Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels
Editorial

Information Systems Research: A Teleological Approach?

The request to write this editorial came at a very opportune time, coinciding as it did with an intense examination of the development of the field of information systems and an analysis of the progress of IS research. I have therefore used this opportunity to focus my thoughts and outline some of my conclusions. By doing so I don't pretend to answer any questions, merely perhaps to stimulate thought amongst those SACJ readers involved in IS research.

The last fifteen years has seen a tremendous growth in the study of information systems. During this period a number of journals devoted to IS research appeared such as MIS Quarterly, The Journal of MIS, Information and Management and Data Base. There are now many research-based activities: the International Conference on Information Systems; the annual IS doctoral dissertation colloquium; and various awards for IS research contributions. Hundreds of universities worldwide have formed information systems departments with (reasonably) standard curricula.

Yet with all this, what has really been achieved from a research viewpoint? Are we any closer to understanding the true nature of information systems? Is there a general unified theory of information systems? Is there even an accepted, unique body of IS knowledge? The answer to all of these must surely be no.

We have, I believe, achieved precious little. Yes, we do understand something of IS development approaches. We understand a little more now than we used to about how users interact with systems. But to get back to the first question, do we really understand what information systems are and how they work? No. Which begs the question: Why not?

There are, again I believe, a number of reasons, but the foremost must be that the majority of people in the IS research community either reside in the business schools of the USA or are drawn from other disciplines. These people, it would appear, are researching for research's sake; to publish in order to secure tenure or develop a research track record, not to further the body of knowledge of the subject. There seems an almost frantic zeal to generate and test hypotheses, trying to adopt and pursue what is seen to be a "scientific approach". But there is very little focus - there can't be, or the answers to my questions earlier would be yes rather than no!

Let me hasten to add that there is nothing unique about these IS researchers. "Publish or perish" is still very much alive and well! But also they are really not all that different from other social scientists. As Nagel [3] observed:

"... in no area of social enquiry has a body of general laws been established, comparable with outstanding theories in the natural sciences in scope of explanatory power or in capacity to yield precise and reliable predictions ..."

Why should this be the case? Is it because the great intellects gravitate to the natural sciences and the social sciences pick up the second best who are incapable of generating these general laws? I hope not! The answer may well be that we have become locked into a particular research approach which is inappropriate to developing a body of social science, and more particularly, IS knowledge. Maybe we should be learning from our own source discipline (systems theory) and be developing a real research approach which complements our field of study.

To explore this further let me go back to the roots of information systems. What is an information system? Do we really have an accepted definition? Probably the most widely referenced is that provided by Davis and Olson [2]:

"an integrated, user-machine system for providing information to support operations, management and decision-making functions in an organization. The system utilizes computer hardware and software; manual procedures; models for analysis, planning, control and decision making; and a database".

Note how this emphasizes the man-machine interrelationship and underscores computers as a core component when they are not even necessarily a part of the information system. The worst aspect is that it does little to describe what a system is, and this may well be one of the causes of our research dilemma. Again, if we draw on systems theory then a more appropriate definition might well be: "a hierarchical set of procedures utilizing information to monitor and control organizational performance". Note that this definition fits with general systems theory that all systems have four basic foundations: cybernetics, hierarchy, control and information [1].
An additional aspect not apparently recognised by IS researchers is that the information system, just like any other system, biological or otherwise, suffers from the problem first identified by our own Jan Christiaan Smuts [4]: that of holism. Simply put, this says that the whole is greater than the sum of the parts. This means that information systems, unlike science, cannot be reduced to simple isolated fields of enquiry and then analyzed or tested using hypotheses and laboratory experiments from which elaborate generalizations may be inferred. They have levels of complexity with new factors emerging at each level. The problem with most of the current research is that it starts out with a reductionist approach and then focuses on the highest (or lowest) level. Thus the majority of the topics have as their target the interaction between user and computer or the management or application of technology. There is very little research that is taking place at fundamental level, that of developing a general theory of information systems. This is the teleological approach, searching for the natural laws and developing the theory based on deduction and logical development. Until we can advance that area of knowledge and, from a basis of these fundamental laws, develop a hierarchy of hypotheses that can then be tested, we will have little focus to our IS research. It will remain a fragmented, uncohesive smattering of the work of individuals who are merely grasping at tenure. There are few people who would today argue against the inclusion of information systems as a field of study at a university or as a fruitful research area. But until such time as we focus on the foundation theory, it will remain unstructured and immature.

