South African
Computer
Journal
Number 9
April 1993

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 9
April 1993

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarvereniging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Shochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Production Editor
Professor Riël Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere:	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Guest Editorial

WOFACS '92: Interdisciplinarity and Collaboration

Chris Brink
Laboratory for Formal Aspects and Complexity in Computer Science,
Department of Mathematics, University of Cape Town
cbrink@maths.uct.ac.za

This edition of SACJ is devoted to the Proceedings of WOFACS '92: the Workshop on Formal Aspects of Computer Science. The event was hosted at the University of Cape Town by FACCS-Lab, the Laboratory for Formal Aspects and Complexity in Computer Science, in July 1992.

The goal of WOFACS '92 was to bring together in a structured learning environment those Southern African computer scientists and mathematicians (academics and graduate students) interested in theoretical computer science. For this event FACCS-Lab imported four researchers eminent in their field, each to give a course of 10 lectures over a two-week period. Topics were carefully chosen so as to appeal to both mathematicians and computer scientists, and to reflect current work in the area of Formal Aspects of Computer Science. Each course was offered at beginning MSc level, and each could be taken by graduate students for credit at their respective home institutions. The guest lecturers and their topics were:

- Prof Roger Maddux (Iowa State University), *Predicate Transformer Semantics and Boolean Algebras with Operators*;
- Prof Austin Melton (Kansas State University), *Domains, Powerdomains and Power Structures*;
- Dr Hans Jürgen Ohlbach (Max Planck Institut für Informatik), *Deduction Systems in Artificial Intelligence*; and
- Prof Jeffery Zucker (McMaster University), *Theory of Computation.*

In addition, a number of Southern African researchers each gave an invited one-hour overview of the research done by themselves and their collaborators at their respective institutions. They were:

- Prof Ian Alderton (UNISA), *Cartesian Closedness*;
- Prof Guillaume Brümmer (University of Cape Town), *Completions in Topology and Elsewhere*;
- Prof Willem Fouché (University of Pretoria), *Identifying randomness given by high descriptive complexity*;
- Prof Johannes Heidema (UNISA), *Some Logics of Semantic Information*;
- Prof Stef Postma (University of the Western Cape), *Octolisp: a set of solutions; a source of problems*;
- Prof Yuri Velinov (University of Zimbabwe), *Categories, Nets and Parallel Computation*;
- Prof Basie von Solms (Randse Afrikaanse Universiteit), *Formal Languages and Automata as the Basis of a Number of Research Projects at the Rand Afrikaans University*; and
- Prof Peter Wood (University of Cape Town), *Detecting Bounded Recursion in Datalog Programs.*

WOFACS '92 was attended by about 75 participants from across the country, roughly half of whom were academics, the remainder being graduate students. Grants were made available to some graduate students who could not obtain funding from their home institutions. Participants who required accommodation were housed in a University Residence, and there was sufficient opportunity to enjoy the beauty of the Cape. Apart from travel and accommodation costs WOFACS '92 was a free service to the community: no fees of any kind were levied.

Simply put, WOFACS '92 was a *developmental* endeavour. The organisers considered Formal Aspects of Computer Science to be an increasingly important field of study, the pursuit of which should be encouraged on a broad front in the Southern African environment. To create a sense of community it was important to bring all interested parties together. On the other hand, we felt that it would be premature to issue a Call for Papers and organise a conference. Thus arose the idea of a Workshop, where people come to learn, and to share information on research projects.

It happens that the WOFACS endeavour fits in with two points of view recently put forward in the editorial pages of the *SA Computer Journal*. In a guest editorial in SACJ 6 (March 1992) Ed Coffman, the FRD-sponsored guest at the 1991 SA Computer Research Conference, gives his impressions of Computer Science research in South Africa. His overriding impression is that South Africa is weak in the area of Formal Aspects (or, as Coffman says, computer and computation theory). Coffman strongly recommends a concerted development in this area. He allows that Mathematics Departments could play an important role in this development – provided they make a major commitment and do not regard the area as a mere service discipline.

Coffman mentions further the practical advantages in a financially constrained environment of inter-departmental and inter-institutional cooperation. At a more fundamental level this point has also been addressed by SACJ Editor

Derrick Kourie. In his Editor's Notes to SACJ 8 (November 1992) Kourie discusses the dual forces of competition and collaboration at play in the Southern African research environment, concludes that there is too much of the former, and makes a strong plea for the latter. In Kourie's view, it is in the nature of scientific research that it will flourish where there is a healthy spirit of collaboration. Moreover, Kourie contends, the benefits of such collaboration should not remain confined to single Departments, but must be extended to an inter-departmental and inter-institutional level.

The foregoing ideas fit well with the basic FACCS-Lab philosophy of interdisciplinarity. We believe that computer scientists and mathematicians can and should collaborate, and that if they do it will be to their mutual benefit. To quote from the FACCS-Lab 1992 Report:

> FACCS-Lab aims to bring Formal Methods and Complexity Theory to bear on problems of Computer Science, in a structured interdisciplinary research programme intended to contribute to the development of research manpower in South Africa. In a developing country such as South Africa it is essential that the more applied sciences (such as Computer Science) should retain a good research base. It is also important that the theoretical sciences (such as Mathematics) should retain links with real-world developmental issues. FACCS-Lab aims to provide a bridge between Mathematics and Computer Science in South Africa, to further its overall developmental aim.

Activities such as WOFACS '92 are intended to contribute to this overall developmental aim. No doubt we can still improve our efforts, and feedback would be welcome.

It is the pleasure and privilege of any organiser to express thanks to those who contribute to the success of an event. I would like to do so, conveying my sincere thanks and that of FACCS-Lab to:

- Roger Maddux, Austin Melton, Hans Jürgen Ohlbach and Jeff Zucker, for coming, for presenting a course, evaluating students, and contributing to this SACJ edition;
- The local speakers, for helping to clarify the picture of what is being done in Southern Africa;
- Janet Goslett, the WOFACS secretary, and Maureen le Sar, my personal secretary, for keeping the show on the road;
- Laurette Pretorius, Hardy Hulley and Janet Goslett, for being actively involved in writing material for this edition;
- The Foundation for Research Development, and the University of Cape Town Research Committee, for funding WOFACS '92; and
- Cliff Moran, Dean of Science at UCT, for finding the money to publish this edition of SACJ.

Editor's Notes

This issue of SACJ is an archive of material presented at a workshop on formal aspects of computer science. The workshop – known as WOFACS '92 – was organised by the Laboratory for Formal Aspects and Complexity in Computer Science (FACCS-Lab) in the Department of Mathematics at the University of Cape Town. As one of the Research Leaders of FACCS-Lab, Prof Chris Brink was a prime mover in getting the event off the ground. As guest editor of this SACJ edition, he has collated the material and, most importantly, organised the funding for this issue.

Consequently, SACJ subscribers are able to reap the benefit of having access to WOFACS material without affecting the production of other editions of the journal. (The next edition is scheduled to appear in the near future.) On behalf of readers, I would like to thank Prof Brink for his initiatives, as well as the four contributors for making their work available. SACJ's production editor, Riël Smit – who has handled the final typesetting in his usual efficient, competent and uncomplaining fashion – also deserves our sincere thanks.

I hope and trust that this archival material will be of lasting value to those who teach and research in the area of formal aspects of computing.

Derrick Kourie
Editor

Production notes

I hope readers are not too perturbed by the fact that half of the articles in this issue is set in single column rather than the usual two column style. This was necessitated by the many wide formulae used in these papers.

Production Editor

Denotational Semantics and Domain Theory

Janet Goslett* Hardy Hulley* Austin Melton†
*Department of Mathematics, University of Cape Town, Rondebosch, 7700
†Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931, USA

Abstract

This tutorial is an introduction to denotational semantics and domain theory. For this reason the presentation is not entirely rigorous, with some proofs being omitted and certain simplifying assumptions being made. Instead, the emphasis here has been on satisfying the computational intuition of the reader — dwelling too long on the mathematical aspects of semantic domains is not always satisfying to somebody attempting to ascertain why these structures are relevant to computer science in the first place. Thus, we have started right at the beginning by discussing the concepts of syntax and semantics, and we have attempted to build the mathematical theory of domains upon this computational foundation in a fluid, intuitive and informal manner. In Section 1 and Section 2 we begin with brief descriptions of the syntax and semantics of computer languages, the roles these notions play in computer science, and how they interact in the quest for a formal methodology for computer programming. Section 3 provides an introduction to denotational semantics in particular — this being illustrated by a very simple example. In Section 4 we get down to the business of defining a language which can be recognised as something approximating a real computer language. Here we also endow this language (which can be found in [8]) with a denotational description. We have attempted to explain why the semantics presented for our toy language should seem natural to the reader. In Sections 5–7 we begin to explore the mathematical complexities of domain theory. Our motivation for introducing all this new theory is the search for a denotational description of the 'while'-loop in our example programming language. We conclude in Section 8 with a discussion of parallelism, non-determinism, and the new demands which these complications make of the theory developed up to then. For more advanced presentations of denotational semantics and domain theory, we recommend [3, 4, 6, 7, 10].

Keywords: *Semantics, Meanings of Programs, Domains*
Computing Review Categories: *(1991) D.3.1, F.3.2, F.3.3*

Received March 1993

1 Syntax

Before we can get to the question of program semantics, we first have to deal with the issue of syntax. This is because the syntax or grammar of a programming language determines which sequences of symbols actually constitute valid programs, and we have to ascertain which these are before we can provide semantic descriptions of them. The standard notation for syntactic descriptions of programming languages is Backus-Naur Form (BNF) (see [1] for an in-depth treatment of this).

