Information Structures

A UNIFORM APPROACH USING PASCAL

B. J. Lings

Information Structures

A UNIFORM APPROACH USING PASCAL

B. J. Lings
Department of Computer Science
University of Exeter

LONDON NEW YORK
Chapman and Hall

First published in 1986 by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE
Published in the USA by
Chapman and Hall
29 West 35th Street, New York, NY 10001

Printed in Great Britain at the
University Press, Cambridge

ISBN 0 412 26490 0 (hardback)
ISBN 0 412 26500 1 (paperback)

British Library Cataloguing in Publication Data

Lings, B. J.
Information structures: a uniform approach using Pascal.—(Chapman and Hall computing)
1. Data structures (Computer science)
I. Title
001.64'42 QA76.9.D35

ISBN 0-412-26490-0
ISBN 0-412-26500-1 (Pbk.)

Library of Congress Cataloging in Publication Data

Lings, B. J., 1950-
Information structures.
Bibliography: p.
Includes index.
1. Data structures (Computer science) 2. PASCAL (Computer program language) I. Title.
QA76.9.D35L56 1986 001.64'2 85-4209

ISBN 0-412-26490-0
ISBN 0-412-26500-1 (Pbk.)

Preface

The study of computer science has become well-enough established for there to be a certain uniformity across the basic course offerings of a wide range of universities and colleges. This book covers those aspects of a computer science course normally referred to as 'Information Structures' or 'Data Structures'.

The book is aimed at students, typically in their first and second years of study, who need a clear presentation of course material in a cohesive framework, which puts the wealth of information about data structures into context. The emphasis is placed on demonstrating the development of ideas and techniques, rather than simply on presenting results.

The approach taken is that of data abstraction, with emphasis on the practical aspects but including discussion, where appropriate, of the underlying theory. The vehicle used for developing examples, of which there is a wealth, is the language Pascal. Although Pascal does not explicitly support data abstraction, it is still the best language readily available for teaching purposes.

The techniques of information hiding, specification, data type realizations and tuning are all covered in a homogeneous structure. The book explores these techniques as well as the construction and analysis of the major data structures. In this way, new material can be quickly fitted into its appropriate place in the materials available to system developers.

The book is divided into three parts. Part One is suitable for students entering an introductory course in information structures after already being exposed to the language Pascal. As an informal approach to data-abstraction theory and practice, it is intended to be self-contained. Part Two introduces the idea of parametrized data types and looks at three which are fundamental in computer science: set, tree and graph. The material is conventional, but the presentation once again stresses the disciplined approach of abstraction techniques. Part Three covers material often placed under the subheading 'Sorting and Searching'. It is a study of a simple associative data type.

My thanks are due to my wife, Pam, who has proof-read the whole typescript in its earliest stages, and to Marlene Teague, whose major efforts transformed my manuscript into machine-readable text.

Contents

PART ONE

The concept of type

1

The Pascal type concept

1.1 INTRODUCTION

This book is all about data types. It is assumed that you have already had an exposure to the language Pascal and that the basic notion of a type is therefore not new to you. What will probably be new is the approach taken to studying such types: we shall be looking at the fundamental concepts involved and learning how we can use this understanding in structuring our programs.

Programs are composed, as far as we are concerned here, of two major components:

(i) An algorithm.
(ii) A set of data items, each item being associated with a data type. These represent the 'state of play' at any given moment during the execution of the algorithm. Only 'appropriate' operations may be performed on these data objects by the algorithm. As an example, addition is an appropriate operation on two objects each associated with type integer.

You should already have some exposure to algorithms and you may even have studied the theory of algorithms to some extent. Only the notion of an algorithm is assumed as a prerequisite to reading this book: the Bibliography lists some suggested texts for this purpose.

Our intention is to study practical aspects of component (ii) above and to present material on data items in a manner consistent with approaches suggested by current research and (to a lesser extent at the present time) current practice.

The scope of the book is that portion of a computer science undergraduate course normally entitled 'Data Structures' or 'Information Structures'. It is suitable as a text for such courses. The approach taken is that of data abstraction, emphasizing the advantages and techniques embodied in the principles of data independence and information hiding. The book is split into three parts.