References


Prof Peter Lay
Assistant Editor: Information Systems
Data Structuring via Functions

B H Venter

Department of Computer Science, University of Fort Hare
Private Bag X1314, Alice, Republic of Ciskei

Abstract

A programming language data model is introduced, based on the notions that variables are functions and types are sets of values. It is shown that, despite the simplicity of the underlying mathematical formalism, the data model is very expressive, and does not violate basic intuitions about variables and types. Keywords: Data Structures, Functional Data Model, Sets, Lists, Records

Computing Reviews Categories: D.3.3, E.2.1

Presented at Vth S A Computer Symposium

1 Introduction

Consider the following statements:

- A programming language type is only a set of values.

- A type constructor is an expression resulting in a set of values.

- Operators such as + and - are functions and do not form part of types.

- Expressions are function applications, and the type of an expression is the set of values representing the co-domain of the last function to be applied.

- A programming language variable is a functional mapping from the empty set to the set of values making up the type of the variable.

- An assignment statement partially redefines the rule according to which a variable maps the empty set to its type set.

From these statements, a formal model of the semantics of types, variables, assignments and expressions can be constructed. In the rest of this paper, this model will be referred to as the functional data model.

There are many other mathematical formalisms that can be used to model variables and types, but none are as simple and intuitive as the functional data model. Of course, since the functional model is so simple, it does not model all aspects of the type systems of typical programming languages. For example, the functional data model does not associate types with representations or with sets of operators. As a consequence, language theorists tend to explore more complicated formalisms.

Some of the more complicated formalisms used to model old programming languages are in turn used as the basis for designing new programming languages, which tend to be more consistent and complete than designs not based on some formalism. The lambda calculus has been a popular choice.

Given the simplicity and intuitive appeal of the functional data model, it is worth asking whether the functional model can profitably be used as the basis for designing a programming language which is consistent, complete and expressive. Earlier work by Shipman[6] and Buneman[1] already goes a long way towards proving that this can be done. However, the functional data model has not yet been used as a basis for designing an imperative language in the Algol tradition. The author is currently designing such a language, and the type system of the preliminary version of this language is presented in this paper.

2 Types as sets

According to the functional data model a variable such as

\[ F : \text{Integer} := 1; \]

should be viewed as syntactic sugar for the function

\[ f : \{\} \rightarrow \text{Integer}; f() = 1 \]

The type, Integer, is thus simply a set of values representing the co-domain of \( f \). Therefore, in the case of

\[ G : \text{set of Integer} := \{1, 2, 3\}; \]
one has to view set of as an operator which results in the powerset of its argument. Hence, the above is syntactic sugar for
\[ g : \{\} \rightarrow \{s | s \subseteq \text{Integer}\}; \ g() = \{1, 2, 3\} \]
A consequence of this view of types is that types A and B where

\[ A = \text{set of Integer}; B = \text{set of Integer} \]

represent the same set of values and are thus identical.
Hence, according to the functional data model, types are simply sets of values which need not be disjoint. This view of types is in marked contrast with the long standing "axioms" of language designers, namely:

- A value belongs to one and only one type[2].
- Types are not sets[4].

The full consequences of abandoning these "axioms" are examined by the author in another paper[7]. The main conclusions drawn there are that a consistent language design based on the view that types are sets that need not be disjoint:

- must use structural type equivalence rules
- cannot be completely type checked at compile time
- must separate operators from types
- must treat types as first class values
- must be fully polymorphic
- must allow types to vary at run-time
- must support abstract data types in a non-traditional way

This paper argues that the following syntax for declarations would suffice to declare types, constants and variables in a programming language design based on the functional data model.

constant-definition =
identifier '=' expression
variable-definition =
identifier ' ' expression ['=' expression]
expression =
expression \{('U' | 'R' | '-') expression\} |
(set | sets) of expression |
(list | lists) of expression |
(map | maps) from expression |
\{' , expression\} to expression |
set-value | non-set-expression

Note that no special syntax is needed to declare types. Where a type is normally expected, any expression resulting in a set of values will do. Type names can be introduced by declaring named constants with sets as values.

