South African
Computer
Journal
Number 8
November 1992

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 8
November 1992

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 Rekenaarverenging 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

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute

Professor Judy Bishop
University of Pretoria

Professor Donald D Cowan
University of Waterloo

Professor Jürg Gutknecht
ETH, Zürich

Professor Pieter Kritzinger
University of Cape Town

Professor Fred H Lochovsky
University of Toronto

Professor Stephen R Schach
Vanderbilt University

Professor Basie von Solms
Rand Afrikaanse Universiteit

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

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: +27 (12) 420-2504 Fax: +27 (12) 43-6454

Sigma Press

Guest Contribution

The paper below was given as an invited address by Prof Roode at the July 1992 Conference of the South African Computer Lectures' Association. (Editor)

The Ideology, Struggle and Liberation of Information Systems

Dewald Roode
Department of Informatics, University of Pretoria

In 1989, Denning *et al* presented the final report of the Task Force on the Core of Computer Science in an article entitled "Computing as a Discipline" [3]. This was said to present a new intellectual framework for the discipline of computing and proposed a new basis for computing curricula.

In the words of the authors, "an image of a technology-based discipline is projected whose fundamentals are in mathematics and engineering." Algorithms are represented as the most basic objects of concern and programming and hardware design as the primary activities. Although there is wide consensus that computer science encompasses far more than programming, the persistent emphasis on programming "arises from the long-standing belief that programming languages are excellent vehicles for gaining access to the rest of the field" [3].

The new framework sets out to present the intellectual substance of the field in a new way, and uses three paradigms to provide a context for the discipline of computing. These paradigms are *theory*, rooted in mathematics; *abstraction*, rooted in the experimental scientific method and *design*, with its roots in engineering.

Programming, the report recommends, should still be a part of the core curriculum and programming languages should be seen and used as vehicles for gaining access to important aspects of computing.

The following short definition is offered of the discipline of computing [3]:

> The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, *"What can be (efficiently) automated?"*

In the same issue of Communications, tucked away towards the end of the journal, an article by Banville and Landry asked the innocent question "Can the Field of MIS be disciplined?" [1]. It is not clear whether the use of the word "discipline" in both articles was purely coincidental – however, the implications were quite clear: computer science was able to talk about "computing as a discipline," and indeed, could present a report which, in a sense, was a culmination of more than twenty years' efforts. Yet, its sister discipline was still asking questions of a very introvertive nature about itself.

It has become quite clear that the fields (leaving aside for the moment the questions of "disciplines") of computer science and information systems (or MIS, informatics, or whatever other name we want to attach to it) have different aims and objectives, different problems that confront it, and, yes, if we want to be truly scientific, different paradigms. To support the latter statement, it is sufficient to contrast the three paradigms of computing with the four paradigms of information systems development described by Hirschheim and Klein [5]. It can be said that a central activity in information systems is the development of information systems, and that therefore, these paradigms have implications for the field of information systems. The four paradigms can be characterized briefly, as follows:

- The analyst as systems expert
- The analyst as facilitator
- The analyst as labour partisan
- The analyst as emancipator or social therapist.

In the same spirit, Lyytinen sees the "systems development process as an instrument in organizational change" [6] and remarks that analysts' principal problems are "in understanding the goals and contents of such change instead of solving technical problems." Already in 1987 Boland [2] observed that: "designing an information system is a moral problem because it puts one party, the designer, in the position of imposing an order on the world of another."

This is clearly a far cry from Denning *et al*'s statement that the fundamental question is "what can be automated?" At the same time, within the context of the field of computing, there is nothing wrong with this question, and it is probably the right question for practitioners of computing to continually ask themselves. But it is a disastrous question for a practitioner of informatics to ask. And it has taken us quite a long time to realise this – that the two disciplines have fundamentally different roles to play. These roles are complementary and supportive, and not destructively opposed.

The liberation of information systems lies in realising this elemental truth: that information systems are man-made objects designed to effect organisational change and that, as such, they can ill be studied using the paradigms of abstraction and engineering mentioned above.

What then is needed? Banville and Landry offer the consolation that we need not concern ourselves too much about the lack of discipline, and that we can indeed even pride ourselves in being a fragmented adhocracy. It is, in fact, even healthy to continue in all sorts of directions. During this process of finding itself, a discipline should be allowed a considerable degree of latitude, and many avenues should be explored. This obviously makes the field of information systems extremely exciting: it is in the process of discovering remarkable truths, discovering that there are in reality people out there using the systems which analysts design and build, and that the most intriguing problems centre around the role of people in all of this: the analyst, the user, their interaction, the impact of sytems on the work lives of workers on all levels, the impact on organizations. These are questions which have mostly been ignored or lightly treated over the years, but which have emerged as *the* problems to be solved. We do not have the tools to solve them – not yet; but a good starting point would certainly be to first understand more about our field and its research tools, for the empirical, positivist approach so often employed will not suffice to solve the above problems.

In the spirit of contributing to the liberation movement of information systems, we have embarked on a study of research on research in Information Systems, and will report on the results more fully in the near future. We define Information Systems as follows [4]:

> Information Systems is an inter-disciplinary field of scholarly inquiry, where information, information systems and the integration thereof with the organisation is studied in order to increase the effectiveness and efficiency of the total system (of technology, people, organisation and society).

In Information Systems then, we see the fundamental question underlying the entire discipline, to be the problem of balancing the need to contribute, through information systems, to the achievement of the mission of the organisation with the moral responsibility to develop and implement socially accepted information systems.

Each of the fields, computer science and information systems, benefits enormously from the activities of the other. Nonetheless, we must recognize the different approaches used by the two disciplines and allow them to complement each other. It should not be our business to convince one another that the universal truth is that which we use in our discipline – whether that be computer science or information systems. Instead, we should seek out the opportunities for synergy, and for complementing each other. If we succeed in doing this at SACLA, then we could indeed do ourselves proud.

References

1. C Banville and M Landry. 'Can the field of MIS be disciplined?'. *Communications of the ACM*, **32**(1):48–60, (1989).
2. R J Boland and R A Hirschheim, eds. *Critical Issues in Information Systems Research*. John Wiley & Sons Ltd., 1987.
3. P J Denning, D E Comer, D Gries, M C Mulder, A Tucker, A J Turner, and P R Young. 'Computing as a discipline'. *Communications of the ACM*, **32**(1):9–23, (1989).
4. N F Du Plooy, L D Introna, and J D Roode. 'Notes on research in information systems'. Unpublished research report, Department of Informatics, University of Pretoria, (1992).
5. R Hirschheim and H K Klein. 'Four paradigms of information systems development'. *Communications of the ACM*, **32**(10):1199–1216, (1989).
6. K Lyytinen. 'New challenges of systems development: A vision of the 90's'. *Data Base*, pp. 1–12, (Fall 1989).

Editor's Notes: To Compete or Collaborate

Human interaction invariably brings with it a blend of competition and collaboration. Competition means that one enjoys the exhilaration of winning while the other endures the shame of loosing. Because of this reward/punishment mechanism, it is a widely assumed that competition enhances performance and efficiency. This dogma pervades not only commerce, sport and politics, but is found in practically all areas of human endeavour, including research.

The competitive spirit in research is found in the well-known saga of Watson and Crick racing to unravel the double helix structure of DNA. Not so well-known, though equally illustrative, is the intensity of Newton's stratagems to oust Leibnitz from receiving any credit for differentiation. Recently there have been reports of scientists who have either tolerated or manufactured fraudulent results in order to win some or other scientific race. The space race, the arms race, the race for an AIDS cure, the scurry for faster smaller hardware, the race for awards, the drive for publications, Nobel prizes: all of this attests to a profoundly competitive international research culture.

But while competition might be the handmaiden of commerce and sport, it is the harlot of research – an unfortunate concomitant of the silly side of human nature. The archetypal researcher not only rises above the incidentals of human accolades; he disdains them. By tradition, the definitive research qualification is a PhD – a Doctor of Philosophy – a lover of thought. Discovery and thought are not only by their very nature rewarding, they are also humbling. When the archetypal researcher moves outside his interior thought-world, it is to share his discoveries. If he is childish, it is not the little boy flexing his biceps and saying: "I'm stronger than you" but the child rushing to

tell everyone: "Wow – look at this!" He is forgetful of self: Pythagoras, oblivious of the invading enemy and his impending death while he researches in the sand; Archimedes shouting "Eureka" without care for his nudity. The competitive spirit is a crass intrusion into this ancient legacy of innocence and selflessness.

