Computational Logic: Memories of the Past and Challenges for the Future

John Alan Robinson

Highland Institute, 96 Highland Avenue, Greenfield, MA 01301, USA.

Abstract. The development of computational logic since the introduction of Frege's modern logic in 1879 is presented in some detail. The rapid growth of the field and its proliferation into a wide variety of subfields is noted and is attributed to a proliferation of subject matter rather than to a proliferation of logic itself. Logic is stable and universal, and is identified with classical first order logic. Other logics are here considered to be first order theories, syntactically sugared in notationally convenient forms. From this point of view higher order logic is essentially first order set theory. The paper ends by presenting several challenging problems which the computational logic community now faces and whose solution will shape the future of the field.

1 Introduction

Although logic and computing are each very old subjects, going back very much more than a mere hundred years, it is only during the past century that they have merged and become essentially one subject. As the logican Quine wrote in 1960, "The basic notions of proof theory converge with those of machine computing. ... The utterly pure theory of mathematical proof and the utterly technological theory of machine computation are thus at bottom one, and the basic insights of each are henceforth insights of the other" ([1], p. 41).

The aptly-named subject of computational logic - no pun intended - subsumes a wide variety of interests and research activities. All have something to do with both logic and computation. Indeed, as the declaration of scope of the new ACM journal Transactions on Computational Logic says, computational logic reflects all uses of logic in computer science. If things are no longer as simple as they seemed to be when logic and computing first came together, it is not because logic has become more complex, but because computing has.

Computational logic seemed simple enough when first order logic and electronic computing first met, in the 1950s. There was just one thing to know: the proof procedure for first order logic. It was clear that now at last it could be implemented on the fantastic new computers that were just becoming available. The proof procedure itself had of course been invented a generation earlier in 1930. It had waited patiently in the literature, supported by beautiful correctness and completeness proofs, for twenty five years until the first electronic digital computers were made available for general research. Then, in the first wave of

J. Lloyd et al. (Eds.): CL 2000, LNAI 1861, pp. 1-24, 2000.

euphoria, it seemed possible that there would soon be proof-finding software which could serve as a computational assistant for mathematicians, in their regular work. Even though that still has not quite happened, there has been remarkable progress. Such automated deduction aids are now in use in developing and verifying programs.

Our small niche within general computer science was at first known simply as "mechanical theorem proving". It really was a small niche. In 1960 only a handful of researchers were working seriously on mechanical theorem proving. Theirs was a minority interest, which hardly counted as "real" computing. The numerical analysts, coding theorists, electronic engineers and circuit designers who ruled mainstream computer conferences and publications tended to be suspicious of our work. It was not so easy to find funding or places to publish. Most people in the computing mainstream considered logic to be boolean algebra, and logical computations to be what you had to do in order to optimize the design of digital switching circuits.

A half century later, our small niche has grown into a busy international federation of associations of groups of small niches housing thousands of researchers. At meetings like the present one the topics cover a broad range of theory and applications reaching into virtually every corner of computer science and software engineering. Such fertile proliferation is gratifying, but brings with it not a little bemusement. No one person can hope any more to follow closely everything going on in the whole field. Each of us now knows more and more about less and less, as the saying goes. Yet there is, after all, a unity under all this diversity: logic itself. We all speak essentially the same language, even if we use many different dialects and a variety of accents. It is just that nowadays logic is used to talk about so many different things.

1.1 First Order Predicate Calculus: All the Logic We Have and All the Logic We Need

By logic I mean the ideas and notations comprising the classical first order predicate calculus with equality (FOL for short). FOL is all the logic we have and all the logic we need. It was certainly all the logic Gödel needed to present all of general set theory, defining in the process the "higher order" notions of function, infinite cardinals and ordinals, and so on, in his classic monograph on the consistency of the axiom of choice and the continuum hypothesis with the other axioms of set theory [2]. Within FOL we are completely free to postulate, by formulating suitably axiomatized first order theories, whatever more exotic constructions we may wish to contemplate in our ontology, or to limit ourselves to more parsimonious means of inference than the full classical repertoire. The first order theory of combinators, for example, provides the semantics of the lambda abstraction notation, which is thus available as syntactic sugar for a deeper, first-order definable, conceptual device. Thus FOL can be used to set up, as first order theories, the many "other logics" such as modal logic, higher order logic, temporal logic, dynamic logic, concurrency logic, epistemic logic, nonmonotonic logic, relevance logic, linear logic, fuzzy logic, intuitionistic logic, causal logic, quantum

logic; and so on and so on. The idea that FOL is just one among many "other logics" is an unfortunate source of confusion and apparent complexity. The "other logics" are simply notations reflecting syntactically sugared definitions of notions or limitations which can be formalized within FOL. There are certain universal reasoning patterns, based on the way that our minds actually work, and these are captured in FOL. In any given use it is simply a matter of formulating suitable axioms and definitions (as Gödel did in his monograph) which single out the subject matter to be dealt with and provide the notions to deal with it. The whole sprawling modern landscape of computational logic is simply the result of using this one flexible, universal formal language FOL, in essentially the same way, to talk about a host of different subject matters. All those “other logics”, including higher-order logic, are thus theories formulated, like general set theory and indeed all of mathematics, within FOL.

There are, of course, many different ways to present FOL. The sequent calculi, natural deduction systems, resolution-based clausal calculi, tableaux systems, Hilbert-style formulations, and so on, might make it seem that there are many logics, instead of just one, but these differences are just a matter of style, convenience and perspicuity.

2 Hilbert’s 1900 Address

On this occasion it is natural to think of David Hilbert’s famous 1900 Paris address to the International Congress of Mathematicians [3]. When he looked ahead at the future of his field, he actually determined, at least partially, what that future would be. His list of twenty three leading open problems in mathematics focussed the profession’s attention and steered its efforts towards their solutions. Today, in wondering about the future of computational logic we have a much younger, and much more untidy and turbulent subject to deal with. It would be very surprising if we could identify a comparable set of neat open technical problems which would represent where computational logic should be trying to go next. We have no Hilbert to tell us what to do next (nor, these days, does mathematics). In any case computational logic is far less focussed than was mathematics in 1900. So instead of trying to come up with a Hilbertian list of open problems, we will do two things: first we will reflect on the historical developments which have brought computational logic this far, and then we will look at a representative sampling of opinion from leaders of the field, who kindly responded to an invitation to say what they thought the main areas of interest will, or should, be in its future.

3 The Long Reign of the Syllogistic Logic

In the long history of logic we can discern only one dominating idea before 1879, and it was due to Aristotle. People had certainly begun to reason verbally, before Aristotle, about many things, from physical nature, politics, and law to metaphysics and aesthetics. For example, we have a few written fragments, composed about 500 B.C, of an obviously detailed and carefully reasoned

philosophical position by Parmenides. Such thinkers surely used logic, but if they had any thoughts about logic, we now have no written record of their ideas. The first written logical ideas to have survived are those of Aristotle, who was active about 350 B.C. Aristotle's analysis of the class of inference patterns known as *syllogisms* was the main, and indeed the only, logical paradigm for well over two millennia. It became a fundamental part of education and culture along with arithmetic and geometry. The syllogism is still alive in our own day, being a proper and valid part of basic logic. Valid forms of inference remain valid, just as 2 plus 2 remains 4. However, the view that the syllogistic paradigm exhausts all possible logical forms of reasoning was finally abandoned in 1879, when it was subsumed by Frege's vastly more general and powerful modern scheme.

