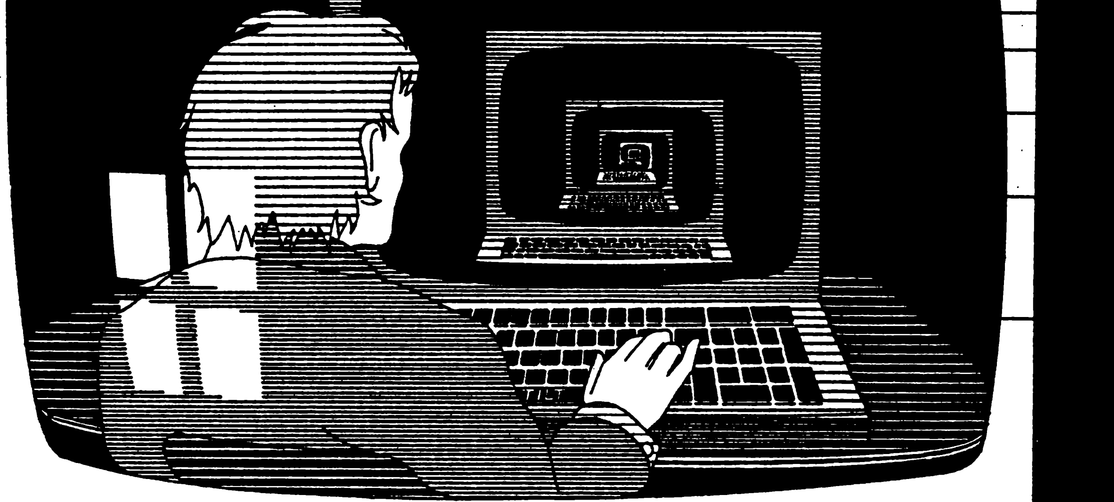
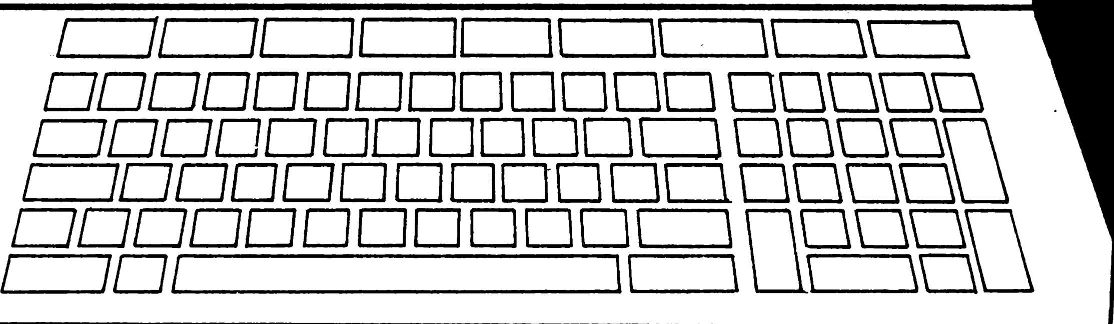
ALGORITHMS IN ARTIFICIAL INTELLIGENCE





D.M.G.

DE CHAMPEAUX

DE LABOULAYE

ALGORITHMS
IN ARTIFICIAL INTELLIGENCE

ALGORITHMS IN ARTIFICIAL INTELLIGENCE

ACADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN
DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN
AAN DE UNIVERSITEIT VAN AMSTERDAM
OP GEZAG VAN DE RECTOR MAGNIFICUS
DR. D.W. BRESTERS
HOOGLERAAR IN DE FACULTEIT DER WISKUNDE EN NATUURWETENSCHAPPEN,
IN HET OPENBAAR TE VERDEDIGEN
OP WOENSDAG 29 APRIL 1981 DES NAMIDDAGS TE 15.00 UUR PRECIES
IN DE AULA DER UNIVERSITEIT
(TIJDELIJK IN DE LUTHERSE KERK, INGANG SINGEL 411, HOEK SPUI)

DOOR

DENIS MARIE GABRIEL de CHAMPEAUX de LABOULAYE

GEBOREN TE TOULOUSE

Promotor : Dr. P. van Emde Boas

Copromotor : Dr. A.B. Frielink

Coreferent : Dr. N.V. Findler

ALGORITHMS IN ARTIFICIAL INTELLIGENCE

Dennis de Champeaux
University of Amsterdam
Jodenbreestr 23
1011 NH Amsterdam
Netherlands
tel 020-525 4262/3

voor mijn moeder
voor Marijke

Contents

	Page	
INTRODUCTION	1.	1
1. Artificial Intelligence in a nutshell	1.	1
1.1 Taxonomy of A.I.	1.	2
1.2 A.I. as a Sub-Discipline of Computer Science	1.	3
1.3 A.I. and Other Sciences	1.	4
1.4 Methodology of A.I.	1.	7
1.5 Achievements and Non-Achievements of A.I.	1.	10
1.6 A.I. in the Future	1.	20
2. A.I. and this thesis	1.	23
BI-DIRECTIONAL HEURISTIC SEARCH	2.	1
1. Introduction	2.	1
2. Bi-directional heuristic front-to-front algorithm	2.	5
3. Minimal path and optimality theorems for BHFFA2	2.	16
4. Worst case analysis	2.	22
5. Implementation results	2.	35
6. Open problems and loose ends	2.	46
SUBSTITUTION IN LISP	3.	1
1. Introduction	3.	1
2. Substitution Functions	3.	6
3. Verification of LISP Functions	3.	12
3.1 The State Description Language	3.	12
3.2 The Symbolic Evaluator	3.	18
3.3 The Deductive Machinery	3.	28
4. Verification of the Substitution Functions	3.	33
4.1 SUBSTSUPF1	3.	33
4.2 SUBSTSUPF2	3.	34
4.3 SUBSTAD1	3.	39
4.4 SUBSTAD2	3.	44
4.5 SUBSTADP	3.	57
5. Implementation Results	3.	75
6. Conclusions	3.	79
APPENDIX	3.	81
TWO THEOREM PROVER PREPROCESSORS	4.	1
1. Motivation	4.	1
2. Compressed Mini-Scope and INSTANCE	4.	6
3. INSURER	4.	18
4. Interplay between INSURER and INSTANCE	4.	30
5. Implementation results	4.	32
6. What next	4.	37
CONCLUSION	5.	1
SAMENVATTING (in Dutch)	6.	1
REFERENCES	7.	1

INTRODUCTION

1. Artificial Intelligence in a nutshell

This introduction begins with a description of Artificial Intelligence (= A.I.) not only because our thesis is concerned with it, but also because we feel that A.I. is not yet well known in the Netherlands. Moreover, every young science traditionally has the urge to demarcate its field, and A.I., about 25 years old, is no exception. There is yet another good reason for a description. A.I. has a broad scope. It, for instance, aims at computer programs which can play chess on champion level, can translate English into French and/or can drive a car in heavy traffic (none of these goals are within close reach, but to make our position immediately clear, we refuse to pronounce them unattainable). By reflecting what A.I. is all about and how goals and achievements relate to one another, one may aspire to keep A.I.´s hubris under control.

After this more general description of A.I., we will give an overview of the subsequent chapters. Although each chapter can stand on its own -- their contents being primarily extractions from published papers -- we sketch how their results could be integrated.

We shall characterize A.I. from a variety of perspectives which include the following:

- a short taxonomic description of A.I., presupposing an intuitive understanding of the topics mentioned;
- the perspective of its (potential) encompassing science Computer Science (called Informatics in the Netherlands);
- its relationships with more established disciplines;
- a look at the methodology and goals of A.I.;
- an overview of achievements and non-achievements of A.I.; and
- some speculations about future developments.

It is our aim that this chapter should be intelligible to everyone, but because of our obligation to specialists, we could not always avoid technical jargon.

1.1 Taxonomy of A.I.

For a short taxonomic description of A.I., we use the session names of the Sixth International Joint Conference On Artificial Intelligence, Tokyo 1979. This taxonomy is an end-of-the-70's snapshot of A.I. since no two A.I. conferences - they are biannual - have had the same collection of sessions. We grouped them into six categories:

-- vision
- image analysis
- region and edge detection
- shape and shading
- texture
- motion
- object detection

-- natural language
- processing
- dialogue and discourse
- question answering
- parsing
- semantics

-- problem solving
- program synthesis
- planning
- program understanding

-- deduction
- theorem proving
- deductive methods
- reasoning models

-- psychological A.I.
- cognitive psychology
- induction and learning

-- miscellaneous
- databases
- robotics
- distributed A.I.
- games
- representation
- architectures for A.I.

To illustrate the arbitrariness of this list, we mention some section names of the 1977 A.I. conference: knowledge acquisition, problem-solving and search, aids to programming, specialized systems, etc. The 1975 conference had sessions like: mathematical and theoretical aspects of A.I., speech understanding, A.I. software, etc. In section 1.5, we follow the latter taxonomy to discuss the achievement level of A.I.

1.2 A.I. as a Sub-Discipline of Computer Science

Investigating Computer Science in order to get a better grip on A.I. is asking for trouble since Computer Science itself is in turmoil. Computer Science resembles Radio Astronomy, a discipline that has emerged as a consequence of a technical invention [35]. Unlike Radio Astronomy, the content and scope of Computer Science is not yet stabilized. This birth from technology initially led to a preoccupation with logic gates and their electronic realization. By now those gadgets have disappeared from the mental luggage of most Computer Science specialists. As with this example, other issues have also appeared as though in need of close investigation but have subsequently drifted to the horizon. Compiler design is a case in point. Operating systems, communication networks and parallel processing are now *en vogue*. In spite of this ongoing fluidity some shapes in the Computer Science landscape are solidifying. The most notable have to do with the intrinsic properties of computers; thus leading to the cluster of automata theory, computability, formal languages, complexity theory, etc. The next cluster, somewhat vaguer in outline, concerns practical aspects, but is still remote from concrete applications. It contains such topics as: architecture, programming languages, algorithms, programming methodology, operating systems, data structures, input, output and file structures, etc. The third cluster is again more practically oriented but does not (yet) belong to other disciplines. It contains such topics as: process control, database management, system design, numerical and statistical algorithms, simulation and modelling, graphics, image processing, computer assisted instruction, etc.

Given this tripartition - for more sophisticated distinctions see e.g. [80] - the question arises whether A.I. actually belongs to this family and if so to which section. In [80] (1974), A.I. was subsumed under fundamental informatics, thus belonging to the first cluster. In [64] (1979), A.I. was thrown out completely and was considered to be a subclass of simulation activities practiced in the social sciences. We consider both choices to be incorrect. A.I. in 1980, using the characterization of the taxonomy above, has parts in common with cluster two and three, and also has subparts that

ultimately belong to other disciplines, some of which may temporarily be rejecting their legitimate off-spring. What complicates things are the strong centrifugal forces now working in A.I. The natural language section is spinning off under the banner of Computational Linguistics. The deduction section is starting to organize its own conferences and is considering setting up its own journal. Psychological A.I. began a Cognitive Science journal in 1977. The vision group has its own conferences, etc. Consequently, it does not make much sense to delineate precisely the relationship between Computer Science and what constitutes A.I. at this moment. One may expect the picture to change drastically within a few years. One may characterize A.I. therefore as a 'laboratory' somewhere between Computer Science and the other sciences, where new problems and new approaches to older problems emerge, which, when matured, drift off and reject their place of birth. So what holds A.I. together is not primarily its object of study, but the extreme tolerance for unorthodox research as long as complicated programs are involved.

1.3 A.I. and Other Sciences

Now let us look at the relationship of A.I. to other sciences. The above mentioned taxonomy already singles out sciences having a potential relationship with A.I.: Linguistics, Psychology, Logic and Mathematics are obvious candidates. Reality however is otherwise. For example there is a strong antagonism, to phrase it politely, between the Computational Linguistics section of A.I. and the hawks of Chomskyan linguistics. For an appalling fight between the two, see [26,88,27]. The Computational Linguists acknowledge the achievements of the Chomskyan Linguists but suggest the possibility to approach language with a different methodology, as outlined below. The Chomskyans hammer that there is only one scientific way to deal with language and they claim the exclusive rights to it. The Chomskyans, after making the language performance/ competence distinction, set out (here simplified deliberately) to describe the competence of a language user with (non-deterministic) generation rules, transformation rules and restriction rules for the grammar of the language. They, however, ignore the competence that should account for the content, the communicative purpose of language usage. They also

never get beyond the unit of a sentence. They frequently confuse the mathematical notion of the (non-deterministic) generative capacity of a grammar with the human capacity to generate sentences, which is certainly a purposeful, non-non-deterministic process [88]. In contrast, Computational Linguistics studies language from the perspective of communication and deals not only with sentences, but with text, stories and dialogues. Grammaticalness, the idol of Chomskyan Linguistics, is not worshipped, if only because non-grammatical sentences still can carry their communicative purpose. Although classical Linguistics clings to Chomskyan views, some (psycho-) linguists [6,43,59,86] do appreciate the study of language by making programs which actually understand text and/or generate sentences in communicative situations.

The relationship between A.I. and Psychology is much healthier. There is A.I. research where the results are expected to have psychological relevance and research where such relevance is not required. What allows the cooperation to be relaxed is that A.I. research usually starts with self observation for a good idea. So even when no psychological relevance is pursued a result can still be a first approximation to a theory that *does* have psychological relevance. The fact that introspection is the beginning of simulation programs ultimately bridges the gap between Behavioristic and pre-Behavioristic Psychology. At last a powerful method becomes available able to structure tentatively the inside of black boxes and pluck from the wealth of intuitions otherwise lying idle (although we hasten to add that their formalization is a painful process).

A relationship between the deduction section (theorem proving) on one hand, and Logic and Mathematics on the other, is virtually non-existent. They are simply not interested in each other. Remarkable is the fact that the tools of Logic, already in development for more than 2000 years, are inadequate for proving theorems in practice. In fact, this phenomenon is slumbering in mathematical circles but receives little attention. Logic has always been interested in the theoretical adequacy of sets of derivation rules and/ or logical axioms schemata. Pragmatical adequacy was never an issue at stake. Logic has still nothing to offer to day-to-day reasoning, as many

people loaded with expectations will have painfully experienced when starting a logic course. [1] Though the rule and not the exception, inconsistent collections of facts have always been avoided like the plague. (See e.g. the primitive attempts to deal with non-monotonic logic [79,25].)

A perhaps surprising relationship of A.I. with philosophy is extensively discussed in [78]. The author shows convincingly that age old problems disappear or are given a new approach, when mental phenomena become more accessible through simulation.

More generally, we believe that in the same way that most sciences in the last five decades, have undergone the process of mathematization - more accurately: formalization - they are now going through the revolution of computerization.

Take for example computer assisted instruction (= C.A.I.) subsumed above under Computer Science. Operational C.A.I. programs - like the PLATO-program - are produced by Computer Science specialists. Somewhat more sophisticated C.A.I. was done in A.I. and resulted in the Sophie program [7]. It should be obvious however that ultimately C.A.I. should be nurtured by Psychology/ Pedagogy. At the moment there are no implementable theories about learning that match different goals with different techniques. There is not much insight into how to build a user model from the user responses as well as a lack of mature natural language interfaces. In particular, there is a lack of insight into the wealth of discourse patterns. Consequently, C.A.I. programs are conceived by the ´technicians´ of Computer Science, who rely on the crudest principles.

----

1) In contrast to the contribution of logic to a better foundation of mathematics.

We know that in making this assertion we will step on many toes, but we feel that a sizeable part of science will be reincarnated in the Procrustean, nuts and bolts environment of Computer Science, under tension from outsiders.

1.4 Methodology of A.I.

´Time´ may be likened to a suppressed minority in Western scientific thinking. Logicians were nearly succesful in eradicating the temporal meaning in ´if ... then´ and ´and´. The question of when a modus ponens *operation* (or any other derivation rule) should be performed is always left to the discretion of the user. The ´possible worlds´ concept, a recent acquisition to the logicians tool box, is fairly clumsy for dealing with time related issues. A physical process is often best described by using a many dimensional space where, coincidentally, one of the axes stands for time. Complex phenomena in time: history, music, thinking, walking, etc. have always been relegated to the future or, at best, have received marginal attention.

Simulation is the method to study the behavior and result of many interacting operations. Computers, in the role of symbol manipulators, have cleared the way for handling those problems which are not amenable to direct simulation. When the duality of states and changes/actions is acknowledged, and we admit that time has always been projected outward, or captured in a state-framework, then we may foresee the addition of executable algorithms/procedures augmenting considerably our capacity to describe reality.

Of course, we exaggerated the lack of attention for the temporal dimension of reality. Indeed, mathematical modelling techniques do exist which capture regularities of changes and which can even be employed for making sophisticated predictions. Yet we believe that the tools offered by Logic/Mathematics are too coarse to describe complex processes.

Instead of merely looking at reality and trying to extract laws describing the behavior of phenomena - actually the job of an empirist - we are now able to *generate* complex behavior. This generation is under control, can be repeated, can be slowed down, can be tuned and can be made part of an ever-expanding generation procedure. Each single transition can be ´classically´ grasped; but when there are long chains of transitions, when the repertoire from which each transition can emerge is large, and when arbitrary cross- or self-recursions are allowed, the total behavior cannot be ´classically´ supervised any longer.

The basic assumption of this game is that when a program behaves - according to a sizable set of input-output pairs - like a process to be explained, the program embodies a first-order approximation of the mechanisms underlying the process. Observe that one has stretched what counts as an explanation. The unwillingness of the spokesmen for classical Chomskyan Linguistics [26,27] to accept the contribution of natural language handling programs to the understanding of language, is rooted in their more restricted view of explanations and their self-restriction to ignore the performance dimension.

Other people, however, are so enthousiastic about algorithms/ procedures for describing complex phenomena that they overshoot and mix things up. Johnson-Laird, a psycho-linguist, describes in [43] the compiler metaphor for processing natural language:

-- a programmer ´utters´ a program in a computer language (ALGOL, FORTRAN, etc), a compiler translates it such that the semantics are preserved in machine excutable code;

-- a speaker utters a sentence in natural language, the hearer activates a procedure (in A.I. called a parser), which translates the sentence into executable code, to be executed subsequently.

First of all, we question the psychological soundness of following this metaphor to the extent that the output of the parser always has the form of a program. Certainly, processing a declarative sentence may require, after having it parsed, an action, for instance a store in the memory, but it is not necessary that the output of the parser be executable code in order to have this action performed.

Secondly, we wonder why Johnson-Laird calls this ´procedural semantics´ for natural language. Also, when the output of the parser is always a procedure, when we acknowledge that the parser is a procedure, and when we allow that the parser may invoke lexical entries also having a mini-procedure form, then we still would not like to label this ´procedural semantics´ for natural language. Instead we are perfectly happy to see it labeled as a procedural account of the *understanding process*. We will illustrate Laird-Johnson´s sloppy usage of this phrase with two quotations: "In order to provide a glimpse of procedural semantics *in action* ..." [emphasis added], "Finally, it should be emphasized that procedural semantics is more a methodology than a specific theory.". These category mistakes/slips are particularly painful when we recall that ´procedural semantics´ has a clearcut interpretation: the meaning of operations/ actions/ commands as meticulously developed for computer languages. Surprisingly, Johnson-Laird knew this as well. On the second page of his paper, he writes: "... procedural semantics deals with the meaning of procedures that computers are told to execute.". Consequently, in the context of a natural language, we would like to restrict ´procedural semantics´ to the description of utterances/ sentences that express actions/ operations/ changes/ etc. rather than apply it to the way how utterances/ sentences are processed.

The methodology of using algorithms/ procedures to describe processes leads to psychologically relevant results when the generated behavior conforms to additional observable characteristics of the simulated real-life process. Without these additional conditions, this methodology is the heart of performance goals. In both cases this methodology has an engineering flavour. Sloman (in [78], page 16) wraps it all up with: (the title of his book is "The Computer Revolution in Philosophy: Philosophy, Science and Models of Mind")

> One of the main themes of the revolution is that the pure scientist needs to behave like an engineer: designing and testing *working* theories. The more complex the process studied, the closer the two must become. Pure and applied science merge. And philosophers need to join in. [emphasis as quoted.]

1.5 Achievements and Non-Achievements of A.I.

We will give a rough sketch here what is going on in A.I. these days (end of 1979) to give some insight into the current achievement level. We follow the IJCAI79 partitioning as presented above.

