

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

189

Mark Steven Sherman

Paragon:

A Language Using Type Hierarchies for the Specification,
Implementation and Selection of Abstract Data Types



Springer-Verlag
Berlin Heidelberg New York Tokyo

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

189

Mark Steven Sherman

Paragon:

A Language Using Type Hierarchies for the Specification,
Implementation and Selection of Abstract Data Types



Springer-Verlag
Berlin Heidelberg New York Tokyo

Editorial Board

D. Barstow W. Brauer P. Brinch Hansen D. Gries D. Luckham
C. Moler A. Pnueli G. Seegmüller J. Stoer N. Wirth

Author

M. Sherman
Department of Mathematics and Computer Science
Dartmouth College, Bradley Hall
Hanover, NH 03755, USA

CR Subject Classification (1982): D.3.2, D.3.3, D.3.4, E2, I.2.2, D.2.2

ISBN 3-540-15212-1 Springer-Verlag Berlin Heidelberg New York Tokyo
ISBN 0-387-15212-1 Springer-Verlag New York Heidelberg Berlin Tokyo

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks. Under § 54 of the German Copyright Law where copies are made for other than private use, a fee is payable to "Verwertungsgesellschaft Wort", Munich.

© by Springer-Verlag Berlin Heidelberg 1985
Printed in Germany

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Table of Contents

Acknowledgements	1
Abstract	3
1. Introduction	5
1.1. Motivation	5
1.1.1. The Use of Abstraction and Refinement	6
1.1.2. Data Abstraction	7
1.1.2.1. The Simple Model of Data Abstraction	8
1.1.2.2. The Limitations Imposed on Abstract Data Type Specifications	8
1.1.2.3. The Limitations Imposed on Abstract Data Type Implementations	10
1.2. Summary of Thesis	12
1.2.1. Data Abstraction Features	13
1.2.2. Representation Selection Features	14
1.2.3. Prototype Translator	16
1.3. Organization of the Thesis	16
2. Goals of Paragon and Their Relation to Previous Efforts	19
2.1. Goals of Paragon	19
2.1.1. Refinements of Specifications	20
2.1.2. Combined Specifications	22
2.1.3. Multiple Implementations	22
2.1.4. Simultaneous Implementations	23
2.1.5. Interacting Implementations	24
2.1.6. Shared Implementations	25
2.1.7. Distinguishing Implementations	25
2.1.8. Variable Description	26
2.1.9. Programmer Accessibility	26
2.1.10. Static Type Checking	27
2.1.11. Automatic Selection of Representation	27
2.1.12. Compile-Time Checking of Program Feasibility	28
2.2. Preliminary Design Restrictions for Paragon	28
2.2.1. Use of a Type Hierarchy	29
2.2.2. Single Identifier/Object Binding	29
2.2.3. Automatic Creation of Representations	30
2.2.4. Automatic Conversion between Representations	30
2.2.5. Run-Time Selection of Representations	30
2.2.6. Prototype Translator	31
3. The Basics of Paragon	33
3.1. Overview of Elaborations	34
3.2. Objects	34