Before Frege, there had also been a few ingenious but (it seems today) hopeless attempts to automate deductive reasoning. Leibniz and Pascal saw that syllogistic analysis could be done by machines, analogous to the arithmetical machines which were feasible even in the limited technology of their time. Leibniz even dreamed of fully automatic deduction engines like those of today. Even now we have still not quite completed his dream – we cannot yet settle any and all disputes merely by powering up our laptops and saying to each other: calculemus. We still have to do the difficult job of formalizing the dispute as a proof problem in FOL. It may not be long, however, before computational conflict resolution turns out to be one more successful application of computational logic.

As modern mathematics grew in sophistication, the limitations of the syllogistic framework became more and more obvious. From the eighteenth century onwards mathematicians were routinely making proofs whose deductive patterns fell outside the Aristotelian framework. Nineteenth-century logicians such as Boole, Schroeder, de Morgan, Jevons, and Peirce tried to expand the repertoire of logical analysis accordingly, but with only partial success. As the end of the nineteenth century approached, it was clear that a satisfactory general logical theory was needed that would cover all possible logical inferences. Such a theory finally arrived, in 1879.

4 Frege's Thought Notation

The new logic was Frege's *Begriffschrifft*, alias (in a weird graphical notation) FOL ([4], pp. 5 - 82). One can reasonably compare the historical significance of its arrival on the scientific scene with that of the integral and differential calculus. It opened up an entirely new world of possibilities by identifying and modelling the way the mind works when reasoning deductively. Frege's German name for FOL means something like *Thought Notation*.

Frege's breakthough was followed by fifty years of brilliant exploration of its strengths and limitations by many mathematicians. The heroic efforts of Russell and Whitehead (following the lead of Peano, Dedekind, Cantor, Zermelo, and Frege himself) to express all of mathematics in the new notation showed that it could in principle be done. In the course of this investigation, new kinds of foundational problems appeared. The new logical tool had made it possible not

only to detect and expose these foundational problems, but also to analyze them and fix them.

It soon became evident that the syntactic concepts comprising the notation for predicates and the rules governing inference forms needed to be supplemented by appropriate *semantic* ideas. Frege himself had discussed this, but not in a mathematically useful way. Russell's theory of types was also an attempt to get at the basic semantic issues. The most natural, intuitive and fruitful approach turned out to be the axiomatization, in the predicate calculus notation, of the concept of a set. Once this was done the way was cleared for a proper semantic foundation to be given to FOL itself. The rigorous mathematical concept of an interpretation, and of the denotation of a formula within an interpretation, was introduced by Alfred Tarski, who thus completed, in 1929, the fundamentals of the semantics of FOL ([5], p. 277). From the beginning FOL had come with the syntactic property, ascribable to sentences, of being formally derivable. Now it was possible to add to it the semantic properties, ascribable to sentences, of being true (in a given interpretation) and logically true (true in all interpretations). The notion of logical consequence could now be defined by saying that a sentence S logically follows from a sentence P if there is no interpretation in which P is true and S is false.

5 Completeness: Herbrand and Gödel

This now made it possible to raise the following fundamental question about FOL: is the following Completeness Theorem provable: for all sentences S and P, S is formally derivable from P if and only if S logically follows from P?

The question was soon positively answered, independently, in their respective doctoral theses, by two graduate students. Kurt Gödel received his degree on February 6, 1930 at the University of Vienna, aged 23, and Jacques Herbrand received his at the Sorbonne on 11 June, 1930, aged 22 ([4], pp. 582 ff., and pp 525 ff.). Today's graduate students can be forgiven for thinking of these giants of our field as wise old greybeards who made their discoveries only after decades of experience. In fact they had not been around very long, and were relatively inexperienced and new to FOL.

The theorem is so important because the property of formal derivability is semidecidable: there is an algorithm by means of which, if a sentence S is in fact formally derivable from a sentence P, then this fact can be mechanically detected. Indeed, the detection procedure is usually organized in such a way that the detection consists of actually constructing a derivation of S from P. In other words, when looked at semantically, the syntactic detection algorithm becomes a proof procedure.

Herbrand did not live to see the power of the completeness theorem exploited usefully in modern implementations. He was killed in a climbing accident on July 27, 1931.

Gödel, however, lived until 1978, well into the era in which the power of the completeness theorem was widely displayed and appreciated. It is doubtful, however, whether he paid any attention to this development. Already by 1931, his interest had shifted to other areas of logic and mathematical foundations. After stunning the mathematical (and philosophical) world in 1931 with his famous incompleteness theorems ([4], pp. 592 ff.), he went off in yet another direction to prove (within FOL) the consistency of the axiom of choice and the generalized continuum hypothesis with each other and with the remaining axioms of set theory ([2]), leading to the later proof (by Paul Cohen) of their independence both from each other and from the remaining axioms ([6]).

Interestingly, it was also Gödel and Herbrand who played a major role in the computational part of computational logic. In the course of his proof of the incompletness theorems, Gödel introduced the first rigorous characterization of computability, in his definition of the class of the primitive recursive functions. In 1934 (reprinted in [7]) he broadened and completed his characterization of computability by defining the class of general recursive functions, following up a suggestion made to him by Herbrand in "a private communication".

6 Computation: Turing, Church, and von Neumann

In this way Gödel and Herbrand not only readied FOL for action, but also opened up the field of universal digital computation. Soon there were others. At the time of the proof of the completeness theorem in 1930, the 18-year-old Alan Turing was about become a freshman mathematics student at Cambridge University. Alonzo Church was already, at age 28, an instructor in the mathematics department at Princeton. John von Neumann had just emigrated, at age 27, to Princeton, where he was to spend the rest of his life. By 1936 these five had essentially laid the basis for the modern era of computing, as well as its intimate relationship with logic.

Turing's 1936 paper (reprinted in [7]) on Hilbert's Decision Problem was crucial. Ostensibly it was yet another attempt to characterize with mathematical precision the concept of computability, which he needed to do in order to show that there is no fixed computation procedure by means of which every definite mathematical assertion can be decided (determined to be true or determined to be false).

Alonzo Church independently reached the same result at essentially the same time, using his own quite different computational theory (reprinted in [7]) of lambda-definability. In either form, its significance for computational logic is that it proved the impossibility of a decision procedure for FOL. The semidecision procedure reflecting the deductive completeness of FOL is all there is, and we have to be content to work with that.

Remarkable as this theoretical impossibility result was, the enormous practical significance of Turing's paper came from a technical device he introduced into his argument. This device turned out to be the theoretical design of the modern universal digital computer. The concept of a Turing machine, and in particular of

a universal Turing machine, which can simulate the behavior of any individual Turing machine when supplied with a description of it, may well be the most important conceptual discovery of the entire twentieth century. Computability by the universal Turing machine became virtually overnight the criterion for absolute computability. Church's own criterion, lambda-definability, was shown by Turing to be equivalent to Turing computability. Other notions (e.g., Kleene's notion of deducibility in his equational-substitutional calculus, Post's notion of derivability within a Post production system) were similarly shown to be equivalent to Turing's notion. There was obviously a fundamental robustness here in the basic concept of computability, which led Alonzo Church to postulate that any function which was found to be effectively computatble would in fact be computable by a Turing machine ("Church's Thesis").