Naturally, some types will have to be predefined. The following seems the most widely expected:

\{True, False\} = \text{Boolean} \subset \text{Entity}
\text{Integer} \subset \text{Number} \subset \text{Entity}
\text{Character} \subset \text{String} \subset \text{Entity}

where \text{Entity} denotes the universal set (the set of all possible values) and

\text{Boolean} \cap \text{Number} = \text{Boolean} \cap \text{String} = \text{Number} \cap \text{String} = \{\}

3 Set values

Most languages supporting sets allow set values to be denoted by explicit enumerations such as \{2, 3, 5, 7\}.

Very few languages, however, allow a version of the most general notation for sets: \{x | \phi(x)\} where \phi(x) is some statement about x. Since the general notation is very expressive, the expressive power of a language will be greatly enhanced if it provides a version of this notation.

When devising a version of the general notation that is suitable for a programming language, the first thing to note is that it is not practical to always interpret x as potentially being any member of the universal domain of discourse. Thus, the programmer should be required to explicitly indicate the set of values that x could possibly be. For example:

\{x \in \{2 \ldots 9\} | \text{is-a-prime}(x)\}

It is also useful to allow x to be an expression, possibly containing more than one local variable. For example

\{a + b, a \in \{a', b'\}, b \in \{1', 2'\}\}

which is equivalent to

\{'a1', 'a2', 'b1', 'b2'\}

It is not difficult to generate the potential members of sets specified in the general format. Nor is it difficult to decide if a potential member qualifies for membership. However, a simplistic compiler faced with

\{x \in \text{Integer} | \text{is-a-prime}(x) \text{ and } x < 10\}

is likely to generate code that will consider all integers from min(Integer) to max(Integer). To do otherwise, the compiler will have to analyze the selection condition. Such an analysis is not trivial for arbitrary conditions. Moreover, in the example, \text{is-a-prime} may well have been defined in such a way that even a sophisticated analysis algorithm will fail to determine that it will return False for every argument value less than 2.

The potential generation of atrocious code for examples such as the one above is probably the reason why so few languages allow set values to be specified
using the \( \{ x \mid \phi(x) \} \) notation. However, potential inefficiency is not a good enough reason to eliminate an otherwise useful language construct. After all, even a simplistic compiler is likely to produce acceptable code for
\[
\{ x \in \{2 \ldots 9\} \mid \text{Is-a-prime}(x) \}
\]

Moreover, in a case where the set cannot be as easily specified by an explicit enumeration of values, for example
\[
\{ x \in \{1 \ldots 1000\} \mid \text{A-complicated-rule}(x) \}
\]
an optimising compiler may well compute the members of this set during compilation, whereas a programmer denied the use of the general notation would probably have used a run-time routine to compute the set, which would be much more difficult to optimise away.

The following syntax is proposed for set-values:
\[
\text{set-value} = \\
\{' \{ \text{expression} [\ldots] \text{expression} \} [\ldots] \}' \\
\{' \{ \text{all} [\text{expression}] [\text{where}] [\text{identifier} \in \text{expression} \} [\ldots] \text{expression} \} [\ldots] \}'
\]
Note that the noise words all and where are needed to make it clear that
\[
\{ \text{all } a+b \text{ where } a \in \{a',b'\}, b \in \{1',2'\} \}
\]
is not a set of three elements, of which the last two are boolean values. Such an interpretation is possible when given only
\[
\{a+b, a \in \{a',b'\}, b \in \{1',2'\} \}
\]

4 Lists

The expression
\[
\text{list of expression}
\]
is simply syntactic sugar for
\[
\text{maps from } \{1 \ldots \max(\text{Integer})\} \text{ to expression}
\]
with the understanding that the list
\[
[\{a', a\}]
\]
represents the mapping
\[
(1, \{a\}) \\
(2, \{a\}) \\
(3, \text{Undefined})
\]
\[
(\max(\text{Integer}), \text{Undefined})
\]
where Undefined is a predefined value not present in any predefined type but Entity.

The above definition allows all the normal list operations to act in the expected way, provided that the length of a list \( L \) is defined to be
\[
\max\{ \text{all } n \in \text{Integer} \mid L(n) \neq \text{Undefined} \}
\]
The same syntax and conventions used to define set values can be used to define list values (but using \([\ldots]\) instead of \(\{\ldots\}\)).