1.5.1 Vision

To deal with inflowing visual data, there are two strategies available. One prescribes combining data components until some meaningful pattern emerges. The other strategy starts out with a group of patterns and checks the patterns one at the time to determine whether the data fits them. The former strategy is called data-driven or bottom-up, the latter is called expectation-driven or top-down. Obviously, a more effective strategy would be a combination of the two, each technique coming into action at the right moment. According to the overview given by D. Marr [55], most research in visual information processing deals with the development of data-driven procedures. He distinguishes three transitions: (a) from the raw image to a so-called primal sketch, in which intensity changes are described, and distinguished locations are labeled; (b) to a so-called 2 1/2 dimensional sketch, which represents contours of surface discontinuity, depth and orientation of visible surface elements, all combined in a coordinate frame that is centered on the viewer; (c) ending in a 3-dimensional model representation with shape description that includes volumetric shape primitives of a variety of sizes, projected onto an object-centered coordinate system. The first two transitions are so to speak, under control. The last one - as far as we can judge - is hampered by a lack of insight into the data representation of the 3-dimensional model. This omission is also responsible for the impossibility of setting up expectation-driven procedures. The role -- or even the existence -- of an iconic-memory (and similarly for olfactic, auditive, tactile, etc. memories) in this type of data processing remains thoroughly unclear.

With the exception of one paper at the IJCAI79 conference, all contributions concerning images dealt with input processing. The one that discussed the generation of images described a running program which was even claimed to produce art. The claim was substantiated by a referral to the remarks of the public at the DOCUMENTA 6 exhibition, 1977, Kassel, Germany, and of those attending a five-month exhibition at the prestigeous (term used by the author) Stedelijk Museum in Amsterdam, where the program continuously generated line drawings. No component of this program might be identified as something akin to an iconic memory. Still, the viewers claimed to recognize image fragments as referring to the real world. He even reports: "Some of the viewers, who knew my work from my pre-computing, European, days claimed that they could ´recognize my hand´ in the new drawings." We feel that this phenomenum resides in the human urge to make sense amidst chaos and cannot be fully attributed to the program. Although we admit that making such programs must be a lot of fun, we fail to see how they contribute to the general problem of image processing (input, output as well as internal representation).

In spite of the rudimentary state of the knowledge representation for image input processing, some applications do exist in industrial environments. A description is given in [87] of a vision-based robot system capable of picking up parts randomly placed on a moving conveyor belt. A built-in training component - sensitive for a wide class of complex curved parts - alleviates the reprogramming task when new objects need to be recognized.

1.5.2 Natural Language

Before discussing A.I.´s attempts at natural language understanding we want to make it clear that natural language is a fairly clumsy tool for inter-human communication. Many prerequisites must be fulfilled, such as common cultural background, ages not too far apart, similar general goals and expectations, etc., before there will be a reasonable chance that inter-human natural language communication can be fully successful. Only against this background is it possible to discuss the nearly complete failure of current programs to interact in *unrestricted* natural language.

Suppose we feed a program the following mini-story:

John is hungry.

He fetches his purse.

Every program, provided its lexicon contains the words from the story, can handle the syntax of these two sentences. Even the anaphoric references ´he´ and ´his´ could be deciphered with a ´hack´: pick the most recent entity of the right type. Most programs could do processing which has a semantic flavour. Thus, they can answer correctly questions like: Is John a man? Who owns the purse? etc. One would be very impressed when a program would answer the question ´Why does John fetch the purse?´ with ´To eat it.´. Even more impressive would be the answer ´To eat its contents.´. There is a program [76] provided with a restaurant-script, which may rush to answer ´To pay the bill´, instead of pondering the possibility that John is heading for a supermarket (or the purse containing chewing-gum).

It should be obvious after this simple example that more complex natural language processing - stories with interacting participants, or man-machine interaction involving shifting initiative; stories with different roles, or with a broad range of topics, etc. - are out of the question. In fact, the failure of a program to do this kind of processing does not arise from its lack of insight into natural language, but in its being unable to evoke appropriate cognitive common-sense processes, like deduction, plausible reasoning, planning, plan recognition, etc.. One is led to suspect that current natural language processing programs know *too much* of the language and try to solve upcoming semantic/ pragmatic problems with inappropriate linguistic knowledge.

In spite of all the above mentioned shortcomings, tools have been developed that are of practical significance. For example, when a certain task domain is well understood and requires an interactive component, it will be easy to develop a sub-natural language pre-processor, by using the special purpose ATN-language [90,91,92]. Several actual applications, making data bases of realistic size more accessible, were reported in [40]. It is even claimed that the reported package requires only minimal adaptation and installation effort in new environments.

Most promising research directions, in our view, are those that break away from natural language proper by regarding natural language usage as a special case of goal-directed behavior while trying to formalize - first in dialogue contexts - the notion of speech acts. An example reported in [2] describes a program which plays the role of a clerk at a train station information booth.

1.5.3 Problem Solving

Problem solving encompasses the most consolidated area of A.I., namely search. This area has been investigated profoundly. Abstract search algorithms have been developed, which work whenever a problem can be represented as a sequence of states on which operators apply. Many techniques such as plan generation, algorithm generation, deduction, etc., ultimately boil down to special cases of search. Its generality is also its weakness: its capability always to find a solution often presupposes irrealistic computational resources. Still, search is important for theoretical reasons, because it unifies many individual algorithms/ techniques.

Most contexts in which problem solving has been experimented with, have been static mini-worlds, static in the sense that only the problem solving program could make changes. Since the mini-worlds have largely been simulated, complete information about the initial situation is assured. Examples are programs which can solve indefinite integrals, can deal with 8/15/24-puzzle configurations, can manipulate in a block-world, and which can generate simple sorting-algorithms.

The power of problem solving programs is crucially dependent on the availability of a heuristic function (or of procedurally built-in knowledge). Such a (strong) heuristic function is also a theoretical weakness, in as much as its format is ad hoc. It is composed of feature detecting functions implemented as some arbitrary programs. Here again, we encounter the problem of representing knowledge such that many different cognitive operations - here hypothetical reasoning - can be supported.

Given this state of affairs, one will not be surprised that problem solving by A.I. programs in a dynamic world and/or with incomplete initial knowledge is not yet too impressive. One project resulted in the design of a simulated taxi-driver in a simulated city, having to go from one location to another, confronted with simulated interruptions like red lights, pedestrians and road blocks [56,57]. Theoretical discussions, concentrating on the question of how to make a model resistant to uncertainty, and how to represent small changes, can be found in [25,51,83]. (Ten years ago these issues were known as the ´frame problem´ [58]; these days one talks about ´truth maintenance with a non-monotonic logic´.)

When problem solving requires dealing (buying, threatening, bribing, commanding, requesting, etc.) with other autonomous intelligent entities, whether human or robots, then we know only of one relevant project [9,11] (ignoring game playing programs). This study resulted in the POLITICS program. It closely resembles an old design of Abelson [1] which permits dynamic plan adaptation and is capable to replan in obstructive and counterplanning situations.

A well-known application of problem solving research is the above mentioned indefinite integration program, MATHLAB. This program has been extended with many other useful mathematical symbolic operations, and is widely available over the ARPA-network. A potential application travel-schedulling, provided one could easily handle the frequent changes of time tables [42].

1.5.4 Deduction

Like problem solving, deduction has important theoretical results. Many algorithms are known and have been experimented with. Every algorithm (or, certainly, almost every algorithm) is complete, i.e. provided with unlimited resources, algorithms can recognize every valid formula from the first order predicate calculus (its expressive power is sufficient for capturing virtually all of mathematics, and presumably all of day-to-day conscious or subconscious deductive demands). The unlimited resources are, of course, here also the problem. Even easy theorems are frequently beyond the effective power

of current deductive machines.

Several other problems are dependent on deduction. Plan generation requires a theorem prover for checking whether an operator can be applied in a hypothetical situation. It is even possible to reformulate plan generation as a deduction problem [37]. The frame problem, however, requires the introduction of so many axioms, expressing explicitly which configurations are preserved when an operator is applied, that this reformulation has no practical significance. People involved with program verification have already introduced the notion of an ´oracle´ as a deductive component in their systems. Such oracles are mostly implemented as an interactive request to the user. A notable exception is promised by the work of Boyer & Moore [4]. Their theorem prover is acquinted with an impressive amount of knowledge about recursive functions, has a rich vocabulary and has access to a data base containing lemmas and instructions as to their use. The authors intend to apply this machinery to the verification of fair-sized FORTRAN programs.

Almost all research on deduction with the resolution technique concerns refining the resolution rule by syntactic means such that generation of instantiations of already available formulas can be prevented. Even when no such spurious formulas are generated, there is still no hope that significant proofs would be found, since the generative power of the resolution rule is immense, and creates unmanageable amounts of data. In general, preventing the generation of garbage is a strategy implausible to success. Deduction, as done by humans, relies on knowledge leading to models, counter-examples, analogical reasoning, hypothesis generation, etc. and various deductive mechanisms, all gracefully cooperating. They are really not available in current programs.

In spite of twenty years of research investment in this branch of A.I., we consider the plane geometry prover [36], written at the end of the fifties, as the best example of how deductive programs should be designed. In contrast with the bulk of later programs, it contained a combination of two techniques: derivation rules and a model for checking the truth-value of formulas. The model allowed to

reject immediately over 99% of subgoals generated by the derivation rules.

For further discussion about deduction see section 1 of chapter 4.

1.5.5 Psychology related A.I.

As argued above all of A.I. is potentially relevant for the description/ simulation of natural intelligence. What goes under "Psychological A.I.", however, is limited to two foci of interest: formalization of intentional behavior, and learning.

The PARRY program, of which there are several incarnations [32], is a full fledged simulation of a personality, a paranoid patient, able to converse about a limited number of topics. An urge to realize primary goals sets up first-order intentions. When their realization somehow gets blocked (frustrated), they will lead to imbalanced affective parameters and a recovery action will have to be initiated, setting up second-order intentions, etc. A more sophisticated model of intentions such as this is implemented in PARRY3 [30,31].

Personality traits label different behavioral patterns, exhibited by different individuals in identical situations (corrected for contextual differences). Their explanation is attempted in [9,10] where it is postulated that different goal hierarchies are associated with different traits. A frequently occurring deviation from a (culture dependent) typical goal hierarchy, which causes a significant behavioral difference, will be codified as a personality trait. Knowledge about such deviations, for instance the ability to set up behavioral predictions when traits are mentioned, is a prerequisite for understanding stories. Different goal hierarchies, hence different traits, may not only generate different behavior - when they are procedurally used - but also may cause different interpretation of events/ stories - when the same hierarchy is interpretatively used. Such goal hierarchies together with planning/ counterplanning strategies are implemented in the already mentioned POLITICS program and demonstrate subjective understanding of simple natural language

accounts of international political conflicts.

PARRY and POLITICS are, by the way, good illustrations of the viewpoint that natural language should be attacked along a ´detour´ after we have first acquired a profound insight into cognitive procedures involved.

Learning was for quite a while a controversial topic in A.I. Proposals for increasing the intelligence of programs through learning had to be firmly rejected when the learning was more ambitious than the tuning of already predesigned parameters of built-in functions. Only after we have acquired more insight into the ´space´/ knowledge representation formalism in which non-trivial learned ´objects´/ structures should fit, can the issue of learning be reconsidered.

The following distinctions can be made along the method-dimension:

-- learning by being ´spoon fed´ (loading a program into a computer is an extremely surgical educational act);
-- learning by being told;
-- learning by teaching, thus providing a sequence of problems of increasing difficulty;
-- learning by self-discovery.

Along the result-dimension one may distinguish [69]:

-- rote learning, input of raw data;
-- parameter tuning;
-- method (plan, algorithm, strategy, ...) learning;
-- concept learning.

Although the teaching method for learning is also not clearcut, A.I.´s interest lies in self-discovery learning. Samuel´s checker program [75] shows that parameter learning is under reasonable control. Simple method learning, by generalization of constructed plans, was demonstrated by the STRIPS problem solver [33]. Automatic programming, self-discovery of algorithms, from input-output specifications, or from input-output examples, still only leads to toy-algorithms [3].

The program BACON [47] can construct invariants from tables with numerical data and was able to ´rediscover´:
the ideal gas law: $PV/NT = k_1$,

Kepler´s third law: $d^3(a - k_2t)^2 = k_3$,

Coulomb´s law: $Fd^2/q1q2 = k_4$,

Galileo´s law: $dP^2/Lt^2 = k_5$ and

Ohm´s law: $Td^2/(1c - k_6c) = k_7$.

An early result in concept formation by teaching was done by Winston [89]. He could ´educate´ the concept ´ARCH´ by giving examples and near-misses. Self-discovery of concepts is ascribed to Lenat´s program AM [49], which can generate concepts from elementary number theory. The relevance of this work is difficult to ascertain as a consequence of the generality of the built in meta-concepts on which the generation was based.

In any case, self-discovery of complex concepts, together with intricate coupling of declarative and procedural aspects is far from being solved.

1.5.6 Miscellaneous Problems

We mention here topics that do not fit into the already mentioned categories but are equally vital.

Intelligent data bases, robotics and games, respectively have a great potential for application, a great layman's appeal and (especially chess) a great debt to A.I. since they contributed less than the credit they got. [2] A more interesting hardware development is the design of architectures supporting A.I. languages. Several hardware LISP-machines are already in operation; the Japanese especially are coming up fast. (The MIT machine with disk, software, etc. sells for a bargain $80,000.) Coupling of and cooperation between several mini-computers have been realized for the HEARSAY project [50], capable of understanding coherent spoken English -- in a limited domain, with a vocabulary of about 1000 words. [3] These days many people feel that most problems of A.I. converge on one issue: knowledge representation. Frequently, research reports contain the conclusion that more knowledge should be effectively accessible to obtain a better performance as well as to break an improvement barrier. Past research has shown that general knowledge representation design has not been done by those who felt the greatest need for it. Their efforts bear the limitations of their specialization, since a sound knowledge representation has to support many cognitive operations rather than just one. The issue has been dealt with,

----

2) Most disappointing is the development of special hardware for chess programs. Having available a match box which can generate a legal chess move in the nano-second range, does not advance an inch towards the defeat by a program of the world chess champion. Recently, interest has been growing in the design of Go-programs. We have not discerned that they build on top of 20 years of chess programming experience.

3) Each mini-computer contained a 'specialist' responsible for one aspect of the understanding process (phoneme analyser, word recognition, syntax checker, semantic checker, etc). Each specialist had access to a central 'blackbord' where requests, hypothesis and solutions to sub-problems could be read and written. This blackbord, residing also in a mini the 'manager', was the sole communication channel for the specialists.

however, from another angle, the design of structures originating from general, minimal and intuitively obvious requirements. Currently the most applauded knowledge representation scheme was designed by Brachman [5]. He succeeded in cleaning up ambiguities, the ´ISA-link´ being a notorious example, which abounded in prior semantic-networks. Nonetheless, there is something seriously wrong with his formalism. Nowhere does he define its scope, the kind of knowledge it is supposed to handle and, more importantly, its raison d´etre. Its ability to interface effectively with cognitive procedures is left to the imagination of the reader as witnessed by a recent in-depth study which has revealed that several of the ´links´ in his formalism are insufficiently defined. Worrysome is the way how he introduces links. Just a few examples are given to justify them. When all is said and done, the reader still wonders whether or not the introduced links actually form a complete set.

Clearly, these two extreme groups of people - those designing knowledge based cognitive procedures and those designing the moulds in which knowledge can be poured - still have a long way to go before meeting each other.

1.6 A.I. in the Future

Before attempting some predictions we refer briefly the predictions obtained by a Delphi-study under A.I. specialists, published in 1973 [34]. Below is a part of a table from that report containing predictions up to the year 2010. This excerpt covers the entries only with median prototype dates up to 1980.

Product	Median prototype date	Median commercial date
Automatic identification system	1976	1980
Automatic diagnostician	1977	1982
Industrial robot	1977	1980
Automated inquiry system	1978	1985
Personal biological model	1980	1985
Computer-controlled artificial organs	1980	1990
Voice response order-taker	1978	1983
Insightful weather analysis system	1980	1985
Universal game player	1980	1985

This table shows that the estimates were overly optimistic. Only the industrial robot - and still hardly to be called as sophisticated - can be encountered outside the laboratory. The other products are in a limited stage of development. For instance, an automatic diagnostician exists only for isolated fragments of medical knowledge. (To prevent horror fantasies: the conclusions worked out by such programs are not ment to be revealed to patients. Although these programs may extract information from patients, interactive reasoning is left to the physicians. Potentialy, these programs bridge the gap between medical frontier knowledge and the obsolete knowledge of individual physicians.)

Now that we have cleared ourselves of any pretentions to making infallible predictions, we are ready for a glimpse into the future.

Confronted with diminishing resources, our society must increase its productivity/ efficiency in order to maintain its standard of living (not to imply that we must adhere to the consumption level attained by the Western societies). We believe that currently available A.I. knowledge is sufficient for the effective increase of productivity (and for maintaining appalling levels of consumption). We have mentioned before the applicability of intelligent databases (databases equipped with simple natural language interfaces and/or simple inference capabilities). For instance, the job market, the housing market, the real estate market, second hand car market, etc.

can be made transparant by relatively simple intelligent databases, and ultimately, directly accessible to the public. We feel that public transportation can similarly be made more efficient by improving the match between supply and demand, or by ´preventing´ transportation (for instance with distributed offices, i.e. people working in their own home, at a terminal).

We also anticipate a great future for garbage collecting robots, given our continuing potential to generate it. It seems that cleaning up the mess of the Three Miles Island nuclear reactor accident awaits the availability of robots. Those robots will need to be robust since computer memories are also sensitive to radiation [93].

The further one looks into the future and/or the wider the scope of the considerations, the more tentative the speculations become. What follows is only one of many possible scenarios.

> Suppose Earth´s capacity to support humanity has been exceeded in this century and as a means of realizing/ maintaining the Western consumption level around the globe one has successfully diminished the world population. We expect that a shortage of labour necessary to maintain the diversity of tasks will have to be supplemented by robots.

The following scenario was taken from [78]:

> The state of the world gives little cause for optimism. Maybe the robots will be generous and allow us to inhabit asylums and reserves, where we shall be well cared-for and permitted to harm only other human beings, with no other weapons than clubs and stones, and perhaps the occasional neutron-bomb to control the population.

Humans possess the fascinating properties of self-improvement and self-consciousness. A.I. programs are not (yet) near these phenomena. Computer components/ chips, however, have been designed by computers, automatic programming is not impossible, and once in a while, self-reflection pops up in the literature as a design goal.

Will humanity find in intelligent machines a rival or a worthy companion? Doesn´t it depend on the view on our fellow men?

2. A.I. and this thesis

Somehow we managed in the former section to circumvent the Laoconian task of characterizing intelligence. Implicitly we suggested one dimension of it: a wide spectrum of capabilities all supporting purposeful - though as yet injected from the outside - behavior. Whether speed of goal attainment is another independent dimension is not even clear since broadness of the behavior spectrum may be inversely related to speed. Although the range of A.I.'s interest is nearly unmanagebly wide it obviously does not cover the entire range of intelligence. Visual imagery, motoric agility, social identification, decision making while incompletely informed or in paradoxical situations, etc. are abilities which are not (yet) studied in A.I. Finally, the major problem, also not even being attacked, is the integration of all the separate achievements. Thus there is a fair chance that the distinct results are yet no more than ad hoc.

The wide range of A.I. is reflected in this thesis. The chapters that follow concentrate on topics which at first sight may seem quite disparate. We will first give a brief overview of the different chapters and then mention some cross relations between them.

Chapter two deals with a particular incarnation of search techniques. We recall that a search technique can be used for finding a sequence of operations that will transform a given start state into a desired goal state. When the goal state is explicitly given, one may attempt to construct a plan by working (pseudo) simultaneous from both sides. This is precisely the topic of chapter two. Theoretical results concern the generalization of theorems which are known to hold for the uni-directional A^*-algorithm [63]. The main one says that when an employed heuristic function satisfies certain conditions, the algorithm will find a 'best' path, i.e. no shorter paths exist.

Search techniques by the way can be used to counterattack the stance that computers cannot really be intelligent since they are not creative, i.e. their outcomes have been (implicitly) built-in beforehand. Search techniques, however, can come up with brand new solutions to arbitrary complex problems. Certainly one could retort

that the space to be searched has to be specified by the programmer. Nevertheless it is fairly easy to formulate a very general space once and for all, covering any kind of well defined problem formulation, such that a program outfitted with such a data structure may be considered to have reached ´adulthood´ with respect to problem understanding (which does not imply that every solvable problem will in fact be resolved since resolution depends on effectively limiting the size of the search space for each particular problem).