During the two years from 1936 to 1938 that Turing spent in Princeton, working on a Ph.D. with Church as supervisor, he and von Neumann became friends. In 1938 von Neumann offered Turing a job as his assistant at the Institute for Advanced Study, and Turing might well have accepted the offer if it had not been for the threat of World War 2. Turing was summoned back to Britain to take part in the famous Bletchley Park code-breaking project, in which his role has since become legendary.

Von Neumann had been enormously impressed by Turing's universal machine concept and continued to ponder it even as his wide-ranging interests and national responsibilities in mathematics, physics and engineering occupied most of his attention and time. He was particularly concerned to improve computing technology, to support the increasingly more complex numerical computing tasks which were crucial in carrying out wartime weapons development. His knowledge of Turing's idea was soon to take on great practical significance in the rapid development of the universal electronic digital computer throughout the 1940s. He immediately saw the tremendous implications of the electronic digital computing technology being developed by Eckert and Mauchly at the University of Pennsylvania, and essentially assumed command of the project. This enabled him to see to it that Turing's idea of completely universal computing was embodied in the earliest American computers in 1946.

The work at Bletchley Park had also included the development of electronic digital computing technology, and it was used in devices designed for the special purposes of cryptography. Having been centrally involved in this engineering development, Turing came out of the war in 1945 ready and eager to design and build a full-fledged universal electronic digital computer based on his 1936 ideas, and to apply it to a wide variety of problems. One kind of problem which fascinated him was to write programs which would carry out intelligent reasoning and engage in interesting conversation. In 1950 he published his famous essay in the philosophical journal *Mind* discussing artificial intelligence and computing. In this essay he did not deal directly with automated deduction as such, but there can be little doubt that he was very much aware of the possibilities. He did not live to see computational logic take off. His death in 1954 occurred just as things were getting going.

7 Computational Logic

In that very same year, 1954, Martin Davis (reprinted in [8]) carried out one of the earliest computational logic experiments by programming and running, on von Neumann's Institute for Advanced Study computer in Princeton, Presburger's Decision Procedure for the first order theory of integer addition. The computation ran only very slowly, as well it might, given the algorithm's worse than exponential complexity. Davis later wryly commented that its great triumph was to prove that the sum of two even numbers is itself an even number.

Martin Davis and others then began in earnest the serious computational applications of FOL. The computational significance of the Gödel-Herbrand completeness theorem was tantalizingly clear to them. In 1955 a particularly attractive and elegant version of the basic proof procedure – the semantic tableau method – was described (independently) by Evert Beth [9] and Jaakko Hintikka [10]. I can still remember the strong impression made on me, as a graduate student in philosophy, when I first read their descriptions of this method. They pointed out the intimate relationship of the semantic or analytic tableau algorithm to the natural deductive framework of the sequent calculi pioneered in the mid-1930s by Gerhard Gentzen [11]. Especially vivid was the intuitive interpretation of a growing, ramifying semantic tableau as the on-going record of an organized systematic search, with the natural diagrammatic structure of a tree, for a counterexample to the universally quantified sentence written at the root of the tree. The closure of the tableau then signified the failure of the search for, and hence the impossibility of, such a counterexample. This diagrammatic record of the failed search (the closed tableau) itself literally becomes a proof, with each of its steps corresponding to an inference sanctioned by some sequent pattern in the Gentzen-style logic. One simply turns it upside-down and reads it from the tips back towards the root. A most elegant concept as well as a computationally powerful technique!

8 Heuristics and Algorithms: The Dartmouth and Cornell Meetings

In the 1950's there occurred two meetings of major importance in the history of computational logic. The first was the 1956 Dartmouth conference on Artificial Intelligence, at which Herbert Simon and Allen Newell first described their heuristic theorem-proving work. Using the RAND Corporation's von Neumann computer JOHNNIAC, they had programmed *a heuristic search* for proofs in the version of the propositional calculus formulated by Russell and Whitehead. This pioneering experiment in computational logic attracted wide attention as an attempt to reproduce computationally the actual human proof-seeking behavior of a person searching for a proof of a given formula in that system. In their published accounts (reprinted in [8], Volume 1) of this experiment they claimed that the only alternative algorithmic proof-seeking technique offered by logic was the so-called "British Museum" method of enumerating all possible proofs in the hope that one would eventually turn up that proved the theorem being considered.

This was of course both unfair and unwise, for it simply advertised their lack of knowledge of the proof procedures already developed by logicians. Hao Wang showed in 1960 how easy the search for these proofs became if one used a semantic tableau method (reprinted in [8], Volume 1, pp. 244-266).

The other significant event was the 1957 Cornell Summer School in Logic. Martin Davis, Hilary Putnam, Paul Gilmore, Abraham Robinson and IBM's Herbert Gelernter were among those attending. The latter gave a talk on his heuristic program for finding proofs of theorems in elementary geometry, in which he made use of ideas very similar to those of Simon and Newell. Abraham Robinson was provoked by Gelernter's advocacy of heuristic, psychological methods to give an extempore lecture (reprinted in [8], Volume 1) on the power of logical methods in proof seeking, stressing the computational significance of Herbrand's version of the FOL completeness theorem and especially of the technique of elimination of existential quantifiers by introducing Skolem function sysmbols. One can see now how this talk in 1957 must have motivated Gilmore, Davis and Putnam to write their Herbrand-based proof procedure programs. Their papers are reprinted in [8], Volume 1, and were based fundamentally on the idea of systematically enumerating the Herbrand Universe of a proposed theorem – namely, the (usually infinite) set of all terms constructible from the function symbols and individual constants which (after its Skolemization) the proposed theorem contained. This technique is actually the computational version of Herbrand's so-called Property B method. It did not seem to be realized (as in retrospect it perhaps ought to have been) that this Property B method, involving a systematic enumeration of all possible instantiations over the Herbrand Universe, is really a version of the British Museum method so rightly derided by Simon and Newell.

9 Combinatorial Explosions with Herbrand's Property B

These first implementations of the Herbrand FOL proof procedure thus revealed the importance of trying to do better than merely hoping for the best as the exhaustive enumeration forlornly ground on, or than guessing the instantiations that might be the crucial ones in terminating the process. In fact, Herbrand himself had already in 1930 shown how to avoid this enumerative procedure, in what he called the Property A method. The key to Herbrand's Property A method is the idea of unification.

Herbrand's writing style in his doctoral thesis was not, to put it mildly, always clear. As a consequence, his exposition of the Property A method is hard to follow, and is in fact easily overlooked. At any rate, it seems to have attracted no attention except in retrospect, after the independent discovery of unification by Prawitz thirty years later.

In retrospect, it is nevertheless quite astonishing that this beautiful, natural and powerful idea was not immediately taken up in the early 1930s by the theoreticians of first order logic.