A BNF definition of a programming language consists of a set of equations. On the left-hand side of each equation is a name or character called a *non-terminal*. The right-hand side describes the syntactic forms which belong to the structural type of the non-terminal in question. These forms are constructed from primitive symbols called *terminal symbols* and also from other non-terminals.

In order to illustrate this discussion we will now provide a syntactic description of a little language. In this grammar we have only one non-terminal type, viz *NUMERAL*, and the first line of the definition states that we are letting N and M be non-terminals of type *NUMERAL*. The second and third lines simply describe what the element N could possibly look like (we read the vertical bar as logical disjunction). Thus N could either be the terminal character 0 or the terminal character 1 or else it could be a 1 followed by a numeral.

Example 1.1

$$N, M \in \mathit{NUMERAL}$$
$$N ::= 0 \mid 1 \mid 1M$$
$$M ::= 0 \mid 1 \mid 0M \mid 1M$$

□

It may seem very obvious that the above little language should be interpreted as a binary representation of the set $\omega = \{\mathit{zero}, \mathit{one}, \mathit{two}, \ldots\}$, of natural numbers. However, care must be taken not to confuse the abstract syntax with the set of natural numbers itself — the natural numbers are well-defined mathematical objects, whereas the syntax of *NUMERAL* is just a definition of notation (for one thing, there are numerous other notational schemes which can also be interpreted as representing the natural numbers). The set of natural numbers is a semantic concept — it can be seen as providing meaning to the abstract notation, i.e., syntax, in the example (so far, the syntax hasn't yet been furnished with any meaning). The syntax of a language only stipulates which symbolic constructions are valid expressions in that language; the syntactic expressions themselves have no built-in meaning. Appreciating the distinction between notation and meaning (or syntax and semantics) is important.

We end this section with the following comment: although it cannot be observed in the above example, BNF grammars do often contain a certain degree of ambiguity (see example 1.2 below). This is because, for the method to be useful, it has to permit a certain degree of 'looseness' in definition — defining a grammar can become very tedious if one is too fastidious about avoiding all possible ambiguity. This isn't a problem, however, since for the purposes of doing semantics, we always assume the existence of a parser which maps each syntactic expression to a unique derivation tree. Thus our 'loose' notation is both adequate and convenient.

An example of an ambiguous grammar is given below.

Example 1.2

$N, M, P \in NUMERAL$

$N ::= 0 \mid 1 \mid 1M$

$M ::= 0 \mid 1 \mid 0M \mid 1M$

$P ::= P_1 + P_2 \mid P_1 \times P_2 \mid M$

□

Given the sentence $1 + 1 \times 0$, the parser could parse it as $(1 + 1) \times 0$ or as $1 + (1 \times 0)$ since the grammar does not indicate which operation should have higher precedence. Which derivation the parser chooses will affect the outcome of the arithmetic.

2 Semantics in General

Having discussed the nature of programming language syntax, we need to develop a method for assigning meaning to a valid expression. A well-developed and successful methodology for determining the meanings of programs would be of great benefit to computer scientists, and the study of formal semantics of computer languages has been motivated by this.

- Initially the idea was that a sufficiently precise method for describing computer languages had to be found so that implementors could construct correct compilers.
- Currently a main objective is to enable programmers to make rigorous statements about the behaviour of their programs (software verification).
- Also, programming semantics can guide language designers towards the design of better computer languages — ones whose formal descriptions are simpler, thus easing the task of reasoning about the correctness of programs.

Mosses [4] offers the following definition of the meaning of a program:

> The semantics of a program is dependent only on the objective behaviour that the program causes (directly) when executed by computers.

Thus a program's meaning is the behaviour it induces in the machine executing it.

There are three major techniques or approaches used to describe the semantics of computer languages and programs. These are the *operational*, *axiomatic* and *denotational* techniques.

- Stoy [10] says the following about the operational approach to formal semantics:

  > We define an 'abstract machine', which has a state, (possibly with several components), and some set of primitive instructions. We define the machine by specifying how the components of the state are changed by each of the instructions. Then we define the semantics of our particular programming language in terms of that. The idea is that although the abstract machine is probably completely unrealistic from a practical point of view, it is so simple that no misunderstanding can possibly arise with respect to its order code. The semantic description of the programming language specifies a translation into this code.

- An axiomatic (Dijkstra-style) description of a language is provided by associating an axiom with each kind of statement in the language. Such an axiom stipulates which conditions had to hold before the execution of its associated statement whenever we assume that certain conditions hold after execution.
- The style of semantics we shall be investigating is denotational semantics (also known as mathematical semantics). Its inspiration lies in the observation that mathematics provides powerful tools for expressing or modelling the effects of syntax. In particular it was realised that the mathematical concept of a function provides a very natural and convenient way of modelling the behaviour of a program. As Mosses [4] remarks:

  > The implementation-independent semantics of a program may typically be modelled mathematically as a function (or relation) between inputs and outputs.

  In their very important contribution to this field, Scott and Strachey [8] express the above ideas as follows:

  > One essential achievement of the method we shall wish to claim is that by insisting on a suitable level of abstraction and by emphasising the right details we are going to hit squarely on what can be called the mathematical meaning of a language.

  The denotational description of a program is given by mapping it onto an element (called its *denotation*) in a set called a *semantic domain*. The denotation of a program is defined in terms of the denotations of its syntactic substructures. At the very lowest level, the denotations of literals are defined as given elements of *primitive* semantic domains, whilst the domains which contain the denotations of more complex syntactic constructs are formed from the primitive domains by a number of *domain constructors*.

3 Denotational Semantics

Mosses [4] provides the following insight into the nature of denotational semantics:

> The characteristic feature of denotational semantics is that one gives semantic objects for all phrases — not only for complete programs. The semantic object specified for a phrase is called the denotation of that phrase.

Thus the meaning of a syntactic construct is some mathematical object which is definable in terms of the denotations of its syntactic subphrases. This technique of building the meanings of complex expressions from the meanings of their subexpressions is called *compositionality*.

At the very lowest level we denote the terminal characters in the grammar of some computer language by fixed mathematical entities, which are elements of sets called *primitive* semantic domains (from now on, when speaking of semantic domains, we shall drop the 'semantic' and speak only of domains). In order to work with more complex structures we require certain operations (known as *domain constructors*) which act upon domains and produce *compound* domains from primitive ones. Also required are operations (referred to as *domain operations*) which map the elements of one domain to another (possibly the same one). Thus, at a high level, complex syntactic constructs are denotable by elements of compound domains. The domain element or denotation corresponding to a particular complex syntactic construct can be obtained in two ways. The first is by direct association of the complex syntactic construct with an element in a compound domain, and the second is by associating simple syntactic substructures with domain elements and then applying domain operations to the simple denotations to obtain the desired element of the compound domain. The domain operations which are applied are in fact the denotations of the syntactic constructors which build up the complex syntactic construct from its simple substructures.

The functions which map syntactic constructs onto their denotations are called the *valuation* functions. There is one valuation function for every nonterminal in the BNF description of a computer language.

Putting all of the above together, the denotational description of a computer language consists of the following components:

1. a description of the syntax of the language,
2. definitions of all the primitive domains (which are simply sets for now, but later we shall see that it is necessary to add some structure to them),
3. definitions of the needed domain constructors,
4. a list of domain equations which describe how the compound domains are formed from the primitive domains via the domain constructors,
5. a description of the required domain operations, and
6. the valuation functions (one for each nonterminal in the grammar) which describe how the syntax is mapped to domain elements in terms of the domain operations.

To illustrate all of this, here follows a denotational description of the language defined in Example 1.1.

Example 3.1

1. *The syntax of numerals is defined in Example 1.1.*
2. *The only primitive domain is the set ω of natural numbers.*
3. *The only needed domain constructor is Cartesian product, $\otimes$. If A and B are sets, then*
   $A \otimes B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
4. *The only domain equation is*
   $D = \omega \otimes \omega$.
5. *The required domain operations are addition and multiplication of natural numbers:*
   $$\_ + \_ \; : \; D \longrightarrow \omega,$$
   $$\_ \times \_ \; : \; D \longrightarrow \omega.$$
6. *The only valuation function is*
   $\mathcal{N} : NUMERAL \longrightarrow \omega$
   defined by:
   $$\begin{aligned} \mathcal{N}[\![0]\!] &= zero \\ \mathcal{N}[\![1]\!] &= one \\ \mathcal{N}[\![N0]\!] &= two \times \mathcal{N}[\![N]\!] \\ \mathcal{N}[\![N1]\!] &= two \times \mathcal{N}[\![N]\!] + one. \end{aligned}$$

□

This formalises our earlier interpretation of the elements of *NUMERAL*. Note that, in the above, the fourth step (i.e. giving a name to the domain $\omega \otimes \omega$) isn't really necessary, since we could simply have achieved everything that followed by referring to the domain (set) $\omega \otimes \omega$ instead of defining the new domain D — we did this, however, because the example is illustrative. Domain equations are only really necessary in a denotational scheme when certain domains are defined as solutions to recursive domain equations. Another thing worth mentioning is that domain constructors are not created within the scope of a denotational definition — there are a number of commonly used constructors (such as Cartesian product, for example), and these are a well-established part of the theory of domains. Thus, a denotational scheme for some language simply makes use of domain constructors which have already been defined (see Section 6 for a treatment of the most commonly used constructors).

We use the brackets $[\![$ and $]\!]$ to insulate the object language (syntax) from the meta-language. (The meta-language includes those symbols which are neither syntactic nor semantic symbols and it is the language of the denotational semantics method or technique itself.) As mentioned before, it is important that we don't confuse syntactic notation with semantic notation.