Chapter three is a bit hybrid. It centers around a few specific substitution functions in the LISP program language. A destructive substitution function SUBSTAD is shown to be much faster than the ´classic´ function SUBST. Moreover, that an unification algorithm - the workhorse for theorem provers and pattern matchers - can benefit from SUBSTAD such that it is faster and consumes less free space than the corresponding unification algorithm with the substitution function SUBST. Due to the destructive property of SUBSTAD, which may cause complicated side effects, its correctness proof is a fullblown research project on its own. Several versions of SUBSTAD are proven to be correct (partially done automatically with the theorem prover described in chapter four, which contains the unification algorithm with SUBSTAD). Although the method for proving them is amenable to automatization and is theoretically adequate, one version of the SUBSTAD function shows that the method is far from being applicable on a large scale in practice. The bottle neck is surprisingly enough not the limitations of available deductive power but the limited expressive power of the predicate calculus used to formulate precisely how that particular version of SUBSTAD is supposed to work. While correctness proofs should eliminate errors in code we are faced with the paradox that the description of that particular SUBSTAD version requires (estimated) more than hunderd times more text than its code. We give some speculations about what might be done to remedy this situation.

Chapter four is about deduction and more particularly, about modules for theorem provers. In contrast with many other practitioners in the theorem proving community, we see a deduction program as made up of many cooperating, special purpose components which will not only

be fed with bare, minimally specified problems, but as well with the theory to which a problem at hand is belonging, with similar theories to provide the food for analogy reasoning, with models to guide control decisions, etc. The uniform approach that is pursued by most, as we see it, is the inheritance of the logician's preoccupation with logic systems per se and therefore remote from applicability in 'real life'. These systems have the feature that while one can reason *about* them and even prove properties *about* them (by hand), they are inadequate for *doing* deductions with them. In spite of these critical remarks, chapter four has the same stigma: properties are proven about deductive modules. A special case theorem prover which is claimed to recognize that a predicate calculus formula is a special case and/or an alphabetic variant of another is shown to be sound. A module which can decompose a predicate calculus formula into an equivalent conjunction (thus leading to subproblems easier to be handled) is shown to produce maximal decompositions. However, in addition to these theoretical results, we offer illustrative examples where these modules drastically simplify the task for the (blind) search component.

Cross-relations between these different topics are numerous. Theorem proving is a special case of search techniques. While in general operators modify the 'state of affairs' by rendering certain facts invalid and adding new facts, in theorem proving no old facts are removed but only new facts are added. At the other hand theorem proving may be necessary in plan formation. Testing for the applicability of an operator in a certain situation may require sophisticated theorem proving. Switching back again: theorem proving without a (global) plan amounts to blind search in rapidly expanding spaces. Proving a theorem having the special form of an equality may benefit from bi-directional search, modifying both sides of the equality.

Algorithm generation can be considered a generalization of plan formation. Not only linear operator sequences are permitted in the solution range, but also case distinctions, loops and recursions. Correctness proof techniques contribute to the algorithm generation problem because it requires the ability to give precise descriptions

of algorithm behavior. This links the material in chapter two and three. The link with chapter four should be obvious since correctness proofs depend on deductive power. The link in the other direction we mentioned already: we focus our attention in chapter three on a specific function which plays a crucial role in unification algorithms - the core procedure of theorem provers. Thus we have all the ingredients for a self-improving program: a theorem prover fitted out with a unification algorithm using the less sophisticated function SUBST can potentially replace its unification algorithm with a version using the more economical function SUBSTAD, after passing through a cyclic process of algorithm generation and correctness proof. This sketch of a self-improving program must not be taken too literally. The level on which the modification is supposed to occur is microscopic. One may not expect that an intelligent program has available a self-description of such a small grain size as given in this SUBST/ SUBSTAD replacement example. (One does not achieve insight into his own DNA-structure by self-observation, nor is he able to remodel his DNA-structure by an internal process.) Yet we maintain the general idea to be a realistic one.

BI-DIRECTIONAL HEURISTIC SEARCH

1. Introduction

Problem solving is considered to be a sub-discipline of A.I. In a sense, it is a Troyan horse since when this sub-discipline lived up to its name, it would swallow the rest of A.I. and ultimately all other sciences. As yet its practitioners are struggling with smaller issues. Problem solving is fairly fashionable because it has consolidated results, abstract algorithms about which even theorems have been proven, and fairly successful (toy) programs written.

The field of problem solving can be partitioned according to different criteria. One way of partitioning might be done by looking close at what constitutes the class of problems. One might distinguish fuzzy versus clear problems, essentially incompletely specifiable versus completely specifiable problems, specific case versus general problems, small versus infinitely large or practically infinitely large problems, decidable versus non decidable problems, one world (accumulative logic) versus many, non-compatible worlds problems, and so on.

Another way of partitioning takes into account the supposed result of the problem-solving activity. One might distinguish yes/no answers, collections of entities which satisfy criteria, plans (= a sequence of operators which, when executed, will fulfil a higher-order goal), conditional plans, algorithms (= a conditional plan augmented with loops and/or recursion, and always halting), procedures (= an algorithm possibly not always halting), analogies, general interesting concepts, interesting conjectures, etc.

Again another way of partitioning focusses on the techniques employed in problem-solving activity. Several approaches have been developed: means-end analysis [62], problem reduction [29] (where one aims to replace a problem P by "P1 and P2" such that each Pi can be solved independently of the other one), problem pseudo reduction [74,82] (where one also aims to replace a problem P by "P1 and P2" but without requiring that solving Pi will not affect a solution to the

other problem), deduction [85,71,46,61], trial and error [8], search [60], heuristic search [41], (postponement to next day, month, year, decade, ..., running away as fast as possible), etc.

All the distinctions above should not be taken too seriously. They are certainly not orthogonal or even exclusive. Problem-solving is a young discipline as yet far removed from the establisment of the equivalent to a periodic table of elements, in which classes of problems, types of outcomes and the conceivable techniques fit together nicely. Obviously, some kinds of problems, answers and techniques are made for each other. Clear problems, formulated in predicate calculus, requiring a yes/no answer can be attacked by deduction, or in fact by a whole range of deductive techniques (see chapter 4). Specific case problems, of a simple nature, where the solution is in the range of specific plans (of course, the solution is ultimately the outcome of the executed plan) may be handled by search. A taxonomy of problem-solving techniques is already beyond current insights. Deduction, for instance, can certainly be seen as search in a single world, accumulative logic space; search is a disciplined and systematic way of performing trials and testing for progress and errors; problem reduction as well as pseudo reduction have also been imbedded in search formalisms [20,12]. At the same time, plan-generation and also algorithm-generation -- at first sight belonging to the realm of search -- have been performed by deductive machinery [37].

In the sequel, we limit ourselves to search techniques. They apply to those problems which:

-- fit the state1-operator-state2 paradigm,

-- have a goal description in the form of a Start State to be transformed into an explicitly described, Desired State, or as a Start State and a testable decidable criterion of Desired States, and

-- possess a mechanism to decide whether two states are essentially equal or different.

Depth-first and breadth-first search are standard techniques. The former is easily implementable within a stack environment, the latter enjoys the property of always ending up with a shortest path to the solution. Both techniques are uni-directional searches. In case the goal is explicitly given, one has the options to do uni-directional search in either direction from Start to Goal State or the opposite way round, *or* working from both sides. We confine ourselves to the last option.

When working from both sides in a breadth-first manner, one may expect a considerable gain as a consequence of a reduced number of states that have to be visited in the search space (see fig 2.1).

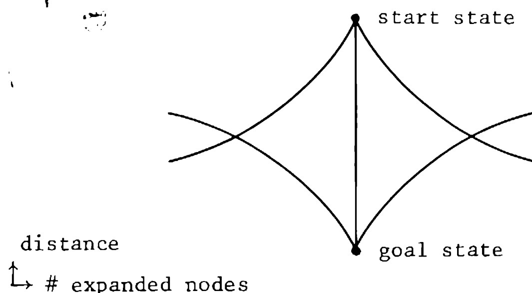


Fig. 2.1. When working from both sides in a breadth-first manner, less states will have to be visited.

A difficulty, however, is that the halting condition is more involved. Instead of checking whether a Frontier State of an uni-directional tree is a Goal State, one has to check whether a Frontier State of one tree is perhaps equivalent to any of the frontier states, at the opposite tree. With e.g. a hash coding trick on states, one may hope that the disadvantage of this more complicated halting condition does not offset the gain of having to visit lesser states.

Uni-directional search improves drastically when a heuristic is available, which allows one to estimate the distance between two states (in terms of the minimum number of operator applications necessary to transform one into the other). Such a heuristic permits to give more attention to promising states and may thus narrow the search tree. The method is often referred to as 'best-first' search.

In [66,67], a first attempt is described to provide a bi-directional algorithm with such a heuristic. This algorithm in fact performs two independent uni-directional searches, a forward search guided by the heuristic toward the Goal State and a backward search guided toward the Start State. The disadvantage is that in a search space where more than one path exists from start to goal, the two searches often proceed along two different paths, and so the two sets of visited states grow into nearly complete uni-directional trees before intersecting each other, see fig 2.2.

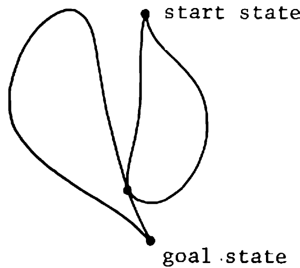


Fig. 2.2. The two search trees miss each other and do not meet in the 'middle' of the space.

In [17], we have described a second attempt. The algorithm presented needs an extensive revision [16] and will be reformulated in the next section. Section 3 is devoted to the generalization of the theorems known about the uni-directional heuristic A^*-algorithm. A worst-case analysis of the bi-directional algorithm will be given in section 4. The results of an implementation geared to the 15-puzzle and one example of the 24-puzzle will be given in section 5.

2. Bi-directional heuristic front-to-front algorithm

As mentioned above, the key disadvantage of Pohl's bi-directional algorithm is that solution components do not meet in the 'middle' of the search space. The reason for this behavior is that the path components are not directed toward each other. The forward path component is directed to the Goal State and the backward path component to the Start State. This fault was remedied in the bi-directional algorithm given in [17] by directing the forward path component to the *most promissing* state for which a path was constructed from the Goal State (and a similar process guiding towards the best state reachable from the Start State). The algorithm halted with a solution when at a certain iteration, a state was considered which had already been reached from the opposite side.

Recently, M. Taunton and T. B. Boffey of the University of Liverpool have convinced us that the algorithm as presented in [17] would not always end up with a shortest path as claimed under the proper condition. The halting condition as formulated was too "eager" and there was a bug in the theorem which claims that, when the heuristic used never overestimates the real distance between pairs of states, a solution found should be of minimal length. Fig. 2.3 shows a simple graph on which the algorithm of [17] will halt with the non-minimal path s-t of length 3 -- instead of finding the minimal path s-x-t of length 2 (while the heuristic is uniformly zero and backward and forward searches alternate).

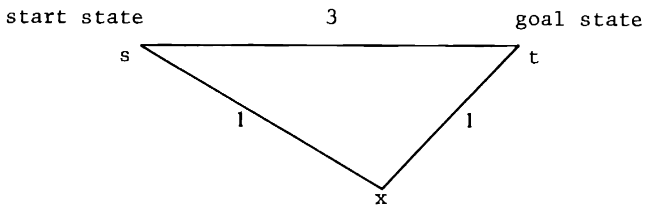


Fig. 2.3. Example of a graph on which a former bi-directional algorithm halts with the non-minimal path s-t of length 3.

We will now give a sketch of the Bi-directional Heuristic Front-to-Front Algorithm (BHFFA2) before presenting precise definitions and its specification. The BHFFA2 consists of two loops. When control resides in the upper loop, the situation is as sketched in fig. 2.4 or fig. 2.5.

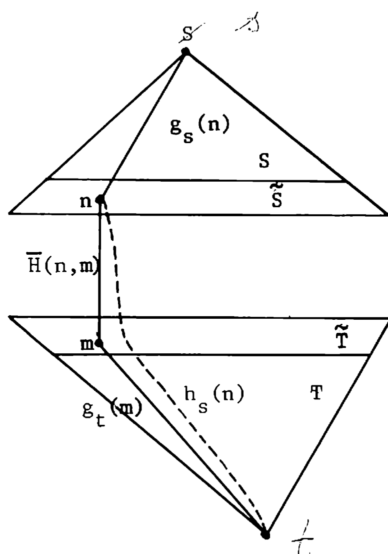


Fig. 2.4. The situation when control is in the upper loop and no path has yet been found.

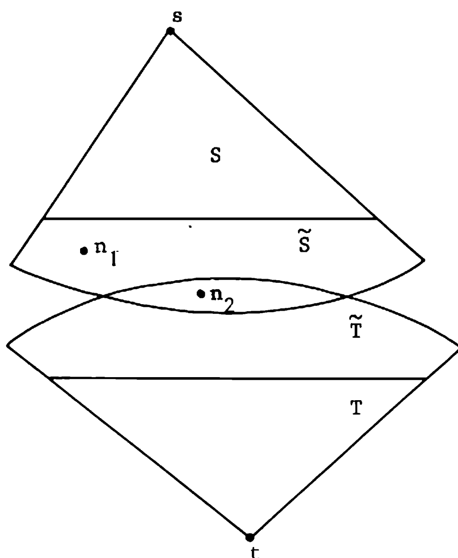


Fig. 2.5. The situation when control is in the upper loop and a path has been found since $\tilde{S}$ and $\tilde{T}$ intersect.

Here, s and t are the unique start and goal nodes respectively (node is a synonym for state), which have to be connected through a (shortest) path. The nodes in $S \cup \bar{S}$ and $T \cup \bar{T}$ are respectively reachable from s and t. Further S and T are nodes which have already been expanded (i.e. for which all states reachable in one step are known); $\bar{S}$ and $\bar{T}$ are nodes from which one is to be selected for expansion. In fig. 2.4, no path between s and t has yet been found since the intersection of $\bar{S}$ and $\bar{T}$ is empty. Expanding, say, node n in fig. 2.4 will either lead to the same configuration or, eventually, the situation in fig. 2.5 will arise. The selection of nodes is governed by a heuristic function $\bar{H}$ which estimates the real distance H. If one is not interested in the length of the solution path or $\bar{H} <=$ H, the precondition of Theorem 1 in the next section, does not hold, the proper halting condition is: halt in case the intersection of $\bar{S}$ and $\bar{T}$ is non-empty. (See, however, also the remark after the introduction of step (3.1) at the end of this section.) In the other case, the search has to continue as fig. 2.3 shows. When the next node to be expanded, say n1, is a member of $\bar{S}$ but not of $\bar{T}$, then control remains in the upper loop of BHFFA2. When the heuristic prescribes the selection of a node in the intersection of $\bar{S}$ and $\bar{T}$, say, n2 then control shifts to the lower loop, for which the situation is as sketched in fig. 2.6.

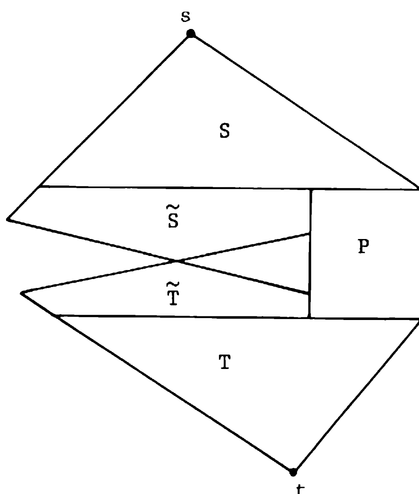


Fig. 2.6. The situation when control is in the lower loop.

Once control is in the lower loop, it stays there until the halting with a shortest path (under the upper-bound condition $\bar{H}$ <= H of Theorem 1). Here, P is the set of closed nodes which have emerged from open nodes in the intersection of $\bar{S}$ and $\bar{T}$ and thus lie on paths from s to t.

To make this presentation self-containing we will redefine the ingredients of BHFFA2 along the lines of [1]. Each node x (we exclude the start node and goal node for obvious reasons) which has been visited has associated with it:

-- one or two pointers $p_s(x)$ and $p_t(x)$, where a pointer $p_i(x)$ indicates a node lying on a path from x to the start node or goal node; and

-- one or two numbers gs(x) and gt(x), where a number gi(x) is an upperbound for the distance along $p_i(x)$ from x to the start node or

goal node, more precisely defined in the sequel.

Let us denote with

s the start node,

t the goal node,

S the collection of nodes reached from s which have been expanded and which do not belong to P,

T the same with respect to t,

$\overline{S}$ the collection of nodes which are neither in S nor in P but are direct successors of nodes in S or P,

$\overline{T}$ the same with respect to T,

P the collection of closed nodes which emerged from nodes in the intersection of $\overline{S}$ and $\overline{T}$,

H(x,y) the minimum distance between nodes x and y,

$\overline{H}$(x,y) a non-negative estimator of the distance between x and y with $\overleftarrow{\overline{H}}$(x,y) = $\overline{H}$(y,x),

gamma(x) the finite set of nodes obtained through applicable operators on x,

gamma(n;x) the set of nodes which are n steps from x,

gammai(x) like gamma(x) but with inverse operators instead,

gammai(n;x) like gamma(n;x),

l(n,x) the nonnegative edge length between n and x, for x in gamma(x) or in gammai(x),

gs(y) for y in S, $\overline{S}$ or P, the sum of l(y,p_s(y)) and the value which gs(p_s(y)) had when the gs(y) value was initialized or most recently updated,

gt(y) the same with respect to t for y in T, $\overline{T}$ or P,

hs(n) the minimum over y in $\overline{T}$ of ($\overleftarrow{\overline{H}}$(n,y) + gt(y)) for all n in $\overline{S}$,

ht(m) the minimum over y in $\overline{S}$ of ($\overline{H}$(m,y) + gs(y)) for all m in $\overline{T}$,

fs(n) = gs(n) + hs(n),

ft(m) = gt(m) + ht(m),

|P| the minimum over p in P of (gs(p) + gt(p)).

Now we give a definition of BHFFA2 interspersed with comment. The phrase "there is an x such that" (existential quantification) will be abbreviated by (E x).

As indicated above, BHFFA2 consists of two loops, FIND A PATH and FIND BEST PATH. Initially, control is in the FIND A PATH loop until control shifts permanently to the FIND BEST PATH loop. Both loops have as main actions:

-- the determination of the set of nodes promissing as to expansion;

-- a control decision concerning the shift to the other loop from the FIND A PATH loop, or halting in the BEST PATH loop; and

-- the expansion of a node.

A node will be expanded in the FIND A PATH loop only when it belongs to $\{\bar{S} - \bar{T}\} \cup \{\bar{T} - \bar{S}\}$. In the other loop, a node can be expanded when it belongs either to the same set or to $\bar{T} \cap \bar{S}$ (so, in fact, when a node belongs to $\bar{S} \cup \bar{T}$; the two cases lead to different actions in the BEST PATH loop, however). To prevent describing twice how a node in $\{\bar{S} - \bar{T}\} \cup \{\bar{T} - \bar{S}\}$ is expanded, we first characterize this as a subroutine, assuming forward search. Thus the node, n, to be expanded belongs to $\bar{S} - \bar{T}$.

```
EXPAND NODE n in S̄ - T̄.
    descendants(n):= gamma(n); S:= S U {n}; S̄:= S̄ - {n}.
    For each x in descendants(n) do:
    If x in P then CHECKgsx(P,{S̄,T̄}) else
    if x in S̄ then CHECKgsx(0,0) else
    if x in S then CHECKgsx(S,{S̄}) else
    [ S̄:= S̄ U {x}; provide a ps(n) pointer at x;
      store gs(n)+l(n,x) as gs-value at x ];
    continue with the next descendant of n.
```

The macro CHECKgsx can be explained by giving the expansion of CHECKgsx(P,{$\bar{S}$,$\bar{T}$}):

```
if gs(n)+l(n,x) < gs(x)
then [ redirect the ps(x) pointer to n;
       store gs(n)+l(n,x) as gs-value at x;
       P:= P - {x};
       S̄:= S̄ U {x};
       T̄:= T̄ U {x}].
```

Thus the first argument of CHECKgsx names the set for which a removal instruction has to be generated (we assume that the macro will not generate 0:= 0 - {x}), while the second argument contains a list of sets for which addition instructions have to be generated.

Remark: The first case ´x in P´ never holds in the FIND A PATH loop.

