add integers regardless of the implementations of the abstract integer, one can write:

```
class CombinedWordIntegerManager of
  WordIntegerManager, ShortWordIntegerManager is
begin
  !....................;
  procedure Addition(L:ShortWordInteger, R:WordInteger)
    return WordInteger is
  begin
  ...
  end;
  !....................;
  procedure Addition(L:WordInteger, R:ShortWordInteger)
    return WordInteger is
  begin
    ...
  end;
  !....................;
end;
```

If CombinedWordIntegerManager were to be selected as the implementation for an abstract IntegerManager object, then abstract integers could be implemented with either the ShortWordInteger or the WordInteger subclasses of Integer. Regardless of the implementation selected for two abstract integers, there will exist an implementation of the Addition procedure that can operate on them.

3.5.3. Shared Implementations for Unrelated Specifications

A third way of sharing in Paragon allows an implementation class to be used as an implementation for multiple specification classes.

One example that illustrates this sharing is the SETL system where sets are implemented by altering the representation of the elements of the set. This is a unique approach to implementing sets and their elements as it requires a shared implementation for two specifications that are not otherwise related: one specification for sets, one specification for the elements of the set. The use of classes and inheritance provides a way to specify this capability as well. Given two separate sets of specification classes, say for integers and sets, one creates a single class that acts as the manager for both and that class contains the representations for the union of the inherited individuals and procedures. An abbreviated illustration is given below:

```
!....................;
! Specification Classes for Integers ;
!....................;
class IntegerManager of AssignableManager is
begin
  procedure Addition(Integer,Integer) return Integer;
  class Integer of Assignable is begin end;
end;
  !....................;
! Specification Classes for Sets ;
!....................;
class SetManager(T:any) is
begin
  procedure Union(Set,Set) return Set;
  class Set is begin end;
end;
```

With these specifications, one may write the following shared implementation for sets and integers (adapted from [Dew 79]):

```
class IntegerSetManager(IntegerManager . T: Integer)
  of IntegerManager, SetManager is
begin
  !....................;
  class IntBlock is
  begin
    ! Reps for the integer and set indication ;
  end;
  !....................;
  ! Shared Data for the Manager;
  !....................;
  var RIBM ⇒ new RefManager(IntBlock);
  var IntValueList ⇒ RIBM . new Reference;
  !....................;
  !....................;
  ! Integer Implementations ;
  !....................;
  class SharedInteger of Integer is
  begin
    var IntValueBlock ⇒ RIBM . new Reference;
  end;
  !....................;
  procedure Addition(SharedInteger, SharedInteger)
    return SharedInteger is
  begin
    ! Implementation for Addition operation;
  end;
  !....................;
  !....................;
  ! Set Implementations ;
  !....................;
```

```
   class SharedSet of Set is
   begin
     var SetNum ⇒ CM. new Word;
   end;
   !...................;
   procedure Union(SharedSet, SharedSet) return SharedSet is
   begin
     ! Implementation for Union operation;
   end;
   !...................;
 end;
```

Although the details are missing, the example above shows that representation combinations can be expressed via the class mechanism whereas most approaches to data abstraction have no way of describing a combined representation.

3.6. Implicitly Shared Implementations

Paragon also supports implicit sharing of representations. Such sharing comes when procedure are written that use only abstract properties of their parameters. For example, an implementation of the Union procedure could have been provided in the SetManager as follows:

```
procedure Union(L:Set,R:Set) return Set is
begin
  var Temp ⇒ new Set;
  var i ⇒ new T;
  for i in Elements(L) do
    Insert(Temp,i);
  end for;
```

```
    for i in Elements(R) do
      Insert(Temp,i);
    end for;
    return Temp;
  end;
```

This procedure implementation assumes that an Elements iterator and an Insert procedure have been specified in SetManager. Note that implementation uses only abstract operations on sets. Each call of this Union procedure may have different or even conflicting representations for the parameters. Paragon permits these calls and checks that all necessary procedure implementations are available for use inside of Union. All of the necessary processing is performed at compile time; no run-time examination of representations is necessary when executing this implementation.