4 A Simple Language

We are now ready to apply the ideas developed in Section 3 to a grammar which bears some resemblance to the syntax of most imperative programming languages. This very simple example comes from [8] and is also dealt with in detail by Stoy in Chapter 9 of [10]. The language consists only of two classes of syntactic object, namely *expressions* and *commands*:

$E \in EXPRESSION$

$C \in COMMAND$

$E ::= (E) \mid \pi \mid tt \mid ff \mid if\ E_0\ then\ E_1\ else\ E_2$

$C ::= (C) \mid \phi \mid dummy \mid if\ E\ then\ C_1\ else\ C_2$

$| C_1; C_2$.

Intuitively speaking, expressions are objects which adopt domain values upon evaluation by a computer, whereas commands are (elementary) programs. In order to achieve an understanding of the semantics of this language, we appeal to the concept of a machine state — we think of a computer as containing a number of internal variables, and their values at any particular time determine the state of the machine at that time. Thus the following primitive domains are needed:

$Bool = \{true, false\}$ and

$State =$ the set of machine states.

In order to determine which compound domains are needed, we need to be more specific about how meaning is to be attached to the syntax of expressions and commands. For this reason we first have to achieve a basic intuitive understanding of how a machine executing programs in our language will behave. When a computer executes a program it starts off from some initial state and then proceeds through a sequence of states, as it follows the instructions laid down in the program, until it finally terminates execution. A program may fail to terminate if, e.g., the instructions with respect to at least one input contain an infinite loop. In this case the computer will continue to proceed through a sequence of states but will not reach a final output state. Each state change in these execution sequences is the result of the execution of one of the commands in the program. Thus it is most natural to interpret commands as state transformations, ie. as functions from *State* to *State* — the state of the machine after execution of a command is a function of its state before execution, and this function is the meaning of the command in question.

Achieving a similar understanding of the nature of expressions in terms of machine states is slightly more difficult. A computer evaluates a logical expression in our simple language as being either *true* or *false*, but which it is depends on the state of the machine when it performs this evaluation process. Thus we could quite reasonably expect to capture the meaning of expressions by letting them be denoted by functions from *State* to *Bool*. This is however slightly unrealistic in that the very task of determining the logical value of an expression may itself alter the internal state of the machine (this is known as a 'side-effect' of the expression evaluation). Consequently it is more appropriate to think of the meaning of an expression as a function from *State* to $Bool \otimes$ `State` — it maps any machine state to a pair consisting of the truth value of the expression, given this state, and the state of the machine after the evaluation of the expression (which may be different from that before as a side-effect of the evaluation procedure).

Therefore the following compound domains are required:

$State \longrightarrow State$ and

$State \longrightarrow Bool \otimes State$.

The first of the above domains is the function-space between *State* and *State* (the set of all functions from *State* to *State*) — all the denotations of programs (and commands) in the language are elements of this domain. The second domain contains all the denotations of expressions.

Before we go ahead and formally define the domain operations and valuation functions needed to describe the language's semantics, an informal description of the meanings of the syntactic constructs will be helpful. The expression π is some arbitrary, but fixed one, e.g. $\pi = (x = true)$, (the same for the command ϕ, e.g. $\phi = (x := 1)$). The command *dummy* is the program which returns its input unchanged and the command $C_1; C_2$ is that program which, when executed, first executes C_1 and then C_2. The rest of the syntax has a natural intuitive interpretation, and the formal description of its semantics should seem natural. The domain operations appear in figure 1 (where $(b, s) \longmapsto s$ means that the element (b, s) is mapped to the element s).

To complete the description of this language, here follow the two valuation functions. We first treat the function which maps expressions to their denotations:

$$\mathcal{E} : EXPRESSION \longrightarrow (State \longrightarrow Bool \otimes State)$$

$$\mathcal{E}[\![(E)]\!] = \mathcal{E}[\![E]\!]$$
$$\mathcal{E}[\![\pi]\!] =$$
some function of type $State \longrightarrow Bool \otimes State$
$$\mathcal{E}[\![tt]\!](s) = (true, s)$$
$$\mathcal{E}[\![ff]\!](s) = (false, s)$$
$$\mathcal{E}[\![if\ E_0\ then\ E_1\ else\ E_2]\!] = (\mathcal{E}[\![E_0]\!] \Rightarrow \mathcal{E}[\![E_1]\!] \bullet_E \mathcal{E}[\![E_2]\!])$$

Now we define the valuation function for commands (here I_{State} is the identity function on the *State*):

$$\mathcal{C} : COMMAND \longrightarrow (State \longrightarrow State)$$

$$\mathcal{C}[\![(C)]\!] = \mathcal{C}[\![C]\!]$$
$$\mathcal{C}[\![\phi]\!] = \text{some function of type } (State \longrightarrow State)$$
$$\mathcal{C}[\![dummy]\!] = I_{State}$$
$$\mathcal{C}[\![if\ E\ then\ C_1\ else\ C_2]\!] = (\mathcal{E}[\![E]\!] \Rightarrow \mathcal{C}[\![C_1]\!] \bullet_C \mathcal{C}[\![C_2]\!])$$
$$\mathcal{C}[\![C_1; C_2]\!] = \mathcal{C}[\![C_2]\!] \circ \mathcal{C}[\![C_1]\!]$$

From the definition of the valuation functions we firstly see that parentheses do not change the semantics of expressions or commands, in the sense that the denotation of an expression, (E), is the same as the denotation of the expression, E, and the denotation of a command, (C), is the same as the denotation of the command, C. Also, the denotations of the atomic expressions *tt* and *ff* indicate that when they are evaluated by a machine there is no state change (evaluating *tt* and *ff* has no side-effects).

The meaning of $if\ E_0\ then\ E_1\ else\ E_2$ is a function which first computes the truth value of E_0 with the machine in some given state, thus leaving it in a second (maybe the same) state. Then, if this truth value is *true*, the truth value of of E_1 is calculated with the computer in the second state (so leaving it in a third state). The function value of the expression is then the pair consisting of this last truth value and the third machine state. If the truth value of E_0 is *false*, then the same procedure is repeated for the expression E_2 instead of E_1. (See figure 1.)

The command *dummy* simply leaves the machine in

$$P_1 : Bool \otimes State \longrightarrow Bool$$
$$(b, s) \longmapsto b,$$
$$P_2 : Bool \otimes State \longrightarrow State$$
$$(b, s) \longmapsto s,$$
$$\_ \Rightarrow \_ \bullet_E \_ : (State \longrightarrow Bool \otimes State) \otimes (State \longrightarrow Bool \otimes State) \otimes (State \longrightarrow Bool \otimes State) \longrightarrow (State \longrightarrow Bool \otimes State)$$
$$(f_0 \Rightarrow f_1 \bullet_E f_2)(s) = \begin{cases} f_1(P_2(f_0(s))) & \text{if } P_1(f_0(s)) = true \\ f_2(P_2(f_0(s))) & \text{if } P_1(f_0(s)) = false \end{cases}$$

(The above function may look a little intimidating, but it is natural when one realises that it is used to give meaning to *expressions* of the form *if* E_0 *then* E_1 *else* E_2.)

$$I_{State} : State \longrightarrow State$$
$$s \longmapsto s,$$
$$\_ \Rightarrow \_ \bullet_C \_ : (State \longrightarrow Bool \otimes State) \otimes (State \longrightarrow State) \otimes (State \longrightarrow State) \longrightarrow (State \longrightarrow State)$$
$$(f_0 \Rightarrow f_1 \bullet_C f_2)(s) = \begin{cases} f_1(P_2(f_0(s))) & \text{if } P_1(f_0(s)) = true \\ f_2(P_2(f_0(s))) & \text{if } P_1(f_0(s)) = false \end{cases}$$

(Again, the above function is used to give meaning to *commands* of the form *if* E *then* C_1 *else* C_2.)

$$\_ \circ \_ : (State \longrightarrow State) \otimes (State \longrightarrow State) \longrightarrow (State \longrightarrow State)$$
$$(f_1, f_2) \longmapsto f_1 \circ f_2$$

(This is just composition of functions.)

Figure 1. The Domain Operations

the same state after execution as it was in before. The denotation of the command *if* E *then* C_1 *else* C_2 is a function which maps one machine state to another by evaluating the expression E with the computer in the input state, and so producing a pair consisting of a truth value and an intermediate state. If the truth value is *true*, then the output state of the command is the result of applying the function $\mathcal{C}[\![C_1]\!]$ to the intermediate state, otherwise it is gotten by applying $\mathcal{C}[\![C_2]\!]$ to the intermediate state. Lastly, we observe how natural the language of mathematics is for modelling the meaning of programs — the semantics of composition of programs is just the composition of the functions which denote them (in the inverse order).

Before proceeding to the next section, we note that the two domains we used for denoting expressions and commands were defined in terms of the function space and Cartesian product constructors. These two constructors are of fundamental importance in denotational descriptions of languages. A category of domains which is closed under function space and Cartesian product (ie. which supports these two constructors) is called *Cartesian-closed*. Much work has been done in domain theory in the search for suitable Cartesian-closed categories of domains.

5 Recursion, Iteration and Cpo's

Unfortunately the computer language introduced in Section 4 is impractical because of its very limited expressive power—its commands are too direct by nature. We could go some way towards remedying this situation by including the following well-understood construct in the syntax of commands:

C ::= *while* E *do* C.

Of course, we are now left with the task of defining the semantics of this new item of grammar, which means extending the definition of the valuation function

$\mathcal{C} : COMMANDS \longrightarrow (State \longrightarrow State)$.