3.2.1. Classes and Simple Objects	38
3.2.2. Inheritance	40
3.2.3. Nested Classes and Objects	42
3.3. Name Expressions	43
3.3.1. Generation of Instances	44
3.3.2. Description of Objects	44
3.3.3. Selection of Objects	45
3.3.4. Other Name Components	45
3.3.5. Other Expressions	46
3.3.6. Integer Literals	47
3.4. Parameters	48
3.4.1. Syntax of Parameters	48
3.4.2. Comparing Objects	49
3.4.2.1. Simple Object Matching	49
3.4.2.2. Object Matching	52
3.4.3. Type Parameters	53
3.4.4. Parameters to Classes	54
3.5. Procedure Specifications	55
3.5.1. Overall Syntax of Procedure Specifications	56
3.5.2. Parameters	56
3.5.3. Return Expression	56
3.5.4. Constraints	57
3.6. Procedure Implementations	58
3.6.1. Overall Syntax of Procedure Implementations	58
3.6.2. Parameters	58
3.6.3. Return Statement	59
3.6.4. Procedure Invocation	59
3.7. Iterators	62
3.7.1. Overview of Iterators	62
3.7.2. Iterator Specifications	63
3.7.3. Iterator Implementations	63
3.7.4. Yield Statement	64
3.7.5. Return Statement	64
3.7.6. For Statement and Iterator Invocations	65
3.7.7. Exitloop Statement	66
3.8. Conventional Statements	67
3.8.1. Labels	67
3.8.2. Procedure Invocation	67
3.8.3. Conditional Looping	67
3.8.4. If Statement	68
3.8.5. Goto Statement	69
4. The Object-Manager Model and Its Implementation	71
4.1. Object Managers and Nested Classes	71
4.1.1. Classes as Manager and Individuals	71
4.1.2. Cross-Implementation Procedures	73
4.2. The Manager Model in Other Languages	74
4.3. Hierarchies for Specifications	76
4.3.1. Generalizations	77
4.3.2. Specifications of Abstract Data Types	78

4.4. Problems with Hierarchies for Specifications	78
4.4.1. Constraints in Procedure Specifications	78
4.4.2. Return Objects of Procedure Specifications	79
4.4.3. Heterogeneous Data Structures	80
4.4.4. Adding Classes to an Existing Hierarchy	82
4.4.5. Refinement by Derivation	84
4.5. Hierarchies for Implementations	85
4.5.1. Multiple Implementations	87
4.5.2. Partial Implementations	90
4.5.3. Shared Implementations	92
4.5.3.1. Shared Implementations via Shared Specifications	92
4.5.3.2. Shared Implementations via Previous Implementations	93
4.5.3.3. Shared Implementations for Unrelated Specifications	94
4.6. Problems with Hierarchies for Implementations	96
4.6.1. Incomplete Implementations	96
4.6.2. Organizing Multiple Implementations	98
4.6.2.1. Using a Single Manager	98
4.6.2.2. Using Multiple Managers	101
4.6.3. Sharing a Representation	103
5. Selection of Implementations	107
5.1. Elaborations	107
5.1.1. Elaboration with Specifications	108
5.1.2. Implementation Selection	108
5.1.3. Elaboration with Implementations	108
5.1.4. Elaboration with Realizations	109
5.2. Variable Declarations and Object Creations	109
5.2.1. Selecting a Variable Implementation	110
5.2.2. Constraints on Variables	116
5.2.3. Checking the Feasibility of Variable Declarations	116
5.2.4. Elaboration of Object Creations with Realizations	119
5.3. Describing Classes and Procedures — Attributes	120
5.3.1. Purpose of Attributes	120
5.3.2. Attribute Variables	121
5.3.3. Attribute Procedures	122
5.3.4. Attributes in Classes	123
5.3.5. Procedure Respecifications	124
5.3.6. Attributes in Procedures	125
5.3.7. Attribute Variables in Expressions	128
5.3.8. Variables with Attributes	128
5.4. Representing the Implementation Choices — The Possibility Tree	130
5.4.1. Abstract Possibility Trees	130
5.4.2. Instances and <i>Instance</i> Classes	139
5.4.2.1. Realized <i>Instance</i> Objects	140
5.4.2.2. Object Instantiations	142
5.4.2.3. Procedure Invocations	146
5.4.3. Bridging <i>Instance</i> Objects and Doppelgangers	147
5.5. Making the Implementation Choices — The Policy Procedure	148
5.5.1. Syntactic Properties of the Policy Procedure	148
5.5.2. Executing a Policy Procedure	150