By its nature, collaboration thrives in a climate of easy social intercourse. It may initially feel uncomfortable for researchers, who are inclined to be socially inept and are wont to bury themselves in work away from society. However, once the plunge to collaborate is taken there is ample evidence that it leads to successful research. In maximizing the use of available talent, it brings about a synergy in which two heads are better than one. All participants enjoy its rewards and no individual has to endure the full weight of its failures. In fact, the notion of collaboration is now so commonplace that significant research seems impossible without it. The tendency, however, is to encourage research collaboration within an organisation, but to emphasize competition in relation to outside organisations.

During a forum discussion at the July South African Computer Lecturers' Association (SACLA) conference, an appeal was made for greater collaboration between universities. Not surprisingly, the information technology disciplines at local universities have always had both a competitive and a collaborative relationship. The competitiveness usually takes the form of friendly rivalry, while the very existence of SACLA bears testimony to a rather unique collaborative relationship. In latter years the competitiveness seems to have intensified, while electronic mail and other developments have improved the prospects for collaboration. At issue, then, is whether there is an imbalance between these dual forces. The appeal at the SACLA forum implied that there is, and I would strongly agree. It is my view (my prejudice, if you will) that competition between universities is a self-indulgent and wasteful dissipation of energy.

Those who are inclined to compete should seriously examine what is to be gained. It is unconvincing to argue that winning makes a significant impact on the way in which students select universities: in the main, this is a matter of geography and language preference. To some extent, the same might be said about staff, although research reputation perhaps plays a more important role here. Neither are research funding agencies (e.g. the FRD) influenced by whether X is "better" in some or other sense than Y. On the contrary, it has wisely been decided to fund on the basis of criteria that are believed to be objective, without any reference whatsoever to the performance of competitors. True enough, funds are limited, but it is precisely for this reason that it is wasteful to divide the little there is between divergent research efforts.

It seems to me that there is a wealth of research talent out there, but that each researcher selects an area of interest almost as a matter of whim. There is an urgent need for well-coordinated collaboration on focussed research areas that have been carefully selected as directly relevant to the country. It is especially incumbent on those who finance, manage and lead research to identify such areas and to encourage collaboration in every possible way.

I look forward to the manifestation of such collaboration in SACJ publications authored by researchers from different university departments. To date there have been none of consequence. If we fail to collaborate, we are in danger of becoming little Don Quixotes who spend our lives attacking windmills and defending castles of xenophobia and irrelevance.

Logic Programming: Ideal vs. Practice

W A Labuschagne* P L van der Westhuizen†‡
*Department of Computer Science and Information Systems, University of South Africa, P O Box 392, Pretoria, 0001
†ABSA, ABSA Park, Gwen Avenue, Sandton, 2199

Abstract

The logician who encounters logic programming expects it to involve 'using predicate logic as a programming language'. He is soon disillusioned. He discovers that it really means 'programming in PROLOG', which in turn seems to mean 'programming in a procedural language combining a peculiar syntax with an unusual flow of control that runs counter to certain basic assumptions of logic'. The baffled logician is left wondering how PROLOG could have come to be viewed as a logic programming language. In this paper we compare Gabbay's logic programming ideal with logic programming in practice and show how they may be reconciled. The reconciliation has implications for the teaching of logic programming and throws some light on the possible nature of future implementations. A crucial role is played in the discussion by the notion of feasibility. It is argued that, as yet, no appropriate theoretical framework for analysing the efficiency of proof procedures exists, but that some indication of what such a framework should look like can be gleaned from practice.
Keywords: *logic programming, resolution, negation-as-failure, closed world assumption, completed database, stratified program.*
Computing Review Categories: *A1, D3.1, F4.1*

Received January 1992

1 Introduction

What is logic programming? The answer depends pretty much on who you ask. The standard text, Lloyd [11], espouses a narrow view, which may be caricatured as the claim that logic programming is PROLOG programming. More precisely, from Lloyd's perspective logic programming involves the use of an SLDNF-resolution theorem-prover to check the consistency of sets of normal clauses, as described in Section 3. A logician encountering the term 'logic programming' would be inclined to the broader view that it should involve programming in some logic language having a well-understood model-theoretic semantics, such as the classical predicate calculus. A suggestion attributed to Dov Gabbay in [18] goes even farther: the semantics does not necessarily have to be classical, and the language may be augmented with control operations provided these are "capable of being implemented in a way that respects the logic, so that the logical meaning of each step is clear". Of course, "above all, it should be feasible", i.e. capable of dealing efficiently with a broad range of problems. In subsequent sections we will look more closely at the *logic programming ideal* embodied in Gabbay's proposal and examine the trend of thought that has led to the general acceptance of the much narrower view of Lloyd. Finally we consider the question whether SLDNF-resolution is the end of the road for logic programming or the beginning.

‡ Research undertaken while registered as an MSc student in the Department of Computer Science of the University of South Africa, in support of a dissertation entitled 'Negation in Deductive Databases.'

2 Predicate logic as programming language

Logic programming in its present form, i.e. PROLOG programming, can be traced back to the joint work of Kowalski and Colmerauer which led *inter alia* to the influential paper [8]. In that paper the ambiguity that characterises the phrase 'logic programming' is already evident. The introduction argues for the broad logic programming ideal, i.e. for the use of a well understood logic language in a way that logicians would find natural:

> "The purpose of programming languages is to enable the communication from man to machine of problems and their general means of solution. The first programming languages were machine languages. To communicate, the programmer had to learn the psychology of the machine and to express his problems in machine-oriented terms ... Concerned with the other end of the man-machine communication problem, predicate logic derives from efforts to formalise the properties of rational human thought ... As a programming language, predicate logic is the only language which is entirely user-oriented. It differs from existing high-level languages in that it possesses no features which are meaningful only in machine-level terms."

In the latter parts of [8], it becomes clear that Kowalski has in mind a particular subset of predicate logic, namely Horn clauses. His view that "clausal form defines a natural and useful language in its own right" is supported by a demonstration that theorem-proving by SLD-resolution has a procedural interpretation. In subsequent sections we shall describe SLD-resolution and its procedural interpre-

tation, and extend the method to SLDNF-resolution. Some consequences of the latter contradict the expectations of logicians, despite which the idea nevertheless constitutes an ingenious response to the question 'Can one really *program* in a logic language?'

The languages of classical propositional and predicate logic are known to be context-free and hence compilable [10]. But programming would surely require that the language possess a *procedural* component, because programs have to do things, i.e. are dynamic entities rather than static descriptions. Languages such as that of predicate logic, on the other hand, are *declarative*. A declarative language may be characterised (e.g. Ullman [21, p.21]) as a language in which one can express what one wants without necessarily saying how to compute it. A sentence of such a language describes, at one or another level of detail, some situation, and hence has a truth value: the description is either accurate, so far as it goes, in which case the sentence is true, or inaccurate, in which case the sentence is false. More subtle distinctions reflected by the assignment of intermediate truth values are possible, but the essential point is that a language becomes declarative when it is provided with a declarative semantics, i.e. a semantics that governs the assignment of truth values to all sentences of the language. How can a static declarative language be used dynamically for programming?

There are two possible approaches, if by 'possible' we mean 'compatible with the logic programming ideal'. One may seek to augment the language by including language elements that represent actions. This *dynamic logic* option carries with it the obligation to work out an acceptable declarative semantics for the augmented language, for example along the lines suggested by Van Benthem [22]. On the other hand, one may choose to remain within a standard logic language with its semantics and to rely on the existence of algorithms which, given a database Δ and a well formed-formula (wf) α that is entailed by Δ, will confirm that α is entailed by Δ. If the algorithm can be chosen so that it provides *constructive* proofs of existential statements (for example, proves 'There exists an even prime number' by actually exhibiting the integer 2), then the algorithm will in effect perform computations, thereby supplying the dynamic aspect one seeks. It is this second alternative that Kowalski selected and that we shall examine.