10 Prawitz's Independent Rediscovery of Unification Ushers in Resolution

In 1960 an extraordinary thing happened. The Swedish logician Dag Prawitz independently rediscovered unification, the powerful secret which had been buried for 30 years in the obscurity of the Property A section of Herbrand's thesis. This turned out to be an important moment in the history of computational logic. Ironically, Prawitz' paper (reprinted in [8], Volume 1) too was rather inaccessible. William Davidon, a physicist at Argonne, drew my attention to Prawitz' paper in 1962, two years after it had first appeared. I wish I had known about it sooner. It completely redirected my own increasingly frustrated efforts to improve the computational efficiency of the Davis-Putnam Herbrand Property B proof procedure. Once I had managed to recast the unification algorithm into a suitable form, I found a way to combine the Cut Rule with unification so as to produce a rule of inference of a new machine-oriented kind. It was machine-oriented because in order to obtain the much greater deductive power than had hitherto been the norm, it required much more computational effort to apply it than traditional human-oriented rules typically required. In writing this work up for publication, when I needed to think of a name for my new rule, I decided to call it "resolution", but at this distance in time I have forgotten why. This was in 1963. The resolution paper took more than another year to reach print, finally coming out in January 1965 (it is reprinted in [8], Volume 1).

The trick of combining a known inference rule with unification can of course be applied to many other rules besides the cut rule. At Argonne, George Robinson and Larry Wos quickly saw this, and they applied it to the rule of Equality Substitution, producing another powerful new machine-oriented rule which they subsequently exploited with much success. They called their new rule "paramodulation". Their paper is reprinted in [8], Volume 2. It and resolution have been mainstays of the famous Argonne theorem provers, such as McCune's OTTER, ever since. See, for example, the 1991 survey ([12], pp. 297 ff.) by Larry Wos.

After 1965 there ensued a somewhat frenetic period of exploration of what could now be done with the help of these new unification-based rules of inference. They were recognized as a big boost in the development of an efficient, practical automated deduction technology. People quite rapidly came up with ways to adapt them to other computational applications quite different from their original application, mechanical theorem-proving.

11 Computational Logic in Edinburgh: The A.I. Wars

Some of my memories of this period are still vivid. In 1965 Bernard Meltzer spent a three-month study leave at Rice University, where I was then a member of the Philosophy Department. Bernard rapidly assimilated the new resolution technique, and on his return to Edinburgh immediately organized a new research group which he called the Metamathematics Unit, with the aim of pursuing full-

time mechanical theorem-proving research. This turned out to be an important event. Edinburgh was already making its mark in Artificial Intelligence. Donald Michie, Christopher Longuet-Higgins, and Rod Burstall, in addition to Bernard Meltzer, were realigning their scientific careers in order to pursue AI full-time. Over the next few years Bernard's group became a lively, intense critical mass of graduate students who would go on to become leaders in computational logic: Bob Kowalski, Pat Hayes, Frank Brown, Donald Kuehner, Bob Boyer, J Strother Moore. Next door, in Donald Michie's Machine Intelligence group, were Gordon Plotkin, David Warren, Maarten van Emden and John Darlington. I spent a sabbatical year with these groups, from May 1967 to September 1968, and thereafter visited them for several extended study periods. Edinburgh was, at that time, the place to be. It was a time of great excitement and intellectual ferment. There was a pervasive feeling of optimism, of pioneering, of great things to come. We had fruitful visits from people in other AI centers: John McCarthy, Bert Raphael, Nils Nilsson, Cordell Green, Keith Clark, Carl Hewitt, Seymour Papert, Gerald Sussman. The last three were advocates of the MIT view (championed by Marvin Minsky) that computational logic was not at all the AI panacea some people thought it was. The MIT view was that the engineering of machine intelligence would have to be based on heuristic, procedural, associative organization of knowledge. The "logic" view (championed by John McCarthy) was that for AI purposes knowledge should be axiomatized declaratively using FOL, and mechanized thinking should consist of theorem-proving computation. This was, at that time, the view prevailing in Edinburgh and Stanford. The two opposing views gave rise to some stimulating debates with the MIT visitors.

Cordell Green's question-answering system in 1969 ([13], pp. 219 ff.) demonstrated the possibilities of intelligent deductive datebases and automatic robot planning using a resolution theorem prover as the reasoning engine. He showed how John McCarthy's 1959 scheme of a deductive Advice Taker (reprinted in [14]) could be implemented.

12 The Colmerauer-Kowalski Encounter

The possibilities of resolution logic for computational linguistics were immediately seen by Alain Colmerauer, who had been working in Grenoble since 1963 on parsing and syntactic analysis ([15]). By 1968 he was working for an Automatic Translation Project in Montreal, where he developed what later could be seen as a precursor of Prolog (the Q-systems formalism). Returning to France in 1970 he took up the theme of making deductions from texts instead of just parsing them, and began to study the resolution principle. This brought him into contact with Edinburgh's Bob Kowalski, who had, in collaboration with Donald Kuehner ([16]), recently devised a beautiful refinement of resolution (SL-resolution) which permitted linear deductions and which could be implemented with great efficiency, as Boyer and Moore soon showed ([17], pp.101 ff.) by the first of their many programming masterpieces, the Edinburgh Structure-Sharing Linear Resolution Theorem Prover.

The result of this Colmerauer-Kowalski encounter was the birth of Prolog and of logic programming. Colmerauer's colleague Philippe Roussel, following discussions with Boyer and Moore, designed the first modern Prolog interpreter using Boyer-Moore's shared data structures. Implemented in Fortran by Meloni and Battani, this version of Prolog was widely disseminated and subsequently improved, in Edinburgh, by David Warren.

13 Computational Logic Officially Recognized: Prolog's Spectacular Debut

By 1970 the feeling had grown that Bernard Meltzer's Metamathematics Unit might be better named, considering what what was going on under its auspices. At the 1970 Machine Intelligence Workshop in Edinburgh, in a paper ([18], pp. 63-72) discussing how to program unification more efficiently, I suggested that "computational logic" might be better than "theorem proving" as a description of the new field we all seemed now to be working in. By December 1971 Bernard had convinced the university administration to sign on to this renaming. From that date onwards, his official stationery bore the letterhead: University of Edinburgh: Department of Computational Logic.

During the 1970s logic programming moved to the center of the stage of computational logic, thanks to the immediate applicability, availability, and attractiveness of the Marseille - Edinburgh Prolog implementation, and to Bob Kowalski's eloquent and tireless advocacy of the new programming paradigm. In 1977, I was present at the session of the Lisp and Functional Programming Symposium in Rochester, New York, when David Warren awed a partisan audience of LISP devotees with his report of the efficiency of the Edinburgh Prolog system, compared with that of Lisp ([19]). A little later, in [20], Bob Kowalski spelled out the case for concluding that, as Prolog was showing in practice, the Horn clause version of resolution was the ideal programming instrument for all Artificial Intelligence research, in that it could be used to represent knowledge declaratively in a form that, without any modification, could then be run procedurally on the computer. Knowledge was both declarative and procedural at once: the "nothing buttery" of the contending factions in the Great A.I. War now could be seen as mere partisan fundamentalist fanaticism.