It should seem quite reasonable to anybody familiar with computer languages that we should want *while E do C* to adopt the same meaning as *if E then (C; while E do C) else dummy*. In other words the function

$\mathcal{C}[\![while\ E\ do\ C]\!] : State \longrightarrow State$

should satisfy the equation

$$\mathcal{C}[\![while\ E\ do\ C]\!] = \mathcal{E}[\![E]\!] \Rightarrow \mathcal{C}[\![while\ E\ do\ C]\!] \circ \mathcal{C}[\![C]\!] \bullet_C I_{State}. \quad (1)$$

Thus we want the denotation of *while E do C* to be a fixed point (a fixed point of a function, $f : X \longrightarrow X$, is an element, $x \in X$, such that $f(x) = x$) of the function

$$\mathcal{E}[\![E]\!] \Rightarrow \_ \circ \mathcal{C}[\![C]\!] \bullet_C I_{State} : (State \longrightarrow State) \longrightarrow (State \longrightarrow State). \quad (2)$$

Up to now the function space $State \longrightarrow State$ has simply been the set of all (total) functions mapping machine states to machine states. Unfortunately this turns out to be unsatisfactory when considering the semantics of *while E do C*, since if we let E be *true* and C be *dummy*, then any $f \in State \longrightarrow State$ will satisfy

$$\mathcal{E}[\![E]\!] \Rightarrow f \circ \mathcal{C}[\![C]\!] \bullet_C I_{State} = f.$$

Clearly we have no way of deciding which of these functions should denote the command in question. This question will be rephrased as Equation 1 – using a function which will be called F. In an effort to answer this question we will introduce an order on domains; we will show how collections of special functions form domains and we will show how functions like F which are defined from a domain to itself can be 'solved' in terms of fixed points.

Note 5.1 *Following a convention used in [3] we write*

$f \in State \longrightarrow State$

to indicate an element of the function space, and

$$f : State \longrightarrow State$$

to mean a function from State to State. In general not every function from State to State will be an element of the function space. Up to now this distinction hasn't been important, but we are soon going to place restrictions upon those functions from one domain into another which are elements of the function space. Given domains D and E, we will also write

$$D \longrightarrow E$$

to mean the function space domain and not in general the set of functions from D to E. □

Now that we have encountered the problem with determining the meaning of *while E do C*, let us backtrack a bit. We can rewrite Equation (1) as

$$f = F(f) \tag{3}$$

where

$$F = \mathcal{E}[\![E]\!] \Rightarrow \_ \circ \mathcal{C}[\![C]\!] \bullet_C I_{State}.$$

We want to solve for $f \in State \longrightarrow State$. Our problem in attempting to solve this equation is that we haven't analysed the structure of domains at all. By electing for generality and allowing $State \longrightarrow State$ to consist of all functions from *State* to *State*, we have created a situation where this equation doesn't have an acceptable solution. To find a satisfactory solution to Equation (3) we have to abandon the comfortable world of sets and total functions between them. As Scott and Strachey [8] note:

> The solution to this problem is easy enough and well known: we modify our idea about the function space. We no longer demand that functions be total but understand the elements of a function space to be partial functions.

Scott realised that this elegant route introduced a degree of undefinedness into the definition of domains. Formally this undefinedness is denoted by a special which is called bottom, is written $\perp$ and is added to every domain. Thus we extend *State* to $State_\perp = State \cup \{\perp\}$ and *Bool* to $Bool_\perp = Bool \cup \{\perp\}$. (We should in fact decorate the two bottom elements with different subscripts since they are not the same object, but our looser notation shouldn't create any confusion).

Naturally we now have to decide what the function space $State_\perp \longrightarrow State_\perp$ should look like (unfortunately if we let it consist of arbitrary functions from $State_\perp$ to $State_\perp$ we will inherit all the problems we had earlier), and so we once again appeal to our computational intuition. We argue (remembering that the elements of $State_\perp \longrightarrow State_\perp$ are intended to be the denotations of programs) that if a program is capable of delivering well-defined output from undefined input (ie. if it maps $\perp$ to some $s \in State$) then, such a program could terminate execution with final state s for any initial state. This is because if the machine's state is $\perp$ we interpret this as meaning that we don't know in what state the machine actually is. So what we are saying is that if a program can deliver a certain well-defined output even if we don't know what its input was, then this is the same as saying that it delivers this output for all input. Thus, if $f \in State_\perp \longrightarrow State_\perp$, then

$$(f(\perp) \neq \perp) \Rightarrow (\forall s \in State)[f(s) = f(\perp)]. \tag{4}$$

Instead of concentrating directly on the function space construction, we shall first add some structure to the primitive domain $State_\perp$. Then we will show how we can make the function space $State_\perp \longrightarrow State_\perp$ absorb enough of this structure so that all its elements satisfy Condition (4).

Definition 5.2 *Let D be a set. A partial order on D is a relation*

$$\sqsubseteq \subseteq D \otimes D$$

satisfying the following three conditions for all d_1, d_2 and d_3 in D:

- *(reflexivity) $d_1 \sqsubseteq d_1$,*
- *(transitivity) if $(d_1 \sqsubseteq d_2)$ and $(d_2 \sqsubseteq d_3)$ then $(d_1 \sqsubseteq d_3)$, and*
- *(anti-symmetry) if $(d_1 \sqsubseteq d_2)$ and $(d_2 \sqsubseteq d_1)$ then $(d_1 = d_2)$.*

We indicate that D is a partially ordered set (poset) with order relation $\sqsubseteq$ by writing it as $(D, \sqsubseteq)$. When $d_1 \sqsubseteq d_2$ we say that d_1 lies below d_2, or that d_2 dominates d_1. □

Note that a poset is a directed graph such that

1. for each node v there is an arc of the form (v, v),
2. there are no cycles except those of the form (v, v) and
3. if (u, v) and (v, w) are arcs then (u, w) is also an arc.

Definition 5.3 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be posets. Then a function $f : D \longrightarrow E$ is said to be* monotone *if for all d_1 and d_2 in D:*

$$(d_1 \sqsubseteq_D d_2) \Rightarrow (f(d_1) \sqsubseteq_E f(d_2)).$$

□

We have now developed enough theoretical apparatus to begin refining our ideas of domains and the (computable) functions between them. Whereas domains were simply sets in the preceding sections, we will now furnish them with partial orderings. Of course, we first have to find some computational justification for doing so.

If we analyse the domain $State_\perp$, we see that it contains the set *State* of perfectly specified (completely defined) machine states together with the state, $\perp$. Although a computer can only ever be in some (well-defined) state $s \in State$, to an observer who isn't familiar with its internal workings, its state could seem to be undefined. In this case, as far as the observer is concerned, the machine could be in any state — in other words, the undefined state is consistent with any of the well-defined states. Of course, if the observer knows that the machine is in some well-defined state, then he also knows that it cannot be in another well-defined state, and so any well-defined state is inconsistent with all other well-defined states. This argument helps motivate the partial order $\sqsubseteq_S$ defined on $State_\perp$ as follows:

$$(\forall s_1, s_2 \in State_\perp)$$
$$[(s_1 \sqsubseteq_S s_2) \Leftrightarrow (s_1 = \perp) \vee (s_1 = s_2)].$$

The domain $Bool_\perp$ can be treated in a similar fashion by defining the partial order $\sqsubseteq_B$:

$$(\forall b_1, b_2 \in Bool_\perp)$$
$$[(b_1 \sqsubseteq_B b_2) \Leftrightarrow (b_1 = \perp) \vee (b_1 = b_2)].$$

Partially ordered sets, such as the above two, where there is a bottom element below everything else in the set and all

the other elements are incomparable are described as being *flat*.

It can be quite easily seen that all the monotone functions from $(State_\perp, \sqsubseteq_S)$ to $(State_\perp, \sqsubseteq_S)$ satisfy Condition (4). Thus the following seems agrees with our discussion of programs in terms of machine states:

Thesis 5.4 *A domain is a partially ordered set $(D, \sqsubseteq)$ which contains an element $\perp \in D$ such that $(\forall d \in D)[\perp \sqsubseteq d]$.* □

Thesis 5.5 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be domains. Then any element of the function space $(D, \sqsubseteq_D) \longrightarrow (E, \sqsubseteq_E)$ is a monotone function from $(D, \sqsubseteq_D)$ to $(E, \sqsubseteq_E)$.* □

We gave an intuitive account of the meaning of the partial order in the case of the domain $(State_\perp, \sqsubseteq_S)$. Domains in general provide a theory of data types, and the order relation is commonly interpreted as a *consistent information* ordering (it places the domain elements in a hierarchy of consistent information content). To understand this, we consider the elements of a domain to be descriptions (or approximations) of a certain class of data objects (the natural numbers for example). Some of the descriptions may be ambiguous (there could be more than one piece of data for which their information is true). Some of the descriptions may also be mutually inconsistent (their information contents are contradictory and so there is no data item which they both approximate). This mutual inconsistency of data approximations is indicated by the domain elements not having an upper bound (ie. an element which dominates all of them).

Among descriptions of the same piece of data, some approximations are better than others, and so we can order them in terms of increasing information content. At the top of such a sequence of improving (but consistent) approximations (if such a sequence has a top element) is a description which cannot be improved upon, ie. it completely and unambiguously specifies a single data item. Such an element of a domain, which is below nothing else in the order, is called a *perfect* element, and the information it contains is inconsistent with that contained by any other perfect description.

At the very bottom, in terms of the domain order, is the element $\perp$. As said before, this element is regarded as being undefined, in other words, it contains no information telling us which piece of data it is actually describing. Thus the bottom element of a domain is completely ambiguous, and we indicate this by placing it below everything else in the domain — the fact that $\perp$ has no information means not only that it has less information than any other data description in the domain, but also that its information content is consistent with that of every other element. As we mentioned before, the elements of a domain are descriptions of the objects belonging to a certain data type, and in this interpretation the bottom element is then the trivial description — it can approximate anything.