3 The procedural reading of clauses

The procedural reading of SLD-resolution described in [8] applies only to the subset of the predicate calculus consisting of statements that can be expressed in a certain way, namely as sets of *Horn clauses*. After describing the details, we will discuss the use of *negation-as-failure* to extend the procedural reading to *normal clauses*.

The well-formed formulas (wfs) of propositional and predicate logic are built up from *atoms* with the aid of connectives ($\neg$ for 'not', $\vee$ for 'or', $\wedge$ for 'and', $\leftarrow$ for 'if') and, in the case of predicate logic, quantifiers ($\forall$ for 'for all', $\exists$ for 'for some'). In the propositional case, which will suffice for illustrative purposes, the atoms are opaque sentence symbols denoted by the metavariables A, B, C, and D, while in predicate logic the atoms are strings consisting of a predicate symbol followed by an appropriate number of terms. Atoms and negations of atoms are called literals and denoted by the metavariable L. A normal clause may be regarded as a wf of the form

$$B \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_n$$

in which the atom B is called the *head* and the conjunction of literals $L_1 \wedge \ldots \wedge L_n$ is called the *body*. (Slight variations are permissible, e.g. the head or the body may be empty.) In propositional logic, any wf is logically equivalent to a finite set (or, if you prefer, a conjunction) of such clauses, since every proposition may be expressed in conjunctive normal form, the conjuncts of which are easily rewritten in the form of clauses (by virtue of the equivalence between $\beta \leftarrow \alpha$ and $\beta \vee \neg\alpha$). In predicate logic, this is not quite the case; every variable occurring in a clause is assumed to be universally quantified by an invisible quantifier carefully not written on the left, and the process one has to use in order to eliminate existential quantifiers (Skolemisation) does not always deliver a logically equivalent wf. In the context of predicate logic, therefore, the set of wfs that can be expressed as conjunctions of normal clauses is a proper subset of the set of all wfs.

A *Horn clause* is a normal clause of the form

$$B \leftarrow A_1 \wedge \ldots A_m$$

in which the body $A_1 \wedge \ldots A_m$ is a conjunction of (unnegated) atoms. Kowalski proposed that a *definite* Horn clause (i.e. a Horn clause with a nonempty head; equivalently, a disjunction of literals exactly one of which is an atom) be read as a procedure declaration, with B the name of the procedure and the body consisting of procedure calls A_i. If $m = 0$, we have a special kind of definite Horn clause, namely a procedure with a head, B, but no body (written $B \leftarrow$). Such single atoms may be thought of as *assertions of fact*. A third case arises if the Horn clause has a body $A_1 \wedge \ldots A_m$ but no head (written $\leftarrow A_1 \wedge \ldots A_m$). Such clauses, which may be thought of as procedures with no names, are called *goals* since they assert the goal of successfully executing all of the procedure calls A_i. Strictly speaking, such goal clauses are disjunctions of negative literals (the variables of which, in the predicate logic case, are all universally quantified). Finally, if one were to attempt to express a contradictory wf as a Horn clause, one would employ the *empty clause*. This nameless procedure with an empty body, which we denote by '$\Box$', can be regarded as a satisfied goal statement and therefore has the effect of a halt statement.

A program consisting of one or more definite Horn clauses switches, when triggered by *procedure invocation*, from being a static description of some situation to being a dynamic entity which answers questions. Procedure invocation involves adding a goal to the program and then relying on an SLD-resolution theorem-prover to check the consistency of the resulting set of clauses. Resolution is

a very simple rule that infers, from premises like $A \leftarrow B$ and $B \leftarrow C$, the conclusion $A \leftarrow C$ obtained by canceling the head of one clause and an identical atom in the body of another. The letters 'SLD' refer to a particular strategy for applying resolution.

Example 1

Given the program consisting of the three definite Horn clauses

$$\begin{aligned} A &\leftarrow B \\ C &\leftarrow \\ B &\leftarrow \end{aligned}$$

the goal

$$\leftarrow A \wedge C$$

would trigger an SLD-resolution theorem-prover to perform the following sequence of steps. From the goal $\leftarrow A \wedge C$ and the clause $A \leftarrow B$, the new clause $\leftarrow B \wedge C$ is inferred by canceling A. The new clause also has the form of a goal, and therefore triggers the next step, in which the new goal $\leftarrow B \wedge C$ is matched with the fact $B \leftarrow$ to infer $\leftarrow C$. This again has the form of a goal, and so another resolution occurs, matching $\leftarrow C$ with fact $C \leftarrow$ to deliver the empty clause $\square$, and the theorem-prover halts. It is clear that the basic strategy is to let the goal clause drive the process. Note that implementations (i.e. PROLOG) impose an additional constraint on the theorem-prover; the clauses in the program are tried in a fixed order from the top of the list. Hence the logically equivalent program with $A \vee \neg A$ at the top of the list in the form $A \leftarrow A$ causes the theorem-prover to loop.

The procedural interpretation illustrated above applies only to Horn clauses. This is a drastic restriction. One can't express 'or', because $A \vee B$ in clausal form would be either $A \leftarrow \neg B$ or $B \leftarrow \neg A$, and in either case one has a negated atom in the body. So it is necessary to work with normal clauses rather than Horn clauses, even though it may not be immediately obvious how to give the negated atoms a procedural reading. The solution built into PROLOG and characteristic of Lloyd's view is to extend SLD- to SLDNF-resolution by interpreting negation as failure. The occurrence of a negated atom like $\neg A$ in a goal statement is no longer an invitation to succeed in executing the procedure call A, but to show exhaustively that it is not possible to succeed in executing the procedure call A.

Example 2

Given the program consisting of the three clauses

$$\begin{aligned} A &\leftarrow (\neg B) \wedge C \\ B &\leftarrow D \\ C &\leftarrow \end{aligned}$$

the goal

$$\leftarrow A$$

would trigger the following sequence of steps. From the goal $\leftarrow A$ and the clause $A \leftarrow (\neg B) \wedge C$, the new goal $\leftarrow (\neg B) \wedge C$ is constructed. Faced with the negated atom $\neg B$, the theorem-prover recursively retreats to determine whether the procedure B can successfully be executed.

From the new temporary goal $\leftarrow B$ and clause $B \leftarrow D$ it constructs the new temporary goal $\leftarrow D$. Note that the theorem-prover has not yet succeeded in executing the procedure call B, since it still has to try to execute the body D. This it fails to do, since there is no program clause with D in its head.

Returning to the main sequence, it takes $\neg B$ to have been established by its failure to execute B and moves to what is left of the goal $\leftarrow (\neg B) \wedge C$, namely $\leftarrow C$. Matching with program clause $C \leftarrow$ produces the empty clause, and the theorem-prover halts.

The procedural interpretation of SLDNF-resolution is inspired by computer science, not by logic. A logician would think of the program as a set of axioms and of the goal clause as the negation of the query, i.e. of the theorem to be proved. If a well-behaved theorem-proving algorithm can massage program plus goal clause into the empty clause, then the goal clause is inconsistent with the program and therefore the query is entailed by the program. This suggests two potentially troublesome questions. Is SLDNF-resolution well-behaved? And even if it is well-behaved, bear in mind that a normal goal clause in predicate logic is a universally quantified disjunction of literals. What does one do if the theorem one wishes to prove is not an existentially quantified conjunction of literals so that its negation does not have the form of a goal clause?

4 Some drawbacks of negation-as-failure

Example 1 illustrates what is usually called SLD-resolution, in which 'S' stands for 'selection rule', 'L' for 'linear input' (phrases referring to the way the goal clause drives the process of resolution) and 'D' for 'definite Horn clauses'. Example 2 illustrates SLDNF-resolution, which is just SLD-resolution with negation-as-failure grafted onto it so as to accommodate normal clauses. In this section we will show that the use of negation-as-failure amounts to rejecting the usual semantic equivalence as the underlying equivalence relation on sentences and the usual semantic entailment as the underlying order relation on sentences; in short, rejecting the classical declarative semantics of propositional and predicate logic.