14 The Fifth Generation Project

These were stirring times indeed. In Japan, as the 1970s drew to a close, there were bold plans afoot which would soon startle the computing world. The 10-year Fifth Generation Project was three years (1979 to 1981) in the planning, and formally began in April 1982. A compact retrospective summary of the project is given by Hidehiko Tanaka in his Chairman's message in [21]. It was a pleasant surprise to learn around 1980 that this major national research and development effort was to be a large-scale concerted attempt to advance the state of the art in computer technology by concentrating on knowledge information processing using Logic Programming as a central concept and technique.

The decade of the 1980s was dominated by the Fifth Generation Project. It was gratifying for the computational logic community to observe the startled response of the authorities in the United States and Europe to such an unexpectedly sophisticated and powerful challenge from a country which had hitherto tended to be viewed as a follower rather than a leader. One of the splendid features of the Fifth Generation Project was its wise and generous provision for worldwide cooperation and interaction between Japanese researchers and researchers from every country where logic programming was being studied and developed.

In 1984 the first issue of The Journal of Logic Programming appeared. In the Editor's Introduction I noted [22] that research activity in logic programming had grown rapidly since 1971, when the first Prolog system introduced the idea, and that a measure of the extent of this growth was the size of the comprehensive bibliography of logic programming compiled by Michael Poe of Digital Equipment Corporation, which contains over 700 entries. The substantial length of Poe's bibliography in [22] was only one sign of the explosion of interest in logic programming. Another sign was the size and frequency of international conferences and workshops devoted to the subject, and the increasing number of people attending them from all over the world. It was inevitable that enthusiastic and in some respects overoptimistic expectations for the future of the new programming paradigm would eventually be followed by a more sober realism, not to say a backlash.

15 The Aftermath: Pure Logic Programming not a Complete Panacea

There is no getting around the fact that a purely declarative programming language, with a purely evaluative deductive engine, lacks the capability for initiating and controlling events: what should happen, and where, when, and how. If there are no side effects, then there are no side effects, and that is that. This applies both to internal events in the steps of execution of the deduction or evaluation computation, and to external events in the outside world, at a minimum to the input and output of data, but (for example if the program is functioning as an operating system) to the control of many other external sensors and actuators besides. One wants to create and manipulate objects which have states.

So although the pure declarative logic programming paradigm may be a thing of beauty and a joy forever, it is a practically useless engineering tool. To be useful, its means of expression must be augmented by imperative constructs which will permit the programmer to design, and oblige the computer to execute, happenings – side effects.

There was a tendency (there still may be, but I am a little out of touch with the current climate of opinion on this) to deplore side effects, in the spirit of Edsger Dijkstra's famous "GO TO considered harmful" letter to the Editor of the Communications of the ACM. There is no denying that one can get into a terrible mess by abusing imperative constructs, often creating (as Dijkstra wrote

elsewhere) a “disgusting mass of undigested complexity” on even a single page of code. The remedy, however, is surely not to banish imperative constructs from our programming languages, but to cultivate a discipline of programming which even when using imperative constructs will aim at avoiding errors and promoting intelligibility.

So the Prolog which looked as if it might well sweep the computing world off its feet was not simply the pure embodiment of Horn clause resolution, but a professional engineering tool crucially incorporating a repertoire of imperative constructs: sequential execution of steps (namely a guaranteed order in which the subgoals would be tried), cut (for pruning the search on the fly), assert and retract (for changing the program during the computation) and (of course) read, print, and so on.

Bob Kowalski’s equation ALGORITHM = LOGIC + CONTROL summarized the discipline of programming needed to exploit the new paradigm - strict separation of the declarative from the imperative aspects. One must learn how to keep the denotation (the presumably intended declarative meaning, as specified by the clauses) of a program fixed while purposefully varying the process by which it is computed (the pragmatic meaning, as specified by the sequence of the clauses and, within each clause, the sequence of the literals, together with the other imperative constructs). Versions of Prolog began to reflect this ideal strict separation, by offering the programmer the use of such features as modes and various other pragmatic comments and hints.

16 Realism: Side Effects, Concurrency, Software Engineering

Others were less concerned with declarative purism and the holy grail of dispensing with all imperative organization of explicit control. In some sense they went to the opposite extreme and explored how far one could go by making the most intelligent use of the clausal formalism as a means of specifying computational events.

Ehud Shapiro, Keith Clark, Kazunori Ueda and others followed this direction and showed how concurrency could be specified and controlled within the new idiom. If you are going into the business of programming a system whose purpose is to organize computations as complexes of interacting events evolving over time, you have to be able not only to describe the processes you want to have happen but also to start them going, maintain control over them once they are under way, and stop them when they have gone far enough, all the while controlling their interaction and communication with each other, and managing their consumption of system resources. An operating system is of course just such a system. It was therefore a triumph of the concurrent logic programming methodology when the Fifth Generation Project in 1992 successfully concluded a decade of work having designed, built and successfully run the operating system of the ICOT knowledge

processing parallel computation system entirely in the Flat Guarded Horn Clause programming language [21].

Much had been achieved technologically and scientifically at ICOT by 1994, when the project (which with a two-year extension had lasted twelve years) finally closed down, but there continued (and continues) to be a splendid spinoff in the form of a worldwide community of researchers and entrepreneurs who were drawn together by the ICOT adventure and remain linked by a common commitment to the future of logic programming. Type "logic programming" into the search window of a search engine. There emerges a cornucopia of links to websites all over the world dealing either academically or commercially (or sometimes both) with LP and its applications.

17 Robbins' Conjecture Proved by Argonne's Theorem Prover

Although logic programming had occupied the center of the stage for fifteen years or so, the original motivation of pursuing improved automatic deduction techniques for the sake of proof discovery was still alive and well. In 1996 there occurred a significant and exciting event which highlighted this fact. Several groups have steadily continued to investigate the application of unification-based inference engines, term-rewriting systems, and proof-finding search strategies to the classic task of discovering proofs for mathematical theorems. Most notable among these has been the Argonne group under the leadership of Larry Wos. Wos has continued to lead Argonne's tightly focussed quest for nontrivial machine-generated proofs for over thirty-five years. In 1984 his group was joined by William McCune, who quickly made his mark in the form of the highly successful theorem proving program OTTER, which enabled the Argonne group to undertake a large-scale pursuit of computing real mathematical proofs. The spirit of the Argonne program was similar to that of Woody Bledsoe, who at the University of Texas headed a research group devoted to automating the discovery of proofs in what Woody always referred to as "real" mathematics. At the symposium celebrating Woody's seventieth birthday in November 1991 [12] Larry Wos reviewed the steady progress in machine proof-finding made with the help of McCune's OTTER, reporting that already copies of that program were at work in several other research centers in various parts of the world. Sadly, Woody did not live to relish the moment in 1996 when Argonne's project paid off brilliantly in the machine discovery of what was undoubtedly a "real" mathematical proof.