We seem to be travelling in the right direction now, because given any flat domain $(D, \sqsubseteq)$, it follows by inspection that $f(\perp) \in D$ is a fixed point for any monotone function, f, from $(D, \sqsubseteq)$ to $(D, \sqsubseteq)$. Also, even though such a function may have other fixed points, $f(\perp)$ is the least fixed point, making it special as a candidate for the solution of a fixed point equation.

We return now to Equation (3) which we wanted to solve in the first place. Following Thesis 5.5, let us assume that $State_\perp \longrightarrow State_\perp$ is the set of all monotone functions from $State_\perp$ to $State_\perp$. We also want, this function space itself to be a domain, which means that we must be able to convert it into a structure satisfying Thesis 5.4. We thus require the following:

- a partial order on $State_\perp \longrightarrow State_\perp$, and
- a monotone function which is less than every other element of $State_\perp \longrightarrow State_\perp$ in terms of the above order, i.e. a bottom element of $State_\perp \longrightarrow State_\perp$.

We let the function space absorb its order from the domain $(State_\perp, \sqsubseteq_S)$ as we said would happen earlier. For all $f_1, f_2 \in State_\perp \longrightarrow State_\perp$, the relation $\sqsubseteq_{func}$ is defined by:

$$(f_1 \sqsubseteq_{func} f_2) \Leftrightarrow (\forall s \in State_\perp)[f_1(s) \sqsubseteq_S f_2(s)].$$

Thus the elements of the function space are ordered pointwise. The monotone function $\perp \in State_\perp \longrightarrow State_\perp$, defined by

$$(\forall s \in State_\perp)[\perp(s) = \perp],$$

is then the bottom element in this order. (The last equation could be written more precisely as $\perp_{func}(s) = \perp_s$.)

Now that we have defined the domain

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func}) \quad (5)$$

we are in a position to start talking about the function space

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func}) \longrightarrow \quad (6)$$

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func}) \quad (7)$$

which, by Thesis 5.5, consists of all the monotone functions from domain (5) into itself. Before we can begin to think about solving Equation (3) we must first extend the definition of the function F defined there so that it too is an element of the above function space (7). (Until now, F has only been defined on total functions mapping the set *State* to itself). We won't go into these details here; we will simply assume that we have found a sensible way of redefining F so that it is an element of the above function space.

We now encounter our second problem: the domain

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func})$$

is not flat. Hence we can't apply the same argument as we did earlier in order to establish the existence of a meaningful fixed point for any monotone function from this domain into itself. To appreciate this problem, let us assume that the set *State* is infinite but countable. We can write

$$State = \{s_0, s_1, s_2, \ldots\}.$$

Suppose that for all $n \in \omega$ the function f_n is defined by:

$$\text{for all } s_i \in State\ , f_n(s_i) = \begin{cases} s_i & \text{if } i < n \\ \perp & \text{otherwise} \end{cases}$$

$$\text{and } f_n(\perp) = \perp,$$

then each of these functions is an element of

$$(State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S).$$

For argument's sake, define F such that $F(\perp) = f_1$, that $F(F(\perp)) = F(f_1) = f_2$, and so on. F is monotone, yet the simple fixed point we found for a monotone function

over a flat domain seems to have evaded us. For reasons we are about to see, we still have a fixed point.

Definition 5.6 *Let $(D, \sqsubseteq)$ be a partially ordered set. An* ω-increasing chain *in this poset is a linearly ordered sequence*

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots$$

of elements of D. To save space, we often denote an ω-increasing chain such as the one above by $(d_i)_{i\in\omega}$ □

Since we have assumed that the set *State* is infinite, we can see that the domain

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func})$$

contains infinite ω-increasing chains — the following is one of these:

$$\perp \sqsubseteq_{func} f_1 \sqsubseteq_{func} f_2 \sqsubseteq_{func} f_3 \sqsubseteq_{func} \cdots \quad (8)$$

where the functions f_n, are defined above. What's more, each of these ω-increasing chains has a least upper bound. If

$$g_0 \sqsubseteq_{func} g_1 \sqsubseteq_{func} g_2 \sqsubseteq_{func} \cdots \quad (9)$$

is such an ω-increasing chain, then its least upper bound, $\bigsqcup_{i\in\omega} g_i$, is defined by

$$(\forall s \in State_\perp)[(\bigsqcup_{i\in\omega} g_i)(s) = \bigsqcup_{i\in\omega} g_i(s)].$$

(The fact that, for all $s \in State_\perp$, the least upper bound, $\bigsqcup_{i\in\omega} g_i(s) \in State_\perp$ exists follows from the definition of the order relation on functions, the fact that the above functions form an ω-increasing chain in this order, and the flatness of $(State_\perp, \sqsubseteq_S)$). Thus the least upper bound of the chain (8) is the identity function from $State_\perp$ to $State_\perp$.

In addition the function F preserves the least upper bounds of the above-mentioned chains. Consider the chain (9) again — the already established monotonicity of F ensures that

$$F(g_j) \sqsubseteq_{func} F(\bigsqcup_{i\in\omega} g_i)$$

for all natural numbers, j. Also by monotonicity of F, we know that

$$F(g_0) \sqsubseteq_{func} F(g_1) \sqsubseteq_{func} F(g_2) \sqsubseteq_{func} \cdots$$

is an ω-increasing chain in the domain

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func})$$

and thus also has a least upper bound, $\bigsqcup_{i\in\omega} F(g_i)$. Hence

$$\bigsqcup_{i\in\omega} F(g_i) \sqsubseteq_{func} F(\bigsqcup_{i\in\omega} g_i). \quad (10)$$

by the definition of a least upper bound.

We will now prove the converse result. First we note that since $(State_\perp, \sqsubseteq_S)$ is a flat domain, for each $s \in S_\perp$ the least upper bound of $g_i(s)$ must be explicitly obtained, i.e.,there must be a natural number, j, such that

$$(\bigsqcup_{i\in\omega} g_i)(s) = g_j(s)$$

for any given $s \in State_\perp$. If this weren't the case, it would contradict $\bigsqcup g_i$ being the least upper bound of the chain (9). Thus for every $s \in State_\perp$ there is a natural number, j, so that

$$F(\bigsqcup_{i\in\omega} g_i)(s) \sqsubseteq_S F(g_j)(s).$$

By the definition of the order placed upon functions, we know that for every $s \in State_\perp$ and every natural number, j,

$$F(g_j)(s) \sqsubseteq_S (\bigsqcup_{i\in\omega} F(g_i))(s),$$

meaning that

$$(\forall s \in State_\perp)[F(\bigsqcup_{i\in\omega} g_i)(s) \sqsubseteq_S (\bigsqcup_{i\in\omega} F(g_i))(s).$$

Hence

$$F(\bigsqcup_{i\in\omega} g_i) \sqsubseteq_{func} \bigsqcup_{i\in\omega} F(g_i). \quad (11)$$

From (10) and (11) we can conclude that

$$F(\bigsqcup_{i\in\omega} g_i) = \bigsqcup_{i\in\omega} F(g_i),$$

which is the result we wanted.

All of these developments now place us in the position to discover the promised least fixed point of the function F. Firstly, we set

$$F_0(\perp) = \perp \text{ and}$$
$$F_n(\perp) = F(F_{n-1}(\perp)) \text{ for } n \geq 1.$$

By the definition of the bottom element,

$$F_0(\perp) \sqsubseteq_{func} F_1(\perp),$$

and so monotonicity of F implies that

$$F_1(\perp) \sqsubseteq_{func} F_2(\perp).$$

By continuing with this inductive application of the monotonicity of F, we can establish the existence of the chain

$$F_0(\perp) \sqsubseteq_{func} F_1(\perp) \sqsubseteq_{func} F_2(\perp) \sqsubseteq_{func} \cdots \quad (12)$$

of functions, which we are assured has a least upper bound, $\bigsqcup_{i\in\omega} F_i(\perp)$. Showing that this function is a fixed point of F is easy, because

$$\begin{aligned} F(\bigsqcup_{i\in\omega} F_i(\perp)) &= \bigsqcup_{i\in\omega} F(F_i(\perp)) \\ &= \bigsqcup_{i\in(\omega-\{0\})} F_i(\perp) \\ &= \bigsqcup_{i\in\omega} F_i(\perp) \text{ because } F_0 = \perp \end{aligned}$$

since F preserves the least upper bounds of chains.

That the above is also the least fixed point of F can also be established without much difficulty, for if $f : State_\perp \longrightarrow State_\perp$ is another fixed point, then we know that $F_0(\perp) \sqsubseteq_{func} f$ to start with. Now, for the inductive step, we assume that

$$F_n(\perp) \sqsubseteq f,$$

for some natural number number n. By monotonicity of F,

$$F_{n+1}(\perp) \sqsubseteq_{func} F(f) = f.$$

Thus, f is also an upper bound of the chain (12), and so it must lie above $\bigsqcup_{i\in\omega} F_i(\perp)$ in the order $\sqsubseteq_{func}$.

Definition 5.7 *A partially ordered set with bottom element in which every ω-increasing chain has a least upper bound is called a* complete *partially ordered set (cpo).* □

Definition 5.8 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be complete partially ordered sets. A function $f : D \longrightarrow E$ is called* continuous *if for every ω-increasing chain, $d_0 \sqsubseteq_D d_1 \sqsubseteq_D d_2 \sqsubseteq_D \ldots$,the function f preserves its least upper bound. Ie., $f(\bigsqcup_{i\in\omega} d_i) = \bigsqcup_{i\in\omega} f(d_i)$.* □

Clearly then,

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func})$$

is a cpo and F is a continuous function over it. The fact that we have been able to locate a fixed point for F with such ease, and that this fixed point presents itself as a natural candidate for the meaning of

while E do C

support our adapting Thesis 5.4 and Thesis 5.5, and our adopting the following:

Thesis 5.9 *A domain is a complete partially ordered set.* □

Thesis 5.10 *Let $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ be domains. Then any element of the function space $(D, \sqsubseteq_D) \longrightarrow (E, \sqsubseteq_E)$ is a continuous function from $(D, \sqsubseteq_D)$ to $(E, \sqsubseteq_E)$.* □

At this point we seem to have encountered a slight hitch: if we accept the above, then $(State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S)$ should also be a set of continuous functions, whereas the entire discussion above was based upon it being the monotone functions from $(State_\perp, \sqsubseteq_S)$ to $(State_\perp, \sqsubseteq_S)$. Fortunately there is no problem here because all monotone functions from one flat domain into another are continuous as well. We also haven't mentioned the fact that $(State_\perp, \sqsubseteq_S)$ should itself be a cpo. Once again though, all flat posets with bottom elements are complete.