Propositional logic may be viewed mathematically as the study of Tarski (also but mistakenly called 'Lindenbaum') algebras. These are Boolean algebras having as elements equivalence classes of strings, which strings in turn are the wfs of a formal language whose alphabet comprises atoms and connectives. The equivalence relation imposed on the wfs is intimately related to the manner in which truth values may be assigned to wfs; two wfs are equivalent iff they will always receive the same truth

value. The Boolean ordering on the equivalence classes is also intimately related to the truth value semantics; a wf α lies below a wf β iff whenever α receives the truth value 'true', so does β. This Boolean ordering is the semantic entailment relation, i.e. a wf α is said to *entail* a wf β iff α lies below β.

How do 'proofs' fit into the picture? A proof architecture is a selection of wfs (known as axioms) and proof rules (such as the familiar 'modus ponens', i.e. 'infer β from α and $\beta \leftarrow \alpha$'). The intention is that the proof architecture should permit the existence of a theorem-proving algorithm that accurately mimics the semantic entailment relation, i.e. that is sound and complete. A theorem-proving algorithm is sound iff every wf obtained from the chosen axioms via the algorithm is semantically entailed by the axioms. Conversely, a theorem-proving algorithm is complete iff each wf semantically entailed by the axioms can be obtained from the axioms via the algorithm. (Interested readers may consult either a standard text like Hamilton [7] or the broader survey of Sundholm [20] for an introduction to non-resolution proof architectures. Wos *et al* [24] is a good introduction to theorem-proving algorithms based on resolution.)

What is the point of a proof architecture if its associated algorithm can at best tell you what the entailment relation already tells you? Well, in a nutshell, a proof architecture may, because its proof rules work by syntactic inspection of the strings presented to it, allow a more efficient demonstration that one wf entails another than would be possible if one were to use truth values. Consider the problem of showing that a wf α entails a wf β. By the definition of entailment, one must show that whenever α is assigned the value 'true', so is β. If n different atoms occur in α and β, the process requires on the order of 2^n steps, since each of the n atoms may be either true or false. A proof architecture may, if for instance the wf $\beta \leftarrow \alpha$ is one of its axioms, confirm that α entails β in linear time (the time taken to inspect the syntactic structure of $\beta \leftarrow \alpha$ and verify that it is an axiom). (The word 'may' is important. It is not automatically the case that an algorithm that works syntactically will be more efficient than one that works semantically: Shepherdson shows in [18] that a resolution method used by Goldberg in [6] is essentially equivalent to an algorithm that constructs truth tables. But the essential point is that proof architectures are of interest solely because they offer the hope of more efficient algorithms for confirming that α entails β.)

While it is acceptable for a proof architecture to be fussy about the form of wfs, for instance to the extent of insisting that all wfs presented for inspection be in conjunctive normal form, it is less acceptable for the proof architecture to be neurotically sensitive to the extent of treating two semantically equivalent wfs (both of which are in the relevant form) entirely differently. The following example shows that SLDNF-resolution is neurotic in this sense. As a consequence, the proof architecture it provides is incomplete.

Example 3

The clauses $A \leftarrow \neg A$ and $A \leftarrow$ are semantically equivalent (recall that $A \leftarrow$ is just the clausal form of the atom A). In order to answer the question 'Is A entailed by the program $A \leftarrow \neg A$?', an SLDNF-resolution theorem-prover would dangle the goal A in front of the database $A \leftarrow \neg A$. Resolution produces the subgoal $\leftarrow \neg A$, upon which negation-as-failure demands that the goal $\leftarrow A$ be investigated again. The theorem-prover loops and therefore does not confirm that A is entailed by $A \leftarrow \neg A$.

On the other hand, given the semantically equivalent program $A \leftarrow$ and the goal $\leftarrow A$, the theorem-prover instantly produces the empty clause and halts.

The next example shows that SLDNF-resolution is unsound.

Example 4

Consider the program consisting of the single clause $A \leftarrow$ and the question 'Is $\neg B$ entailed by the program?'. To answer the question, the goal $\leftarrow \neg B$ is dangled in front of the program. Failure to succeed with procedure call B is interpreted as success with $\neg B$, delivering the empty clause. The answer, according to SLDNF-resolution, is 'Yes'. However, it is easy to see that it is possible for A (i.e. the program $A \leftarrow$) to be true while $\neg B$ is false; just take the case in which A and B are both true. Thus the atom A does not entail the atom $\neg B$.

Our examples have been drawn from propositional logic, for simplicity. The situation is the same for predicate logic, except that much more telling examples of incompleteness may be given; the reader may consult Shepherdson [19] in this regard. Essentially, the additional problems arise because application of negation-as-failure to negated atoms containing variables confuses $\forall x \neg P(x)$ with $\neg \forall x P(x)$, but restricting the application of negation-as-failure to negated atoms that don't contain variables makes the proof architecture very incomplete.

Why, in view of the defects illustrated by examples 3 and 4, would anyone persist with SLDNF-resolution as a basis for logic programming? The following arguments in support of SLDNF-resolution may be gleaned from Shepherdson [19]:

- a full theorem-prover which is sound and complete with regard to the classical entailment relation will be less efficient than a theorem-prover obtained by grafting some restricted form of negation onto "the very efficient SLD-resolution used in PROLOG";
- negation-as-failure is more compatible with the view of logic programs as databases than classical negation would be;
- soundness and completeness results for SLDNF-resolution are possible if one either replaces the classical entailment relation by a relation derived from 3-valued logic or restricts consideration to an appropriate subclass of the class of normal programs.

We consider these arguments in turn.

5 Efficient theorem-proving

The basis for the perception that SLD-resolution is very efficient is experience with smallish problems. (Theoretical analyses of efficiency will be discussed in Section 8.) Example 5 illustrates a bottom-up approach in which clauses are systematically resolved wherever possible, while example 6 illustrates the goal-directed approach characteristic of SLD-resolution.

Example 5

Consider the set of clauses (numbered for convenience of reference)

(1) $A \leftarrow$
(2) $B \leftarrow A$
(3) $C \leftarrow B$
(4) $\leftarrow C.$

Simplemindedly resolving wherever possible, we get the following steps:

(5) $B \leftarrow$ (from 1 and 2)
(6) $C \leftarrow A$ (from 2 and 3)
(7) $\leftarrow B$ (from 3 and 4)
(8) $C \leftarrow$ (from 5 and 3)
(9) $C \leftarrow$ (from 6 and 1)
(10) $\leftarrow A$ (from 6 and 4)
(11) $\leftarrow A$ (from 7 and 2)
(12) $\square$ (from 7 and 5)

Example 6

Consider the same set of clauses, but think of the first three as a program triggered by the goal clause.

(1) $A \leftarrow$
(2) $B \leftarrow A$
(3) $C \leftarrow B$
(4) $\leftarrow C$ (the goal clause)
(5) $\leftarrow B$ (from 4 and 3)
(6) $\leftarrow A$ (from 5 and 2)
(7) $\square$ (from 6 and 1)

SLD-resolution efficiently produced the empty clause in example 6 because of its use of a limited form of the *set of support strategy*: all of the algorithm's reasoning is required to be recursively traceable to the goal clause. Wos and McCune show in [23] that this limited form of the set of support strategy, impressive as it seems in example 6, is much less effective in restricting the reasoning of the theorem-prover than more general versions of the strategy if the problems are hard. Indeed, they argue convincingly that the apparent efficiency of SLD-resolution is an illusion generated by applying the method mainly to shallow problems, and that much greater power could be conferred on logic programming by taking into account the following lessons learned from experience with general theorem-provers:

- The efficient treatment of equality requires the use of paramodulation (at present logic programmers are compelled to avoid equality wherever possible).
- It is unwise to restrict the set of support to headless goal clauses. For one thing, it severely constrains the kind of query you may pose, and for another, you may need to incorporate such clauses in your database and use them during the reasoning process, which is not possible if the reasoning is steered by a headless goal clause.
- In addition to the set of support strategy, which restricts the algorithm's reasoning, one needs to use something like the weighting strategy to direct the algorithm's reasoning (essentially the user employs the weighting strategy "to impose knowledge and intuition on an automated theorem-proving program's attack on a problem"). PROLOG has control operations like cut which allow the user to prune the search tree, but such operators are limited compared with successful strategies used in automated theorem-proving.
- Whereas SLD-resolution makes no provision for providing intermediate results with more than the evanescent existence required for backtracking, "...without the retention of some intermediate information, an attempt at solving a hard problem will ordinarily lead to the pursuit of an inordinately large number of unprofitable paths" and therefore an attack on hard problems would require a departure from the usual practice of logic programming of discarding intermediate lemmata. Naturally, unrestrained accrual of intermediate information must be avoided by strategies for information purging, such as demodulation and subsumption.