5.5.3. Attribute-Procedure Invocations	151
5.5.4. The Pattern Matching Statement	154
5.5.5. Feasibility of a Program	155
5.5.5.1. Selecting a Procedure Invocation	156
5.5.5.2. Limiting the Size of the Possibility Tree	158
5.5.5.3. Selecting the Implementations of Return Objects	160
5.5.5.4. Hidden Implementations	163
5.5.5.5. Another Way to Terminate Recursive Procedure Calls	164
6. A Complete Example Using Paragon	167
6.1. Program Structure and Processing	167
6.2. Predefined Environment	168
6.2.1. Input and Output	168
6.2.2. Assignment	168
6.2.3. Logical Objects	169
6.2.4. Ordered Objects	170
6.2.5. Hashable Objects	170
6.2.6. Integer Objects	171
6.2.7. Word Objects	172
6.2.8. Arrays	173
6.2.9. Pointers	174
6.2.10. Selection Facilities	175
6.3. An Abstract Data Type: List	175
6.3.1. A Specification for Lists	176
6.3.1.1. Redundant Attributes	179
6.3.1.2. Attributes that Abstract Representation Differences	180
6.3.1.3. Gathering Usage Data	181
6.3.1.4. Default Attributes	182
6.3.2. An Implementation of Lists with Arrays	182
6.3.2.1. Local Declarations and Statements	185
6.3.2.2. Refining an Attribute	186
6.3.2.3. Use of a Manager Parameter	186
6.3.2.4. Requiring an Implementation Class as a Parameter	186
6.3.2.5. Implementing Generalization Classes	187
6.4. A Program: Sort	187
6.4.1. Explicit Manager Presence	188
6.4.2. User-Defined Representation Information	189
6.5. A Policy: Minimum Time and Space	190
6.5.1. Policy Algorithm	193
6.5.2. Global Properties	194
6.5.2.1. Separate Evaluation Functions	194
6.5.2.2. Use of Local Procedures	195
6.5.2.3. Block-at-a-Time Analysis	195
6.5.3. Local Properties	195
6.5.3.1. Selections within a Block	195
6.5.3.2. Using Attributes	196
6.5.3.3. Trying all Implementations	196
6.6. Transformed Program	196
6.6.1. Annotated Program	197
6.6.2. Object Listings	201

6.7. General Procedures	204
6.8. Recursive Procedures	207
6.8.1. Application Program	207
6.8.2. Object Listings	208
6.9. Some Alternative Policies	210
6.9.1. Dynamic Programming	210
6.9.2. Branch and Bound	213
6.9.3. Hill-Climbing Heuristic	216
6.9.4. Simple Constraint	221
7. Implementation	225
7.1. Phase Descriptions	225
7.1.1. ML: Parser	227
7.1.2. PURIFY: Input Reader	228
7.1.3. NAME: Scope Linking	228
7.1.4. SETUPC: Setup Class Declarations	229
7.1.5. SETUPP: Setup Procedure Declarations	229
7.1.6. SETUPI: Setup Procedure Implementations	229
7.1.7. ELABS: Type Checking and Semantic Analysis	230
7.1.8. PREDEF: Locate and Bind Predefined Identifiers	230
7.1.9. MARKC: Create <i>ClassDecl</i> Objects	231
7.1.10. RPOLIC: Implementation Selection	231
7.1.11. ELABI: Feasibility Checking	231
7.1.12. WALK: Write Implementation Decisions	231
7.2. Component Descriptions	232
7.2.1. Name Components	232
7.2.1.1. Create_Class	232
7.2.1.2. Create_Call	233
7.2.1.3. Create_Local_Instance	233
7.2.2. MYLET: Function Call Utility	233
7.2.3. LOOKUP: Symbol Table Processing	233
7.2.4. COMP: Comparing Objects	234
7.2.5. GC: Garbage Collector, TIMER: Metering, SW: Switches	234
7.3. Translator Performance	234
7.3.1. Static Measurements of the Translator	235
7.3.2. Static Measurements of Some Programs	237
7.3.2.1. Predefined Environment	238
7.3.2.2. Abstract Data Type Specifications	238
7.3.2.3. Abstract Data Type Implementations	238
7.3.2.4. Application Programs	239
7.3.2.5. Measured Sizes of Programs	239
7.3.3. Dynamic Measurements of Translator	240
7.3.3.1. Measuring Elaboration with Specifications	240
7.3.3.2. Measuring Elaboration with Implementations	246
7.3.3.3. Measuring Elaboration with Realizations	248
7.4. Conclusions about the Prototype	253
8. Retrospective on the Language Design and Implementation	255
8.1. Abstract Data Type Features	255
8.2. Describing and Selecting Abstract Data Types	258
8.2.1. Attributes	258