5 Strings

The predefined set String can be regarded as equivalent to
\[
\text{lists of Character}
\]
and the predefined set Character can be regarded as equivalent to
\[
\{ \text{all } S \in \text{String} \mid \text{length}(S) = 1 \}
\]
This is, of course, a circular definition. However, it doesn't matter, so long as it is not used to formalise the semantics of a language. To do the latter, one would have to specify Character as an enumeration, typically the ASCII character set.

The circular definition does have the advantage of making it clear that character 'a' and string 'a' denote the same value. This seems rather more convenient than continually having to convert characters into strings and strings into characters, even if this is only done conceptually. It also has the advantage that 'a' = ['a'], 'ab' = ['a', 'b'], and so on.

6 Arrays

The array construct of classical languages is simply a special case of the map type constructor provided by the proposed syntax. Thus, the classical declaration
\[
A : \text{array } 1 \ldots 10 \text{ of Integer};
\]
would be written as
\[
A : \text{map from } \{1 \ldots 10\} \text{ to Integer}
\]
and should be viewed as syntactic sugar for the function
\[
a : \{\} \rightarrow \{f|f : \{1 \ldots 10\} \rightarrow \text{Integer}\}
\]
Furthermore, A should be understood as \( a() \) and \( A[1] \) as \( (a())(1) \). Finally, assignments to elements of A should be viewed as incremental redefinitions of \( a \).

Note, however, that the construction of generalised array operators, makes it necessary to specify that type expressions may contain free variables in certain circumstances. The reason for this is that types Vector10 and Vector20, where
Vector10 = maps from \{0...9\} to Integer
Vector20 = maps from \{0...19\} to Integer

represent disjoint sets of values, and hence operator Sum20, where

\[ \text{Sum20} : \text{Vector20} \rightarrow \text{Integer} \]

may not be applied to any array of type Vector10. Using free variables instead, one may have

\[ \text{Vector} = \text{maps from \{L...U\} to Integer} \]
\[ \text{Sum} : \text{Vector} \rightarrow \text{Integer} \]

where type Vector is understood to be the set \( \bigcup_{L, U \in \text{Integer}} \{f | f : \{L,...,U\} \rightarrow \text{Integer}\} \)

The above definition would allow Sum to be applied to any map from a single domain, consisting of a contiguous range of integers, to Integer. Furthermore, by specifying that L and U are parameters of Sum, and that L and U must automatically be set to integer values representing the lower bound and upper bound of the first parameter, it becomes possible for Sum to determine the index range of an actual parameter.

7 Records

Most modern programming languages provide some form of record constructor, and most implementations of abstract data types make very heavy use of records. More recently, record-based type extensions have been used to integrate object-oriented programming ideas into a language of the Algol family[8].

The reader may therefore have been surprised by the omission from this proposal of a type constructor for records. The reason for this is that such constructors are very difficult to model as operators resulting in sets of values.

Consider the following (Ada-like) type definition:

\[ \text{type Stack is record} \]
\[ \text{Top : range 0..100;} \]
\[ \text{Elems : array 1..100 of Integer;} \]
\[ \text{end record;} \]

If the intention is to use Stack as an abstract data type, then the irrelevant detail that Stack values are records must be hidden from the client code. Specifically, the client code must not be able to do something like this:

\[ \text{S: Stack; put(S.Elems[S.Top-1]); put(S.Elems[S.Top]); S.Top := S.Top - 2;} \]

In conventional languages like Ada this is achieved by hiding the component selector operators, .Top and .Elems, from the client code. Correspondingly, a language based on the functional data model could provide a means of exporting the name, Stack, without also exporting the names Top and Elems.

However, what happens if a client module includes a definition of type Stack1? If type Stack and Stack1 represent the same set of values, the names Top and Elems are not hidden from the client, since they have to be available for use with variables of type Stack1.

Thus to support both records and information hiding in a language based on the functional data model, it is necessary to specify that different applications of the same record constructor yield different sets of values. This is much like saying that \(5+6\) should yield a different value every time it is evaluated.

8 Living without records

Records are not only difficult to marry with the functional data model. They cause problems even in conventional languages. For example, it is difficult to use records to define an abstract data type Person, as well as subtypes Student and Staff, so that some values can belong to all three sets while certain operations are restricted to values of type Student and others to values of type Staff. Wirth's type extensions[8] come close to allowing this, but fall short because he requires an extended type to be disjoint from the type it is extending. Thus, a record value of type Student, defined as
an extension of Person, may be assigned to a variable of type Person, but only with loss of information. Furthermore, a value of type Student cannot also be a value of type Staff.