In general, the claim that SLD-resolution is efficient has empirical grounds provided the problem domain is of a certain kind, i.e. contains basically shallow problems such as may be encountered in, say, standard database applications, whereas theorem-provers relying on additional strategies have demonstrated themselves to be more efficient than SLD-resolution on hard problems (a selection of which are described in [23]). The experimental evidence therefore does not unequivocally substantiate a claim that grafting negation-as-failure onto SLD-resolution will deliver a more efficient theorem-prover than could be obtained in some other way.

Shepherdson's claim was subtly different, however. He suggested only that SLDNF-resolution would be more efficient than a sound and complete theorem-prover. Here he may indeed have a point, since the use of a strategy that chops out large chunks of the search space will, in general, make the theorem-prover incomplete. Nevertheless, the possibility of a sound (though incomplete) theorem-prover that is more efficient than SLDNF-resolution (due to the former's use of powerful restrictive, directive, retentive and purgative strategies) appears to be an attainable and inviting prospect. Moreover, there are different degrees of incompleteness. SLDNF-resolution is very incomplete. In the first place, queries must be existentially quantified conjunctions of literals, so that their negations have the

form of goal clauses. Within the bounds of this severe restriction, not only are some problems too hard for it, but on some very simple problems it fails to make obvious inferences. For example, Shepherdson [19] shows that a query L_1 may fail although $L_1 \wedge L_2$ succeeds, contrary to the logical principle that L_1 may be inferred from $L_1 \wedge L_2$. Given that efficient theorem-provers are doomed to incompleteness, one can at least insist on soundness and try to ensure that the incompleteness should manifest itself on hard or irrelevant problems rather than trivial but relevant ones.

6 Deductive databases

It is easy to represent a relational database as a set of normal clauses; the records in the basic relations become facts while the desired views are encoded as more general clauses. (See Maier and Warren [12, pp. 153–160] for an elementary, and Ullman [21, ch. 3] for a more fundamental treatment.) Clausal form logic is in fact more than a relationally complete query language (since recursive clauses permit the construction of transitive closures of relations), and the use of variables allows efficient storage of information that would occupy many separate records in a relational database. Moreover, users seem to prefer, all other things being equal, the convenience of a single declarative language which integrates query language and host language. Databases therefore represent an application of logic programming that deserves serious scrutiny, particularly in order to determine whether the overhead imposed by drawing inferences nullifies the advantages. Of particular interest is the role of negative data.

In a conventional employee database or airline reservation database, say, one has far more negative data about the domain than positive data. In the relational model, if no record expresses the information that Joe Bloggs has a company car, we are entitled to deduce that Joe Bloggs does not drive a company car. This implicit representation of negative data is far more efficient than an explicit representation would be, since the majority of employees fail to qualify for a company car. How can such an implicit representation of negative data be made a part of the logic programming approach to databases? Two approaches were suggested in the landmark volume [5].

Reiter [15] suggested the *closed world assumption (cwa)*. Given a database Δ (construed as a set of normal clauses, say) and a query α, one desires an efficient algorithm that will confirm, not whether α is entailed by Δ, but whether α is entailed by $cwa(\Delta)$, where $cwa(\Delta)$ is obtained by adding to Δ all negative facts $\neg A$ where A is not entailed by Δ. Shepherdson [16] showed that negation-as-failure is sound with respect to the closed world assumption, i.e. that if SLDNF-resolution applied to Δ establishes the query α, then α is indeed entailed by $cwa(\Delta)$. So SLDNF-resolution is a candidate for the algorithm we seek.

Clark [3] suggested the *completed database*. The key idea is to read the various clauses $B \leftarrow L_1 \wedge \ldots \wedge L_m$ which share the head B as defining the relation represented by B in terms of the relations represented by $L_1, \ldots$ (In predicate logic, of course, the atoms involve predicate symbols which may be thought of as names for relations, so the idea makes sense.) Now the sad fact is that, in mathematics as often as in everyday life, people tend to use 'if' when they mean 'if and only if'. Look at the definitions in your favourite calculus text, or consider whether the promise 'I'll buy you an ice cream if you behave' is intended any differently from 'I'll buy you an ice cream iff you behave'. Clark therefore suggests rewriting the definitions of relations that make up the program Δ as 'iff' definitions, thus delivering the completed database $comp(\Delta)$. Given a query α, we no longer seek an efficient algorithm that will confirm whether α is entailed by Δ but one to confirm whether α is entailed by the completed database $comp(\Delta)$. Clark showed that SLDNF-resolution is sound with respect to the completed database, i.e. if SLDNF-resolution applied to Δ establishes the query α then α is indeed entailed by $comp(\Delta)$. So SLDNF-resolution becomes a candidate for the algorithm sought under this approach also.

In the context of deductive databases, the objectionable unsoundness of SLDNF-resolution may be argued away by adopting either of the two approaches outlined above. The incompleteness of the method remains of a high order. By way of illustration, consider the following examples, based on Shepherdson [19], which show that $cwa(\Delta)$ and $comp(\Delta)$ can be very different, and reflect on the fact that SLDNF-resolution is sound with regard to both of them.

Example 7

If Δ consists of the single clause $B \leftarrow \neg B$ then $cwa(\Delta)$ contains no additional literals, because B is entailed by $B \vee B$, i.e. by $B \leftarrow \neg B$. Hence $cwa(\Delta)$ is consistent: giving the atom B the truth value 'true' makes all the wfs in $cwa(\Delta)$ true. However $comp(\Delta)$ is inconsistent, since no assignment of a truth value to B can make $B \leftrightarrow \neg B$ true.

Example 8

If Δ consists of the single clause $B \leftarrow \neg A$, then $comp(\Delta)$ is consistent, since the assignment of the value 'true' to B and 'false' to A makes $B \leftrightarrow \neg A$ true. However, $cwa(\Delta)$ is inconsistent, since neither A nor B is entailed by $A \vee B$, i.e. by $B \leftarrow \neg A$, and so $cwa(\Delta)$ contains not only the clause $B \leftarrow \neg A$ expressing the disjunction of the two atoms but also the negative literals $\neg A$ and $\neg B$.

Example 9

Suppose Δ consists of the three clauses

$$\begin{aligned} B &\leftarrow A \\ A &\leftarrow \neg B \\ A &\leftarrow A \end{aligned}$$

Then $cwa(\Delta)$ and $comp(\Delta)$ are both consistent but their union is not. To see this, note that B is entailed by Δ whereas A is not, so $cwa(\Delta)$ adds $\neg A$ to Δ, and $cwa(\Delta)$ can therefore only be satisfied by assigning truth values 'true' to B and 'false' to A. $comp(\Delta)$ on the other hand consists of the wfs $B \leftrightarrow A$ and

$A \leftrightarrow A \vee \neg B$. (The definition of A consists of two clauses with A as head, and before changing 'if' to 'iff', the disjunction of the bodies must be formed.) $comp(\Delta)$ can therefore only be satisfied by assigning 'true' to both A and B. Hence the union of $cwa(\Delta)$ and $comp(\Delta)$ cannot be satisfied by any assignment of truth values, i.e. is inconsistent.

We have seen that if a wf is derived from a database Δ by SLDNF-resolution then the wf is entailed by $cwa(\Delta)$ and by $comp(\Delta)$. Unfortunately, many wfs entailed by $cwa(\Delta)$ or $comp(\Delta)$ cannot be confirmed by SLDNF-resolution to be such. One obvious reason is that SLDNF-resolution is driven by goal clauses, so queries are limited to existentially quantified conjunctions of literals. But this suggests that perhaps SLDNF-resolution may be complete in the restricted sense that, whenever an existentially quantified conjunction of literals is entailed by $cwa(\Delta)$ or by $comp(\Delta)$, then SLDNF-resolution can confirm it. Unfortunately matters are not so simple.