Expansion of a node in $\bar{S} \cap \bar{T}$ in the lower loop proceeds similarly and will also be described as a subroutine to simplify the description of the main algorithm.

```
EXPAND NODE n in S̄ ∩ T̄.
    P:= P U {n};
    S̄:= S̄ - {n}; T̄:= T̄ - {n};
    descendants(n):= gamma(n) U gammai(n).
    For each x in descendants(n) do:
    If x in P then CHECKgstx(P,{S̄,T̄}) else
    if x in S then
        [ CHECKgsx(0,0);
          S:= S - {x}; S̄:= S̄ U {x}; T̄:= T̄ U {x};
          provide a pt-pointer and store
          gt(n)+1(n,x) as gt-value at x ] else
    if x in T then
        [ CHECKgtx(0,0);
          T:= T - {x}; S̄:= S̄ U {x}; T̄:= T̄ U {x};
          provide a ps-pointer and store
          gs(n)+1(n,x) as gs-value at x ] else
    if x neither in S̄ nor in T̄ then
        [ S̄:= S̄ U {x}; T̄:= T̄ U {x};
          provide ps- and pt-pointers and
          gs- and gt-values at x ] else
    [ if x in S̄ then CHECKgsx(0,0)
      else { S̄:= S̄ U {x};
             provide a ps-pointer and gs-value at x };
      if x in T̄ then CHECKgtx(0,0)
      else { T̄:= T̄ U {x};
             provide a pt-pointer and gt-value at x } ];
    continue with the next descendant of n.
```

The macro CHECKgtx works like CHECKgsx but takes gt and p_t instead of gs *and* p_s. The macro CHECKgstx works like CHECKgsx and like CHECKgtx, thus whenever the gs- and/or gt-value needs to be updated then the appropriate action will be taken.

Now the stage is set for BHFFA2:

(1) INITIALIZATION.
$\overline{S}$:= {s}; $\overline{T}$:= {t}; P := S := T := 0; |P| := infinite.

(2) FIND A PATH LOOP which extends up to step (5).
If $\overline{S}$ = 0 or $\overline{T}$ = 0 then halt without a solution.
Decide to go forward, step (3), or backward, step (5). (E.g., go forward when the size of $\overline{S}$ is less than the size of $\overline{T}$. Every other decision procedure however is also allowed, e.g. going forward continuously.)

(3) FORWARD SEARCH; determine the subset of nodes from $\overline{S}$ which are plausible for expansion and try to postpone entering the lower loop (possibly by shifting to backward search).
aa:= min over x in $\overline{S}$ of fs(x).
A:= { x | x in $\overline{S}$ with fs(x) = aa }.
If (E n){ n in A - $\overline{T}$ }
then [let n be such a node and continue with step (4)].
If (E a)(E c){ a in A and c in $\overline{T}$ - $\overline{S}$ and
aa = gs(a) + $\overline{H}$(a,c) + gt(c) }
then [go backward, step (5), skip determining aa and A, and expand such a node c in $\overline{T}$ - $\overline{S}$].
Go to the FIND BEST PATH loop, step (10).

(4) EXPAND NODE n in $\overline{S}$ - $\overline{T}$. (see above)
Go to step (2).

(5) BACKWARD SEARCH.
Do step (3) through (4) with (s, S, $\overline{S}$, $\overline{T}$, gamma) replaced by (t, T, $\overline{T}$, $\overline{S}$, gammai); CHECKgsx should be replaced by CHECKgtx and vice versa.

As argued in the sequel, the following FIND BEST PATH component of BHFFA2 is only relevant when the $\overline{H}$ <= H condition holds.

(10) FIND BEST PATH LOOP.
If (E a){ a in A and gs(a) + gt(a) = aa } then halt with a path through a.
Select n in A with minimal bb:= gs(n) + gt(n).
|P|:= min(|P|, bb).

(11) EXPAND NODE n in $\overline{S} \cap \overline{T}$. (see above)

(12) DECIDE FORWARD2/BACKWARD2 EXPANSION
If $\overline{S}$ = 0 or $\overline{T}$ = 0 then
[halt with a shortest path through a node in P].
Decide to go forward2, step (13) or backward2, step (15); see the comment at step (2).

(13) FORWARD2 SEARCH.
aa:= min over x in $\overline{S}$ of fs(x).
If |P| <= aa then
[halt with a shortest path through a node in P].
A:= { x | x in $\overline{S}$ and aa = fs(x) }.
If (E n){ n in A - $\overline{T}$ } then
[let n be such a node and go to step (14)].
If (E a)(E c){ a in A and c in $\overline{T}$ - $\overline{S}$ and
aa = gs(a) + $\overline{H}$(a,c) + gt(c) }
then [go to backward2, step (15), skip determining aa and A and expand such a node c in $\overline{T}$ - $\overline{S}$].
Go to step (10).

(14) EXPAND NODE n in $\overline{S}$ - $\overline{T}$. (see above)
Go to step (12).

(15) BACKWARD2 SEARCH.
Do step (13) through (14) with (s, S, $\overline{S}$, $\overline{T}$, gamma) replaced by (t, T, $\overline{T}$, $\overline{S}$, gammai); CHECKgsx should be replaced by CHECKgtx and vice versa.

The following invariants hold: the sets S, T, $\overline{S}$, $\overline{T}$, P are all pairwise disjoint, with the exception of the pair $\overline{S}$, $\overline{T}$, which may intersect.

As already stated above, if one is not interested in the shortness of solution path or if $\overline{H}$ <= H does not hold, the lower loop, step (10) - step (16), can be eliminated and step (3) should be replaced by:

(3.1) FORWARD SEARCH.
If $\overline{S} \cap \overline{T}$ = 0 then [halt with a solution path].
Let n be a node for which fs(n) = min over x in $\overline{S}$ of fs(x) and continue with step (4).

If $\overline{H}$ <= H does not hold, it does no harm to use step (3) together with the lower loop. Halting will be postponed and, consequently, shorter paths may be found in the meantime.

Whether $\overline{H}$ <= H holds or not, one may replace step (3) for efficiency reasons by:

(3.2) FORWARD SEARCH.
Determine aa and A as in (3);
let n be in A with preference that n be also in $\overline{T}$;
if n in $\overline{T}$ then [go to the lower loop, step (10)]
else [continue with step (4)].

The advantage of using (3) instead of (3.2) is that execution of the upper loop is cheaper than execution of the lower loop, since P is not yet around. The advantage of (3.2) is that halting with an optimal path (shortest when $\overline{H}$ <= H) may occur earlier.

We will use the formulation with step (3) for the theoretical discussion in the next section. The results apply also when (3.2) is used instead.

3. Minimal path and optimality theorems for BHFFA2

Before proceeding with the minimal path theorem, we have to prove two lemmas that will be needed. They have to do with the properties of optimal paths holding before BHFFA2 halts.

LEMMA 1. *If* $\overline{H}(x,y) \le H(x,y)$ *and* q *is an optimal path from* s *to* t *then when control is in the upper loop, step (2)-(5), there exist open nodes* n *in* $\overline{S}$, m *in* $\overline{T}$ *on* q *with* $fs(n) \le H(s,t)$ *and* $ft(m) \le H(s,t)$.

The proof is similar to the proof of lemma 1 in [17].

PROOF. Let n be the first node on q, counted from s, with n in $\overline{S}$. Let m be the first node on q, counted from t, with m in $\overline{T}$. (They exist because otherwise all nodes on q would be closed and control would be in the lower loop.)

$fs(n) = gs(n) + hs(n)$ by definition of fs,

$= gs(n) + \overline{H}(n,y) + gt(y)$ for the y in $\overline{T}$ where hs realizes its minimum,

$\le gs(n) + \overline{H}(n,m) + gt(m)$ by definition of hs,

$\le gs(n) + H(n,m) + gt(m)$ precondition of lemma,

$= H(s,t)$ since we are on an optimal path.

$ft(m) \le H(s,t)$ is proved in the same way. <<

The next lemma is like lemma 1 but deals with the lower loop instead.

LEMMA 2. *If* $\overline{H}(x,y) \le H(x,y)$ *and* q *is an optimal path then when control is in the lower loop, step (10)-(15), either there exists open nodes* n *in* $\overline{S}$, m *in* $\overline{T}$ *on* q *with* $fs(n) \le H(s,t)$ *and* $ft(m) \le H(s,t)$ *or there is a node* n *on* q *with* n *in* P *and* $|P| = H(s,t) = gs(n) + gt(n)$ (thus the path q is already found).

PROOF. If all nodes on q are closed then one of them, say n, was the last one. Thus n is in P and the required properties hold since we are on an optimal path. If not every node is closed the argument of lemma 1 applies. <<

The phrasing of the minimal path theorem is the same as in [17], the proof is only slightly different.

<u>THEOREM 1</u>. *If* $\bar{H}(x,y) <= H(x,y)$, *if all edge labels are at least a positive* d, *and if there is at least one path between* s *and* t *then* BHFFA2 *halts with a shortest path between* s *and* t.

<u>PROOF</u>. Suppose theorem 1 does not hold. Then we have three cases: (1) BHFFA2 does not halt; (2) BHFFA2 halts without a solution path; (3) BHFFA2 halts without a shortest path.

Case 1: According to lemma 1, lemma 2 and the halting condition in step (13), only those nodes will be expanded which have f-values less than or equal to H(s,t). Consequently, their g-values are less than or equal to H(s,t). Thus BHFFA2 only expands nodes at most H(s,t)/d steps away from s or t, and this is a finite number. Let Ms and Mt be the sets of all nodes which are ever generated from s and t, respectively. As every node has only a finite number of successors, and the maximum number of steps any node is away from s and t is finite, both Ms and Mt can only contain a finite number of nodes, and so M = Ms U Mt is of finite size c. Let r_m be the (necessarily finite) maximum number of paths from s to m and from t to m for m in M, and let r be the maximum over all r_m. Then r is the maximum number of different times a node can be reopened. After r.c iterations of BHFFA2, all nodes are permanently closed. So $\bar{S}$ U $\bar{T}$ = 0 and the BHFFA2 halts, which produces a contradiction.

Case 2: If BHFFA2 halts without a solution then $\bar{S}$ = 0 or $\bar{T}$ = 0 and control must have been in step (2). Lemma 1 prohibits this however.

Case 3: The BHFFA2 can only halt with a solution in step (10), step (12) and in step (13), all in the lower loop.

Step (10): Thus we have aa > H(s,t). If step (10) was entered from the upper loop, we have an immediate contradiction, because lemma 1 prescribes that aa <= H(s,t). Otherwise, step (10) was entered from step (13). In case no optimal path has been found yet, lemma 2

prescribes that $aa <= H(s,t)$ should hold. In case an optimal path already crosses P, BHFFA2 would already have halted in step (13) with an optimal path since then $aa > |P| = H(s,t)$.

Step (12): Lemma 2 prescribes that when $\overline{S} = 0$ or $\overline{T} = 0$, an optimal path should cross P and would thus be selected.

Step (13): If an optimal path does not yet crosses P then according to lemma 2 $aa <= H(s,t) < |P|$. Contradiction. <<

To bring all the theoretical results together, we will restate and prove here the bi-directional version of the optimality theorem (which first appeared in [18]). Again, we first present two lemmas.

<u>LEMMA 3</u>. *If* $\overline{H}(x,y) <= H(x,y)$ *then for every node closed, coming from* $\overline{S}$ *(respectively* $\overline{T}$*),* $fs(n) <= H(s,t)$ *(and, respectively,* $ft(n) <= H(s,t)$*) holds.*

<u>PROOF</u>. Immediate consequence of lemma 1, lemma 2 and the $|P| <= aa$ condition in step (13). <<

<u>LEMMA 4</u>. *If* $\overline{H}(x,y) <= H(x,y)$ *and* $\overline{H}(x,z) <= \overline{H}(y,z) + H(x,y)$ *(the so-called consistency property of* $\overline{H}$*), then for every node* n *coming from* $\overline{S}$ *(respectively* $\overline{T}$*) which is closed, in the upper as well as in the lower loop, it is the case that* $gs(n) = H(s,n)$ *(and, respectively,* $gt(n) = H(t,n)$*).*

Consequently, when the preconditions of lemma 4 are fulfilled, all the checks '$gs(n) + l(n,x) < gs(x)$' which pertain to nodes in S or T may be inactivated from BHFFA2, since a shortest path to them is already found.

<u>PROOF</u>. Suppose the opposite, see fig 2.6A.

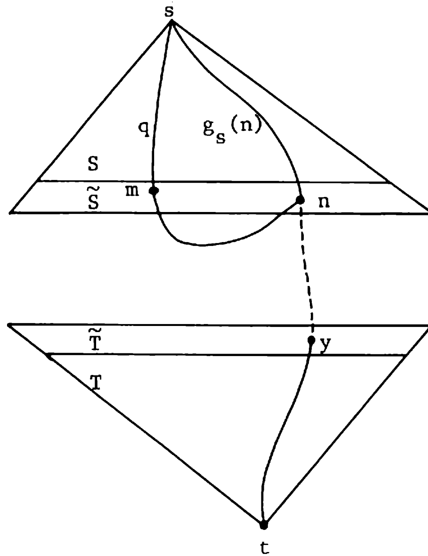


Fig. 2.6A. See the proof of lemma 4.

Let n be a node (from say $\overline{S}$) which will be closed and for which gs(n) > H(s,n). Let q be an optimal path from s to n. Let m be the first node on q in $\overline{S}$ (m exists, otherwise gs(n) would be equal to H(s,n)). fs(n) = gs(n) + $\overline{H}$(n,y) + gt(y) for some y in $\overline{T}$.

fs(m) = gs(m) + hs(m) by definition of fs,
<= gs(m) + $\overline{H}$(m,y) + gt(y) by definition of hs,
= H(s,m) + $\overline{H}$(m,y) + gt(y) because m is on the optimal path to s,
= H(s,m) + H(m,n) + $\overline{H}$(m,y) - H(m,n) + gt(y),
= H(s,n) + $\overline{H}$(m,y) - H(m,n) + gt(y),
< gs(n) + $\overline{H}$(m,y) - H(m,n) + gt(y) is our assumption,
<= gs(n) + $\overline{H}$(n,y) + gt(y) as a consequence of consistency,
= fs(n).

Contradiction because m would have been chosen for expansion instead of n. <<

Two heuristics can be compared when the estimates they produce differ in a uniform way; for instance, when one of the heuristics persistently estimates distances smaller than the other heuristic. The heuristic $\bar{B} \equiv 0$, which does not provide any information, underestimates consistently with respect to the 'maximally informed' real distance H. More generally $\bar{H}$ is a better heuristic than $\bar{B}$ when for all unequal pair of nodes x and y we have $\bar{B}(x,y) < \bar{H}(x,y) <= H(x,y)$.

The optimality theorem restricts the behavior of BHFFA2 with a 'good' heuristic $\bar{H}$ with respect to a 'bad' heuristic $\bar{B}$. Since the decision procedure for forward and backward search is not specified in BHFFA2, it should not be surprising that the decision procedure is mentioned when the set of nodes expanded by $\bar{H}$ and by $\bar{B}$ are related. The strategy is to show that the good heuristic will not expand more nodes than the bad heuristic.

THEOREM 2. *If two heuristics* $\bar{H}$ *and* $\bar{B}$ *are related by:*
$\bar{H}(x,y) <= H(x,y)$,
$\bar{H}(x,z) <= \bar{H}(y,z) + H(x,y)$ *(consistency of* $\bar{H}$*) and*
$\bar{B}(x,y) < \bar{H}(x,y)$ *for* $x \neq y$ *(which makes* $\bar{H}$ *a better heuristic than* $\bar{B}$*), and there is a solution, then for every decision procedure employed by* $\bar{B}$ *there is a decision procedure for* $\bar{H}$ *such that* $\bar{H}$ *will at most expand the nodes that will be expanded by* $\bar{B}$ *(which justifies the goodness of* $\bar{H}$*)*.

PROOF. Suppose the opposite. So there is at least one pair of nodes (n,m) in $\bar{S} \times \bar{T}$ for which it holds that $\bar{H}$ is forced to expand at least one node of such a pair while $\bar{B}$ will never expand them.
Let at iteration i_1, when $\bar{H}$ is used, n in $\bar{S}$ and m in $\bar{T}$ be the first pair of nodes for which this holds. Since (n,m) is the first pair of nodes, (n,m) will at some iteration i_2, when $\bar{B}$ is used, belong to $\bar{S}_B$ and $\bar{T}_B$, and they will remain in $\bar{S}_B$ and $\bar{T}_B$. (If they would not surface in $\bar{S}_B$ and $\bar{T}_B$ then their parents would not have been expanded by $\bar{B}$. Also it would hold for another pair that $\bar{H}$ has to expand one of them but $\bar{B}$ none of them, and that would occur at an iteration before i_1.)
Suppose n = m, thus $\bar{H}$ finds a path with length gs(n)+gt(m) = fs(n). No

shorter solution path exists, since lemma 1 or lemma 2 would prescribe the expansion of another node or to halt with such a shorter path. Consequently, $\bar{B}$ has to find another path with equal length, but before being allowed to halt with that path, $\bar{B}$ has to expand node n. Contradiction, thus n is unequal to m.

Denote the fs-values, using $\bar{B}$ on iteration j, by fBj and when $\bar{H}$ is used by fHj.

Denote the gs-value, using $\bar{B}$ on iteration j, by gsBj and when $\bar{H}$ is used by gsHj.

So we get on each iteration j when i_2 <= j:

fBj(n) = gsBj(n) + min over y in $\bar{T}_B$ of ($\bar{B}$(n,y) + gtBj(y)),

<= gsBj(n) + $\bar{B}$(n,m) + gtBj(m),

= gsHi$_1$(n) + $\bar{B}$(n,m) + gtHi$_1$(m), since according to lemma 4, gs(n) and gt(m) cannot improve anymore.

< gs(n) + $\bar{H}$(n,m) + gt(m), since n is unequal m,

= fHi$_1$(n).

$\bar{B}$ will stop, say on iteration k (with i_2 <= k, since otherwise (n,m) would not be the first pair of nodes, etc.) with a path of length H(s,t) and with an f-value equal to H(s,t). So H(s,t) <= fBk(n), because otherwise $\bar{B}$ would have expanded n.

Thus H(s,t) <= fBk(n) < fHi$_1$(n).

Contradiction with lemma 3. <<

4. Worst case analysis

A crude technique to compare different algorithms is to investigate how they behave in worst case circumstances. For a certain search space, we give formulas for the number of expanded nodes with the uni-directional algorithm, the bi-directional Pohl algorithm and with BHFFA2. We assume that the heuristic functions used will give a maximum error within *relative* bounds.

Let the search space be an undirected graph containing a countable collection of nodes; two nodes, the start and goal nodes, have m edges ($m > 1$), and there is a unique path of length K between them. From all other nodes emanate m+1 edges; there are no cycles; and all the edge lengths are one, see fig. 2.6B.

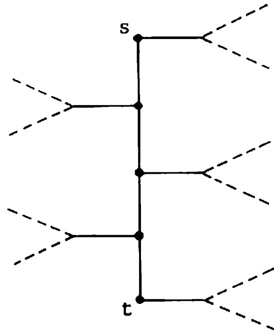


Fig. 2.6B. Example of a graph with m=3 and K=4.

So all nodes, except the start and the goal node, have m+1 successors, of which one is the direct ancestor. Since for the uni-directional, the bi-directional Pohl and BHFFA2 algorithms, the direct ancestor will be found in the set of closed nodes and will subsequently be ignored (in this space, there is only one g-value possible so that cannot be improved), we consider only the remaining m successors. From an uni-directional point of view, the space is a tree with branching rate m since the algorithm will not look beyond the goal node.

Due to the error of the heuristic function, nodes, which form side trees hanging off the solution path (=s.p), are expanded. The depth, n, of a side tree at node x on the s.p. in the uni-directional case depends on the distance, R, of x to the goal node; thus n=u(R) for some function u. Similarly in the case of Pohl's bi-directional algorithm, we get n=u(R), where R is the distance of x to the goal node or start node dependent of the side to which x belongs. Suppose that, in the case of BHFFA2, the heuristic forces complete expansion of a side tree at x before the successor of x on the s.p. will be expanded. Then the depth of a side tree is also a function n=u(R), where R is the distance of x to the opposite front.