Problems such as the above also occur when record types are used to model persistent information (databases), and have been discussed at length in [3]. Various solutions have been proposed, including some which are still record based (for example, semantic data models[5]). The "record free" functional data model introduced by Shipman[6], however, offers a far simpler solution.

In short, when using the functional data model, one declares groups of functions in the place of records. For example, instead of declaring:

```plaintext
type Person is
    record
        Name : String;
        Date-of-birth : Integer;
    end;
P : Person
```

one declares a group of functions:

```plaintext
Name : Person → String
Date-of-birth : Person → Integer
P : {} → Person
```

and declares Person to be a set of abstract values, for which no operators exist, apart from explicitly defined functions. Information hiding is achieved by exporting only those functions that are needed by clients.

Note furthermore that it is possible for a client module to define types Staff and Student to be subsets of type Person, and to define additional functions:

```plaintext
Student-number : Student → Integer
Staff-number : Staff → Integer
```

Furthermore, if PS denotes a value which is a member of type Staff and type Student, then all of the following function calls are valid:

```plaintext
Name(PS)
Date-of-birth(PS)
Student-number(PS)
Staff-number(PS)
```

It is very cumbersome to achieve the same functionality using a record-based data model.

9 Pointers

Another type constructor that is very heavily used in implementing abstract data types, but absent from this proposal, is pointer to ...

It is not difficult to view a pointer variable as a function. For example:

```plaintext
P : pointer to Person;
```

can be viewed as a function of the following type:

```plaintext
P : {} → {f | f : {} → Person}
```

However, pointers are error-prone and troublesome. Moreover, a language supporting sets, lists, generalised functional mappings and abstract values, has no need of pointers.

10 Dynamic Types

The author has concluded elsewhere[7] that one of the consequences of adopting the functional data model is that a consistent language design must allow types to vary at run-time. In other words, the language must allow the set of values making up a type to be computed (and recomputed, if necessary) at run-time. Such types are called dynamic types, since they represent dynamically varying sets of values.

Dynamic types make good sense in database-oriented languages. For example, if one declares

```plaintext
Parts = {1, ..., 1000}
Suppliers = {1, ..., 200}
P : set of Parts
S : set of Suppliers
```

and one may also declare

```plaintext
Supplies : map from S to set of P
```

instead of

```plaintext
Supplies : map from Suppliers to set of Parts
```

then the constraints that Supplies may only return sets of values that are in P, and may only be applied to values that are in S, are expressed in their most natural form.

Note that no special notation is needed to allow dynamic types. Any expression which can only be evaluated at run-time, and is used in the role of type, represents a dynamic type.

11 Abstract values

The astute reader will have noticed by now that, although the existence of sets of "abstract" values have been referred to, no syntax for generating such sets have been presented. The reason is that dynamic types make it unnecessary to have such a syntax.

For example, instead of declaring Parts to be a large set of values, and P to be a variable of type set of Parts, as is done above, one would declare P to be a variable of type set of Entity, and Parts to be a constant function that returns the current value of P. Initially, P would have no elements: As new parts are added to the database, new abstract values...
(elements of Entity that are not yet included in any set but Entity) are added to \( P \).

To support this, the language only has to supply a function that delivers a different, unused element of Entity every time it is called.

12 Conclusion

Not much needs to be said to define the functional data model: Variables are functions whose definitions may be changed by assignment statements. Types are sets of values that act as membership constraints on the domains and ranges of functions. That's it.

Despite the almost ridiculous simplicity of its definition, the functional data model provides as much security and power of expression as any other data model. The same can be expected of a programming language based on this data model.

At this stage it cannot be stated with absolute certainty that such a language would be efficient or even implementable. However, there are many good techniques for implementing sets and functions mapping sets to sets. It therefore seems likely that an efficient programming language, based on the functional data model, can be designed and implemented.

References


FRD Investment in Advanced Computer Science Training

FRD press release

The Foundation for Research Development (FRD) will invest almost R1 million over the next three years in advanced training in computer science at three South African universities, namely the Rand Afrikaans University, Rhodes University and the University of Cape Town. The main objective of the programme is to promote advanced training of research manpower in the field, and to establish a research infrastructure for computer science in South Africa. Technology transfer between the academic world and the computer industry is also an important facet of the programme.