7 Completeness relative to $cwa(\Delta)$ and $comp(\Delta)$

Let us first note a negative result relating to $cwa(\Delta)$. If one wishes to apply predicate logic via logic programming to such fields as group theory, then it is desirable to incorporate function symbols like '+' into the formal language. This has the consequence of making the Herbrand universe associated with the language infinite. (The Herbrand universe is the set of all variable-free terms of the language, and if it is finite then the predicate language can be made to behave much like a propositional language; if not, not.) As a result one can show no algorithm exists that will accept an arbitrary database Δ and atom A as input and decide whether A is a consequence of Δ or not (Apt, Blair and Walker [1] and Shepherdson [19]). (Algorithms that are guaranteed to confirm that an atom A is entailed by Δ if it is indeed entailed, do exist; the problem is that the algorithm may fail to terminate when given as input an atom that is not entailed by Δ.) But a theorem-proving algorithm wishing to confirm that a wf is a consequence of $cwa(\Delta)$ must first know what $cwa(\Delta)$ consists of. So even a watered-down algorithm that seeks only to confirm those consequences of $cwa(\Delta)$ that are queries cannot guarantee to do so.

Matters improve if we turn to $comp(\Delta)$. Several completeness results are proved in Clark [3] and Shepherdson [17] for various classes of queries. The conditions imposed on the queries in these papers are often not easily testable (particularly the conditions of having the finite tree property and of being allowable). The most useful result is that if we restrict ourselves to programs Δ that are allowed and satisfy the hierarchical condition, then SLDNF-resolution is complete for weakly allowed queries entailed by $comp(\Delta)$.

A program is said to satisfy the hierarchical condition if its predicate symbols can be assigned to levels so that in each clause

$$B \leftarrow L_1 \wedge \ldots \wedge L_m$$

about the predicate symbol in B, the predicate symbols occurring in $L_1 \wedge \ldots \wedge L_m$ are of level less than the predicate symbol in B. The idea is that such clauses describe how a relation (denoted by the predicate symbol in the head of the clause) may be built up from other relations (those denoted by predicate symbols mentioned in the body), and these building blocks must have their existence established before they are employed to construct new relations. But this is of course very restrictive, since it forbids any kind of recursive definition.

A program is *allowed* if every variable occurring somewhere in a clause also occurs in an unnegated atom of the body of the clause. To see the sense in this restriction, let us first look at the notion of a *weakly allowed* query. Because $\neg\forall x P(x)$ is not equivalent to $\forall x \neg P(x)$, and because SLDNF-resolution theorem-provers cannot distinguish between the two (the quantifier being invisible), soundness with respect to $comp(\Delta)$ requires that negation-as-failure be applied only to negative literals which are variable-free. The literal need not originally have been variable-free; it may be that the process of matching (called unification) earlier in the sequence of resolution steps caused the variables to be replaced by constants. But given a query like $\exists x(\neg P(x) \wedge Q(y))$, which becomes the goal clause $\leftarrow \neg P(x) \wedge Q(y)$, there is no way to replace x by a constant; even if we try to 'execute procedure call' $Q(y)$ first, the variable x does not appear in $Q(y)$ and so is unaffected by any unification that ensues. So such a query cannot succeed, i.e. there is no way to get rid of $\neg P(x)$. Hence we define a query to be weakly allowed if every variable that occurs in a negative literal also occurs in a positive literal (i.e. in an unnegated atom). The restriction that programs be 'allowed' can now be seen to have a specific purpose: in an allowed program, clauses of the form $P(x) \leftarrow$ cannot appear. If facts containing variables were to appear in the program, then a weakly allowed query like $\leftarrow P(x) \wedge \neg Q(x)$ would "lose its cover" and go into $\neg Q(x)$, whereupon the theorem-prover would flounder, i.e. be unable to proceed.

Attempts to generalise the completeness result by weakening the hypotheses have naturally followed. Perhaps the greatest interest has been generated by a weakening of the hierarchical condition to the notion of a *stratified program* by Apt, Blair and Walker [1]. In a stratified program, a clause

$$B \leftarrow L_1 \wedge \ldots \wedge L_m$$

defining the predicate symbol in B is permitted to contain in its body predicate symbols of the same level as the predicate symbol in B, provided they occur in unnegated atoms of the body. Predicate symbols that occur in negative literals of the body must belong to a level lower than the predicate symbol in B. The basic idea is again simple. The existence of a relation must be established before its complement can used as a building block. Recursive definitions become possible, although not recursion through

negation. Unfortunately, the completeness result does not extend to stratified programs, at least not without the additional restrictions mentioned in Cavedon and Lloyd [2].

An interesting attempt to derive soundness and completeness results from a different angle of attack is made by Kunen [9]. Using a 3-valued logic devised by Kleene, Kunen states that SLDNF-resolution is sound with respect to $comp(\Delta)$ in the sense that if a query succeeds (i.e. is established via SLDNF-resolution) then it is true in all 3-valued models of $comp(\Delta)$. (When we say that a wf β is true in all 3-valued models of a wf α, we mean that if we assign truth values according to the rules of the logic and α gets the value 'true', so must β.) Moreover, for propositional programs Δ, SLDNF-resolution is complete in the sense that any query (i.e. conjunction of literals) entailed by $comp(\Delta)$ (i.e. true in all 3-valued models of $comp(\Delta)$) will succeed (i.e. trigger SLDNF-resolution to produce the empty clause). This suggests that perhaps the correct underlying logic for SLDNF-resolution is Kleene's 3-valued logic. What is also interesting is the light cast on the degree of incompleteness of SLDNF-resolution; the entailment relation under consideration is much narrower than the usual entailment relation of classical logic. To see this, simply note that every allocation of the values 'true' and 'false' to atoms that is acceptable according to the rules of classical 2-valued logic is also acceptable according to 3-valued logic (i.e. 2-valued models are 3-valued models), but any allocation of truth values to atoms employing the intermediate truth value takes us outside 2-valued logic (i.e. those 3-valued models that take seriously the possibility of computations that do not return an answer are not 2-valued models). So to ask that a wf β be true in all 3-valued models of a wf α is to ask more of β than just that it be true in all 2-valued models of α.

8 Conclusions: the role of archetypical problems

We have seen that from two perspectives SLDNF-resolution is a disappointment to the classical logician: it does not mimic the usual entailment relation of classical logic, and it does not have the power to tackle deep problems. On the other hand, SLDNF-resolution seems suited to database applications, being sound with respect to both $cwa(\Delta)$ and $comp(\Delta)$ and having limited completeness with respect to $comp(\Delta)$. For propositional programs, SLDNF-resolution represents logic programming based on a non-classical system of logic, namely Kleene's 3-valued system. In this sense it fits into the picture, suggested by the logic programming ideal, of an array of logic languages, based on both classical and non-classical systems of logic. Two questions offer themselves. What are the implications of its non-classical semantics for teaching logic programming, and can a language based on classical logic be feasibly implemented (where by 'feasibly' we mean 'with enough power to solve non-trivial problems')?

For the time being, teaching logic programming will remain synonymous with teaching PROLOG programming. But recognition of the non-classical nature of the logic reflected by PROLOG suggests that the introductory texts which introduce PROLOG programming by looking at classical propositional and predicate logic may be doing the novice a disservice by furnishing an inappropriate paradigm. Some familiarity with 3-valued logic may serve much better, while incidentally also being relevant for parallel languages (as shown in [14]).

Another mistake is the suggestion in some texts that PROLOG attains (or that some future implementation will attain) what might be called the *declarative ideal*. This is the belief, surely quite amusing in its naivety, that "a program should simply be the statement of the problem. The way the problem is solved and the sequence of instructions that the computer must go through to solve it, are decided by the system", McDonald and Yazdani [13]. The fixed sequencing built into PROLOG bears little resemblance to this description, which in any case is unrealistic. Three thousand years of recorded problem solving by humans establishes the simple fact that finding 'the' right way to approach a non-routine problem is the hardest part. The reasoning involved may be vague and depend on insights that are specific to the problem domain. From expert systems we learn the lesson that problem-solvers which do not provide the user with plenty of opportunity to transfer domain-specific insight to the program simply won't fly. The strategy built into PROLOG, coupled with its primitive pruning operator 'cut', is pretty good for, say, deductive database applications. Why not replace sweeping claims by an emphasis on the domain-dependence of successful strategies?