Part One is suitable for students entering an introductory course in

information structures after already being exposed to the language Pascal. It covers the major concepts of data-abstraction techniques, and discusses basic data types (integer, boolean, char), simple user-defined data types and the 'structured' types related to those provided by Pascal. As an informal introduction to data-abstraction theory and practice it is intended to be self-contained.

Part Two introduces the idea of parametrized data types and looks at four such information structures which are fundamental in computer science: set, tree, graph and list. Once again the basic material is conventional: a study of these data types and a comparison of the various implementations (we will use the term realizations for reasons that will become clear) open to us. The presentation, however, will once again stress the approach of data abstraction. As an example, the tenth chapter of the book uses type graph in order to demonstrate how to develop programs from abstract algorithms, in such a way as to improve their longevity and reliability.

Part Three covers material often placed under the subheading 'Sorting and Searching'. It is a study of a simple associative data type. Associative (or key) retrieval is a simple abstract notion. Its realizations can be complex and are very varied.

One aspect of this study which will pervade all sections is the effect usage has on choosing an optimum realization for a data type. Each different realization will, in general, favour a different subset of its defined operations. It is important to bear this in mind when reading the book, and to consciously assess each suggested realization by determining its behaviour with each operation.

1.2 WHAT IS A TYPE?

1.2.1 A type as operations + domain

You will have come across the notion of a data type in Pascal. Pascal is a strongly typed language, by which we mean a language in which every defined data item is associated with a specified data type. For example, in introducing two variables **SUBTOTAL** and **TOTAL** we may have a definition of the form

```
var SUBTOTAL,TOTAL:INTEGER
```

at the head of a Pascal program block. Let us recap on the implications such a statement has in a Pascal program.

(i) It identifies those values which **SUBTOTAL** and **TOTAL** may tal
For example

```
TOTAL := 6
```

is perfectly legal, whereas

SUBTOTAL := TRUE

is not. The set of legal values associated with a type is called its *domain*. The number of elements in a domain is called the cardinality of the domain. The cardinality of the boolean domain is 2, that of the Pascal character domain 128 and so on.

(ii) It is a message to the compiler concerning the way in which **SUBTOTAL** and **TOTAL** are to be represented. In this case the compiler may, perhaps, deduce that one word of memory should be allocated to each data item, and that values will be stored in twos-complement binary representation. We will see later that this 'implication' is in fact of a different nature to (i) and (iii).

(iii) It is a message to the compiler concerning the way in which **SUBTOTAL** and **TOTAL** are to be used. Each of the pre-defined object types has associated with it a set of operations. In the case of integer these will include +, −, **MOD**. All manipulations of integers must (ultimately) be specified in terms of these operations.

Therefore we have as a fact in Pascal that

SUBTOTAL + TOTAL

is a meaningful expression whose value is obtained by adding together the current values of **SUBTOTAL** and **TOTAL**, whereas

SUBTOTAL AND TOTAL

is not a meaningful expression (it is in some weakly typed languages) and is therefore defined to be 'illegal'. Such a statement will be identified as erroneous by the compiler.

We note three things about the legal expression

SUBTOTAL + TOTAL (*)

(i) It has a parallel in mathematical integers. By looking at the mathematics of the expression and the values of the two variables we can deduce the value that the expression should have (and will normally have – see (ii)). In mathematics type integer can be specified starting from Peano's axioms. Later we will attempt to specify all types that we use in a program, many of them informally at this stage. To specify a type is to give a method, independent of the given system, by which we can establish the expected result of each operation performed. Normally we try to be formal in our specifications (mathematical) because the more rigorous we are the more confident we can be of our pronouncements.

(ii) The parallel with mathematical integers is not complete. Peano's

axioms are for integers with an infinite domain. On the other hand, computers are finite. The immediate impact of this is that (*) may yield a perfectly 'normal' value when calculated using our specification, but may cause *overflow* on a typical machine when, for example, **SUBTOTAL** and **TOTAL** are both very large integers. We note the problem here, but shall not attempt to develop these thoughts at this stage.

