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

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

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

Table of Contents

List of Figures

List of Tables

Acknowledgements

יהי ביתך בית ועד לחכמים
והוי מתאבק בעפר רגליהם
והוי שותה בצמא את דבריהם

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 I: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. 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 implement[1] 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

---

[1]The 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 to too restrictive. The specification for sorting might require that the elements to be sorted be tuples in a well found 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

[2]The 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.