At depth 1 off the s.p., there are m - 1 nodes, at depth i there are $m^{i-1}(m-1)$ nodes, and so the total number of nodes in one such side tree of depth u(R) is given by:

$$V_R = \sum_{i=0}^{u(R)-1} m^i(m-1)$$

$$= (m-1)(m^{u(R)}-1)/(m-1)$$

$$= m^{u(R)}-1.$$

If we denote by F[a] the total number of nodes erroneously expanded by algorithm a, we get the following results:

1. Uni-directional: R is the distance to the goal node, going from s to t on the s.p.; R decreases in steps of 1 from K till 1 and so

$$F[\text{uni-directional}] = \sum_{R=1}^{K} V_R$$

2. Bi-directional Pohl: For some node lying on the s.p. and expanded by the forward algorithm, R is the distance to the goal node, and this distance decreases in steps of 1 from K for the start node to K/2+1 for the intersection node. So the forward algorithm expands a total of

$$\sum_{R=K/2+1}^{K} V_R$$

nodes off the s.p.. The same number is expanded by the backward algorithm. (For convenience, we assume that for the bi-directional cases K is even and that forward and backward search is alternated; slight changes are required when K is odd.) So

$$F[\text{bi-directional Pohl}] = 2.\sum_{R=K/2+1}^{K} V_R.$$

3. BHFFA2: First s is expanded then t. Suppose a is the successor of s lying on the s.p. and b is the predecessor of t on the s.p. Then for the side tree of s, R=R(s,b)=K-1, and so the depth of the side tree hanging off s is u(K-1). The same goes for the tree hanging off t. When a and b are expanded after the side trees at s and t are completed, the distances from a and b to the opposite front are K-3, etc. So in this case, R is decreasing in steps of 2 from K-1 until 1 is reached; therefore the total number of nodes in all the side trees is given by

$$F[\text{BHFFA2}] = 2.\sum_{R=1}^{K/2} V_{2R-1}.$$

These results hold independently to the form of u(R). It is reasonable to assume that the smaller the real distance, the more accurate (or the less erroneous) will be the estimated distance. Thus we expect that the depth of the side tree hanging off a node on the s.p. will become smaller as R becomes smaller. In that case, u(R) is a monotonic function. Figures 2.7, 2.8 and 2.9 give an idea of the depths of the side trees (represented by the lengths of the bars) for all nodes on the solution path if this assumption holds.

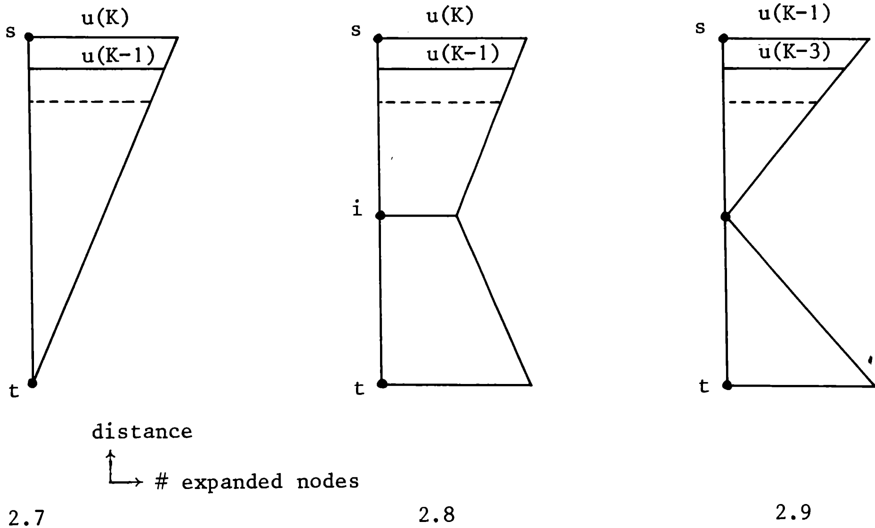


Fig. 2.7. Uni-directional. R decreases with steps of 1.
2.8. Bi-directional Pohl. Since at i, R is still K/2, n never gets very small.
2.9. BHFFA2. R decreases with steps of 2.

If R1<R2 implies u(R1)<u(R2), then also R1<R2 implies $V_{R1}<V_{R2}$ (V_R denotes the number of nodes in the side tree with depth R), and so

$$2 \cdot \sum_{R=1}^{K/2} V_{2R-1}$$

$$< \sum_{R=1}^{K/2} V_{2R-1} + \sum_{R=1}^{K/2} V_{2R}$$

$$= \sum_{R=1}^{K} V_R$$

$$< \sum_{R=K/2+1}^{K} V_R + \sum_{R=K/2+1}^{K} V_R$$

or F[BHFFA2] < F[uni-directional] < F[bi-directional Pohl].

Now we show that u(R) is indeed a monotonic function if we assume that the heuristic $\bar{H}$ gives a maximum error within relative bounds. We have studied two heuristic functions. Let $d > 0$:

1) $\bar{H}(n,m) = H(n,m)/(1+d)$ if at least one of n and m is not on the s.p; $\bar{H}(n,m) = H(n,m)$ when n and m are both on the s.p.; thus $\bar{H}$ only under estimates H;
2) $\bar{H}(n,m)$ is as above if at least one of n and m is not on the s.p.; $\bar{H}(n,m) = H(n,m).(1+d)$ when n and m are both on the s.p.; thus this $\bar{H}$ is more erroneous than the former one.

We give proofs and derive formulas for the more erroneous heuristic. Only slight changes are necessary for the other heuristic. Some formulas for the other heuristic will be given but without their derivation.

We first deal with the monotonicity in the uni-directional algorithm and the bi-directional Pohl algorithm (see fig 2.10).

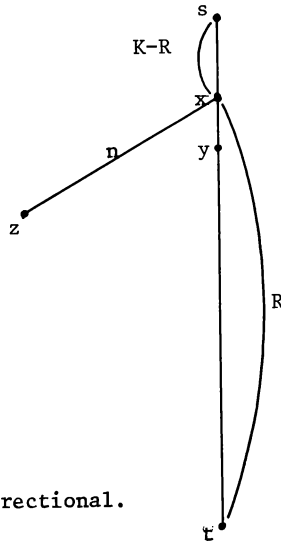


Fig. 2.10. H(s,t) = K, uni-directional.

If x on s.p., y on s.p., y in gamma(x), and z not on s.p. but n steps away from x, then z will be expanded iff

$g(z)+h(z) <= g(y)+h(y)$, or

$K-R+n+(R+n)/(1+d) <= K-R+1+(R-1)(1+d)$, or

$n(1+1/(1+d)) <= -R/(1+d)+(R-1)(1+d)+1$, or

$n(1+d+1) <= -R+(1+d)+(R-1)(1+d)^2$, or

$$n(2+d) <= -R+1+d+R+2Rd+Rd^2-1-2d-d^2, \text{ or}$$
$$n(2+d) <= Rd(2+d)-d(1+d), \text{ or}$$
$$n <= Rd-d(d+1)/(d+2), \text{ or}$$
$n <= INT\{Rd-d(d+1)/(d+2)\}$, if $INT\{x\}$ denotes the largest integer smaller than or equal to x. And so we see that in this case the monotonicity condition holds. { A similar derivation for the less erroneous heuristic yields: $n <= INT\{Rd/(2+d)\}$. }

Now we deal with the monotonicity in case of BHFFA2. When BHFFA2 expands nodes coming alternatively from $\bar{S}$ and $\bar{T}$, and when it uses the heuristic $\bar{H}(x,y)=H(x,y)(1+d)$ with both x and y are on the s.p., and $\bar{H}(x,y)=H(x,y)/(1+d)$ in all other cases, then
(a) when a node x, x on s.p., x in $\bar{S}$, is expanded, it realizes its minimum in a node y on s.p., y in $\bar{T}$;
(b) whenever a node x in $\bar{S}$, x on s.p., is expanded, then at the next iteration a node y in $\bar{T}$, y on s.p., will be expanded with $H(t,y)=H(s,x)$;
(c) a node in the side tree hanging off x on s.p. will always realize its minimum in a node y', y' in gammai(y), y on s.p., y' on s.p., and a node in the side tree of y on s.p. will always realize its minimum in a node x' in gamma(x), x on s.p., x' on s.p.;
(d) after expansion of x' in gamma(x), x and x' on s.p., and y' in gammai(y), y and y' on s.p., no other nodes in the side trees of x and y will be expanded.

(a), (b) and (c) will be proved by induction. First, we check the statements for s and t. (a) at the moment s is expanded, it realizes its minimum in t; (b) s and t are expanded immediately after each other; (c) for x in gamma(n;s), y in gammai(n;t), s' in gamma(s), t' in gammai(t), s' and t' on s.p., x and y not on s.p.,

$$\begin{aligned} gs(x)+\bar{H}(x,t')+gt(t') &= n+(K-1+n)/(1+d)+1 \\ &< n+(K+n+n)/(1+d)+n \\ &= gs(x)+H(x,y)+gt(y). \end{aligned}$$

So x realizes its minimum in t' and not in y in gammai(n;t). The same holds for y with respect to s'.

Now we proceed with the induction steps (see fig. 2.11).

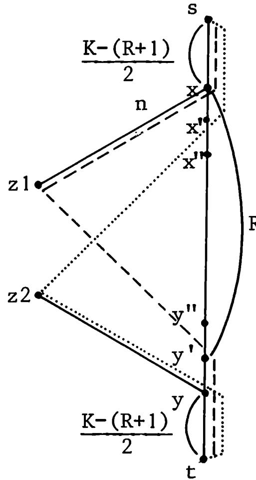


Fig. 2.11. BHFFA2. Dashed line is fs(z1), dotted line is ft(z2), both before expansion of x' and y'.

Suppose (a), (b) and (c) hold for nodes x and y, x in $\bar{S}$, y in $\bar{T}$, x and y on s.p.; then we prove that they also hold for x′ in gamma(x), x′ on s.p., and for y′ in gammai(y), y′ on s.p.

Let H(x,y′)=R, then H(s,x)=H(t,y)=[K-(R+1)]/2 (they are equal according to (b)). Inductive proofs for (a), (b), (c) and (d) follow.

(a) As the branching rate is constant, and as x and y were expanded immediately after each other, the side trees at x and y will grow in exactly the same way and so, for some nodes z1 in gamma(n;x) and z2 in gamma(n;y), at even iterations fs(z1)=ft(z2). Since, according to (c) they realize their minimum in y′ and x′ respectively expansion of z1 does not change the value of z2; descendants of z1 will have larger f-values than z1, and so z2 will be expanded next. Therefore, y′ will not be expanded as long as x′ is not expanded. Furthermore, x′ will

not be expanded as long as it realizes its minimum in some node in the side tree at y, because when this is the case, there is always some node z1 in the side tree at x with the same fs-value to y′ and then this z1 will be chosen for expansion instead of x′. (We had a worst case so ties are always resolved in the most unfavorable way.) So, at the moment x′ is expanded, it realizes its minimum in y′, and at the same moment y′ realizes its minimum in x′, and $fs(x')=ft(y')$ is the minimum of all distances between nodes from $\bar{S}$ and $\bar{T}$. <<

(b) At that iteration, $fs(x') < fs(z1) = ft(z2)$, or
$gs(x')+H(x',y')+gt(y') < gs(z1)+H(z1,y')+gt(y')$,
and thus

$$[K-(R+1)]/2+1+(R-1)(1+d)+[K-(R+1)]/2+1 <$$
$$< [K-(R+1)]/2+n+(R+n)/(1+d)+[K-(R+1)]/2+1,$$

or

$$[K-(R+1)]+2+(R-1)(1+d) <$$
$$< [K-(R+1)]+n+1+(R+n)/(1+d). \qquad \text{(A1)}$$

At the next iteration, we have in $\bar{S}$ three kinds of nodes: x" in gamma(x′), x" on s.p.; v in gamma(x′), v not on s.p.; z1 in gamma(n;x), z1 not on s.p.. So

$$\begin{aligned}
ft(z2) &= gt(z2)+\{\text{min over } u \text{ in } \bar{S} \text{ of } (gs(u)+\bar{H}(u,z2))\}\\
&= n+[K-(R+1)]/2+\min\{[K-(R+1)]/2+2+[n+(R-1)]/(1+d);\\
&\qquad [K-(R+1)]/2+2+[n+(R+1)]/(1+d);\\
&\qquad [K-(R+1)]/2+n+[2n+(R+1)]/(1+d)\}\\
&= n+[K-(R+1)]/2+[K-(R+1)]/2+[n+(R-1)]/(1+d)+2\\
&= [K-(R+1)]+n+(R+n)/(1+d)-1/(1+d)+2,
\end{aligned}$$

while

$$\begin{aligned}
ft(y') &\le gs(x")+gt(y')+\bar{H}(x",y')\\
&= [K-(R+1)]/2+2+[K-(R+1)]/2+1+(R-2)(1+d).
\end{aligned}$$

So

$$\begin{aligned}
ft(y') &\le [K-(R+1)]+3+(R-2)(1+d)\\
&= [K-(R+1)]+2+(R-1)(1+d)-d\\
&< [K-(R+1)]+2+(R-1)(1+d)\\
&< [K-(R+1)]+n+1+(R+n)/(1+d) \quad \text{(according to (A1))}\\
&< [K-(R+1)]+n+(R+n)/(1+d)+2-1/(1+d)=ft(z2),
\end{aligned}$$

and so now y′ will be expanded. <<

(c) A node in the side tree of x' always realizes its minimum in y'' and vice versa, because if z1 in gamma(n1;x') and z2 in gammai(n2;y'), $gs(z1)+gt(y'')+\bar{H}(z1,y'')=$

$< gs(z1)+gt(z2)+\bar{H}(z1,z2)$ because $gt(y'') <= gt(z2)$ and $\bar{H}(z1,y'') < \bar{H}(z1,z2)$. <<

(d) $fs(x'') <= gs(x'')+gt(y'')+\bar{H}(x'',y'')$

$= gs(x'')+gt(y')+1+\bar{H}(x'',y'')$

$< gs(x'')+gt(y')+\bar{H}(x'',y')$ because $d>0$

$= ft(y')$

$< ft(z2)$ because y' was expanded instead of z2;

and so z2 will not be expanded any more. <<

Suppose we have x in $\bar{S}$, x' in gamma(x), z in gamma(n;x), x on s.p., x' on s.p., z not on s.p., y in $\bar{T}$, y' in gammai(y), y on s.p., y' on s.p. (see fig 2.12); then z will be expanded iff

$gs(z)+hs(z) <= gs(x')+hs(x')$, or

$gs(z)+gt(y')+\bar{H}(z,y') <= gs(x')+gt(y')+\bar{H}(x',y')$, or

$[K-(R+1)]/2+n+[K-(R+1)]/2+1+(R+n)/(1+d)$

$<= 2([K-(R+1)]/2+1)+(R-1)(1+d)$

and this yields $n=INT\{Rd-d(d+1)/(d+2)\}$, the same as for the uni-directional and the bi-directional Pohl algorithms. (This could also be derived from the fact that side trees in the case of BHFFA2 become fully developed before the next node on the s.p. is expanded and consequently a translation argument reduces the BHFFA2 case to the uni-directional case.)

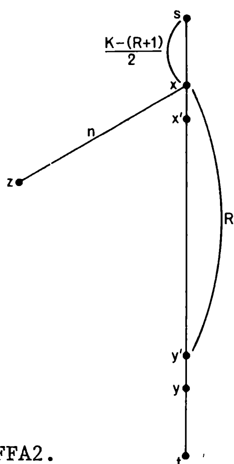


Fig. 2.12. H(s,t) = K, BHFFA2.

If we denote d(d+1)/(d+2) by cd, we get $V_R = m^{INT(Rd-cd)} - 1$. Now we give approximations for the Fa´s assuming n = Rd−cd instead of n = INT{Rd−cd}. We will here also give the approximations for the less erroneous heuristic by using n = R.qd instead of n = INT{R.qd}, where qd stands for d/(d+2).

1. F[uni-directional] = $\sum_{R=1}^{K} V_R$

$= m^{d-cd}.(m^{Kd}-1)/(m^d-1)-K;$

the less erroneous heuristic yields:

$m^{qd}.(m^{qd.K}-1)/(m^{qd}-1)-K;$

2. F[bi-directionalPohl] = 2. $\sum_{R=K/2+1}^{K} V_R =$

$= 2.m^{(1+K/2)d-cd}.(m^{Kd/2}-1)/(m^d-1)-K;$

the less erroneous heuristic yields:

$2.m^{qd(1+K/2)}.(m^{qd.K/2}-1)/(m^{qd}-1)-K;$

3. F[BHFFA2] = 2. $\sum_{R=1}^{K/2} V_{2R-1} =$

$2.m^{d-cd}.(m^{Kd}-1)/(m^{d}-1)-K;$

the less erroneous heuristic yields:

$2.m^{qd}.(m^{qd.K/2}-1)/(m^{qd}-1)-K.$

Examples for K=4, m=2, d=1 are shown in Figures 2.13 (uni-directional), 2.14 (bi-directional Pohl) and 2.15 (BHFFA2).

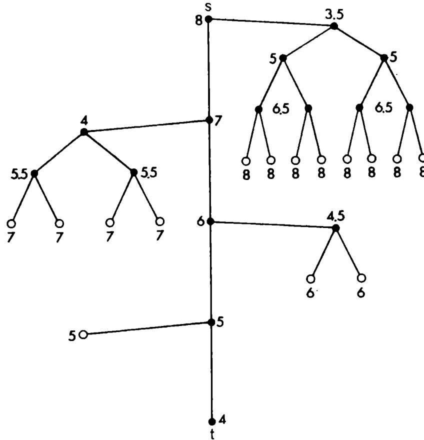


Fig. 2.13. Examples of worst-case situations.
$\bar{H}(x,y) = H(x,y)(1+\delta)$ if both x and y are on the solution path, and $\bar{H}(x,y) = H(x,y)/(1+\delta)$ otherwise (unidirectional, y = t always; bidirectional Pohl, y = s or t). Filled-in circles are nodes visited and expanded; empty circles are nodes visited but not expanded. For all the examples K=4, m=2, and $\delta=1$. The numbers beside the nodes are the F-values; the order of expansion for the first two algorithms is then easily derived. INT is a function that produces the integer part of the expression following it.
(a) Unidirectional.
$F = m^{INT(\delta-c\delta)}(m^{K\delta}-1)/(m^{\delta}-1)-K = 2^{0}\times(2^{4}-1)/(2^{1}-1)-4=11.$

See below for part (b) and for part (c).

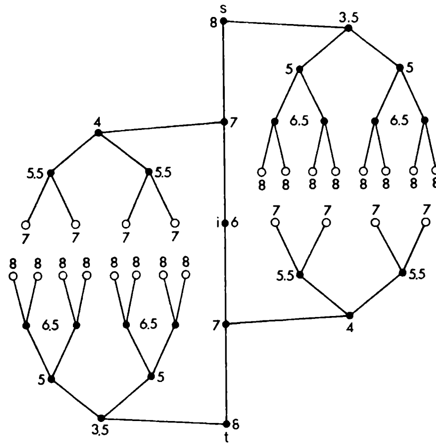


Fig. 2.14. Bidirectional Pohl.

$\bar{F} = 2m^{\mathrm{INT}\{(1+K/2)\delta - c_\delta\}}(m^{(K/2)\delta}-1)/(m^{\delta}-1)-K =$

$= 2\times 2\times(2^2-1)/(2^1-1)-4 = 20.$

i must be expanded before the algorithm terminates.

δ → d

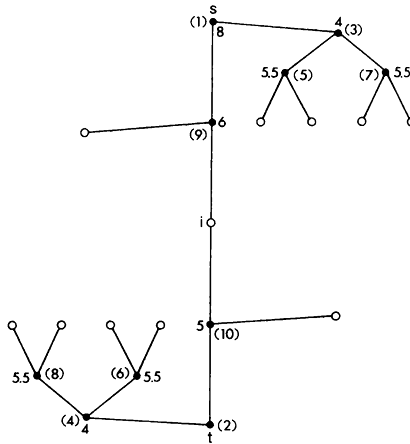


Fig. 2.15. BHFFA2. The order of expansion is shown in brackets; the F-values are those at the moment of expansion.
$F = 2m^{INT(\delta - c_\delta)}(m^{K\delta}-1)/(m^{2\delta}-1)-K=2\times 2\times(2^4-1)/2^2-1)-4=6.$

$\delta \to d$

5. Implementation results

We report here the results of an implementation that accompanied the presentation of the BHFFA-algorithm [17] since we feel that the changes of BHFFA2 with respect to BHFFA do not significantly change the comparison between the empirical results obtained by an uni-directional implementation and our results (moreover the implementation deviated already from BHFFA). The modified algorithm has been implemented as a FORTRAN program geared towards the 15-puzzle. In this search space, all edge lengths are taken as unity. The modifications made are the following:

(1) If the program has not found a solution path after expanding 1000 nodes, it gives up.