8.2.2. Policies and Possibility Trees	259
8.2.3. Anonymous Possibility Tree Nodes	260
8.2.4. Parse Tree Availability	261
8.2.5. Decorating the Possibility Tree	263
8.2.6. Simpler Models	263
8.2.7. External Selection Language	265
8.2.8. Program Creation Systems	266
8.3. Automatic Processing of Paragon Programs	266
8.3.1. Heterogeneous Data Structures	267
8.3.2. Global Feasibility Checking	267
8.4. Summary	269
8.4.1. Contributions: Abstract Data Types	269
8.4.1.1. Refining Specifications	269
8.4.1.2. Implementing Abstract Data Types	270
8.4.1.3. Combining Representations	270
8.4.1.4. Uniform Object Notation	270
8.4.2. Contributions: Representation Selection	271
8.4.2.1. Describing Abstract Data Types	271
8.4.2.2. Organizing Global Program Optimization	271
8.4.2.3. Programmer Control of Selection Criteria	271
8.4.2.4. Feasibility Analysis	272
8.4.3. Future Areas for Related Work	272
8.4.3.1. Uniform Procedure, Iterator, Object Semantics	273
8.4.3.2. Value of Multiple Representations	273
8.4.3.3. Program Representations for Programmer Manipulation	273
8.4.4. Conclusions	274
Bibliography	275
Appendix A. Additional Paragon Features	283
A.1. Lexical Elements	283
A.1.1. Character Set	283
A.1.2. Identifiers	283
A.1.3. Literals	284
A.1.4. Special Symbols	284
A.1.5. Reserved words	284
A.1.6. Comments	284
A.2. Object Creation Expressions	285
A.3. Most Preferred Match	285
A.4. Initial Environments	287
A.5. Restricting Environments	287
A.6. Environments for Parameter Elaboration	288
A.7. Inheriting Parameters	290
A.8. Sharing Implementations	293
A.8.1. Subsuming Implementation Paths	294
A.8.2. The Environment of the Object	295
A.8.3. Parameters in a Shared Implementation	296
A.8.4. Variable Interaction	296
A.8.5. Elaboration of a Shared Implementation	296
A.9. Procedure Constraints	297
A.9.1. Constraints that Check Matching	297

A.9.2. Combining Constraints	297
A.10. Self-References	298
A.11. Statements	299
A.11.1. Statement Structure	299
A.11.2. Expressions as Statements	299
A.11.3. Subprogram Control Statements	300
A.11.3.1. Return Statement	300
A.11.3.2. Yield Statement	301
A.11.4. Conditional Statement	302
A.11.5. Loop and Loop Control Statements	303
A.11.5.1. For Loops	303
A.11.5.2. While Loops	304
A.11.5.3. Exiting Loops	305
A.11.6. Goto Statement	305
Appendix B. Paragon BNF	307
B.1. Notation	307
B.2. Program Structure	308
B.3. Declarations	308
B.4. Statements	310
B.5. Expressions	311
B.6. Name Components	311
Appendix C. Conventional Design Issues	313
C.1. Iterators	313
C.2. Type Parameters	314
C.3. Literals	316
C.4. Declaration Verbosity	318
C.5. Expression Verbosity	319
Appendix D. Glossary	321
Appendix E. Abstract Data Types Used in the Examples	331
E.1. Overview of Sets	331
E.2. Overview of Lists	332
E.3. Assumptions about Attribute Procedures	332
Appendix F. Applications Programs	335
F.1. Set Maximum	335
F.2. Insertion Sort # 1	336
F.3. Insertion Sort # 2	336
F.4. Merge Sort	337
F.5. Transitive Closure	338
F.6. Huffman Encoding	339
Appendix G. Sample Output of Translator	343
Index	357