The wisdom of accepting Thesis 5.9 and Thesis 5.10 is confirmed by the next very important result. It describes how one may construct least fixed points for arbitrary continuous functions over cpo's. This enables us to give meaning to all recursive and iterative commands in a computer language.

Theorem 5.11 (Least Fixed Point Theorem) *If $(D, \sqsubseteq)$ is a cpo and $f : D \longrightarrow D$ is continuous, then there exists a point, $fix_f \in D$, such that $fix_f = f(fix_f)$ and $fix_f \sqsubseteq d$ for any $d \in D$ such that $d = f(d)$. In other words, fix_f is the least fixed point of f.*

Proof. *We have already proved this result for the continuous function F.* □

According to our idea of interpreting the order of a domain as relating the consistent information content of its elements, we view an ω-increasing chain as a sequence of increasingly informative (yet consistent) information elements or packages. The effect which Thesis 5.9 has on our computational understanding of domains is that there has to be some domain element which contains precisely all the information contained by all elements of any chain.

Insisting that all chains have upper bounds also means that every element in a domain must lie below some perfect element in the domain order. Hence every partial description of a data object is in fact consistent with at least one complete description — something quite pleasing.

6 Domain Constructors

Having decided that domains should be complete partially ordered sets, we will now describe a number of the most common domain constructors. These constructors are used to build complex domains from 'simple' domains.

Function space

Given two cpo's, $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$, the function space,

$$((D, \sqsubseteq_D) \longrightarrow (E, \sqsubseteq_E), \sqsubseteq_{D \longrightarrow E}),$$

consists of all continuous functions from $(D, \sqsubseteq_D)$ to $(E, \sqsubseteq_E)$, with the relation $\sqsubseteq_{D \longrightarrow E}$ defined point-wise by

$$(f_1 \sqsubseteq_{D \longrightarrow E} f_2) \Leftrightarrow (\forall d \in D)[f_1(d) \sqsubseteq_E f_2(d)],$$

for all $f_1, f_2 \in (D, \sqsubseteq_D) \longrightarrow (E, \sqsubseteq_E)$.

The bottom element of this cpo is the continuous function

$$\perp : (D, \sqsubseteq_D) \longrightarrow (E, \sqsubseteq_E)$$

defined by

$$(\forall d \in D)[\perp(d) = \perp].$$

Product of cpo's

Let $\{(D_i, \sqsubseteq_i) \mid i \in I\}$ be an indexed family of cpo's. Their product,

$$(\prod_{i \in I} D_i, \sqsubseteq_{prod}),$$

is a cpo where

$$\prod_{i \in I} D_i = \{(d_i)_{i \in I} \mid (\forall i \in I)[d_i \in D_i]\}.$$

The partial ordering, $\sqsubseteq_{prod}$ is defined by

$$((d_i)_{i \in I} \sqsubseteq_{prod} (e_i)_{i \in I}) \Leftrightarrow (\forall i \in I)[d_i \sqsubseteq_i e_i],$$

for all $(d_i)_{i \in I}, (e_i)_{i \in I} \in \prod_{i \in I} D_i$

The bottom element here, $(\perp_i)_{i \in I}$, is the I-tuple containing the bottom elements from each of the cpo's $(D_i, \sqsubseteq_i)$.

Sum of cpo's

The sum of an indexed family, $\{(D_i, \sqsubseteq_i) \mid i \in I\}$, of cpo's is a cpo

$$(\sum_{i \in I} D_i, \sqsubseteq_{sum}),$$

where

$$\sum_{i \in I} D_i = \bigcup_i \{(d, i) \mid d \in D_i - \{\perp\}\} \cup \{\perp\}.$$

Thus the sum of the above family of cpo's consists of the disjoint union of their underlying sets with bottom elements removed, together with a new element appended. This new element will feature as the bottom element of the sum.

The order relation, $\sqsubseteq_{sum}$, is simply defined by

$$((d, i) \sqsubseteq_{sum} (e, j)) \Leftrightarrow ((i = j) \wedge (d \sqsubseteq_i e)),$$

for all $(d, i), (e, j) \in \sum_{i \in I} D_i$, and

$$(\forall (d, i) \in \sum_{i \in I} D_i)[\perp \sqsubseteq_{sum} (d, i)].$$

7 Finite Elements and Algebraicity

In Section 5 we described the elements of a domain as data descriptions. The strength of each description is determined by the amount of information the element contains about the data objects it could possibly be describing. (An element may (partially) describe more than one data object.) We will now refine our ideas about the information content of domain elements by introducing the notion of finite information.

Firstly, what does it mean for a data description to contain only a finite amount of information? To gain some idea of how to answer this question, consider once again the function space domain

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func}).$$

Our intuition tells us that in this example an element with finite information is a continuous function which maps all but finitely many elements of $State_\perp$ to $\perp$. In other words, finite information means undefinedness almost everywhere. Let us now rigorously formulate the condition which an element of a cpo must satisfy in order for it to be regarded as finite.

Definition 7.1 *Let $(D, \sqsubseteq)$ be a cpo. An element $d \in D$ is finite if for any ω-increasing chain, $(d_i)_{i\in\omega}$, we have*

$$d \sqsubseteq \bigsqcup_{i\in\omega} d_i \Rightarrow (\exists i \in \omega)[d \sqsubseteq d_i].$$

We write the set of finite elements of $(D, \sqsubseteq)$ as D^0. □

If we think of the above definition in terms of our ideas of finite information content, it seems natural. Notice also that the finite elements of

$$((State_\perp, \sqsubseteq_S) \longrightarrow (State_\perp, \sqsubseteq_S), \sqsubseteq_{func})$$

which we identified as continaing only finite information are precisely the finite elements of this domain by Definition 7.1.

Now that we have defined the finite elements of a domain, it remains only to specify the relationship between these elements and those whose information content is not finite. In this respect it seems most natural to think of the non-finite domain elements as being built up out of bits of finite information via some kind of limit process.

Definition 7.2 *A cpo, $(D, \sqsubseteq)$, is* algebraic *if every $d \in D$ is the least upper bound of some ω-increasing chain of finite elements of $(D, \sqsubseteq)$. Further the cpo is called an* ω-algebraic *cpo if the set D^0 is countable.* □

It is generally accepted that domains should at least be ω-algebraic cpo's. These structures have a number of attractive features, for example

- they allow for the existence of powerdomain constructors, needed for the semantics of concurrency and nondeterminism, and
- in the words of Plotkin [6], they allow us to vizualize domains as completions of structures of finite information.

We will investigate the content of the last of the above two features of ω-algebraic cpo's here — the powerdomain constructors will be dealt with in Section 8.

Definition 7.3 *Let P be a set. A relation $\leq \subseteq P \otimes P$ is called a* quasiorder *if it is reflexive and transitive. We call $(P, \leq)$ a quasiordered set.* □

Definition 7.4 *Let $(P, \leq)$ be a quasiordered set. For any subset, $P' \subseteq P$, the* down-closure *of P' is defined by*

$$\downarrow P' = \{p_1 \in P \mid (\exists p_2 \in P')[p_1 \leq p_2]\}.$$

We often write $\downarrow p$ for $\downarrow \{p\}$. □

Definition 7.5 *Let $(P, \leq)$ be a quasiordered set. An* ideal *of $(P, \leq)$ is a set $I \subseteq P$ satisfying the following two conditions:*

- *(down-closedness) $I = \downarrow I$, and*
- *(directedness) $(\forall p_1, p_2 \in I)(\exists p_3 \in I)[(p_1 \leq p_3)$ and $(p_2 \leq p_3)]$.*

We write $\mathcal{I}(P, \leq)$ for the set of ideals of the quasiordered set. □

It can be easily verified that for any quasiordered set, $(P, \leq)$, and any element, $p_1 \in P$, the set

$$\downarrow p_1 = \{p_2 \in P \mid p_2 \leq p_1\}$$

is an element of $\mathcal{I}(P, \leq)$.

Definition 7.6 *An ideal I of a quasiordered set $(P, \leq)$ is called a* principal *ideal if it has a single generator, i.e. there is a $p \in P$ such that $I = \downarrow p$.* □

Definition 7.7 *Let $(P, \leq)$ be a quasiordered set. The complete partially ordered set $(\mathcal{I}(P, \leq), \subseteq)$ is called the* ideal completion *of $(P, \leq)$.* □

It turns out that $(\mathcal{I}(P, \leq), \subseteq)$ may have very nice properties. The next theorem is proved in [6].

Theorem 7.8 *Let $(P, \leq)$ be a countable quasiordered set with a bottom element in the order. Then the ideal completion, $(\mathcal{I}(P, \leq), \subseteq)$, of $(P, \leq)$ is an ω-algebraic cpo.* □

Since for any ω-algebraic cpo, $(D, \sqsubseteq)$, the set of finite elements, D^0, forms a quasiordered set (a partially ordered set in fact) with bottom element under the order relation induced by $\sqsubseteq$, we get the following theorem.