The Argonne group's achievement is comparable with that of the program developed by the IBM Deep Blue group which famously defeated the world chess champion Kasparov. Both events demonstrated how systematic, nonhuman combinatorial search, if organized sufficiently cleverly and carried out at sufficiently high speeds, can rival even the best human performance based heuristically and associatively on expertise, intuition, and judgment. What the Argonne program did was to find a proof of the Robbins conjecture that a particular set of three equations is powerful enough to capture all of the laws of

Boolean algebra. This conjecture had remained unproved since it was formulated in the 1930s by Herbert Robbins at Harvard. It had been tackled, without success, by many "real" matheticians, including Alfred Tarski. Argonne attacked the problem with a new McCune-designed program EQP embodying a specialized form of unification in which the associative and commutative laws were integrated. After 20 hours of search on a SPARC 2 workstation, EQP found a proof. Robbins' Conjecture may not be Fermat's Last Theorem, but it is certainly "real" mathematics. Woody Bledsoe would have been delighted, as we all were, to see the description of this noteworthy success for computational logic written up prominently in the science section of the New York Times on December 10, 1996.

18 Proliferation of Computational Logic

By now, logic programming had completed a quarter-century of fruitful scientific and technological ferment. The new programming paradigm, exploding in 1971 following the original Marseille-Edinburgh interaction, had by now proliferated into an array of descendants. A diagram of this genesis would look like the cascade of elementary particle tracks in a high-energy physics experiment. We are now seeing a number of interesting and significant phenomena signalling this proliferation.

The transition from calling our field "logic programming" to calling it "computational logic" is only one such signal. It was becoming obvious that the former phrase had come to mean a narrower interest within a broader framework of possible interests, each characterized by some sort of a connection between computing and logic.

As we noted earlier, the statement of scope of the new ACM journal TOCL points out that the field of computational logic consists of all uses of logic in computer science. It goes on to list many of these explicitly: artificial intelligence; computational complexity; database systems; programming languages; automata and temporal logic; automated deduction; automated software verification; commonsense and nonmonotonic reasoning; constraint programming; finite model theory; complexity of logical theories; functional programming and lambda calculus; concurrency calculi and tools; inductive logic programming and machine learning; logical aspects of computational complexity; logical aspects of databases; logic programming; logics of uncertainty; modal logics, including dynamic and epistemic logics; model checking; program development; program specification; proof theory; term rewriting systems; type theory and logical frameworks.

19 A Look at the Future: Some Challenges

So the year 2000 marks a turning point. It is the closing of the opening eventful chapter in a story which is still developing. We all naturally wonder what will happen next. David Hilbert began his address to the International Congress of

Mathematicians by asking: who of us would not be glad to lift the veil behind which the future lies hidden; to cast a glance at the next advances of our science and at the secrets of its development during future centuries? The list of open problems he then posed was intended to – and certainly did – help to steer the course of mathematics over the next decades. Can we hope to come up with a list of problems to help shape the future of computational logic? The program committee for this conference thought that at least we ought to try, and its chairman John Lloyd therefore set up an informal email committee to come up with some ideas as to what the main themes and challenges are for computational logic in the new millennium. The following challenges are therefore mainly distilled from communications I have received over the past three months or so from members of this group: Krzysztof Apt, Marc Bezem, Maurice Brynooghe, Jacques Cohen, Alain Colmerauer, Veronica Dahl, Marc Denecker, Danny De Schreye, Pierre Flener, Koichi Furukawa, Gopal Gupta, Pat Hill, Michael Kifer, Bob Kowalski, Kung-Kiu Lau, Jean-Louis and Catherine Lassez, John Lloyd, Kim Marriott, Dale Miller, Jack Minker, Lee Naish, Catuscia Palamidessi, Alberto Pettorossi, Taisuke Sato, and Kazunori Ueda. My thanks to all of them. I hope they will forgive me for having added one or two thoughts of my own. Here are some of the challenges we think worth pondering.

19.1 To Shed Further Light on the P = NP Problem

The field of computational complexity provides today's outstanding challenge to computational logic theorists. Steve Cook was originally led to formulate the P = NP problem by his analysis of the complexity of testing a finite set S of clauses for truth-functional satisfiability. This task certainly seems exponential: if S contains n distinct atoms and is unsatisfiable then every one of the 2^n combinations of truth values must make all the clauses in S come out false simultaneously. All the ways we know of for checking S for satisfiability thus boil down to searching this set of combinations. So why are we are still unable to prove what most people now strongly believe, that this problem is exponentially complex? We not only want to know the answer to the problem, but we also want to understand why it is such a hard problem.

This challenge was only one of several which call for an explanation of one or another puzzling fact. The next one is another.

19.2 To Explain the Intelligibility/Efficiency Trade-off

This trade-off is familiar to all programmers and certainly to all logic programmers. Why is it that the easier a program is to understand, the less efficient it tends to be? Conversely, why do more computationally efficient programs tend to be more difficult to reason about? It is this curious epistemological reciprocity which forces us to develop techniques for program transformations, program verification, and program synthesis. It is as though there is a deep mismatch between our brains and our computing devices. One of the big advantages claimed for logic programming – and more generally for declarative programming – is the natural intelligibility of programs which are essentially just

declarations - sets of mutually recursive definitions of predicates and functions. But then we have to face the fact that "naive" programs written in this way almost always need to be drastically reformulated so as to be made computationally efficient. In the transformation they almost always become more obscure. This is a price we have become used to paying. But why should we have to pay this price? It surely must have something to do with the peculiar constraints under which the human mind is forced to work, as well as with objective considerations of the logical structure and organization of the algorithms and data structures. An explanation, if we ever find one, will presumably therefore have to be partly psychological, partly physiological, partly logical.

The nature of our brains, entailing the constraints under which we are forced to perceive and think about complex structures and systems, similarly raises the next challenge.

19.3 To Explain Why Imperative Constructs are Harmful to a Program's Intelligibility

This is obviously related to the previous challenge. It has become the conventional wisdom that imperative programming constructs are at best an unpleasaant necessity. We are supposed to believe that while we apparently can't do without them, they are a deplorable source of confusion and unintelligibility in our programs. In Prolog the notorious CUT construct is deployed masterfully by virtuoso logic programmers but regretted by those who value the elegance and clarity of "pure" Prolog. Why? The sequential flow of control is counted on by programmers to obtain the effects they want. If these features are indispensible to writing effective programs, in practice, then why does our theory, not to say our idealogy, treat them as pariahs? This is a kind of intellectual hypocrisy. It may well be that imperative constructs are dangerous, but we are acting as though it is impossible that our programming methodology should ever be able to manage them safely. If true, this is extremely important, but it has to be proved rather than just assumed. The same issue arises, of course, in the case of functional programming. "Pure" declarativeness of a program is promoted as an ideal stylistic desideratum but (as, e.g., in classical Lisp) hardly ever a feature of practical programs. The imperative features of "real" Lisp are in practice indispensible. The dominance of unashamedly imperative languages like C, C++, and Java in the "outside world" is rooted in their frank acceptance of the imperative facts of computational life. But this is precisely what earns them the scorn of the declarative purists. Surely this is an absurd situation. It is long overdue to put this issue to rest and to reconcile lofty but unrealistic ideals with the realities of actual software engineering practice.