These examples have shown how multiple representations for variables and procedures may be declared and used in Paragon. After a program is written, the selection of an appropriate representation must be made for each variable and each procedure call in the program. To satisfy this need, Paragon provides a representation selection mechanism that the programmer may use to guide the translator in picking appropriate implementations. The discussion of this selection system is beyond the scope of this summary (and paper). The interested reader is referred to the complete description of Paragon [She 85].

4. A Retrospective

Paragon was designed and implemented as a proof-of-concept and was never intended to become a production system. Its design is complete and a prototype implementation exists. Several thousand lines of Paragon code have been programmed. Both abstract data types and application programs have been programmed and processed by the prototype. This code also provides test cases for performance measurements that were taken of the prototype translator. The results of the test are described in [She 85]. The source code for all of the tests can be found in [She 85]. Generally, the system interprets Paragon code at about the same speed as the initial Ada interpreters, and outputs a transformed source that is equivalent to Pascal.

Over the last four or five years, several recurring themes have merged in the discussions about hierarchies in languages. A taxonomy of many of the discussions can be found in [Weg 87]. Four of these issues relate to the Paragon experience and each will be discussed in turn.

4.1. Compiled vs Interpretive Languages

There is a continuing debate about whether object-oriented languages should be interpreted or compiled. From the outset, there were examples of both: Simula is compiled and Smalltalk is interpreted. Paragon performs a kind of execution simulation to check feasibility. Another approach is to try to translate an interpreted language into a compiled language [Cox 87]. It seems that the camp favoring interpreted languages is assuming that with clever partial compilations (e.g., compilation as needed), and with better hardware support, the interpreted languages can perform as well as compiled languages ([Suz 83, Deu 83]). However, it also seems to me that for the incremental performance gained from better algorithms or hardware can be applied usually to traditionally compiled languages, the result being that both sets of languages perform better than before, but with compiled languages outperforming interpreted languages. Further, I have not yet seen a convincing answer to the problem of safety in an interpreted language: a type error may not be located until well into program execution. The usual answer of "one can immediately fix the program and continue execution" is not helpful in a production environment.

4.2. Theoretical vs Applied Languages

Although the design of Paragon was clearly influenced by heterogeneous algebras, the work itself does not attempt to advance that formalism. These technique provide a basis for determining if an implementation meets a specification, and provide one kind of framework for extending specification by adding domains and axioms. However, these techniques do not provide any notion of combining together different implementations, nor discuss how to apply these theories in practice. Attempts have been made to provide an additional theoretical framework for generating instances from specifications, for example [Agn 85], but these efforts are usually just careful renaming systems rather than anything can be applied in practice. There are so few systems that can provide multiple representations of an abstract data type, and even fewer that provide a way to select them, that trying to abstract a theory on such systems may be premature. Demanding that working systems must first have a complete theoretical treatment before construction may begin seems too restrictive.

4.3. Using Hierarchies for Refinement vs Implementation

In her keynote address at the 1987 OOPSLA conference, Barbara Liskov noted that languages encourage several uses of inheritance, two of which are refinement and implementation. Roughly, refinement means the addition of new specifications while implementation refers to using a superclass as an implementation of a subclass. An example of the former is that a mathematical group is a subclass of semigroup, while the later is that a set is subclass of list. She correctly points out that the former is consistent with the strategies of information hiding and data abstraction while that latter is an implementation trick that can cause maintenance problems later. In this paper, the discussion of Paragon uses its classes in a very stylized way, for example, in the object-manger model. However, the programmer is not restricted to using Paragon in this way. Arbitrary nesting and inheritance of classes is permitted, but very confusing, resulting in programs that are as bad as in any other language. I believe that design would have been better if it had explicitly distinguished between specifications and implementations, and between managers and individuals.