Theorem 7.9 *$(\mathcal{I}(D^0, \sqsubseteq), \subseteq)$ is an ω-algebraic cpo. In fact, it can be shown that $(\mathcal{I}(D^0, \sqsubseteq), \subseteq) \cong (D, \sqsubseteq)$.* □

The net result of all of this is that every ω-algebraic cpo can be regarded as the completion by ideals of its set of finite elements — this is what we were referring to earlier when we claimed that all the elements of an ω-algebraic cpo could be constructed as completions of structures of finite information.

It follows that any countable quasiordered set with bottom element defines the finite elements of its ideal completion and this defining is very elegant.

Proposition 7.10 *Let $(P, \leq)$ be a countable quasiordered set with bottom element, and define the order embedding*

$$i : (P, \leq) \longrightarrow (\mathcal{I}(P, \leq), \subseteq)$$

by

$$i(p) = \downarrow p.$$

Then $\mathcal{I}(P, \leq)^0 = \{\downarrow p \mid p \in P\}$. □

As we shall see in Section 8, this feature of ω-algebraic cpo's (that they can be constructed by completing the ideals of quasiordered sets with bottom elements) is precisely what we need in order to define the powerdomain constructors.

8 Nondeterminism, Parallelism and Powerdomains

As a further extension to the language developed in Section 4 we add the command

$$C ::= C_1 \text{ or } C_2$$

to the syntax. Intuitively, this command instructs the computer to choose arbitrarily one of the commands C_1 and C_2 and then to execute it. A program containing this command could, from a single given input state, end in one of several possible output states and is therefore said to be *nondeterministic*. Our language is now said to support *nondeterminism*.

We introduce nondeterminism into our programming language because situations do arise where nondeterminism is acceptable, desirable and even unavoidable. Parallel programming is an example. Here different program segments are executed simultaneously (often on separate processors). If these program segments are allowed to interact with each other (e.g. access the same variable store) then their speeds of execution may affect the outcome of the

program, i.e., may force nondeterminism. For example, if we let 'C_1 *par* C_2' mean 'execute C_1 and C_2 in parallel' and '$x := a$' mean 'assign the value of a to the variable x', then the program

$$(x := 1)\ par\ (x := 0; x := x + 1)$$

could produce as output $x = 1$ or $x = 2$. This example is taken from Smyth [9] and in this example it is possible for '$x := 1$' to execute after '$x := 0$' and before '$x := x + 1$'.

Returning to our expanded programming language, we need to establish what the semantics of the command *or* should be. In general this command could produce one of two different outputs, so we could take its meaning to be a mapping from a state to a set containing both possible output states. However, none of the domain constructors discussed thus far have sets of states as elements. How then should we construct its corresponding domain operation? We have to construct a domain on the *power set* (i.e. the set of subsets) of a domain. This is a *powerdomain.*

There are three standard powerdomains, called the *Smyth, Hoare* and *Plotkin* powerdomains. We will construct each of these powerdomains on our state space, $(State_\perp, \sqsubseteq)$. Recall that $(State_\perp, \sqsubseteq)$ is a flat domain with countably many elements (see Section 5).

To form the three powerdomains of $(State_\perp, \sqsubseteq)$, we must first decide which elements in the power set of $State_\perp$ to include in our powerdomain. This selection/elimination process is necessary because in a powerdomain two different elements of a power set may with respect to a given semantics be equivalent. Thus, to ensure that each powerdomain is a partial order — and not just a quasi-order — we select only certain elements of the power set for a given powerdomain. To be more exact we select one element, i.e., one subset of $State_\perp$, from each collection of quivalent answers. The different powerdomains include different sets. We, however, always exclude the empty set, since even if a program aborts or fails to terminate it will have the output state $\perp$.

To motivate the three constructions consider the following programs. (We have taken this example from Plotkin [6].)

Program 1. $x := 1$

Program 2. $x := 1$ *or* $(b := tt;\ while\ b = tt\ do\ dummy)$

Program 3. $b := tt;\ while\ b = tt\ do\ dummy$

We pose the question: Intuitively, do any of these programs have the same meaning? We will consider three different possible answers to this question, each of which will give rise to a different powerdomain.

Answer 1: For our first answer, which will be the basis for constructing the Smyth powerdomain, we say that programs 2 and 3 are equivalent because they both may (or will) fail to terminate. Thus for our first answer we identify all programs that have the possibility of failing to terminate. Thinking of answers as sets of states, we are saying that all sets are considered to be equivalent in this sense. For this reason we equate these sets and identify them with the set $State_\perp$.

What about programs that always terminate? A program may contain only finitely many commands. The command *or* allows only finitely many choices to be made so if a program always terminates then it may do so in one of at most finitely many possible output states. To see why the above statement is true think of an execution tree in which each path from the root node to a leaf node represents an execution of the program. All the paths represent all possible executions for a single input. Whenever the program executes an *or* command, the tree branches to represent the possible choices of the *or* statement. If the program always terminates for the given input, then the tree must have only finite height. Thus, the number of leaf nodes (representing possible output states) is also finite. See [7, page 299] for a slightly different proof argument.

Hence we exclude from consideration infinite sets not containing $\perp$, because no program may have such a set as a set of possible output states. Therefore the set of elements of the Smyth powerdomain of $State_\perp$ is

$$\mathcal{P}_S(State_\perp) = \{X \subseteq State_\perp \mid X = State_\perp \text{ or } (X \neq \emptyset, \perp \notin X \text{ and } X \text{ is finite})\}.$$

We need to choose an ordering for $\mathcal{P}_S(State_\perp)$. Inclusion is not suitable since we then do not have a least element. Let us turn first to our intuition. Programs 2 and 3 may fail to terminate so we are uncertain of their outcomes. We think of such programs as being the *worst* because they convey to us the least information. In keeping with this idea, we say that one program is *better* than another if we can be more certain of its outcome, i.e. if it has fewer possible outcomes. Therefore we define the ordering $\sqsubseteq_S$ on $\mathcal{P}_S(State_\perp)$ by

$$X \sqsubseteq_S Y \text{ iff } Y \subseteq X,$$

i.e. $\sqsubseteq_S$ is simply reverse inclusion.

The ordering $\sqsubseteq_S$ is equivalent to the ordering $\sqsubseteq_S^+$ on $\mathcal{P}(State_\perp)$ defined by

$$X \sqsubseteq_S^+ Y \text{ iff } (\forall y \in Y)(\exists x \in X)[x \sqsubseteq y].$$

The ordering $\sqsubseteq_S^+$ is called the *Smyth* ordering. The reason for mentioning this second ordering now will be seen when we discuss constructions of powerdomains for domains in general (i.e. for domains that are not necessarily flat). The two definitions are in general not equivalent.

The following proposition shows that we have in fact constructed a domain on a subset of the power set of $State_\perp$.

Proposition 8.1 $(\mathcal{P}_S(State_\perp), \sqsubseteq_S)$ *is a domain.*

The Smyth powerdomain of a domain is named after M.B. Smyth, who first constructed it [9].

Answer 2: Our second answer to the question of which of the three programs have the same meaning will be to say that programs 1 and 2 have the same meaning because they are able to produce the same output (if any). Thus we are equating two programs if barring nontermination they have the same set of possible output states. For this reason we identify the set X, where $\perp \notin X$, with the set $X \cup \{\perp\}$. Hence every set in the second powerdomain that we are constructing contains $\perp$. Do we want to include every set that contains $\perp$? Certainly the finite sets are included since a program may end in one of finitely many states. What about the infinite sets? Can a program end in one of infinitely many states if it has the possibility of not

terminating? The answer is 'yes' as the following example illustrates.

Example 8.2

$x := 0;$
$b := tt;$
while $b = tt$ *do* $(x := x + 1$ *or* $b := ff)$

□

This program may not terminate, or it may terminate with $b = ff$ and $x = n$ for any $n \in \omega$. Hence it is possible for a program to have infinitely many possible output states but only if the program may not terminate.

So our second powerdomain contains precisely all subsets of $State_\perp$ that contain $\perp$. This powerdomain is called the Hoare powerdomain and we denote it by '$\mathcal{P}_H(State_\perp)$'. We must decide how these sets should be ordered. In this case we are not interested in how certain we are of the outcome, rather we are concerned with how likely the program is to terminate. We say that the more terminating possibilities a program has, the better the program is because it is more likely to terminate. This means that we order the elements of $\mathcal{P}_H(State_\perp)$ by inclusion. Hence our second powerdomain is

$$\mathcal{P}_H(State_\perp) = \{X \subseteq State_\perp \mid \perp \in X\}$$

ordered by inclusion. On $\mathcal{P}_H(State_\perp)$, the inclusion ordering is equivalent to the *Hoare* ordering $\sqsubseteq_H^+$ defined by

$$X \sqsubseteq_H^+ Y \text{ iff } (\forall x \in X)(\exists y \in Y)[x \sqsubseteq y].$$

Again the significance of this second ordering will be seen later.

The following proposition shows that we have in fact constructed another domain on the power set of $State_\perp$.

Proposition 8.3 $(\mathcal{P}_H(State_\perp), \subseteq)$ *is a domain.*

The reason this powerdomain is called the Hoare powerdomain is given in Winskel [11].