(2) The number of open nodes in a front is restricted to some maximum m, which is given to the program as an input parameter but must be less than 100. This restriction is realized by deleting ("pruning") the worst node of a front whenever inserting a new one would mean that the front would contain more than m nodes. The pruning is mainly necessary to save time, as the number of comparisons needed to calculate hs(x) for a node in $\overline{S}$ is equal to the number of nodes in $\overline{T}$ and vice versa. But this also means that the algorithm is no longer admissible (an algorithm satisfying Theorem 1 is called admissible), since it is possible that some node on the optimal path will be thrown out because it looked bad at some iteration. In a search space where only one path exists from start to goal node, some backtracking mechanism would be required to ascertain that this path is found. In the case of the 15-puzzle, the actual influence did not appear to be very large when m was set at 50 (or larger), as can be seen in the sequel.

(3) The section of step (5) of BHFFA2, in which occurrence of a new node in the collection of closed nodes on its own side is checked, was eliminated. This was done because we thought that the time possibly gained by expanding a few nodes less would not balance the loss caused by searching through the set of closed nodes for every open node.

(4) The lower loop was eliminated and step (3) was replaced by:

(3I) Determine aa and A as in (3);

while in (5) another test was added:

"If x in $T \cup \overline{T}$ then halt with a solution path."

That the testing of x is in $T \cup \overline{T}$, instead of just in $\overline{T}$, is a necessary consequence of the pruning, as it is possible that a descendant of a closed node is deleted from the front of open nodes. The addition of x to $\overline{S}$ in step (5) was made by estimating all distances to the opposite front and inserting node x in one of the ordered fronts of open nodes (the ordering is given by the f-values of the open nodes). A nasty side effect was that the insertion of a new node in $\overline{S}$ could imply a reordering of $\overline{T}$ and vice versa. The ordering was done by using a square matrix in which all combinations of the $\overline{H}(x,y)$-values of the fronts were stored.

An $(n^2 - 1)$ puzzle, for n => 3, consists of an n x n square with $(n^2 - 1)$ tiles, identifiable for instance by numbers. Tiles can change locations using the rule that one of the tiles adjoining the empty position can be moved into the empty position. Thus for example in Table 2.0, configuration A1 allows the movement of tile 11 or 12, while configuration A10 allows changing the tiles 7, 15, 3, or 1.

Three heuristic functions were implemented in the program:

(1) $P(x,y) = \sum_i p_i$,

with p_i being the Manhattan distance between the positions of tile i in x and in y.

(2) $S(x,y) = \sum_i p_i^2 h_i^{0.5}$,

where p_i is as in (1) and h_i is the distance in x from tile i to the empty square.

(3) R(x,y) is the number of reversals in x with respect to y, where a reversal has the meaning that x(i)=y(j) and x(j)=y(i), and i and j are adjacent tiles.

Of these functions, (1) and (2) come from [24] and (3) comes from [66].

In order to compare our results with the uni-directional case, the program was run with the same 15-puzzle problems as were used in [66]. (See Table 2.0, for the tile configurations of problem A1 - A10.) Furthermore, the same heuristic function was used with the w-values (the value infinite is a weight implying the elimination of g):

f1= g + wP with w= 1,2,3,4,8,16,inf,

f2= g +w(P+20R) with w= 1,2,3,4,8,16,inf,

f3= g+wS with w= 0.5,.75,1,1.5,2,3,4,16,inf,

f4= g+w(S+20R) with w= 0.5,0.75,1,1.5,2,3,4,16,inf,

(this, in fact, means f1(x)= gs(x)+min{over y in $\overline{T}$ of (w.P(x,y)+gt(y))}, etc.). As there were ten different 15-puzzles, this amounts to a total of 320 problems, of which our program solved 240 wheras the uni-directional program of Pohl solved 203 of them. It can be seen that in nearly all these cases, the heuristic is not a lower bound on the real effort to be made. This is the main reason why many of the solutions found are not optimal, both for Pohl's program and ours.

The results are given in Tables 2.1-2.4. Table 2.1 gives the number of problems solved for each of the ten puzzles with each function. Subtotals are made for each puzzle and each function. Table 2.2 gives a score for the path lengths. It was obtained as follows: The program with the shortest path for some problem scored 1 and the other program scored 0 (and any path is counted shorter than no path at all). If the same path length was found, both programs scored 1. If a problem was not solved by either of them they both scored 0. Table 2.3 gives a similar score for the number of nodes expanded. Table 2.4 gives for each problem the shortest path found, the average path length over all solved cases, and the average number of expanded nodes over all solved cases.

Insofar as the solution quality is concerned, BHFFA2 is an improvement over the uni-directional algorithm: it solves more problems, finds in general shorter paths, and expands fewer nodes on the average, although the last effect is less prominent than we expected. BHFFA2 performs particularly well with a strong heuristic function; with f4, the total number of nodes expanded by our program was 32% less than that by Pohl's program.

The front length adequate for the problems was found empirically. Experimental runs were made with front lengths of 25, 32 and 50. An increasing number of problems was solved and higher stability was reached (by stability, we mean the chance that a longer front length preserves a solution obtained with a shorter front length; pruning tricks are the obstructing force here). As could be expected, the performance with respect to the front length depends on both the solution path length and the heuristic used: the better the estimator, the smaller the front length required. All problems have been run with a front length of 50, and the least satisfactorily solved problems were run again with a front length of 99, in order to see whether the maximum number of 1000 expanded nodes or the pruning in the fronts created the bottleneck. In general, the former seems to be the case since no significant improvements were made. (An exception was f1 on A9, were six, instead of one, out of seven problems were solved.)

The main disadvantage of Pohl's bi-directional algorithm, mentioned in section 2.1 appeared to be remedied. The fronts now did meet near the middle of the search space, which we could see by comparing gs and gt of the intersection nodes.

By removing the 1000 expanded nodes limitation and changing a simple parameter, it was possible to run the 'most difficult' example from the 24-puzzle. The solution characteristics are given in Table 2.5. Again the path components meet near the middle of the search space.

A serious disadvantage of our algorithm is the time consuming calculation of the distance estimator. How much more expensive BHFFA2 is depends on the heuristic used: the more complicated this function is, the larger will be its share in the total computation time needed for a solution, and the smaller will be the share of the other computations, which have to be done for BHFFA2 as well as for the uni-directional algorithm. So, with a front length of m, BHFFA2 will be in the limit (with an infinitely complicated heuristic) m times as expensive as the uni-directional algorithm when they end up with a solution path of the same length. In general, the loss of efficiency will not be sufficiently offset by the shorter paths found. Nevertheless, it may well pay off in, for example, an ABSTRIPS-like environment (see [73]), where it is crucial to find an optimal path from among many different existing paths, as the number of subproblem searches depends on the path length found in the dominating problem space. There BHFFA2, or a similar algorithm with a strong heuristic, may find an optimal path more efficiently than an uni-directional program with a heuristic satisfying the lower bound condition because this kind of heuristics tends to be rather weak and results in a fast explosion of the number of nodes expanded.

A1	1	2	3	4	A2	9	5	1	3
	5	6	7	8		13	7	2	8
	13	15	14	11		14	6	4	11
	10	9	12			10	15	12	
A3	6	2	4	7	A4	1	3	7	4
	5	15	11	8		9	5	8	11
	10	1	3	12		13	6	2	12
	13	9	14			10	14	15	
A5	2	5	6	4	A6	1	2	3	4
	9	1	15	7		5	6	7	8
	14	13	3	8		10	12	11	13
	10	12	11			15	14	9	
A7	1	4	2	3	A8	7	13	11	1
	6	5	8	11			4	14	6
	14	9	12	15		8	5	2	12
	10	13	7			10	15	9	3
A9	15	11	14	12	A10	9	2	13	10
	7	10	13	9		5	12	7	4
	8	4	6	5		14	1		15
	3	2	1			11	6	3	8

Table 2.0 The initial configurations of the 15-puzzles.

	f1		f2		f3		f4		Sum	
	P	B	P	B	P	B	P	B	P	B
A1	7	7	7	7	9	9	9	9	32	32
A2	7	7	7	7	1	9	9	9	24	32
A3	3	4	3	7	5	3	8	9	19	23
A4	7	7	7	7	9	8	9	9	32	31
A5	2	6	5	7	3	8	9	9	19	30
A6	3	2	6	6	6	3*	8	9	23	20
A7	5	4	6	6	0	3	8	9	19	22
A8	0	3*	3	5	0	1	7	9	10	18
A9	2	6*	4	5	1	0	6	9	13	20
A10	0	1	3	3	2	1	7	7	12	12
Sum	36	47	51	60	36	45	80	88	203	240

Table 2.1 The number of problems solved for Pohl's uni-directional program (P) as well as for BHFFA2 (B). The maximum achievable score for the columns headed by f1 and f2 is 7 and for the columns headed by f3 and f4 is 9 (see text). The entries marked with a * are those problems which were run with a front length of 99. All others were run with front length 50.

	f1		f2		f3		f4		Sum	
	P	B	P	B	P	B	P	B	P	B
A1	7	7	7	7	9	9	9	9	32	32
A2	1	7	5	7	1	9	1	9	8	32
A3	1	4	1	6	4	2	0	9	6	21
A4	5	6	5	4	9	3	9	5	28	18
A5	0	6	1	6	1	8	0	9	2	29
A6	3	1	1	5	5	3*	1	9	11	17
A7	3	3	4	4	0	3	2	7	9	17
A8	0	3*	2	3	0	1	0	9	2	16
A9	2	4*	3	3	1	0	0	9	6	16
A10	0	1	1	3	2	1	1	7	4	12
Sum	22	42	30	48	32	39	23	82	107	211

Table 2.2 The score for solution path lengths. A program scored 0 for a problem when it did not found a solution or its solution was longer than a solution of the other program. Otherwise, a program scored 1.

	f1		f2		f3		f4		Sum	
	P	B	P	B	P	B	P	B	P	B
A1	6	7	0	7	8	7	9	0	30	14
A2	5	2	6	1	0	9	0	9	11	21
A3	1	4	1	6	4	2	2	7	8	19
A4	0	7	3	4	9	2	8	1	20	14
A5	0	6	2	5	2	7	2	7	6	25
A6	3	1	2	4	5	2*	5	4	15	12
A7	4	1	5	1	0	3	2	7	11	12
A8	0	3*	2	3	0	1	1	8	3	15
A9	1	5*	3	2	1	0	0	9	5	16
A10	0	1	2	2	2	1	2	6	6	10
Sum	20	37	33	28	31	34	30	58	115	157

Table 2.3 The score for numbers of nodes expanded. A program scored 0 for a problem when it did not found a solution or when it expanded more nodes than the other program had to expand for finding a solution. Otherwise, a program scored 1.

	I		II		III	
	P	B	P	B	P	B
A1	12	12	12	12	12.6	12.9
A2	26	26	42.8	27.3	161.5	54.4
A3	36	34	64.8	44.0	389.1	320.9
A4	20	20	21.8	24.3	90.2	108.6
A5	38	32	56.7	39.7	310.4	250.1
A6	32	32	42.8	37.1	335.7	333.4
A7	36	36	56.6	53.7	365.4	429.3
A8	85	61	132.0	93.0	605.6	485.9
A9	86	88	152.9	118.5	551.2	563.5
A10	64	60	92.3	90.0	609.3	532.3

Table 2.4 In column I, the shortest path found for each problem; in column II, the average path length; and in column III, the average number of nodes expanded (only over the solved problems).

Start node						Goal node				
	24	23	22	21		1	2	3	4	5
20	19	18	17	16		6	7	8	9	10
15	14	13	12	11		11	12	13	14	15
10	9	8	7	6		16	17	18	19	20
5	4	3	2	1		21	22	23	24	

Front length 75.
Nodes visited 6896.
Nodes expanded 2671.
Solution path length 340 (177+163).
Solution time 1130 cpu sec (C.D.C.).

Table 2.5 A 24-puzzle example with the solution characteristics.

6. Open problems and loose ends

BHFFA2 can be further simplified by not calculating the heuristic distance to every node in the opposite front but only to the better half or even fewer of them. This idea is inspired by the fact that, in the limited number of cases where we checked it, a node realized its minimum nearly always in a node which belonged to the best ten of the opposite front. Another simplification would be to delete the resequencing of the opposite front as the consequence of adding a node to a front. The sensitivity of the solution quality to these attempts to reduce the computation time, needs still to be tested.

A crude solution to the costs of calculating the heuristic distances to all nodes of the opposite front may come from the rising tide of multi-processors. One may even expect that with an unlimited number of processors available, a solution may be found twice as fast when working from both ends.

ACKNOWLEDGEMENT. Discussions with Ira Pohl, when he was a Visiting Lecturer at the Vrije Universiteit, are appreciated.

The coding of the implementation was done by Marleen Sint.

Mr M. Taunton and dr T.B. Boffey of the University of Liverpool notified us in June 1979 about the incorrectness of the BHFFA algorithm as presented in [17].

SUBSTITUTION IN LISP

1. Introduction

LISP is one of the oldest languages around emerging like FORTRAN at the end of the fifties. But here the resemblance ends. The most obvious difference between the two languages is that FORTRAN programs need to be compiled while those of LISP are mostly interpreted. Because of the inefficiency of interpretation the latter are not likely to be used in a production environment. In a research environment, on the other hand, not needing the intermediate step of compilation adds up to an attractive advantage and here LISP flourishes. One proviso must be made however: numerical operations can be done in LISP but when long calculations have to be done, FORTRAN is to be preferred. LISP is effective for non-numerical symbol manipulation, where the processing is not a matter of depth but of breadth.

A unique feature of LISP is its incremental character. For other computer languages, there is an urge to standardize. Not so in LISP. Certainly, every LISP implementation will contain a core of basic functions but every implementation also contains functions which might be available in most other implementations but not necessary to all, and there may even be functions which are unique to a particular implementation. Contrary to what one might expect, there is no mob in the LISP community promoting standardization, the reason being that missing functions can always easily be simulated by user-provided functions. There is more incrementality. Most LISP implementations are accompanied by compilers which allow selective compilation of parts of LISP programs. Thus it is possible to have both the advantages of compilation and of interpretation. Programming is easy in a high level language like LISP and inefficiency can be reduced by spotting the code where control resides most of the time - often a small part of the total code - and subsequently compiling that part.

Sometimes LISP compilers produce intermediate assembler code in a language which is also available to the programmer making more optimization possible. Stated otherwise: the user has a tool to extend LISP according to his personal needs. A further step is the addition to the standard LISP repertoire of new functions, thus making them available to every user. In this fashion, LISP grew during the last 20 years from an initial 80 built-in functions to the approximately 600 functions that are available in INTERLISP.

In this chapter, we shall describe the fare of SUBSTAD, a substitution function, as added to the repertoire of a LISP implementation. The aim of this addition was to increase the efficiency of the theorem prover described in the next chapter. This operation in turn led to an optimization of the core procedure, the unification algorithm. We had noticed that the originally implemented version of this general pattern matcher generated a lot of garbage and accounted for a large measure of unnecessary condition-testing resulting from the generality of the employed SUBST substitution function. By tailoring a substitution function and slightly changing the unification algorithm, we have realized savings in the unification procedure.

We have developed two versions of the SUBSTAD function. The first and more complicated one, uses pointer-reversal instead of recursion, which would have been the more obvious method since the data have a recursive structure -- trees of arbitrary depth. Of course, we still have disguised recursion since the stack is dynamically constructed inside the data. Later we realized that this pointer reversal technique is only worthwhile when the tree depth is so large that the stack may overflow. Marking during garbage collection is a typical example where this pointer reversal technique is appropriate. Other operations, however are always recursively implemented, using the stack, and there is no particular reason why substitution should be an exception. Therefore the second, faster version is a mixture of recursion and iteration.

Both versions are interesting from the perspective of program verification because they both are designed with support functions having side effects the most tricky of which is the pointer-reversal version. In order to verify both versions we have chosen the method of symbolic execution [44,22]. In brief, this method requires the addition of input/output descriptions to the program code and of invariants to each loop. Subsequently the code must be executed with symbolic input values conforming to the input specifications, producing a symbolic value for every branch through the code for which the output condition has to be verified, 'manually' or with a theorem prover. Analogous to the EFFIGY-program, which symbolically interprets PL/1 style programs [44], we have developed a symbolic interpreter for a subset of LISP, that includes the primitives used in several SUBSTAD versions. This interpreter is an indispensable tool for obtaining rigorous proofs. It automatically deals with many petty details doing all the book-keeping while simultaneously following the different branches of the computation tree.

We must admit that verifying these 'small' versions was a non-trivial task, in spite of this sophisticated tool (and having available a theorem prover - the one partly described in chapter 4 - which could automatically decide many verification conditions). Specifying the input/output conditions and the loop invariants and defining the (specific !) properties of the intermediate data structures requires a formalism more than a hundred (estimated) times as long as the mere code in the case of the pointer reversal SUBSTAD function.

For this reason we are sympathetic to the point raised in [23] that program verification is (as yet) unworkable (we consider non-formalized input/output specifications and informal proofs irrelevant, since it is precisely in the fine points that bugs are hiding.) Nevertheless, we do not agree with their conclusion that program verification will remain forever an illusion. We wish with good reasons that research in this field will be continued.

In a previous publication [13] we reported some magnificent results obtained by comparing unification algorithms using SUBSTAD versus SUBST. Two years later, it turned out that the results were biased by a bug in the machine implementation of SUBST. This bug must have been slumbering for at least 5, if not more than 10, years. The total LISP interpreter is ´gigantic´, about 14000 lines of assembler code. It consists of a large number of small packages for which, in most cases, small equivalent formulations exist in LISP code. Although no easy task, it is certainly not inconceivable, as suggested in [23], that these LISP formulations can be rigorously verified. A compiler (and an optimizer) would subsequently produce faultless code. Greater justification lies in the realization that this bug in SUBST was not the only one in the interpreter. In fact, we have devoted about 10% of our time during the last five years to debugging the LISP interpreter (not to mention the time lost due to errors in the operating system).

At the same time we are ill at ease with the tools developed by the theoreticians of program verification (for an overview of this subject, see [38,39]). The languages developed (Dynamic Logic, etc.) abstract away from real application, concern toy-like programming languages and tend to be seen as interesting objects in themselves. The proliferation of notations used does not make it any easier to remain sympathetic to these efforts.

We feel closer to concrete efforts, such as the one expounded in [84] to verify the correctness of the Schorr-Waite marking algorithm [77], which also works with pointer reversals. On the one hand, this algorithm is more complicated since cyclic data structures are allowed. On the other hand, the side effects in our program are more complicated than the side effects of just marking accessible structures. Another difference is that Topor is concerned with a correctness proof of an *algorithm* instead of a program. Thus bugs which would have otherwise been introduced by implementing the algorithm into a concrete language will not be captured.

The proof itself is in the style one encounters in mathematics books, so we hesitate to accept his statement "We believe however, that the present proof is the first one which is rigorous, easy to follow, and amenable to machine checking". Indeed, the proof is reasonable to follow but machine checking requires another level of rigour, as we will illustrate with the many painstaking details in our own proof fragments.

A radically different strategy to verify pointer reversal marking and copying algorithms is given in [48]. They begin with a definition of marking in set-theory language, deliberately ignoring most aspects of control and efficiency. Subsequently, they give an informal correctness proof of this 'archetypical' algorithm which is gradually modified by the addition of more details and making more commitments about control. They present those modifications with general applicable transformations, suggesting that their faithfulness need be demonstrated once only, and so reducing the verification effort when they can be applied. However, the generality of these transformation rules did not strike us as their main characteristic. We consider their main contribution their ability to *derive* different but related algorithms from one rudimentary algorithm. From the perspective of machine verification, their correctness proofs were not as rigorous as one would have desired.

In the next section we define several versions of the standard substitution function SUBST, of recursive destructive substitution functions and of a pointer-reversal destructive substitution function. The subsequent section 3 gives the framework of verification of LISP code by symbolic execution. This is then applied in section 4 to several versions of the substitution functions. Section 5 gives the results of experiments comparing SUBST with the recursive destructive SUBSTAD, and the results obtained by equipping a unification algorithm with SUBSTAD.

2. Substitution Functions

We will describe the different functions primarily by giving their coding in elementary LISP functions, since we assume familiarity with that language. To pave the way for the symbolic evaluator, to be described in the next section, we summarize the basic characteristics of LISP:

- The syntax is in preorder Polish notation, e.g. a+b is coded as (PLUS A B); sometimes (+ A B) is also allowed;
- The data objects are mostly trees, called S-expressions, having two branches at non-terminal nodes, while a terminal node, also called an atom, can be a number or a string; however, cycles leading to directed bi-graphs - each non-terminal node has exactly two out-going edges - are also allowed;
- The parameter passing is call-by-reference with either (1) evaluation of the parameters from left to right and passing the pointers thus obtained to the calling function, which happens with functions like CAR, CDR, CONS, EQ, EQUAL, RPLACA, RPLACD; or (2) non-evaluation of the parameters and passing their pointers to the calling function, for instance with COND, SETQ, QUOTE, PROG. In this case, the calling function itself can decide when to evaluate arguments. The examples given above pertain to built-in functions but user functions can also have either of these parameter mechanisms. Since the LISP repertoire contains the function EVAL, the evaluation of arguments can range from complete and automatic to arbitrary selective.
- User functions are accessible as S-expressions (and it is even possible to construct dynamically new functions to be executed in the same run). This feature makes it easy to program a symbolic evaluator for LISP code in LISP. Eventually, one might even attempt to have the verifier verify itself. Of course, a positive outcome is only to be trusted when the verifier is in fact correct.
- There is an ´unlimited´ number of free cells available for CONS-ing up, tree structures (or constructing bi-graphs).