List of Figures

Figure 3-1: . An Object Consisting of 3 Simple Objects	35
Figure 3-2: Nested Simple Objects that are not an Object	36
Figure 3-3: Another Object with 3 Simple Objects	36
Figure 3-4: A Simple Object with Parameters	37
Figure 5-1: Simple Possibility Tree	132
Figure 5-2: Selecting <i>Implementation1</i> for <i>x</i>	134
Figure 5-3: Changing <i>x</i> to <i>Implementation2</i>	135
Figure 5-4: Adding Procedure Implementations to the Possibility Tree	136
Figure 5-5: Reusing old Procedure Local Instances in a Possibility Tree	137
Figure 5-6: A Possibility Tree with only <i>Implementation4</i>	138
Figure 5-7: A Possibility Tree with <i>Implementation4</i> and <i>LV1</i>	138
Figure 5-8: Picking <i>Implementation3</i> after <i>Implementation4</i>	139
Figure 5-9: Part of an Infinite Possibility Tree	159
Figure 7-1: Phase Diagram for the Paragon Translator	226

List of Tables

Table 7-1: Static Sizes of Translator Phases	235
Table 7-2: Static Sizes of Translator Components	236
Table 7-3: Static Sizes of Program Fragments	240
Table 7-4: Phase Measurements for Semantic Analysis	241
Table 7-5: Component Measurements for Semantic Analysis	243
Table 7-6: Dynamic Performance of Feasibility Checking	247
Table 7-7: Dynamic Performance of Policy Procedure Execution	251
Table 7-8: Unit Execution Times of Policy Procedure	252

Acknowledgements

יהי ביתך בית ועד לחכמים
והיו מתאבק בעפר רגליהם
והיו שותה בצמא את דברייהם

פרקי אבות א:ד

Let your house be a meeting place
for scholars,
and sit in the dust of their feet,
and drink in their words with thirst.

Pirkei Avos 1:4

One usually acknowledges ones committee, friends and relatives in the acknowledgement section of a thesis. Such a narrow view unfairly reflects the way that research is conducted at CMU. During my tenure here in Pittsburgh, I have spent a great deal of time listening to and learning from other people: fellow students, faculty, official visitors and random hanger-ons. Each contributed to my education and is in some way responsible for my completing my Ph.D. degree. The wealth of opportunities at the Computer Science Department made my studies an exciting and memorable adventure.

Nevertheless, my day-to-day contact with several people helped organize and advance my work on this thesis. Andy Hisgen and Jonathan Rosenberg were always willing to listen to each bizarre new idea and provide helpful suggests and criticisms. Elaine Kant patiently watched my research go through its ups and downs and provided me with technical feedback and encouragement whenever each was needed. Peter Hibbard and John Nestor reminded me that the art of language design is still largely an art. Like all apprentices, I was glad to have these masters around me for advice and help. Cynthia Hibbard read through hundreds of pages of drafts, checking my writing and offering suggestions to improve the writing style. Michael Conner carefully read initial drafts of this thesis and offered helpful technical suggestions.

Nearly everyone who works on a thesis can attest to the frustrating and overwhelming effort it requires. My wife Vera took care to ensure that I never let this thesis oppress me. Without her, this thesis might have never been finished. Last, but not least, I wish to thank her.