A similar issue arises within logic itself, quite apart from computation. If logic is to be a satisfactory science of reasoning, then it must face the following challenge.

19.4 To Explain the (Informal, Intuitive) / (Formal, Rigorous) Trade-off in Proofs

Why is it that intuitive, easy-to-understand proofs tend to become more complex, less intelligible and less intuitive, when they are formalized in order to make them more rigorous? This is suspiciously like the trade-offs in the previous three challenges. What, anyway, is rigor? What is intuition? These concepts are epistemological rather than logical. We all experience this trade-off in our everyday logical experience, but we lack an explanation for it. It is just something we put up with, but at present it is a mystery. If logic cannot explain it, then logic is lacking something important.

The prominence of constraint logic programming in current research raises the following challenge.

19.5 To Understand the Limits of Extending Constraint Logic Programming

As constraint solving is extended to even richer domains, there is a need to know how far automatic methods of constraint satisfaction can in fact feasibly be extended. It would appear that directed combinatorial search is at the bottom of most constraint solution techniques, just as it is in the case of systematic proof procedures. There must then be ultimate practical limits imposed on constraint-solving searches by sheer computational complexity. It would certainly seem to be so in the classic cases, for example, of Boolean satisfiability, or the Travelling Salesman Problem. The concept of constraint logic programming is open-ended and embarrassingly general. The basic problem is simply to find, or construct, an entity X which satisfies a given property P. Almost any problem can be thrown into this form. For example the writing of a program to meet given specifications fits this description. So does dividing one number Y by another number Z: it solves the problem to find a number R such that R = Y/Z. So does composing a fugue for four voices using a given theme.

It is illuminating, as a background to this challenge, to recall the stages by which the increasingly rich versions of constraint logic programming emerged historically. Alain Colmerauer's reformulation of unification for Prolog II originally led him to consider constraint programming [15]. Just as had Herbrand in 1930, and Prawitz in 1960, Colmerauer in 1980 saw unification itself as a constraint problem, of satisfying a system of equations in a domain of terms. After loosening the constraints to permit cyclic (infinite) terms as solutions of equations like $x = f(x)$, which had the virtue of eliminating the need for an occur check, he also admitted inequations as well as equations. It became clear that unification was just a special case of the general concept of constraint solving in a given domain with given operations and relations, which led him to explore constraint solving in richer domains. Prolog III could also handle two-valued Boolean algebra and a numerical domain, involving equations, inequalities and infinite precision arithmetical operations. Thus logic programming morphed by stages into the more general process of finding a satisfying assignment of values to

a set of unknowns subjected to a constraint (i.e., a sentence) involving operations and relations defined over a given domain. The computer solves the constraint by finding or somehow computing values which, when assigned to its free variables, make it come out true. Later in the 1980s Joxan Jaffar and Jean-Louis Lassez further elaborated these ideas, thus deliberately moving logic programming into the general area of mathematical programming which had formed the heart of Operations Research since the 1940s. Constraint solving of course is an even older mathematical idea, going back to the classical methods of Gauss and Fourier, and to Lagrange, Bernouilli and the calculus of variations. In its most general form it is extremely ancient: it is nothing other than the fundamental activity of mathematics itself: to write down an open sentence of some kind, and then to find the values of its free variables, if any, which make it true.

In summary: the challenge is to pull together all the diversity within the general notion of constraint satisfaction programming, and to try to develop a unified theory to support, if possible, a single formalism in which to formulate all such problems and a single implementation in which to compute the solutions to the solvable ones.

It has not escaped the notice of most people in our field that despite our better mousetrap the world has not yet beaten a path to our door. We are challenged not just to try to make our mousetrap even better but to find ways of convincing outsiders. We need applications of existing LP and CLP technology which will attract the attention and enthusiasm of the wider computing world. Lisp has had an opportunity to do this, and it too has fallen short of perhaps over-hyped expectations of wide adoption. Logic programming still has vast unrealized potential as a universal programming tool. The challenge is to apply it to real problems in ways that outdo the alternatives.

Kazunori Ueda and Catuscia Palamidessi both stress the importance of Concurrent Constraint Programming. Catuscia maintains that concurrency itself poses a fundamental challenge, whether or not it is linked to constraint programming. She even suggests that the notion of concurrent computation may require an extension or a reformulation of the theory of computability, although she notes that this challenge is of course not specific to computational logic. It is interesting to note that the early computing models of Turing and Post avoided the issue of concurrency entirely.

19.6 To Find New Killer Applications for LP

This challenge has always been with us and will always be with us. In general, advances and evolution in the methodology and paradigms of logic programming have repeatedly been stimulated and driven by demands imposed by applications. The first such application was the parsing of natural languages (Alain Colmerauer and Veronica Dahl both say this is still the most natural LP application). Future applications to natural language processing could well be killers. Jack Minker points out that the deductive retrieval of information from databases was one of the earliest applications and today still has an enormous unrealized potential.

Following the example of Burstall and Darlington in functional programming, Ehud Shapiro and others soon demonstrated the elegance and power of the paradigm in the specification, synthesis, transformation, debugging and verification of logic programs. Software engineering is a hugely promising application area. Kung-Kiu Lau and Dale Miller emphasize the need for software engineering applications dealing with programming in the large - the high level organization of modules in ways suitable for re-use and component-based assembly by users other than the original composers.

Shapiro, Clark, Ueda and others then opened up the treatment of concurrency, shedding new light on the logic of message-passing and the nature of parallel processes through the remarkable properties of partial binding of logic variables. Shapiro and Takeuchi showed how the LP paradigm could embody the object-oriented programming style, and of course LP came to the fore in the representation of knowledge in the programming and control of robots and in other artificial intelligence systems. Already there are signs of further development in the direction of parallel and high-performance computing, distributed and network computing, and real-time and mobile computing. The challenge is to anticipate and stay ahead of these potential killer applications by enriching the presentation of the basic LP methodology to facilitate the efficient and smooth building-in of suitable special concepts and processes to deal with them. This may well (as it has in the past) call for the design of special purpose languages and systems, each tailored to a specific class of applications. Whatever form it takes, the challenge is to demonstrate the power of LP by actual examples, not just to expound its virtues in the abstract.

Examples of killer applications will help, but it is better to teach people to fish for themselves rather than simply to throw them an occasional salmon or halibut. The challenge is to provide the world with an irresistably superior set of programming tools and accompanying methodology, which will naturally replace the existing status quo.

19.7 To Enhance and Propagate the Declarative Programming Paradigm

Major potential growth areas within CL include the development of higher order logic programming. Higher order logic has already been applied (by Peter Andrews and Dale Miller, for example) to the development of higher order theorem provers. The functional programming community has for years been preaching and illustrating the conceptual power of higher order notions in writing elegantly concise programs and making possible the construction and encapsulation of large, high level computational modules. The consequences for software engineering are many, but as yet the software engineers, even the more theoretically inclined among them, have not responded to these attractions by adopting the higher-order paradigm and incorporating it into their professional repertoire.