Most built-in functions can be expanded as compositions made up of a small set of core functions. As a warming-up we give the expansion of the function EQUAL, which returns T iff its two arguments are trees of the same form and have identical terminal nodes at corresponding leaf positions, and NIL otherwise (text between question marks is comment):

```
(EQUAL(LAMBDA(S1 S2)(COND
     ((EQ S1 S2)T)
     ((ATOM S1)NIL)
     ((ATOM S2)NIL)
     ((EQUAL(CAR S1)(CAR S2))
      (EQUAL(CDR S1)(CDR S2)))
     (T NIL)
)))?end of EQUAL?
```

We have noted already that there is no standard LISP. The example above shows we assume that EQ will not hiccup when non-atomic arguments are given. It is supposed to be generalized in such a way that it can recognize whether two arguments have identical ´addresses´ and thus need not be descended all the way down to their leaves. Otherwise the first two lines of the COND-ition would have to be replaced by the more clumsy formulation:

```
((ATOM S1)
 (COND((ATOM S2)(EQ S1 S2))
      (T NIL)))
```

Now we are ready for the standard version of SUBST:

```
(SUBST(LAMBDA(S1 S2 S3)(COND
     ((EQUAL S2 S3)S1)
     ((ATOM S3)S3)
     (T(CONS(SUBST S1 S2(CAR S3))
            (SUBST S1 S2(CDR S3))))
)))?end of SUBST?
```

which can be phrased as follows: replace in S3 every subtree which is EQUAL to S2 by S1 (and do not destroy S3).

This version eats up three times as much stack space as necessary. The next version using the support function SUBSTSUPF1 remedies this fault:

```
(SUBST(LAMBDA(S1 S2 S3)(SUBSTSUPF1 S3)))

(SUBSTSUPF1(LAMBDA(S3)(COND
     ((EQUAL S2 S3)S1)
     ((ATOM S3)S3)
     (T(CONS(SUBSTSUPF1(CAR S3))
            (SUBSTSUPF1(CDR S3))))
)))?end of SUBSTSUPF1?
```

Another nasty property is still present in this version. It copies all of the non affected S3, even when no substitutions are performed. With a little additional computational effort, this space consumption can be curbed, saving garbage collection computation later on. In the next version, CONS-ing happens only when a change occurred in a subtree (using as a check the liberated EQ function (!) and using the auxiliary variable X for saving intermediate results):

```
(SUBSTSUPF2(LAMBDA(S3)(PROG(X)(RETURN(COND
     ((EQUAL S2 S3)S1)
     ((ATOM S3)S3)
     ((EQ(CAR S3)(SETQ X(SUBSTSUPF2(CAR S3))))
      (COND((EQ(CDR S3)(SETQ X(SUBSTSUPF2(CDR S3))))S3)
           (T(CONS(CAR S3)X))))
     (T(CONS X(SUBSTSUPF2(CDR S3))))
)))))?end of SUBSTSUPF2?
```

We now switch to the non-copying, destructive substitution functions. In these functions, only non-numeric atoms are allowed as their second argument, because that is how substitutions are mostly used in applications. This restriction allows us to remove the EQUAL-testing on the inside of the S3-tree. The definition function is formulated with a support function.

```
(SUBSTAD(LAMBDA(S1 LAT S3)(COND
     ((NOT(ATOM LAT))
      (EXIT with an error))
     ((NUMBERP LAT)
      (EXIT with an error))
     ((ATOM S3)
      (COND((EQ LAT S3)S1)
           (T S3)))
     (T(SUBSTAD1 S3)S3)
)))?end of SUBSTAD?
```

```
(SUBSTAD1(LAMBDA(S3)(PROG2
     (COND((ATOM(CAR S3))
           (COND((EQ LAT(CAR S3))(RPLACA S3 S1))))
          (T(SUBSTAD1(CAR S3))))
     (COND((ATOM(CDR S3))
           (COND((EQ LAT(CDR S3))(RPLACD S3 S1))))
          (T(SUBSTAD1(CDR S3))))
)))?end of recursive SUBSTAD1?
```

The quintessence of this support function is that it works by side effect. It does not need to return a significant value because at the top level SUBSTAD still ´knows´ the root of the S3-tree.

The next version capitalizes on this feature by throwing out the recursion on the CDR-branch, while another small modification reduces CAR/CDR-actions:

```
(SUBSTAD2(LAMBDA(S3)(PROG(HH)
AGAIN
     (COND((ATOM(SETQ HH(CAR S3)))
           (COND((EQ LAT HH)(RPLACA S3 S1))))
          (T(SUBSTAD2 HH)))
     (COND((ATOM(SETQ HH(CDR S3)))
           (COND((EQ LAT HH)(RPLACD S3 S1))))
          (T(SETQ S3 HH)
            (GO AGAIN)))
)))?end of half recursive/iterative SUBSTAD2?
```

It would of course also have been possible to maintain recursion on the CDR-branch and to throw out CAR-branch recursion. Measurements in practice however, have shown that the atom/list ratio for the CAR was 1.33, while this ratio for the CDR was 0.37 [21]. Consequently the choice embodied in SUBSTAD2 reduces stack usage.

The next version of the support function for SUBSTAD is radically different since it uses pointer reversal.

```
(SUBSTADP(LAMBDA(S3)(PROG(EX HH)
     (SETQ EX $)                                   /1
L2
     (SETQ HH(CAR S3))
     (COND((NOT(ATOM HH))                          /2
           (MARK S3 1)                             /3
           (RPLACA S3 EX)                          /4
           (SETQ EX S3)
           (SETQ S3 HH)
           (GO L2))
          ((EQ LAT HH)(RPLACA S3 S1)))             /5
L4
     (SETQ HH(CDR S3))
     (COND((ATOM HH))                              /6
          ((NOT(EQ EX $))                          /7
           (RPLACD S3 EX)
           (SETQ EX S3)
           (SETQ S3 HH)                            /8
           (GO L2))
          (T(SETQ S3 HH)                           /8
            (GO L2)))
     (COND((EQ LAT HH)(RPLACD S3 S1)))             /9
     (COND((EQ EX $)(RETURN)))                     /10
L5                                                 /11
     (SETQ HH S3)
     (SETQ S3 EX)
     (COND((MARKB S3)                              /12
           (MARK S3 0)                             /13
           (SETQ EX(CAR S3))                       /14
           (RPLACA S3 HH)
           (GO L4)))
     (SETQ EX(CDR S3))                             /15
     (RPLACD S3 HH)
     (GO L5)
)))?end of the pointer reversal SUBSTADP?
```

The numbers after the slashes refer to the following comments:

(1) $ should be an atom non-accessible by the user. In the initial S3 the CAR- respectively CDR-part, if these are non-atomic, will be replaced by $ to indicate the end of a reversed pointer chain; see also (7).

(2) Go down along CAR if CAR is not atomic.

(3) A bit associated with memory cell S3 is temporarily turned on. If there is no parallel garbage collection the garbage collection bit can be used.

(4) Store end of the chain or reversed pointer.

(5) Replace CAR terminal if identical to LAT.

(6) CAR part finished thus now inspect CDR part.

(7) In case EX=$ then the current S3 is the current top lying on the CD...DR chain of the original input S3. Since SUBSTAD still ´knows´ the root of the input S3 the reversed pointer chain will not be extended. Thus after reaching the bottom of CD...DR´s, one does not need to climb back and restore the original pointers. In case EX is not equal to $, S3 is somewhere inside the CAR of the current top and the reversed chain has to be extended.

(8) Go down along CDR.

(9) Replace CDR terminal if identical to LAT.
(10) Quit if CD...DR´s terminal of original top is reached.
(11) Go up instead.
(12) MARKB returns T iff the mark bit of S3 is on, indicating that the CDR-part still has to be treated.
(13) Set mark bit off.
(14) Restore original downward pointer.
(15) Go further up.

In the next section we develop the tools for the formal verification of these functions.

3. Verification of LISP Functions

Since we take the position that verification of programs should be done rigorously or not at all, we have decided to develop a program which could keep track of the many details that are involved when checking all possible branches of computation trees. We have chosen the method of symbolic execution because it guarantees that every branch is visited and that all preconditions to operations are considered. Hand simulation is so cumbersome that one is willing to skip obvious checks, and it is of course exactly in these that bugs hide. The program is by no means complete. Side effects as generated by RPLACX (= RPLACA or RPLACD) are only provisionally dealt with. Nonetheless the program shaped our ideas on the proper treatement of side effects.

The method of symbolic execution requires the following:
-- to have a language in which abstract states can be described;
-- to have a symbolic evaluator which embodies the semantics of the built-in operators and which for every applicable operator from the considered program language can transform one state configuration into another faithfully describing the new situation;
-- to have a deductive component, which can confirm that preconditions of operators are fulfilled, that loop invariants hold and that specified output conditions are fulfilled for every computation path through the code being verified.

3.1 The State Description Language

In order to facilitate the deductive requirements, the state description language uses first-order predicate calculus. We start off with a countable domain of cells C and a countable domain of atoms A, where C and A are disjunct. Let D be their union: D = C U A. We will have the partial functions:
-- car and cdr, with domain C and range D; and
-- addr, with domain D and range N, the natural numbers.
We will have the partial predicate:

-- atom with domain D, and which, where defined, coincides with the characteristic predicate of A.

Using the addr-function, we define the relation eqa with:

(d)(e){ eqa(d,e) <--> addr(d)=addr(e) },

for d, e in D where addr is defined. It is easy to see that eqa is an equivalence relation.

We will have the following axioms:

AXIOM 1.

(d)(e){ eqa(d,e) --> [atom (d) --> d=e] },

for d, e in D; i.e. two elements of D are identical when they have the same address and one of them is atomic.

AXIOM 2.

```
(d)(e){ [ ~atom(d) & eqa(d,e) &
          car(d) = car(e) & cdr(d) = cdr(e) ] -->
        d = e };
```

for d, e in D. Observe that the ~~former~~ latter axiom ensures that e is also non-atomic. This axiom says that two elements of D are identical, when they have the same address and their respective car´s and cdr´s are identical. It precludes configurations similar to those found in figure 3.1.

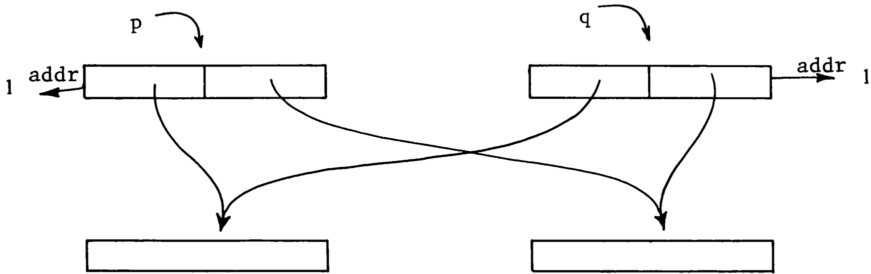


Fig. 3.1. Axiom 2 prohibits p unidentical q.

DEFINITION.

A *data object* D_r *is an element of the power* set of D:

1) *with* D_r *of finite size,*

2) *with* C_r *and* A_r *the elements of* D_r *respectively in* C *and* A,

3) *with* car(C_r) *and* cdr(C_r) *subsets of* D_r, *and*

4) *with a unique element* r *in* D_r, *the root of* D_r, *which has the property that all other members of* D_r *can be reached from* r *by finite* car/cdr *chains.*

Alternatively each data object can be seen as a directed bi-graph with a unique node from which all other nodes can be reached along the directed edges.

From now on we mention data objects by referring to their roots.

Recursive definitions on data objects run the risk of being undefined due to infinite regress because data objects may contain cycles - a cell which reaches itself along a car/cdr chain. The finiteness of data objects is the way out of this problem. Most recursive definitions we will give in the sequel, apply to data objects that have the special format of a tree. The generalization for

some of them will be given in the appendix to this chapter.

Recursive definitions on trees invoke in proofs an appeal to the so-called car/cdr induction. Whenever a formula P(x) reduces to a formula P(car(x)) and/or P(cdr(x)) then car/cdr induction allows the conclusion that P(x) has been inferred. This is justified by the observation that a well founded relation can be constructed (mostly the number of reachable cells from x) which decreases on each recursive reference. A similar trick requiring somewhat more care applies on recursive definitions in which non-tree type of arguments are also allowed.

Since we are dealing with partially defined functions and predicates we frequently require a selective reading of the logical connectives. For example an implication P --> Q, we take to be true when P is false, although the falseness of P renders Q in fact undefined, such as in ~atom(x) --> car(x)=y.

Next we give definitions of the predicates partof and loopfree. The definition of partof works only on trees. The appendix contains a generalization to arbitrary data objects (& is the conjunction connective):

```
(d)(e){ partof(d,e) <-->
        [ partofcar(d,e) OR partofcdr(d,e) ] }

(d)(e){ partofcar(d,e) <-->
        [ ~atom(e) &
          ( d=car(e) OR partof(d,car(e)) ) ] }

(d)(e){ partofcdr(d,e) <-->
        [ ~atom(e) &
          ( d=cdr(e) OR partof(d,cdr(e)) ) ] }

(d){ loopfree(d) <--> loopfreel(d,0) }

(d)(V){ loopfreel(d,V) <-->
        [ atom(d) OR
          { ~(d in V) &
            loopfreel(car(d),{d} U V) &
            loopfreel(cdr(d),{d} U V) } ] }.
```

The expression partof(d,e) signifies that the data object e contains a cell or atom identical to the root of d. Loopfree defines the property that a data object does not contain a cycle.

Figure 3.2 shows data objects p and q for which simultaneously partof(p,q) and partof(q,p).

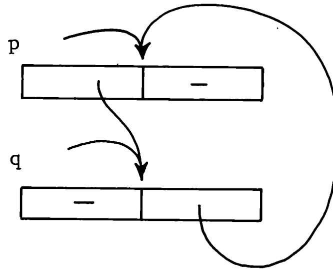


Fig. 3.2. For p and q we have partof(p,q) as well as partof(q,p).

We continue with the definition of the equal-predicate for cycle free data objects:

```
(d)(e){ equal(d,e) <-->
        [ d = e OR
          ( ~atom(d) & ~atom(e) &
            equal(car(d),car(e)) &
            equal(cdr(d),cdr(e)) ) ] }.
```

When we assume that p and q in fig. 3.1 do have different addr´s then we have an example of two non-identical data objects p and q for which still equal(p,q) holds.

<u>DEFINITION</u>.
A <u>state description</u> is a conjunction of facts referring to a finite number of data objects, always containing the data objects nil *and* t, *corresponding with* NIL *and* T, *members of* A, *for which holds*: atom(nil), atom(t) *and* ~(t=nil).

A state description may refer to ´virtual´ data objects, which are inherited from former state descriptions but no longer exist. We now define the compatability of two data objects (not containing cycles) a property which implies that they can co-exist:

```
(d)(e){ compatible(d,e) <-->
        [ atom(d) OR atom(e) OR
          ( eqa(d,e) & d=e ) OR
          ( ~eqa(d,e) &
            compatible(d,car(e)) & compatible(d,cdr(e)) &
            compatible(car(d),e) & compatible(cdr(d),e) )]}.
```

In other words, d and e are compatible when they do not contain cells having the same address unless those cells are identical. When two data objects are non-compatible at least one must be virtual. The appendix contains the generalization of compatible which allows arguments having cycles.

The RPLACX operations are responsible for ´killing´ data objects which then become virtual.

DEFINITION.
An *alist* *is a finite list of pairs* $((a_1,r_1), \ldots, (a_n,r_n))$ *with* a_i *atoms unequal* nil *and* r_i *the roots of data objects, and for each pair* r_i, r_j *we have*: compatible(r_i,r_j).

The alist (= association list) contains the current bindings of the atoms. A data object is virtual with respect to an alist if it is non-compatible with an r_i from that alist. An atom may occur more than once as a lefthand side of a pair, which may happen for instance as a consequence of recursion. LISP functions retrieve and update leftmost occurrences. Side effects may propagate to the right in the alist. Extensions and contractions, as a consequence of entering a respectively higher and lower stack level, also occur at the lefthand side.

DEFINITION.
A *state configuration* *is a pair* (AL,FL) *with* AL *an alist and* FL *a state description*. Atomicity of nil, t and all atoms (i.e. lefthand side of pairs) on the alist is implicitly a part of the FL, which embodies the current fact list.

Example 1. When AL contains only one pair (A1.r1) and FL is empty [the fact atom(A1) is implicitly given] then (AL,FL) is a state configuration because the compatible-requirement is trivially fulfilled.

Example 2. If AL = ((A1.r1)(A2.r2)) and
FL = ~atom(r1) & eqa(r1,r2) & car(r1)=nil & car(r2)=r1
then (AL,FL) is <u>not</u> a state configuration because the requirement r1=r2 cannot be fulfilled.

Example 3. If AL = ((A1.r1)) and
FL = ~atom(r1) & eqa(r1,r2) & car(r1)=nil & car(r2)=r1
then (AL,FL) is a state configuration because the compatibility requirement is automatically fulfilled. The data object r2 is virtual since we have eqa(r1,r2), atom(car(r1)) and ~atom(car(r2)), which excludes r1=r2.

3.2 The Symbolic Evaluator

The symbolic evaluator generates, when given LISP-code and a state configuration, a tree of state configurations, corresponding to all possible computation paths through the code. The symbolic evaluator works like a real LISP evaluator. It has a code pointer, corresponding to a program counter, pointing to that part of the code which has to be executed, it contains modules which correspond to built-in LISP functions and it knows what to do with user defined functions.

A non-numerical atomic form is evaluated by retrieving the most recent (leftmost) binding from the current alist.

For built-in functions, the recipe consists of checking whether preconditions, parametrized for the current arguments, are fulfilled; when the check succeeds, the state configuration is updated. An exception is made when treating the COND-function since this function leads to the generation of one or more bifurcations of the current

state configuration. The correctness of a bifurcation (satisfiability of a test expression and its negation) is not proven by means of the deduction machinery but by the construction or availability of two models that possess opposite truth values with respect to the test expression and that are both consistent with the current state configuration. A simple technique for constructing these models is to let the user provide several examples, which are processed concurrently with the symbolic input specification for the code. The models also play another role, as will be explained in the sequel. We will see that testing by running examples and formal verification should not be considered as two mutually exclusive methods. They must go hand in hand.

User provided functions have to be accompanied by input conditions which should be fulfilled before the function is entered, and by an output assertion which is the description of how the state configuration ought to be updated in terms of the actual argument bindings in the context (a function with side effect needs in addition a procedure (!) for updating the alist, see the sequel). The user should indicate for each user function whether he also wishes it to be verified in which case he will have to provide its body, to allow ´opening´ it. For obvious reasons recursive user functions will be opened no more than once. A well-founded relation, user provided, should be used when verifying that arguments of a recursive call score less with respect to that well-founded relation than the arguments at the top level call. This was not implemented; the number of non-terminal nodes in trees will be the well-founded relation for most of the cases we will encounter.

Modules are implemented for the following subset of standard LISP functions: ATOM, CAR, CDR, COND, CONS, EQ, EQUAL, GO, NOT, NULL, PROG, PROGN, QUOTE, RETURN, RPLACA, RPLACD and SETQ. The functions COND, GO, PROG, PROGN, QUOTE and SETQ are of type FSUBR, i.e. evaluation of their arguments is to their own discretion. The other functions have automatic - left to right - argument evaluation before module-specific actions are taken.

An essential requirement for the modules is that the compatability property of state configurations is preserved. We have to worry about RPLACA, RPLACD and SETQ because only those functions affect the alist.

Here follows a description of some of these modules.

ATOM
Let the argument of ATOM evaluate to x. A new symbolic value will be generated, say g1, which will be returned as the value, while the fact list will be expanded with:
{ g1=t & atom(x) } OR { g1=nil & ~atom(x) }.