Mark Sherman
July 6, 1983

Abstract

This thesis describes a set of language features that supports the specification, implementation and selection of data abstractions. The effectiveness of these features is illustrated through a language, called Paragon, developed for the thesis. Novel features of Paragon include:

- Multiple inheritance of classes (the basic encapsulation mechanism);
- Multiple procedure implementations for a procedure specification;
- Iterators;
- User-provided descriptions of abstract data types;
- User-provided strategies for making representation-selection decisions;
- Compile-time selection of a procedure implementation for each procedure call;
- Compile-time selection of variable representations.

Representative Paragon programs illustrate how this language can be used for defining multiple, simultaneous and interacting implementations of abstract data types. In addition, some refinements of the data abstraction paradigm, such as generalized specifications and shared specifications, are defined in the thesis and illustrated with Paragon programs. I then show how the type-hierarchy facilities in Paragon can be combined with a semi-automated, representation-selection mechanism and some representation-selection strategies using Paragon's notation are provided. To show how Paragon can be implemented, I describe the design of a translator and provide some measurements of a prototype. This prototype demonstrates that the conventional compiler technology can be used for implementing type hierarchies, though it does illustrate possible problems with separate compilation when using multiple, simultaneous implementations of abstract data types. Finally, a critique of the language is provided.

Chapter 1

Introduction

This thesis discusses a new programming language called *Paragon* that supports the specification, implementation and selection of data abstractions. The language uses type hierarchies to specify and implement abstract data types. Further, the Paragon language design integrates the abstract data type facilities with a semi-automatic procedure for making implementation choices for the variables in a program. A prototype for the Paragon design was written and run on several example programs. All of these aspects are considered in detail in this thesis.

In this introductory chapter, the motivation for pursuing this work is presented, followed by a summary of the main results of the thesis. This chapter ends with a discussion of how the rest of the thesis is organized.

1.1. Motivation

Modern software has grown to such size and complexity that programmers can no longer manage all of the details of the programs they write. This lack of management causes the programs being created to be improperly specified (they do not accomplish what the user intended), incorrectly implemented (they do not accomplish what the programmer intended), and inefficient (they produce the wrong answer slowly and at great cost). Programming methods that promote the management of the details of a program can help control the size and complexity of modern software, and in turn, promote the production of correct and efficient systems.

1.1.1.1. The Use of Abstraction and Refinement

A successful method of controlling complexity in other disciplines is *abstraction*, that is, the suppression of irrelevant details. Various abstraction methods have been introduced into the programming task, notably control abstraction and procedural abstraction. Control abstraction usually takes the form of *while* loops, *repeat* loops, and *if* statements, each of which suppress the details of specifying nonsequential program flow. Procedural abstraction provides a way for a programmer to specify a black box that can transform some set of values into another set of values while suppressing the details of how the transformation is accomplished.

Although the abstractions initially suppress some details, these details are needed in the final program. The process of introducing details is called *refinement*. Sometimes the refinement is automated, as when a compiler automatically translates a *while* loop into an appropriate sequence of test and jump instructions. Sometimes the refinement is performed by the programmer, as when the programmer writes the code that describes how the specified black box actually works.

Refinement does more than introduce the details suppressed by abstraction. Refinement is also a selection and binding process. There are usually many different models that meet the requirements of an abstraction. For example, a common procedural abstraction is *Sort*. In an abstract sense, a sort procedure accepts a sequence of data and produces a permutation of that sequence that meets a specified ordering relation. There are many different algorithms that meet such a specification, any one of which meets the abstract requirements. The binding of a sort black box in a program to the selected algorithm is a refinement of the program.

Binding details to abstractions reduces the number of choices that a programmer can make for further refinements in the program. For example, if a choice is made to represent an ordered sequence of data as a linked list, a search procedure operating on that sequence can not use a binary search method. The refinement of the abstract sequence to a linked list reduces the number of choices for a searching procedure. As a program is refined further, the program becomes less abstract, more filled with details and more constrained. Therefore refining a program introduces inflexibility.