4.4. The Role of Scope Nesting in Languages

In order to allow Paragon's mechanisms of class nesting and hierarchies to serve many roles, the mechanisms had few restrictions. It has become apparent that allowing such arbitrary scopes has little practical value but can make programs harder to read and harder to process. For example, Paragon explicitly checks for an infinite recursion of data structures and flags it as an error. Another system encountered a different problem: attempts to provide in-line substitution in Loglan [Kre 87] for nested classes resulted in another variant of the funarg problem. As a result of these observations, many newer languages, such as C++, do not provide arbitrary nesting of scopes.

5. Summary

The paper demonstrates how a type hierarchy [Technically, a directed acyclic graph of types, but type hierarchy is a more commonly used term.] can be used for writing programs using the object-manger model to specify abstractions, refine the specifications, write representations for the abstractions and combine representations as desired. These capabilities are not available in current languages, so the Paragon design shows how type hierarchies can be used in new language designs. A number of programs were written and translated with a prototype system that processes Paragon, thus the suggested language is not a mere paper design, but a complete language that can be implemented and used for programming. However, its age in a rapidly advancing field is showing, and a significant redesign would be required to be used as a production system.

Bibliography

[Agn 85] Snorri Agnarsson and M. S. Krishnamoorthy: *Towards a Theory of Packages*, Proc. ACM Symp. on Language Iss. in Programming Environments, pgg 117-130. ACM SIGPLAN, June, 1985. Also SigPlan Notices, Vol. 20, No. 7, July 1985.

[Che 79] Cheatham Jr., Thomas E., Townley, Judy A. and Holloway, Glenn H.: *A System for Program Refinement*, Proc. 4th Intern. Conf. on Software Engineering, pgg. 53-62. IEEE Comp. Soc. September, 1979.

[Cox 87] Brad J. Cox and Kurt J. Schmucker: *A Tool for translating Smalltalk-80 to Objective-C*, Object-Oriented Progr. Systems, Languages and Applications Conf. Proc., pgg. 423-429. ACM, October, 1987. Also SigPlan Notices, Vol. 22, No. 12, December 1987.

[Cur 82] Curry, Gael, Baer, Larry, Lipkie, Daniel and Lee, Bruce: *Traits: An Approach to Multiple-Inheritance Subclassing*, Limb, J.O. (editor), Proc. SIGOA Conf. on Office Information Systems, pgg. 1-9. ACM, SIGOA, June, 1982. Also SIGOA Newsletter, Vol. 2, Nos. 1 and 2.

[Dah 68] Dahl, O.-J.: *Simula 67 Common Base Language*, Techn. Rep., Norwegian Computing Center, Oslo, 1968.

[Deu 83] L. Peter Deutsch and Allan M. Schiffman: *Efficient Implementation of the Smalltalk-80 System*, Proc. ACM Symp. on Principles of Programming Languages, pgg. 296-302. ACM SIGACT and SIGPLAN, January, 1983.

[Dew 79] Dewar, Robert B. K., Grand, Arthur, Liu, Ssu-Cheng and Schwartz, Jacob T.:*Programming by Refinement, as Exemplified by the SETL Representation Sublanguage*, ACM Trans. on Progr. Languages and Systems 1(1):27-49, July, 1979.

[Gol 81] Goldberg, Adele: *Introducing the Smalltalk-80 System*, Byte 6(8):14-22, August, 1981.

[Her 80] Herlihy, Maurice Peter: *Transmitting Abstract Values in Messages*, Techn. Rep. MIT/LCS/TR-234, Lab. for Comp. Science, MIT, April, 1980.

[Ich 80] Ichbiah, Jean, et. al: *Reference Manual for the Ada Programming Language*, US Government, Washington, D.C., 1980.

[Ing 78] Ingalls, Daniel H. H.: *The Smalltalk-76 Programming System: Design and Implementation*, Conf. Record Fifth Annual ACM Symp. on Principles of Programming Languages, pgg. 9-16. ACM, January, 1978.