Although there is no great shortage of undergraduate students in computer science, very few continue with postgraduate studies. The reason for this is probably the very attractive job opportunities which exist for graduates in computer sciences in this country. As a result, the development of new expertise and high level research in the field is badly neglected.

'The FRD, together with the universities, has now given recognition to the urgent need for more research manpower in computer science. We hope that industry will support this initiative', said Professor Daan van Wyk, a member of the Executive of the FRD.

Amongst other things, the funds being made available will be used for bursaries, equipment, running expenses and the appointment and encouragement of research personnel as well as for the strengthening of ties with experts abroad through student exchange and visits by international leaders in the field.

Another important application of funds is the special provision being made for financial assistance to promising young students from all population groups who would otherwise not be able to continue their studies to postgraduate level. In particular, postgraduate studies for black students will be actively supported.

Regular forums where students and academics being supported by the programme can meet with the industry will be arranged by the FRD. 'This will not only ensure that the work being done at our universities is made known to the industry but also that it remains relevant to the requirements of the industry,' said Professor van Wyk.

The programme is to be established for a three year period. Thereafter the progress and products will be evaluated to determine whether it should continue for a further term.

ADA Courses and Workshop

July 1990

The Department of Electronic Engineering at the University of Natal, is offering three ADA courses and workshops in July 1990.

1. Full ADA Course and Workshop.
   5 Days. 2-6 July 1990. R1350-00.
   In this course, the student will learn the ADA programming language, ADA's support for data abstraction, information hiding, localization, and modularity. The student will gain experience with the ADA syntax and semantics for data and program structure, error handling, and the ADA task structure. This course is for practising programmers with experience in Pascal or C. Attendance is limited to 20.

2. Embedded/realtime Programming in ADA
   This course covers the ADA compilation systems, the role of the underlying runtime support environment, and various realtime issues associated with ADA. The course will include a case study. This course is for experienced ADA programmers. Attendance is limited to 30.

3. ADA Software Design and Design Tools
   This course presents the tools available for Object-Oriented programming with ADA, program design with ADA, and developing reusable ADA software. The ADA-9x program will be reviewed. This course is for software designers with some ADA experience. Attendance is limited to 30.

The courses will be given by Dr J L Tokar, Pyrrhus Software, Pittsburgh, USA, and Mr D C Levy, University of Natal.

Dr Tokar has been involved with the development, implementation and use of ADA for the past 10 years. She worked at Gensoft on one of the first ADA compilers to be validated, and at AT&T on ADA, UNIX and fault tolerance. She has a PhD in Computer Engineering from Clemson University, and now runs her own ADA consulting business in Pittsburgh.

Mr Levy is a Senior Lecturer in Electronic Engineering at the University of Natal. He has worked with embedded realtime systems for 12 years and ADA for 5 years.

Enquiries: Miss B le Breton
Phone: 031-816 2755
Fax: 031-811 733
Book Reviews

A Software Engineering Approach

Database technology has developed considerably over the past few years through the adoption of the relational mode. As a result database management systems are no longer limited to the larger installations. Performance improvements and developing standards are facilitating the spread of database usage. Effective utilisation of the data resource is a prime concern, but the subject area receives so much attention that it is often difficult to separate 'hype' from fact.

The emergence of software engineering, together with a methodological approach, puts database technology in context and it can be seen as a convergence of technologies rather than as an add-on to the conventional systems development life-cycle.

Hughes's book is a most readable text despite the nature of the principles involved, and the exercises provide a useful means of ensuring understanding. The book presents a well developed explanation of the subject by identifying and explaining the hierarchic and network models before looking at the relational model.

Not only are the important aspects presented but the shortfalls are clearly explained, and some attempt is made to project developments to illustrate the continuous refinement process.

The book focuses on the specifics, and each chapter becomes a useful guide in itself. Chapter 9, Commercial Systems an Database Machines, is an excellent summary of the current state of play, together with some suggestions as to further advances. By concentrating on the emerging standards the book avoids becoming a catalogue, and provides sufficient detail to be informative.

As a text for students it remains general enough to allow the individual to readily assimilate the information and apply it in a real environment. One does not need to have a detailed knowledge of Modula-2 since the examples are easy to understand and help to clarify the text. For the practitioner it is a useful text, since it provides an excellent understanding of the major concepts together with insight into trends. It is clear and concise, uncluttered by unnecessary information, and with sufficient references to encourage further reading.