This inflexibility adversely affects program development and maintenance. As a program is

being developed, a programmer may not know which refinement to choose but a programmer has to choose one so that development may continue. Later the programmer might discover that the wrong decision was made, but the inflexibility introduced by previous refinements hinders a better approach from being implemented. This problem is exacerbated for program maintenance since only the fully refined program is available. Because the costs of maintaining a program are far greater than the cost for initial development, inflexibility in a program can exact a high price over the lifetime of a program.

Clearly, an approach is needed that introduces the refinements for constructing a program without eliminating the abstractions. Techniques for introducing details without obscuring control and procedural abstraction are being widely adopted. In control abstraction, the abstraction is provided by the programmer using structured programming techniques and the details are mechanically generated by a compiler. Because of the mechanical nature of the refinement process, a programmer can confidently change an abstraction and rely on the compiler to insert faithfully new details as necessary. In procedural abstraction, the programmer adopts a convention that the interface of a subroutine will remain an invariant abstraction that may be used by the rest of a program. Further, *only* the abstract interface of the procedure may be used by the rest of the program. Because the program using the subprogram relies only on the abstract interface, the refinements inside of the subprogram may be changed without affecting the rest of the program. So for both control and procedural abstraction, there are refinement techniques that retain much of the abstraction, and hence, much of the flexibility.

However, control and procedural abstractions have been used for many years. A newer form of abstraction, data abstraction, is becoming widespread and its refinement techniques are not well developed.

1.1.2. Data Abstraction

Data abstraction is based on the observation that programs conceptually operate on abstract objects that have specific properties unrelated to a computer. For example, a program simulating a traffic intersection operates on objects that represent cars, trucks, streets, and traffic lights. Since the program is ultimately run on a computer and does not manipulate concrete cars, some transformation must be made from the abstract objects to concrete objects that a computer manipulates. The refinements that effect this transformation usually require the addition of a great number of details, and unless carefully

done, will cause confusion in the programmer, inflexibility in the program and ultimately, errors in the finished product.

1.1.2.1. The Simple Model of Data Abstraction

There are emerging methods for refining data abstractions that provide a limited way to control the inflexibility and confusion that results from transforming program objects into computer objects. These methods require that each kind of object manipulated by the program have two parts: a *specification* that describes the actions that may be performed on the object, for example, start a car or stop a car; and a *representation* of the object in terms of computer objects, for example, a car is represented by three integers that hold data about the number of people in the car, the serial number of the car and the make of the car. A special piece of a program, called a *module*, provides a set of subprograms that implement¹ the operations that may be performed on a car. Inside of this module, a programmer may refer to the representation of the object in terms of the computer objects. Outside of this module, only the specified operations may be used to manipulate the representation of the object.

Unfortunately, the view that each kind of object be split into two parts is too simple. Although the methodology for building systems recognizes the need for layering for many purposes [Cheatham 79, Parnas 74], the view of providing layers of specifications for abstract objects has not been widely embraced. Yet the single layer of specification is inadequate for many kinds of specifications. Further, multiple representations of an object are not well supported and interactions between representations are not permitted. Each of these problems will be considered in turn.

1.1.2.2. The Limitations Imposed on Abstract Data Type Specifications

The single, isolated specification in a module is too restrictive. Other kinds of specifications that a programmer may wish to write include a specification that is a refinement of another, related specifications that are not refinements of one another and implementation-independent specifications. Each of these three kinds of specifications is illustrated below.

First, one kind of program object may be a refinement of another. For example, a Plymouth

¹The data abstraction literature sometimes uses the word *representation* for the definition of local storage of an object and the word *implementation* for the code that makes up the procedures in a module. It is now becoming accepted that the information in an abstract object may be encoded in either the state of the local storage or in procedures that operate on local storage and so the words *implementation* and *representation* have become interchangeable. They are used interchangeably in this thesis.

object is refinement of an Automobile object. Thus the specification for a Plymouth should be some refinement of the specification for an Automobile. Yet the described method of data abstraction allows only disjoint pairs of specifications and representations, not collections of related specifications and representations. The simple data abstraction method requires different kinds of program objects to be refined independently even when one specification may be a refinement of another's specification.