The experimental evidence acquired in the field of automated theorem-proving suggests that it ought to be possible to devise a programming language based on classical logic (actually a language-metalanguage pair in which the former suffices for descriptions of problems while the metalanguage component allows the imposition of strategies). Although the underlying theorem-prover is unlikely to be complete, it can certainly be made sound and the incompleteness need not be of the high degree characterising SLDNF-resolution. Indeed, if the incompleteness is caused by the imposition of strategies geared to a particular problem domain, then it may be that the incompleteness manifests itself increasingly as problems become more remote from the intended domain.

This view of inefficiency getting progressively worse as the distance from some archetypical problem increases (but being tolerable for precisely that reason), is alien to conventional complexity theory. In view of the requirement that the proof procedure forming the basis of a logic programming language should be feasible, and the fact that feasibility is usually interpreted in terms of complexity theory, it is necessary to bring the relevance (or rather, irrelevance) of complexity theory into perspective. Complexity theory aims to classify decidable problems according to degrees of 'hardness'. The classification is achieved by reference to the existence of algorithms (decision procedures) of one or another order of complexity. In most cases, the complexity of the algorithms is determined by a worst case

analysis. (In a few cases, enough information about the instances of the decision problem is available to permit the construction of a plausible probability measure in terms of which an average case analysis may be performed. But this is exceptional.)

A worst case analysis only makes sense for decision procedures, i.e. for algorithms that are guaranteed to terminate whatever their input. For propositional logic, the problem whether a wf α entails β is decidable, the construction of a truth table for $\beta \leftarrow \alpha$ being the essence of the most familiar algorithm for deciding the question. The algorithm terminates no matter what wfs α and β we may choose, and in the worst case 2^n steps are required (where α and β between them contain n atoms), since this represents the case in which the whole truth table has to be constructed. For predicate logic, the problem whether α entails β is known not to be decidable except in very special cases such as Presburger arithmetic (Enderton [4]). Theorem-proving algorithms for predicate logic therefore aspire to be not decision procedures but proof procedures, i.e. can at best guarantee that if α does entail β, then they will terminate and return confirmation of entailment. In the worst case, the input will be such that the algorithm does not terminate, suggesting that the algorithm is infinitely inefficient. This is of course quite misleading, since such proof procedures can be very efficient on particular kinds of problem.

May one not restrict the worst case analysis to the set of inputs on which the algorithm will terminate? In the case of predicate logic, after all, such inputs can be characterised semantically, so that one can avoid the circularity of describing the set of inputs on which the algorithm terminates as the set of inputs on which the algorithm terminates. However, bear in mind that undecidability means inter alia that there is no algorithm to decide whether an input is acceptable in the sense that a proof procedure will terminate on it. If a worst case analysis restricted to the set of acceptable inputs were possible, an upper bound on the length of the computation required for a given input would be found, expressed in terms of the length of the input. But this would convert the proof procedure to a decision procedure; given any input, if the upper bound on the length of the computation is passed the algorithm may terminate with the message 'No, β is not entailed by α'. Hence no such analysis is possible.

If the edifice of complexity theory does not provide a suitable arena for the theoretical analysis of the feasibility of proof procedures, what does? Two examples from practice suggest an alternative approach on which it may be possible to base a useful theoretical framework for the analysis of algorithms that do not necessarily terminate, including proof procedures. Consider the simple problem of solving a system of linear equations. As is well known, Gaussian elimination is a pretty good direct method. On the other hand, if the coefficient matrix is sparse enough and the zeros don't form a useful pattern, iterative methods are more efficient. The numerical mathematician brings to bear her insight in deciding which method is appropriate for a given system. In some vague way, this seems to involve judging how far the given system is from being ideally suited for the one method or the other, using for the purpose experience gained in the particular field. Similarly, experience gained in a particular field can be used, via the weighting strategy described in Wos *et al* [24], to help a theorem-prover to decide where to focus its attention. The weights assigned to predicates and terms on occasions when the problems are similar would also tend to be similar. The examples suggest that, in practice, informed users go through a (mostly unsystematic) process of comparing a candidate problem to archetypical problems (for which there is some proven approach) and applying the approach if the candidate seems close enough in structure to the archetypical problems. Perhaps this process can be formalised and a geography gradually established in which the archetypical problems constitute landmarks and some metric gives the distance from a candidate problem to its nearest landmarks.

References

1. K R Apt, H A Blair, and A Walker. 'Towards a theory of declarative knowledge'. In J Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, (1988).
2. L Cavedon and J W Lloyd. 'Completeness results for SLDNF-resolution'. *J. Logic Programming*, **7**:177–191, (1988).
3. K L Clark. *Negation as failure*, pp. 293–322. In Gallaire and Minker [5], 1978.
4. H B Enderton. *A Mathematical Introduction to Logic*. Academic Press, London, 1972.
5. H Gallaire and J Minker, eds. *Logic and Databases*. Plenum Press, New York, 1978.
6. A Goldberg. 'Average complexity of the satisfiability problem'. In W S Joyner, ed., *Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas*, pp. 1–6, (1979).
7. A G Hamilton. *Logic for Mathematicians*. Cambridge University Press, revised edition, 1988.
8. R A Kowalski. 'Predicate logic as programming language'. In *Information Processing 74*, pp. 569–574. North-Holland, (1974).
9. K Kunen. 'Negation in logic programming'. *J. Logic Programming*, **4**:289–308, (1987).
10. H R Lewis and C H Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New York, 1981.
11. J W Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd extended edition, 1987.
12. D Maier and D S Warren. *Computing with Logic*. Benjamin/Cummings, Menlo Park, California, 1988.
13. C McDonald and M Yazdani. *Prolog Programming*. Blackwell, Oxford, 1990.
14. S W Postma and N C K Phillips. 'The parallel conditional'. *Quaestiones Informaticae*, **6**(3):109–112, (1988).
15. R Reiter. *On closed world data bases*, pp. 55–76. In

Gallaire and Minker [5], 1978.
16. J C Shepherdson. 'Negation as failure: a comparison of Clark's completed database and Reiter's closed world assumption'. *J. Logic Programming*, **1**:51–81, (1984).
17. J C Shepherdson. 'Negation as failure II'. *J. Logic Programming*, **2**:185–202, (1985).
18. J C Shepherdson. 'Introduction to the theory of logic programming'. In F R Drake and J K Truss, eds., *Logic Colloquium '86*, pp. 277–318. North-Holland, Amsterdam, (1988).
19. J C Shepherdson. 'Negation in logic programming'. In J Minker, ed., *Foundations of Deductive Databases and Logic Programming*, pp. 19–88. Morgan Kaufmann, (1988).
20. G Sundholm. 'Systems of deduction'. In D Gabbay and F Guenther, eds., *Handbook of Philosophical Logic*, pp. 133–188. D Reidel, Dordrecht, (1983).
21. J D Ullman. *Principles of Database and Knowledge-base Systems*, volume I. Computer Science Press, 1988.
22. J van Benthem. *Language in Action (Categories, Lambdas and Dynamic Logic)*. North-Holland, 1991.
23. L Wos and W McCune. 'Automated theorem proving and logic programming: a natural symbiosis'. *J. Logic Programming*, **11**:1–53, (1991).
24. L Wos, R Overbeek, E Lusk, and J Boyle. *Automated Reasoning*. Prentice-Hall, 1984.

Book Reviews

Data Structures: An Object-Oriented Approach, by *William J Collins*, Addison Wesley, 1992, 624pp, ISBN 0-201-56953-1.
Reviewer: Professor Judy M Bishop, University of Pretoria.

A lecturer in search of a data structures text book at about second year level these days probably has two concerns. The first is that the book should support a movable infrastructure. The language in which Computer Science 2 is being offered is under question in most departments, and in many it is hotly debated. Thus if possible the book should not be tied too tightly to any one language. In practice this means that it should support a pseudo-code, which brings its own disadvantages of extra layers of, possibly ill-defined, notation. The second concern when choosing a book is that it should reflect an object-oriented approach. This is now essential: students and the marketplace demand it, even if lecturers are still a bit bemused by all the hoopla.