(iii) Becoming less esoteric, expression (*) tells the compiler what code to generate. The compiler knows that **SUBTOTAL** and **TOTAL** are both integers, so that + refers to integer arithmetic. The 'integer addition' instruction from the machine instruction set may therefore be called for. This instruction is an *implementation* of the integer '+' operation. Each of the defined operations for a type must have such an implementation defined.

None of this is new: it is merely documenting what we already know about types from our Pascal experience. Much of it we may never have consciously formulated: it will have remained implicit in our own model of what our programs are actually about.

Let us summarize, then, what we mean by a (Pascal) type:

- A type *specifies* a domain: a set of legal values.
- A type *specifies* a set of legal operations on those values and the results of applying these operations (the semantics).
- A type *prescribes* a representation for values from its domain: this representation is chosen by the compiler writer. An example would be the representation of integer values by twos-complement binary values.
- A type *prescribes* an implementation for each operation from its operation set: these implementations are also chosen by the compiler writer, who in turn is constrained by the architecture of his machine.

We chose our words carefully above, because whereas the domain and operation set for, say, an integer are fixed for all Pascal implementations (and for those of most other languages) the representation of integers and the implementation of integer operations varies widely and is particularly dependent on the hardware of a system. Obviously the implementation of an operation is intimately bound up with the representation of its values; change this representation and you must change the implementation.

Before we leave this section, note the correspondence which will form the basis of an approach to building our own data types. If we look at the summary it is not hard to see that, for each data type

- Operations are *specified* on values from the domain.

- Values from the domain are given a *representation* which associates each value from the domain with a value in a different domain (for example, that of twos-complement binary values in the case of integer).
- Operations are *implemented* by defining which operations are to be performed on values from the representation domain in order to achieve the desired (specified) result (for example, Logical Shift to achieve multiplication by 2).

In other words, there is an isomorphism defined between a type and its realization (the representation and implementation combined).

1.2.2 A type as operations only

It is a fact that the concept of a data item (a *variable* in Pascal) to denote a data value is not strictly necessary, and we could concern ourselves solely with operations. A variable is only 'visible' because operations are available which make it so, for example

```
WRITE(J)
```

In this case J is simply a shorthand notation for the expression from which its 'value' was constructed: say

```
MULTIPLY (2,2)
```

But 2 is itself only the shorthand (denotation) for another expression:

```
SUCCESSOR (SUCCESSOR (ZERO))
```

(see Peano's axioms). In this *applicative* notation we can see that data items are indeed unnecessary and can always be replaced by functional expressions. In particular, our **WRITE(J)** can be replaced by

```
WRITE (MULTIPLY (SUCCESSOR (SUCCESSOR (ZERO)), SUCCESSOR
(SUCCESSOR (ZERO))))
```

As we are dealing with a Pascal environment, presenting practical as well as theoretical aspects in that environment, we will retain the notion of a data item. If you are interested in the 'pure' approach you will find references in the Bibliography.

1.2.3 Defining type INTEGER

For us, then, type integer is realized by choosing a more basic type to represent it and by defining a correspondence (Fig. 1.1).

(i) Between integers and values from the representation domain.

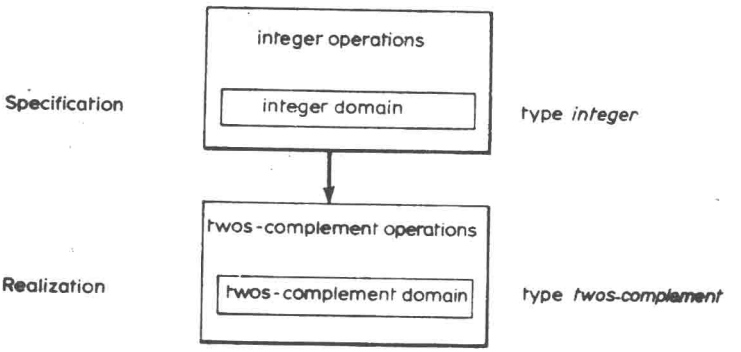

Figure 1.1 Type **INTEGER**.

(ii) Between integer operations and operations on the representation domain.