By adhering to the band of emerging standards it should remain current for some time.

Programming Language Theory and its Implementation
Reviewer: Willem van Biljon, The EFT Company

The title of this book is misleading. The book is less concerned with the theory of programming languages per se than with theorising about programs written using various programming language paradigms. The implementation spoken of in the title is thus concerned with the implementation of theories developed about programs rather than with the implementation of programming languages, which was my first impression. However, this does not preclude the book from being an excellent text on program theory and verification.

The book is divided into three parts: Part I (Proving Programs Correct) deals with the problems of program specification (as is usual practice for texts like this, a toy language is defined for this purpose), the Floyd-Hoare logic, and a discussion of the mechanism of the proof of program correctness. Part II (The Lambda-calculus and Combinators) switches attention away from imperative programming to functional programming and discusses the fundamentals of functional programming, including the theory behind it (Lambda-calculus) and the theory of the modern implementation of functional languages (combinators). This last section is in fact the only part of the book that is in keeping with the expectations created by the title. Part III (Implementing the Theories) gives the reader an introduction to Lisp, and proceeds to present a complete verification condition generator for the language presented in Part I as well as a theorem prover to prove the validity of these verification conditions. These programs are written in Lisp, and the author also discusses how different versions of Lisp currently available may affect the implementation of these programs.

It is this last section that impressed me most. This is the first text on program verification that I have seen that gets down to the nitty gritty and provides actual code that implements a program verifier, albeit for a very simple language. In fact, throughout the book the emphasis is on the practical aspects of the theory, rather than its mathematical curiosity, so that little space is devoted to laborious proofs of results. This aspect of the book should appeal to computer scientists.

I see this book being of use in an undergraduate first course on program verification. The section on Lambda-calculus covers topics like undecidability and the halting problem, which are typically also covered in an undergraduate course on computation - providing a bridge to other theory courses. In contrast, the implementation of the theories will keep the student firmly aware of the utility and practical application of the field.
NOTES FOR CONTRIBUTORS

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems. However, non-refereed review and exploratory articles of interest to the journal’s readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in tripli
cate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines.

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author’s initials and surname;
  - author’s affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin. Distinguish clearly between such cases as:
  - upper and lower case letters;
  - the letter O and zero;
  - the letter I and the number one; and
  - the letter K and kappa.
- References should be listed at the end of the text in alphabetic order of the (first) author’s surname, and should be cited in the text in square brackets. References should thus take the following form:

Manuscripts accepted for publication should comply with the above guidelines, and may provided in one of the following three formats:
- in a typed form (i.e. suitable for scanning);
- as an ASCII file on diskette; or
- in camera-ready format.
  A page specification is available on request from the editor, for authors wishing to provide camera-ready copies.

Charges
A charge per final page, scaled to reflect scanning, typesetting and reproduction costs, will be levied on papers accepted for publication. The costs per final page are as follows:
- Typed format: R80-00
- ASCII format: R60-00
- Camera-ready format: R20-00
  These charges may be waived upon request of the author and at the discretion of the editor.

Proofs
Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should signed, and should be limited to about 500 words.

Advertisement
Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.
EDITORIAL.......................................................................................................................................................... 1

GUEST CONTRIBUTION
Opening Address: Vth SA Computer Symposium ............................................................... 3
  Prof H C Viljoen

RESEARCH ARTICLES

Homological Transfer: an Information Systems Research Method ........................................... 6
  J Mende

On the Generation of Permutations ........................................................................................... 12
  D Naccache de Paz

Coping with Degeneracy in the Computation of Dirichlet Tessellations ............................... 17
  J Buys, H J Messerschmidt and J F Botha

An Estelle Compiler for a Protocol Development Environment .............................................. 23
  P S Kritzinger and J van Dijk

Predicting the Performance of Shared Multiprocessor Caches .............................................. 31
  H A Goosen and D R Cheriton

Implementing UNIX on the INMOS Transputer ................................................................... 39
  P J McCullugh and G de V Smit

Data Structuring via Functions .................................................................................................. 45
  B H Venter

COMMUNICATIONS AND REPORTS

FRD Investment in Advanced Computer Science Training ...................................................... 51
ADA Courses and Workshop ................................................................................................... 51
Book Reviews ......................................................................................................................... 52
A Review of the Use of Computers in Education in South Africa
  Part I: Primary and High Schools ......................................................................................... 53