Answer 3: Our third and final answer to the question regarding the three programs is that the three programs are all different. This may seem to be the most reasonable answer, but it results in the most complicated of the three powerdomains. Again our first step is to decide which subsets of $State_\perp$ to include in our powerdomain. This time, though, we make no identifications and so we must include every set that could be the set of possible output states of a program. Recall that if a program always terminates, it may do so in one of at most finitely many possible output states. Thus we exclude infinite sets that do not contain $\perp$. We know also that if a program may not terminate, it could terminate in one of infinitely many possible output states (from example 8.2). Hence we include in our third powerdomain, $\mathcal{P}_P(State_\perp)$, all sets containing $\perp$. Thus

$$\mathcal{P}_P(State_\perp) = \{X \subseteq State_\perp \mid \perp \in X \text{ or } (X \neq \emptyset, \perp \notin X \text{ and } X \text{ is finite})\}.$$

To define the ordering $\sqsubseteq_P$ on $\mathcal{P}_P(State_\perp)$ we first consider programs that always terminate. We say that two such programs are comparable if and only if their sets of possible output states are identical in which case they will, of course, be considered to be equivalent. Next we say that termination is *better* than nontermination so no program that may fail to terminate will be above a program that always terminates. Thus for any set $X \subseteq State_\perp$, $\perp \notin X$

$$X \sqsubseteq_P Y \text{ iff } X = Y. \quad (13)$$

This tells us only that no set is above a set not containing $\perp$. This implies that termination is *good* and a program is better than another if it is more likely to terminate. Let us, therefore, order sets containing $\perp$ by inclusion. Hence if $\perp \in X$ and $\perp \in Y$ then

$$X \sqsubseteq_P Y \text{ iff } X \subseteq Y. \quad (14)$$

We are not finished yet since we do not know whether a set containing $\perp$ could be below a set not containing $\perp$. It seems reasonable to take the following approach. Suppose program A always terminates and program B may fail to terminate. Then we can say that program B approximates program A if every terminating state of B is a terminating state of A. In terms of sets, if $\perp \notin Y$ and $\perp \in X$ then

$$X \sqsubseteq_P Y \text{ iff } X - \{\perp\} \subseteq Y. \quad (15)$$

Putting the partial definitions of $\sqsubseteq_P$ (see equations (13), (14) and (15)) together, if $X, Y \in State_\perp$ then

$$X \sqsubseteq_P Y \quad \text{iff} \quad (\perp \notin X \text{ and } X = Y) \text{ or } (\perp \in X \text{ and } X - \{\perp\} \subseteq Y - \{\perp\}).$$

Of course, in the sentence above '$Y - \{\perp\}$' could be replaced by 'Y'. Then we have the following:

Proposition 8.4 $(\mathcal{P}_P(State_\perp), \sqsubseteq_P)$ *is a domain.*

Thus we have constructed a third powerdomain. This powerdomain is called the Plotkin powerdomain and is named after G. Plotkin. He not only constructed it, but also introduced the powerdomain constructions [5].

The ordering $\sqsubseteq_P$ is equivalent to the *Plotkin* ordering $\sqsubseteq_P^+$ on $\mathcal{P}_P(State_\perp)$ defined by

$$X \sqsubseteq_P^+ Y \quad \text{iff} \quad (\forall y \in Y)(\exists x \in X)[x \sqsubseteq y] \text{ and } (\forall x \in X)(\exists y \in Y)[x \sqsubseteq y]$$
$$\text{iff} \quad X \sqsubseteq_S^+ Y \text{ and } X \sqsubseteq_H^+ Y.$$

We have constructed the three standard powerdomains on the flat domain $State_\perp$. What about domains that are not flat?

The situation with non-flat domains is more complicated. We can no longer simply use inclusion and reverse inclusion as this ignores the structure on the original domain. For example, if in a domain $a \sqsubseteq b \sqsubseteq c$ where a, b and c are distinct elements then $\{a, b\} \not\sqsubseteq \{a, c\}$ and $\{a, c\} \not\sqsubseteq \{a, b\}$. We may, however, want $\{a, b\} \sqsubseteq \{a, c\}$ as a approximates a and b approximates c. To obtain the three powerdomain structures when beginning with a general domain, we use the method of completion by ideals which was defined in Section 7 (Definition 7.7).

We will describe the three powerdomain constructions for an arbitrary domain $(D, \sqsubseteq)$. We cannot directly apply theorem 7.8 to $(D, \sqsubseteq)$ as D may not be countable. Therefore, we must first construct a quasiordered set from $(D, \sqsubseteq)$ to which we can apply theorem 7.8 . Recall that a domain, being an ω-algebraic cpo, has countably many finite elements. Recall also that every element of a domain is the lub of an ω-increasing chain of finite elements. Thus, the set of finite elements, D^0, of D seems to be a good place to start our construction. It is not enough, though, to take the ideal completion of $(D^0, \sqsubseteq)$ for as shown in Section 7

$$(\mathcal{I}(D^0, \sqsubseteq), \subseteq) \cong (D, \sqsubseteq),$$

which would leave us back where we started.

Rather, we begin by taking the set of all finite, nonempty sets of finite elements of D. Denote this set by $M(D)$. To use theorem 7.8 on $M(D)$, we need to know that it is countable.

Proposition 8.5 *If $(D, \sqsubseteq)$ is a domain then $M(D)$, the set of finite, nonempty sets of finite elements of D, is countable.*
Proof: See, for example, Enderton [2] which shows that a countable union of countable sets is countable.

Next we need to place an ordering on $M(D)$ that will make it into a quasiordered set. The inclusion ordering would accomplish this but then we would have lost the original ordering $\sqsubseteq$ on D. Instead we use the ordering $\sqsubseteq$ on D to obtain orderings on $M(D)$. We, in fact, obtain three different orderings which give rise to the three different powerdomains (these orderings were mentioned in our earlier discussion but we define them formally here). First we give some motivation for the definitions of these orderings. Let $X, Y \in M(D)$. Then we say that X is less than Y in the Smyth ordering if Y is *above* X, in the sense that every element of Y is above or approximated by some element of X. In the Hoare ordering, X is less than Y if X is *below* Y, in the sense that every element of X is below or approximated by some element of Y. The Plotkin (or Egli-Milner) ordering combines these two orderings. Hence X is less than Y in the Plotkin ordering if X is below Y and Y is above X. Formally, we have the following definition.

Definition 8.6 *Let $(P, \leq)$ be a quasiordered set. The Smyth $(\leq_S^+)$, Hoare $(\leq_H^+)$ and Plotkin $(\leq_P^+)$ orderings are defined on $\mathcal{P}(P)$, the power set of P, respectively by*

$$X \leq_S^+ Y \quad \text{iff} \quad (\forall y \in Y)(\exists x \in X)[x \leq y]$$
$$X \leq_H^+ Y \quad \text{iff} \quad (\forall x \in X)(\exists y \in Y)[x \leq y]$$
$$X \leq_P^+ Y \quad \text{iff} \quad X \leq_S^+ Y \text{ and } X \leq_H^+ Y.$$

Having defined these orderings, we note that they are quasiorderings. This is the subject of our last proposition.

Proposition 8.7 *The Smyth, Hoare and Plotkin orderings on the power set of a quasiordered set are again quasiorderings.*

We are finally in a position to construct the three standard powerdomains on an arbitrary domain. For each of these powerdomains we could as before determine which elements, ie., which sets of original domain elements, are equivalent and choose one representative from the equivalent ones. In this way we could turn the quasiorders into partial orders, and then the powerdomains would be domains.

Definition 8.8 *Let $(D, \sqsubseteq)$ be a domain.*

1. *The Smyth powerdomain is given by the ideal completion of $M(D)$ with the Smyth ordering.*
   i.e. $(\mathcal{I}(M(D), \sqsubseteq_S^+), \subseteq)$ is the Smyth powerdomain.
2. *The Hoare powerdomain is given by the ideal completion of $M(D)$ with the Hoare ordering.*
   i.e. $(\mathcal{I}(M(D), \sqsubseteq_H^+), \subseteq)$ is the Hoare powerdomain.
3. *The Plotkin powerdomain is given by the ideal completion of $M(D)$ with the Plotkin (or Egli-Milner) ordering.*
   i.e. $(\mathcal{I}(M(D), \sqsubseteq_P^+), \subseteq)$ is the Plotkin powerdomain.

We have constructed three different domains on the power set of a domain. Our original aims were in particular to find a semantics for the command *or* and in general to define a semantics for nondeterminism. We can accomplish these goals by defining functions from $State_\perp$ to a powerdomain of $State_\perp$. Which powerdomain to choose depends on the application the language designer has in mind.

References

1. A Aho and J D Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass, 1977.
2. H B Enderton. *Elements of Set Theory.* Academic Press, New York, 1977.
3. C Gunter and D Scott. 'Semantic domains'. In J van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics.* Elsevier, Amsterdam, (1990).
4. P Mosses. 'Denotational semantics'. In J van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B, Formal Models and Semantics.* Elsevier, Amsterdam, (1990).
5. G Plotkin. 'A powerdomain construction'. *SIAM Journal of Computing*, **5**, (1976).
6. G Plotkin. 'The category of complete partial orders: A tool for making meanings'. Post-graduate lecture notes, Computer Science Department, University of Edinburgh, Edinburgh, (1982).
7. D Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn and Bacon, 1986.
8. D Scott and C Strachey. 'Towards a mathematical semantics for computer languages'. In *Proceedings of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series Vol 21.* Polytechnic Institute of Brooklyn, (1971).
9. M Smyth. 'Power domains'. *Journal of Computer and System Science*, **16**, (1978).
10. J Stoy. *Denotational Semantics. The Scott-Strachey Approach to Programming Languages.* MIT Press, Cambridge, 1977.
11. G Winskel. 'Powerdomains and modality'. *Theoretical Computer Science*, **36**, (1985).

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as 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 triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- 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 numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- 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 [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above 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.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

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

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

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.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, **9**:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African
Computer
Journal

Number 9, April 1993
ISSN 1015-7999

Suid-Afrikaanse
Rekenaar-
tydskrif

Nommer 9, April 1993
ISSN 1015-7999

Contents

Pretoria Drukkers