[Ing 81] Ingalls, Daniel H. H.: *Design Principles Behind Smalltalk*, Byte 6(8):286-298, August, 1981.

[Ingr 75] Ingargiola, Giorgio P.: *Implementations of Abstract Data Types.* In Proceedings of the Conference on Computer Graphics, Pattern Recognition, & Data Structure, pages 108-113. IEEE Computer Society, May, 1975.

[Joh 76] Johnson, Robert T. and Morris, James B. *Abstract Data Types in the MODEL Programming Language.* In Proceedings of Conference on Data: Abstraction, Definition and Structure, pages 36-46. ACM, March, 1976. Also Sigplan Notices, Vol. 8, No. 2, 1976.

[Kat 83a] Katzenelson, J.: *Introduction to Enhanced C (EC).* Software

Practice and Experience 13(7), July, 1983.

[Kat 83b] Katzenelson, J. *Higher Level Programming and Data Abstractions A Case Study Using Enhanced C.* Software Practice and Experience 13(7), July, 1983.

[Kre 87] Antoni Kreczmar. *The programming language LOGLAN-82.* Technical Report, Institute of Informatics, Warsaw University, Warsaw, Poland, 1987.

[Lis 81] Liskov, B., Moss, E., Schaffert, C., Scheifler, R. and Snyder, A. *The CLU Reference Manual.* Springer-Verlag, New York, N.Y., 1981. Lecture Notes in Computer Science No. 114.

[Low 74] Low, James R. *Automatic Coding: Choice of Data Structures.* Technical Report CS-452, Stanford University Computer Science Department, August, 1974.

[Mor 81] Morgan, Chris. *Smalltalk: A Language for the 1980s.* Byte 6(8):6-10, August, 1981.

[Myl 80] Mylopoulos, John, Bernstein, Philip A. and Wong, Harry K. T. *A Language Facility for Designing Database-Intensive Applications.* ACM Transactions on Database Systems 5(2):185-207, June, 1980.

[Sch 77] Schonberg, E. and Liu, S. C. *Manual and Automatic Data-Structuring in SETL.* In Andre, Jacques and Banatre, Jean-Pierre (editor), Implementation and Design of Algorithmic Languages: Proceedings of the 5th Annual III Conference, pages 284-304. IRISA, May, 1977.

[Sha 81] Shaw, Mary (editor). *ALPHARD: Form and Content.* Springer Verlag, New York, New York, 1981.

[She 85] Sherman, Mark Steven. Lecture Notes in Computer Science. Volume 189: *Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types.* Springer-Verlag, Heidelberg, 1985.

[Smi 77] Smith, John Miles and Smith, Diane C. P. *Database Abstrac-*

tions: Aggregation and Generalization. ACM Transactions on Database Systems 2(2):105-133, June, 1977.

[Sut 87] Robert S. Sutor and Richard D. Jenks. *The Type Inferenced and Coercion Facilities in the Scratchpad II Interpreter.* In Proceedings of the ACM Symposium on Interpreters and Interpretive Techniques, pages 56-63. ACM SIGPLAN, June, 1987. Also SigPlan Notices, Vol. 22, Nol. 7, July 1987.

[Suz 83] Norihisa Suzuki and Minoru Terada. *Creating Efficient Systems for Object-Oriented Languages.* In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 290-296. ACM SIGACT and SIGPLAN, January, 1983.

[Weg 87] Peter Wegner. *Dimensions of Object-Based Language Design.* In Object-Oriented Programming Systems, Languages and Applications Conference Proceedings, pages 168-182. ACM, October, 1987. Also SigPlan Notices, Vol. 22, No. 12, December 1987.

[Wei 81] Weinreb, Daniel and Moon, David. *Lisp Machine Manual.* Symbolics Inc., California, 1981. Fourth Edition.

[Wul 81] Wulf, W.A., Shaw, M., Hilfinger, P.N. and Flon, L. *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

[Xer 81] Xerox Learning Research Group. *The Smalltalk-80 System.* Byte 6(8):36-48, August, 1981.