Under both these criteria, Collins' book scores well. It uses pseudo-code for all the high-level design of data structures, but comes down to earth and Turbo Pascal for the implementation, and it is firmly based in objects, which are nicely supported by the later versions of Turbo Pascal used. To my mind, this text book is a serious contender to the established market, and deserves a close look.

The overall strategy is to consider each classic data structure in turn, from lists through to graphs. The data type is defined in terms of the relationship between its elements and the abstract operations which it offers. These are presented as method names with incoming and outgoing parameters, followed by the pre- and post-conditions associated with the method, and a short example of its use in practice. Thereafter, alternative designs of the definition are presented, most commonly a linked and a contiguous design. Here, each method is expanded with a subalgorithm presented in free-form Pascal, a verification of its correctness phrased as a reasoning in English, and an analysis of its performance if appropriate. Diagrams are used to support each design.

Only after the designs are given, are the actual data representations and method implementations discussed. Here we see the advantage of the earlier design phase: a linked design can be represented by an array or pointer implementation. The separation of concerns between specification, design and implementation works very well in Collins' hands, and is likely to be helpful to students as they come to grips with the myriad different possibilities.

The implementations are given in Turbo Pascal 6, and make good use of procedure types for achieving the abstraction of those methods that require visiting every node. However, the book could quite easily and profitably be used in a course which is teaching Ada or C++ as a second language. Students would then be encouraged to develop the implementations themselves, based on a firm design and an example in another (known) language.

The data structures and algorithms covered include lists, ordered lists, stacks, queues, binary trees, heaps, sorting, searching, tables, graphs. Each data structure is given a full treatment, and I did not find any important algorithm or technique of my acquaintance that had been omitted. All the sorts are there, all the chaining methods, the tree implementations and the graph traversals.

The book is amply supported by appendices on the predefined types such as files and strings, but does not go into the niceties of integers, reals, characters and booleans. I found this a refreshing omission. On the other hand, the Turbo Pascal assistance is considerable, and the book actually starts with a full chapter on the more advanced features of Turbo that will be needed, and might not have been stressed in an introductory course. These include pointers, units and objects. The next two chapters wisely revise recursion and software engineering: once again topics that might have had a less than thorough treatment in first year. The book ends with a full chapter on Turbo Vision, creating a sort of Turbo-sandwich which is both satisfying and attractive.

The examples used throughout are apt and interesting. Some are classic (expression evaluation) and some

are novel (simulating queues at Bob's checkout counter). Every chapter is followed by extensive problems in three categories, ranging from simple paper work exercises to full blown programming assignments. The wealth and range of these is impressive. Answers to some are given at the back of the book.

All in all, this is a most interesting book, coming at a time of great change in the market. I have prescribed it for my second years next year, and look forward to using it. I can certainly recommend it for any second year course on data structures and algorithms, and also as a handy book to have on one's shelf for look-up purposes.

Computer Architecture: A Quantitative Approach, by *John L. Hennessy and David A. Patterson*, Morgan Kaufmann Publishers Inc., 594pp + appendices, ISBN 1-555880-069-8.
Reviewer: Associate Professor Henk A Goosen, University of Cape Town.

"The authors have gone beyond the contributions of Thomas to Calculus and Samuelson to Economics. They have provided the definitive text and reference for computer architecture *and design.*" So writes Gordon Bell in the foreword to this very readable book.

For the first time there is now a book on computer architecture that goes beyond mere descriptions of instruction sets and hardware structures, and instead provides a quantitative basis for the design of computer systems. The central message of the book is that the execution time of a real workload is the only true measure of computer performance. Indirect measures can be used to provide insight and to analyze performance bottlenecks, but the final judgment on the performance has to be based on the execution time of a realistic workload.

The authors were two of the pioneers of RISC architectures in the early 1980s: Hennessy leading the design of the MIPS processor at Stanford University, and Patterson leading the design of RISC-I and RISC-II at the University of California, Berkeley. The fact that the authors are two of the leading practising computer architects adds immensely to the entertainment value of the book, and means that much of the most recent advances in computer architecture is covered.

The first two chapters in the book introduce the quantitative principles of computer design, including speeding up the common case, Amdahl's Law, and locality of reference. Ways of measuring and comparing performance are explained, and the costs of computer components, and how they influence the final price of the machine, are discussed.

The rest of the book covers instruction set design, processor implementation techniques (including pipelining and vector processors), memory hierarchy design, and the design of input/output systems. Illustrative examples of architectures used in the book include the 8086, IBM 360, VAX, and DLX (pronounced "Deluxe" a canonical RISC architecture).

A major theme of the implementation chapters is the extent to which instruction set design impacts the implementation, and specifically how complex and long-running instructions complicate the design of a pipelined processor.

The chapter on pipelining includes sections on advanced topics like scoreboarding, Tomasulo's algorithm, superscalar and VLIW machines, and software techniques for increasing instruction-level parallelism (for example, loop unrolling and trace scheduling), while the memory hierarchy chapter explains the latest thinking about multilevel cache structures and other ways of reducing cache miss penalties.

In addition, there is an appendix on computer arithmetic, written by David Goldberg of the Xerox Palo Alto Research Center. This chapter includes a thorough discussion of the IEEE floating point standard.

The organization of the book is unique: each chapter has a "putting it all together" section, which applies the material in that chapter to a realistic example, as well as a "fallacies and pitfalls" section, pointing out some of the frequent traps and commonly held misbeliefs. Each chapter concludes with a section on historical perspectives and references.

The book is suitable for an intermediate or advanced computer architecture course, and will make an excellent self study book. Readers familiar with Knuth's books will appreciate the "difficulty-graded" exercises, and an instructor's manual is available from the publishers. Software tools, including a trace-driven cache simulator, a DLX simulator, address traces, and benchmark programs are also available from the authors. A final plus for the classroom is the use of non-sexist language throughout the book. The book is currently used as textbook in many North American universities and locally at least at UCT and Wits, and is well debugged.

Book Reviewers Wanted

The following books are available for review. Potential reviewers who would like to review any of these books should contact the editor. A book that has been reviewed may be retained by the reviewer.

- P Chaudhuri, *Parallel Algorithms: Design and Analysis*, Prentice-Hall, 1992.
- W M Collier, *Reasoning About Parallel Architecture*, Prentice-Hall, 1992.
- F J Dodd, *Computerised Accounting Systems*, NCC Blackwell, 1992.
- R Goody, *Programming and Interfacing the 8086/8088 Microprocessor*, Maxwell Macmillan, 1992.
- G M Karam and J C Bryant, *Principles of Computer Systems*, Prentice-Hall, 1992.
- D M Kroenke, *Database Processing, Fourth Edition*, Maxwell Macmillan International Editions, 1992.
- N S Matloff, *IBM Microcomputer Architecture and Assembly Language: A Look Under the Hood*, Prentice-Hall, 1992.

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 8, November 1992
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 8, November 1992
ISSN 1015-7999

Contents

GUEST CONTRIBUTION

The Ideology, Struggle and Liberation of Information Systems
JD Roode 1

Editor's Notes 2

RESEARCH ARTICLES

Selection Criteria for First year Computer Science Students
AP Calitz, GdeV de Kock, and DJL Venter 4

A New Algorithm for Finding an Upper Bound of the Genus of a Graph
DI Carson and OR Oellermann 12

A Model Checker for Transition Systems
PJA de Villiers 24

Beam Search in Attribute-based Concept Induction
H Theron and I Cloete 32

PEW: A Tool for the Automated Performance Analysis of Protocols
G Wheeler and PS Kritzinger 37

Evaluating the Motivating Environment for Information Systems Personnel in South Africa Compared to the United States (Part II)
JD Couger and DC Smith 46

An Evaluation of the Skill Requirements of Entry-level Graduates in the Information Systems Industry
DC Smith, S Newton, and MJ Riley 52

COMMUNICATIONS AND REPORTS

Social Responsibility for Computing Professionals and Students
MC Clarke 63

Qualitative Reasoning: An Introduction
J Bradshaw 70

Logic Programming: Ideal vs. Practice
WA Labuschagne and PL van der Westhuizen 77

Book Reviews 86

Sigma Press