Examples

	INTEGER	TWOS-COMPLEMENT (16 bit)
Value:	−32768	1000000000000000
Value:	0	0000000000000000
Operation:	+	IADD (integer add)
Operation:	**DIV**	IDIV: return quotient
Operation:	**MOD**	IDIV: return remainder

We now introduce a notation for explicitly expressing all the facts contained in this correspondence. Such a definition is implicit in every Pascal program using integers: it will become more useful as and when we introduce our own data types. For simplicity the specification given as Fig. 1.2 is not complete.

We make the following observations about it:

(i) Under *operations* we give the *form* of each operation: + acts on two integers to give a result which is an integer.
(ii) The only operations allowed on the right of '→' in an implementation are those defined for the type in the *representation* or the type in question. Here, we are at the level of the Assembly language. If we were not, then a specification and realization for *binary twos-complement* would have to appear as well.

That concludes our section on 'What is a type?'. If you are more confused than when you started it is because we are having to force many issues into the consciousness which have hitherto, quite rightly,

Specification for **INTEGER**

Domain

−32768...32767

Operations

+, −, *, **DIV, MOD** : − **(INTEGER, INTEGER): INTEGER**

<, <=, =, >=, > : − **(INTEGER, INTEGER): BOOLEAN**

:= : − **(INTEGER, INTEGER)**

Semantics

Based on Peano's axioms, suitably modified because of finite domain

Realization of **INTEGER**

Representation

Binary twos-complement

Implementation

+(*X*, *Y*) → load *X*
iadd *Y*, etc.

Figure 1.2 Type **INTEGER**.

been left covered by 'black boxes'. As always, opening a black box leads to more complexity in our model of what a program actually means, and an increase in the jargon so necessary to allow brevity in future developments. As we move on you will see that the benefits of opening this box far outweigh the initial cost, so the exercise is well worth while.

It will be time well spent if you study this section until its concepts and technical terms are absorbed into your modelling kit. Once this has been done the rest of the book will make far easier reading.

Technical terms

- Strongly typed.
- Domain.
- Cardinality.
- Specification.
- Realization.
- Representation.
- Implementation.

1.3 THE BASIC PASCAL TYPES

In Pascal, as in all strongly typed languages, a number of data types are both specified and realized implicitly. These can be used by a programmer with no further definition.

On the positive side this means that the programmer is saved the problem of delving into details of the machine architecture in order to devise a realization for these very fundamental and frequently used types. Together they form the blocks from which the user's own types can be built.

On the negative side, it means that the user must accept whichever realization has been chosen by the compiler writer for these types. The chosen realization will almost certainly be a compromise which may be far from optimum for the particular circumstances which an algorithm dictates. For example, integers may be used primarily in input and output instructions; multiplication by 2 may use the general multiply instruction rather than a *shift*.

In Pascal the basic types include

INTEGER
BOOLEAN
CHAR
REAL

We say that these types are defined 'in the language prelude', that is, all programs can be written as if their specifications and realizations form a part of the code.

We have already seen what the language prelude could look like for type integer. Of course, the notation we used is not in the Pascal language and so we have to adapt it when we define our own types. We return to this issue in Chapter 2. For the moment, however, we keep the notation and look at another basic type **CHAR**.

The first thing we notice is that **CHAR**, unlike **INTEGER**, has no universal specification. Indeed, in most languages type **CHAR** has its domain specified in the language prelude by listing all elements in the domain. The cardinality of **CHAR** in Pascal is 128. As with **INTEGER** the domain is totally ordered. Each value in the domain has a unique predecessor (except one which we call 'low') and each value in the domain has a unique successor (except one which we call 'high'). The comparison operations ($<$, $=$, $>$, etc.) are therefore applicable. Two further operations are applicable, as indeed they are for type **INTEGER** and any other ordered type. These are the successor and predecessor operations. We left them out of the discussion of type **INTEGER** purely to simplify the exposition. We introduce them shortly, but first a remark on notation.

The operations listed in Fig. 1.2 for **INTEGER** are all binary operators (that is they all operate on two **INTEGER** values). We are used to writing expressions in infix notation, so that notation is adopted by Pascal. However, there is nothing particularly special about the notation. In fact