The computational logic community should feel challenged to carry out the necessary missionary work here too, as well as to develop the necessary

programming tools and technical infrastructure. Dale Miller points out that mechanical theorem provers for higher order logics have already been successfully applied experimentally in many areas including hardware verification and synthesis; verification of security and communications protocols; software verification, transformation and refinement; compiler construction; and concurrency. The higher order logics used to reason about these problems and the underlying theorem prover technology that support them are also active areas of research. Higher order logic is however still a relatively esoteric corner of computational logic. Its conceptual basis has perhaps been unduly obscured in the past by an over-formal theoretical approach and by unnecessarily metaphysical conceptual foundations. The computational formalisms based on the elegant Martin-Löf theory of types are not yet ready to be launched on an industrial scale. It is a challenge to bring higher order methods and ideas down to earth, and to demonstrate to the computing world at large that the rewards are great and the intellectual cost is reasonable. Lambda-calculus computing has a simple operational rationale based on an equality logic of term rewriting. Its essential idea is that of the normalization of a given expression. Namely, a directed search is made for an equivalent expression which is in a form suitable as an answer. This process is also thought of as evaluation of the given expression. Under suitable patterns of rewriting (e.g., by so-called lazy evaluation) a successful search for a normal form is guaranteed to succeed if a normal form exists. In constraint satisfaction the essential idea is instantiation of a given expression: a directed search is made for an instance which is in a form suitable as an answer (e.g., if the given expression is a sentence, then the instance should be true). Both these ideas are simple yet powerful, and are extremely suitable as the basis for a pedagogical account of high level computation.

It has been a source of weakness in logic programming that there have been two major paradigms needlessly pitted against each other, competing in the same marketplace of ideas. The challenge is to end the segregation and merge the two. There is in any case, at bottom, only one idea. So our final challenge is the following one.

19.8 To Integrate Functional Programming with Logic Programming

These are two only superficially different dialects of computational logic. It is inexplicable that the two idioms have been kept apart for so long within the computational logic repertory. We need a single programming language in which both kinds of programming are possible and can be used in combination with each other. There have been a number of attempts to devise programming formalisms which integrate the two. None have so far been taken up seriously by users. This may well be because the systems offered so far have been only experimental and tentative. Some experiments have grafted evaluation on to a logic programming host, as for example, in the form of the limited arithmetical expressions allowed in Prolog. Others have grafted constraint satisfying onto a functional programming host in the form of the "setof" construct. More recently there have been attempts to subsume both idioms within a unified system based on term rewriting, for example, Michael Hanus's integrated functional logic programming language

Curry [25], the Vesper formalism of Robinson and Barklund (in [23]), the Escher system of Lloyd (in [23]) and the Alma system of Apt and Bezem [24]. These unified systems have hitherto been mere experiments, and have attracted few if any actual users. The system of Hanus is the currently most promising one. May its momentum increase and its user community thrive. The challenge is to create a usable unified public LP language which will attract every kind of user by its obvious superiority over the alternatives.

Those are only some of the challenges which await us.

20 Hilbert Has the Last Word

Hilbert's ending for his 1900 address was a stirring declaration of faith in the continuing unity of his field. If we substitute "computational logic" for "mathematics", what he said on that occasion becomes exactly what we should be telling ourselves now. So let us borrow and re-use his words to end this address on a similar note.

> The problems mentioned are merely samples of problems, yet they will suffice to show how rich, how manifold and how extensive the [computational logic] of today is, and the question is urged upon us whether [computational logic] is doomed to the fate of those other sciences that have split up into separate branches, whose representatives scarcely understand one another and whose connection becomes ever more loose. I do not believe this nor wish it. [Computational logic] is in my opinion an indivisible whole, an organism whose vitality is conditioned upon the connection of its parts. For ... we are still clearly conscious of the similarity of the logical devices, the relationship of the ideas in [computational logic] as a whole and the numerous analogies in its different departments. ... So it happens that, with the extension of [computational logic], its organic character is not lost but only manifests itself the more clearly. But, we ask, with the extension of [computational logic] will it not finally become impossible for the single investigator to embrace all departments of this knowledge? In answer let me point out how thoroughly it is ingrained in [computational logic] that every real advance goes hand in hand with the invention of sharper tools and simpler methods which at the same time assist in understanding earlier theories and cast aside older more complicated developments. ... The organic unity of [computational logic] is inherent in the nature of this science ... . That it may completely fulfil [its] high mission, may the new century bring it gifted masters and many zealous and enthusiastic disciples!

References

1. Quine, W.V. The Ways of Paradox and Other Essays, Random House, 1966.
2. Gödel, K. The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis with the Axioms of Set Theory. Princeton, 1940.
3. Hilbert, D. Mathematical Problems. Bulletin of the American Mathematical Society, Volume 8, 1902, pp. 437 - 479. (Engish translation of original German version).
4. Heijenoort, J. van (editor). From Frege to Gödel; A source Book in Mathematical Logic, 1879 - 1931. Harvard, 1967.
5. Tarski, A. Logic, Semantics, Metamathematics. Oxford, 1956.
6. Cohen, P. J. Set Theory and the Continuum Hypothesis. Benjamin, 1966.
7. Davis, M. (editor). The Undecidable. Raven Press, 1965.
8. Siekmann, J. and Wrightson, G. (editors). The Automation of Reasoning: Classical Papers on Computational Logic. 2 Volumes, Springer, 1983.
9. Beth, E. Semantic Entailmentand Formal Derivability, North Holland, 1955.
10. Hintikka, J. Form and Content in Quantification Theory. Acta Philosophica Fennica 8, 1955, pp. 7-55.
11. Gentzen, G. Untersuchungen über das logische Schliessen. Mathematische Zeitschrift 39, 1934, pp. 176-210, 405-431.
12. Boyer, R. S. (editor). Automated Reasoning. Essays in Honor of Woody Bledsoe. Kluwer, 1991.
13. Walker, D. and Norton, L. (editors). Proceedings of the International Joint Conference on Artificial Intelligence, Washington D.C., 1969.
14. Brachman, R. and Levesque, H. (editors). Readings in Knowledge Representation. Morgan Kaufmann, 1985.
15. Colmerauer, A. Curriculum Vitae, January 1999. Private communication.
16. Kowalski, R. and Kuehner, D. Linear Resolution with Selection Function. Artificial Intelligence 2, 1971, pp. 227-260.
17. Meltzer, B. and Michie, D. Machine Intelligence 7, Edinburgh, 1972.
18. Meltzer, B. and Michie, D. Machine Intelligence 6, Edinburgh, 1971.
19. Warren, D. and Pereira, L. PROLOG: The Language and Its Implementation Compared with LISP. SIGPLAN Notices 12, No 8, August 1977, pp. 109 ff.
20. Kowalski, R. Logic for Problem Solving. North Holland, 1979.
21. International Conference on Fifth Generation Computer Systems 1992, Proceedings, ICOT, Tokyo, 1992.
22. The Journal of Logic Programming, 1, 1984.
23. Furukawa, K., Michie, D. and Muggleton, S. Machine Intelligence 15, Oxford University Press, 1999.
24. Apt, K. and Bezem, M. Formulas as Programs. Report PNA-R9809, CWI, Amsterdam, October 1998.
25. Hanus, M. A Unified Computation Model for Functional and Logic Programming. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), 1997.