Second, objects may be related even if one is not a refinement of another. This relationship might be made explicitly by the specification of several objects in a single module or might be made implicitly by the specification of type parameters a module. Neither is permitted in the simple model of data abstraction.

In the simple model of data abstraction, each module may specify exactly one kind of object. However, some specifications are related, such as keyboards and displays. They are clearly separate objects: one might desire many displays to be attached to one keyboard or many keyboards to share a display. Yet they are related: when operating in half-duplex mode, typing a character on a keyboard causes a character to appear on the display. Since the abstract objects, keyboard and display, are related, their specifications should be related and a data abstraction facility should allow both specifications to appear in a single module.

The simple model of data abstraction also provides no facilities for families of specifications. Yet many objects have similar structures. For example, nearly all symbol tables have the same structure: a collection of pairs, where each pair consists of a key and some data. Typically, the keys belong to one type and the data to another type. In the simple approach of data abstraction, every symbol table that uses a different key type must have its own specification and representation. There is no way of defining a class of symbol tables that can be related with another class of objects, namely the different types of keys. Yet the specifications and representations for all symbol tables are nearly identical. It should be possible to factor out the common parts of the specifications and representations into a single specification and representation. Later, a programmer should introduce those details necessary for any particular symbol table as parameters rather than by creating new specifications and representations.

A third way in which specifications in the simple model are too restrictive is their lack of implementation independence. The simple model places strict rules on the relationship between specifications and implementations. In particular, the information available to an

implementation is exactly that information provided by the specification, no more and no less. A simple example can illustrate this. The specification of a typical sort procedure requires that the elements to be sorted have a comparison procedure. Any implementation of the sort procedure may use such a comparison procedure, but nothing else. Because the specification is not restrictive, it prohibits bucket sorting, since the bucket sort algorithm requires that the elements to be sorted come from a cross product of ordered sets and that the set of resulting tuples be well founded. Sometimes the opposite problem occurs and the specification is too restrictive. The specification for sorting might require that the elements to be sorted be tuples in a well founded set. This limits the types of elements that may be sorted since many objects may be compared without having a tuple structure. Such a specification effectively prohibits sorting to be done on objects where bucket sort is not feasible. These problems also occur with data abstractions. The specifications for the elements to be stored in a symbol table may require the elements to have a hash procedure defined on them (for example, see the symbol table example on page 164 of the Alphard Book [Shaw 81]). Such specifications limit the possible implementations of symbol tables to those that use hashing functions and those that do not use any element specific functions. In all of these cases, the problem is that information about the refinement process has leaked from the implementation to the specification. A more general facility would include details of refinement where they are appropriate.²

1.1.2.3. The Limitations Imposed on Abstract Data Type Implementations

Besides the inadequate support for writing specifications, the simple model of data abstraction does not adequately support multiple implementations of a specification. However, these multiple implementations can be quite useful. For example, an abstract array object allows the assignment and retrieval of data via a list of indices. Two common representations of arrays in linear memory are row-major order and column-major order. Normally it makes little difference which order is used. Sometimes one representation gives a better program performance, for example, because of paging requirements. Sometimes a representation is necessary for properties unrelated to the operations given in the specification. For example, another program may be providing the array in a predetermined format, such as a Fortran subroutine providing an array in column-major order. Therefore it is

²The restriction imposed by the simple model is not unmotivated. By insisting that all specifications available for the implementation be present in the specification, a compiler may separately check at compile time that the use of a data abstraction is legal, that an implementation that meets the specification, and that both checks are sufficient for guaranteeing that the resulting program can execute.