The implemented version behaves differently for efficiency reasons. It deals immediately with the atomicity of x. It returns t, nil or generates a bifurcation of the current computation branch with t in one branch and atom(x) is added to the fact list, while nil in the other branch and ~atom(x) added to the fact list belonging to that branch. The first or second option is chosen when atomicity or non-atomicity of x can easily be derived from the given fact list. When this cannot be decided, the user is consulted. He may indicate apart from t or nil that both possibilities are to be pursued in case the current fact list does not determine the truth value of atom(x).

CAR (and analogously CDR)
Let the argument of CAR evaluate to x. In contrast with ATOM there is a precondition check for CAR: ~atom(x) should be derivable from the current fact list. If that derivation succeeds then a new symbolic value, say g2 is generated, which will be returned, and g2=car(x) will be added to the fact list.

COND
This function leads to bifurcation(s) of the current computation branch, as described for the implemented version of ATOM. See also the remark above concerning the availability of models in which test expressions have opposite truth values.

CONS

Let the arguments of CONS evaluate to x and y. A new symbolic value, say g3, is generated and will be returned, while the fact list will be extended with:
~atom(g3), car(g3)=x and cdr(g3)=y.

EQ
Let the arguments evaluate to x and y. Let g4 be a fresh symbolic value, which will be returned. The fact list will be extended with:
{ g4=t & x=y } OR { g4=nil & ~(x=y) }.
A similar modification was made and implemented as described under ATOM.

EQUAL
This module works in the same way as for EQ, but generates instead:
{ g5=t & equal(x,y) } OR { g5=nil & ~equal(x,y) }.

GO
We assume only backward jumps. The loop invariant, associated with the label to which GO refers - provided by the user - parametrized for the current bindings, should be derivable from the current fact list. A non-looping check, based on a well founded relation should also be performed. After a jump the current computation branch can be ignored.

RPLACA (and analogously RPLACD)
Let the arguments of RPLACA evaluate to x and y. The precondition for RPLACA is ~atom(x). A new symbolic value, say g6, is generated, which will be returned, and the fact list will be extended with:
eqa(x,g6), car(g6)=y and cdr(g6)=cdr(x).
The next step is updating the alist, possibly by adding more new facts to the fact list, a necessary step since bindings on the alist identical to x or 'above' x are affected indirectly by the RPLACA operation. Consequently a non-atomic binding z1, which is affected, has to be replaced by a new binding z2 for which eqa(z1,z2) minimally holds. In general, when a RPLACX operation causes x1 to be replaced by x2 then each binding on the alist y1, will be replaced by a fresh binding y2, (unless it can be proven that the original binding is not affected by the x1-x2 replacement, for instance with lemma 1 and lemma 2) while the fact list will grow with:

eqaupto(y1,y2,x1,x2),
which says y2 is identical with y1 unless there is a substructure of y1 that is identical with x1. The predicate eqaupto is defined (for non cyclic arguments) as:

```
(y1)(y2)(x1)(x2){ eqaupto(y1,y2,x1,x2) <-->
[ eqa(y1,y2) &
  {y1=x1 --> y2=x2} &
  {[~(y1=x1) & ~atom(y1)] -->
   [eqaupto(car(y1),car(y2),x1,x2) &
    eqaupto(cdr(y1),cdr(y2),x1,x2) ]}]}.
```

The definition of eqaupto that allows arguments with cycles is again to be found in the appendix.
Remark: when the original binding y1 is atomic then according to axiom 1 the new binding y2 will be identical with y1, as can be checked by opening the eqaupto definition.

When the replaced value x2 is identical with x1 then the net effect is nihil as expressed by:

<u>LEMMA 1</u>.
{x1=x2 & eqaupto(y1,y2,x1,x2)} --> y1=y2.

<u>PROOF</u>. According to the above remark we can exclude the case that y1 is atomic. Obviously we can also skip the case that y1=x1 since we then immediately obtain y2=x2=x1=y1. Thus we get:
eqaupto(car(y1),car(y2),x1,x2) and
eqaupto(cdr(y1),cdr(y2),x1,x2).
Induction yields respectively car(y1)=car(y2) and cdr(y1)=cdr(y2).
Together with eqa(y1,y2) and axiom 2 we obtain y1=y2. <<

<u>LEMMA 2</u>.
{~(x1=y1) & ~partof(x1,y1) & eqaupto(y1,y2,x1,x2)} -->
y1=y2.

<u>PROOF</u>. Again we can skip the case that y1 is atomic, and we can also exclude x1=y1. Therefore we obtain:
eqaupto(car(y1),car(y2),x1,x2) and
eqaupto(cdr(y1),cdr(y2),x1,x2),
while ~partof(x1,y1) yields:

~(x1=car(y1)) & ~partof(x1,car(y1)) &
~(x1=cdr(y1)) & ~partof(x1,cdr(y1)).
Induction settles, as in lemma 1, with axiom 2 the consequence. <<

These lemmas can be used to curb updating activities.

An essential requirement for the symbolic evaluator and therefore for its modules, is that it preserves the compatability of the alist bindings. Up to RPLACX we did not have to worry about this property being violated, since the alist was not modified. The next theorem ensures that an updated alist inherits compatability from the former alist when an RPLACX induced modification occurs.

THEOREM 1. *Let* y1 *and* z1 *be old bindings which are respectively replaced by* y2 *and* z2 *due to an* RPLACX-*operation causing* x1 *to be changed into* x2, *thus with* eqa(x1,x2), *then* compatible(y1,z1), eqaupto(y1,y2,x1,x2) *and* eqaupto(z1,z2,x1,x2) *implies* compatible(y2,z2).

PROOF. Although we can give a precise formal proof here we prefer a 'loose' one. The property compatible(y1,z1) says that when two cells p and q are reachable from respectively y1 and z1 and for them holds eqa(p,q), then they must be identical, p=q. Suppose that we have such a pair (p,q) which is affected by an RPLACX-operation, causing x1 to be replaced by x2. Let p' and q' be the cells that replace respectively the cells p and q. Obviously we have p'=q' and therefore the property compatible(y2,z2) is secured. <<

Remark: Theorem 1 holds also for bindings possibly containing cycles.

SETQ
Let the second argument evaluate to x. The precondition for SETQ is that the non-evaluated first argument is atomic, say A. The binding of the leftmost occurrence of A on the alist will be replaced by x. If A does not occur on the alist - which only makes sense when A is a globally accessible variable - then (A.x) will be added to the righthand side of the alist. Preservation of alist-compatability is

ensured when the evaluation of the second argument yields a value compatible with the current bindings, i.e when the values produced by the built-in and user functions yield compatible results. Above, we treated the least obvious RPLACX, of the built-in functions. In the sequel, we treat a most troublesome class of user functions in a similar vein.

The modules not described trigger obvious updatings.

System and user functions which generate RPLACX-type side effects will have even more complicated alist updating schemes than the one given above for RPLACX. To show what is involved, we give an example of a side effect generated by the system function NCONC. This function concatenates two S-expressions by destructively modifying its first argument, using the function RPLACD.

Suppose we execute (NCONC LIS S1), where the bindings of LIS and S1 are respectively lis and s1. The rightmost leaf of S1, which must be NIL, will be replaced by a pointer to its second argument S1.

In figure 3.3, bindings for A1 and A2, a1 and a2, are introduced with a1 ´above´ lis and a2 referring to a cell on the ´spine´ of lis. The bindings of A1 and A2 are affected by the (NCONC LIS S1) action for these particular choices of lis, a1 and a2.

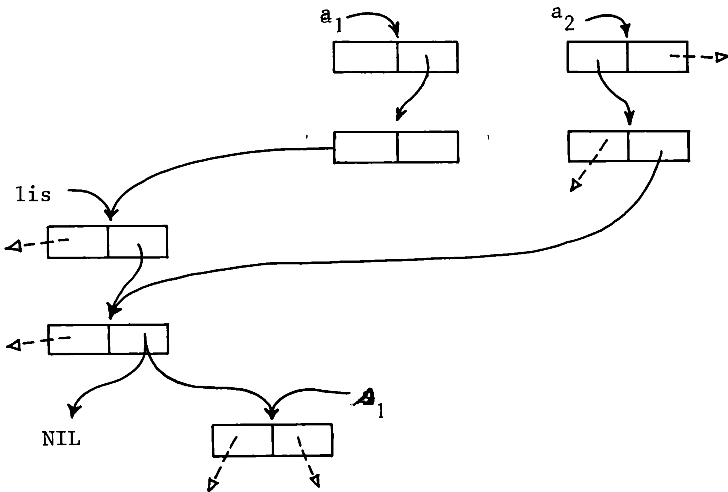


Fig. 3.3. When A_1, A_2, LIS, S_1 are bound to a_1, a_2, lis, s_1 then (NCONC LIS S1) affects the bindings of A_1 and A_2.

We will describe an alist update scheme for a class of side effect generating functions, including NCONC, EFFACE and our SUBSTAD support functions SUBSTAD1 and SUBSTAD2. It applies to those functions which cause replacement of a cell, say x1, by a cell, say x2, (thus we have eqa(x1,x2)).

Every binding, z1, on the alist will be replaced by a fresh binding, z2, and the fact list will be expanded with:
transf(z1,z2,x1,x2).
The predicate transf and its supporting predicate tr1 and tr2 works by double recursion. First, it is checked whether z1 is identical with x1, or - using tr1 - identical with a cell reachable from x1. If the tr1-case applies then the predicate tr2 is invoked to relate z1 and z2. Second, when z1 is not identical with x1 or a subcell of x1 then transf is called recursively to test whether subcells of z1 are

affected by the x1-x2 replacement.

The predicate transf is defined (for non cyclic arguments) as:

```
(y1)(y2)(x1)(x2){ transf(y1,y2,x1,x2) <-->
[ eqa(y1,y2) &
  {x1=y1 --> y2=x2} &
  {[~atom(y1) & ~(x1=y1) & tr1(y1,x1,x2)] -->
   tr2(y1,y2,x1,x2)} &
  {[~atom(y1) & ~(x1=y1) & ~tr1(y1,x1,x2)] -->
   [transf(car(y1),car(y2),x1,x2) &
    transf(cdr(y1),cdr(y2),x1,x2)]}]},
```

with tr1 defined as:

```
(y1)(x1)(x2){ tr1(y1,x1,x2) <-->
[ ~atom(x1) &
  eqa(x1,x2) &
  {y1=x1 OR
   tr1(y1,car(x1),car(x2)) OR
   tr1(y1,cdr(x1),cdr(x2))}]}, and
```

and with tr2 defined as:

```
(y1)(y2)(x1)(x2){ tr2(y1,y2,x1,x2) <-->
[{y1=x1 --> y2=x2} &
 {~(y1=x1) -->
  [{tr1(y1,car(x1),car(x2)) -->
    tr2(y1,y2,car(x1),car(x2))} &
   {tr1(y1,cdr(x1),cdr(x2)) -->
    tr2(y1,y2,cdr(x1),cdr(x2))}]}]}.
```

The meaning of the transf(z1,z2,x1,x2) formula can be phrased as follows: let y1 be z1 or a subcell of 1, let u1 be x1 or a subcell of x1, while u1 has been replaced by u2 (so u2 is identical with x2 or with a subcell of x2), then, when y1 is identical with u1, there is a corresponding cell in z2, which is identical with u2. The generalization of transf, tr1 and tr2 is to be found in the appendix.

In analogy with lemma 1 and lemma 2, we have:

<u>LEMMA 3</u>.
{x1=x2 & transf(y1,y2,x1,x2)} --> y1=y2.

<u>PROOF</u>. When y1 is atomic then we need only to invoke axiom 1; when x1=y1 we are through also. Hence we can assume ~atom(y1) and ~(x1=y1). In case we have ~tr1(y1,x1,x2), we apply car/ cdr induction as in lemma 1 and 2.

In case of tr1(y1,x1,x2) the problem reduces to:
{~(x1=y1) & tr1(y1,x1,x2) & tr2(y1,y2,x1,x2) & x1=x2} -->
y1=y2.
After opening tr1(y1,x1,x2), we may assume w.l.o.g.:
tr1(y1,car(x1),car(x2)),
while opening the tr2-formula gives:
tr2(y1,y2,car(x1),car(x2)).
Certainly, we also have: car(x1)=car(x2). When y1=car(x1), we can easily get y1=y2 from the last tr2-formula. Thus the problem reduces to:
{~(y1=car(x1) & tr1(y1,car(x1),car(x2)) &
 tr2(y1,y2,car(x1),car(x2)) & car(x1)=car(x2)} -->
y1=y2.
Car-induction settles this subproblem. <<

<u>LEMMA 4</u>.
[(z){[z=x1 OR partof(z,x1)] -->
 [~(z=y1) & ~partof(z,y1)]} &
 transf(y1,y2,x1,x2)] -->
y1=y2.

<u>PROOF</u>. The reasoning is analogous to the proof of lemma 3. <<

Invariance of the compatible-requirement for the alist is seen to ~~by:~~ be

<u>THEOREM 2</u>. *Let* y1 *and* z1 *be old bindings which are respectively replaced by* y2 *and* z2 *due to a side-effect operation causing* x1 *to be changed into* x2, *thus with* eqa(x1,x2), *then* compatible(y1,z1), transf(y1,y2,x1,x2) *and* transf(z1,z2,x1,x2) *implies* compatible(y2,z2).

<u>PROOF</u>. Case reasoning and induction on the structures of y1, y2, z1 and z2 along the lines of the proof of theorem 1. <<

Remark: Theorem 2 also holds when the bindings contain cycles.

The limitations of this updating scheme can be seen from the function NCONC2, defined as:

```
(NCONC2(LAMBDA(LIS1 LIS2 S1)
    (NCONC LIS1(NCONC LIS2 S1))))
```

A binding referring to the 'spine' of the input binding of LIS2 cannot be recognized and therefore will not be updated, even though it is no longer up to date.

Apparantly, we cannot avoid the conclusion that the user must be given the option of specifying for a function causing side-effects, a specific alist updating mechanism. The latter increments the verification burden since the compatible-requirement for the updated alist will have to be shown. Updated bindings which are in fact non-affected need potentially complicated proofs to show their invariance. In the face of these complications, one can hear the siren call to extend the PC with a formalism embodying "oldbinding=newbinding provided consistency of this assumption" [79].

3.3 The Deductive Machinery

It must be clear from the description above that deductive machinery is the backbone of program verification. Preconditions from system and user functions as well as loop invariants and output assertions need to be deduced.

A code verification program is best designed with two cooperating components: a language specific verification condition generator and a general deductive component (possibly implemented only as an interactive request to the user). The interaction between the verification condition generator and the deductive component, when accomplished by throwing the fact list and the conjecture at the deductive component, is conceptually nice and an easy task for the generator. From the perspective of the deduction component, it must be considered minimal. A cross-indexed structure would already simplify

searches for matings. Direct availability of redundant knowledge, concerning for instance virtualness of data objects, may prevent the attack of hopeless subgoals. For further discussion on the necessity of non-minimal problem specifications, see the next chapter.

Since general deductive machines are fairly weak, it pays to give the deduction component language specific knowledge. In our case, it is of great help to have a procedural ´watch dog´ for car/cdr induction when recursive definitions are expanded. This conforms with the methodology of distributing deductive power over separate, cooperating, procedural specialists.

Furthermore, it is advisable to build some simple deductive knowledge, specific for LISP, into the precondition check generators. An obvious tactic is to check for membership in the fact list when a verification condition is a simple literal formula.

We end this section with an example showing that a deductive specialist which selectively opens recursive definitions simplifies proofs.

Assume that the EQUAL function of section 2 is a user defined function instead of a system function. It would not have preconditions since there are no restrictions on the two arguments. Suppose the arguments evaluate to x and y and out is the value returned by EQUAL. We now have to check for every path through the EQUAL code:
{ out=t & equal(x,y) } OR { out=nil & ~equal(x,y) }.
We repeat the EQUAL definition:

```
(EQUAL(LAMBDA(S1 S2)(COND
     ((EQ S1 S2)T)
     ((ATOM S1)NIL)
     ((ATOM S2)NIL)
     ((EQUAL(CAR S1)(CAR S2))
      (EQUAL(CDR S1)(CDR S2)))
     (T NIL)
)))?end of equal?
```

The alist for all paths will be:
((S1.x)(S2.y)).

Path 1. This path will be taken when the first test expression (EQ S1 S2) is assumed to return T. As a consequence, the fact list will have been expanded with x=y. EQUAL will return with T, which leads to the substitution of t for out, yielding as output assertion to be checked:

```
{ t=t & equal(x,y) } OR { t=nil & ~equal(x,y) }.
```

As a consequence of ~(t=nil), there is no hope that the second part of the disjunction can play a role. So we have to prove:

```
x=y --> { t=t & equal(x,y) }.
```

The first subproblem is trivial and remains:

```
x=y --> equal(x,y).
```

Opening up equal(x,y) gives:

```
x=y --> { x=y OR
          [ ~atom(x) & ~atom(y) &
            equal(car(x),car(y)) &
            equal(cdr(x),cdr(y)) ]},
```

which is obviously correct.

Path 2. Since path 1 is not chosen, the fact list will have been expanded with ~(x=y), due to (EQ S1 S2)=NIL, and with atom(x), due to (ATOM S1)=T. EQUAL returns with NIL, so we have to deal with:

```
[ atom(x) & ~(x=y) ] -->
[ { nil=t & equal(x,y) } OR { nil=nil & ~equal(x,y) }].
```

For the same reasons as above, we better try:

```
[ atom(x) & ~(x=y) ] --> ~equal(x,y),
```

or after expansion:

```
[ atom(x) & ~(x=y) ] -->
~{ x=y OR
   [ ~atom(x) & ~atom(y) &
     equal(car(x),car(y)) &
     equal(cdr(x),cdr(y)) ]},
```

or after working the negation inwards:

```
[ atom(x) & ~(x=y) ] -->
{ ~(x=y) &
   [ atom(x) OR atom(y) OR
     ~equal(car(x),car(y)) OR
     ~equal(cdr(x),cdr(y)) ]}.
```

This again is correct.

Path 3. This requires proving:

```
[ atom(y) & ~atom(x) & ~(x=y) ] -->
[ { nil=t & equal(x,y) } OR { nil=nil & ~equal(x,y) }].
```

This is analogous as in path 2.

Path 4. This is the interesting case. EQUAL will return with out2. The fact list will be:

```
(I) ~(x=y) & ~atom(x) & ~atom(y) &
    [ { out1=t & equal(car(x),car(y)) } OR
      { out1=nil & ~equal(car(x),car(y)) } ] & out1=t &
    [ { out2=t & equal(cdr(x),cdr(y)) } OR
      { out2=nil & ~equal(cdr(x),cdr(y)) } ].
```

The symbolic evaluator when confronted with the code:

```
(EQUAL(CAR S1)(CAR S2))
```

will recognize that the function EQUAL is under consideration and should not be opened again. As argued before, it takes for granted that {(CAR S1) (CAR S2)} is ´less´ than {S1 S2} according to some well ordered relation. The output assertion parametrized for the current arguments will then be added to the fact list. The recursive call on the CDR´s is treated similarly. The instantiated output assertion will be:

```
(II) { out2=t & equal(x,y) } OR { out2=nil & ~equal(x,y) }.
```

We have to show: (I) --> (II).

First we simplify (I) to:

```
~(x=y) & ~atom(x) & ~atom(y) &
equal(car(x),car(y)) &
[ { out2=t & equal(cdr(x),cdr(y)) } OR
  { out2=nil & ~equal(cdr(x),cdr(y)) } ].
```

Case reasoning with out2=t requires:

```
[ ~(x=y) & ~atom(x) & ~atom(y) &
  equal(car(x),car(y)) &
  equal(cdr(x),cdr(y)) ] -->
equal(x,y).
```

Expanding equal(x,y) leads to success.

The other case with out2=nil requires:

```
[ ~(x=y) & ~atom(x) & ~atom(y) &
  equal(car(x),car(y)) &
  ~equal(cdr(x),cdr(y)) ] -->
```

~equal(x,y).

Opening ~equal(x,y) again and subsequently working ~ inwards settles this case as well.

Path 5. The fact list will have grown to:

(I) ~(x=y) & ~atom(x) & ~atom(y) &
[{ outl=t & equal(car(x),car(y)) } OR
{ outl=nil & ~equal(car(x),cdr(y)) }] & outl=nil.

The instantiated output assertion will be:

(II) { nil=t & equal(x,y) } OR { nil=nil & ~equal(x,y) }.

Formula (I) simplifies to:

~(x=y) & ~atom(x) & ~atom(y) &
~equal(car(x),car(y)).

Formula (II) simplifies to: ~equal(x,y). Expanding this formula and working ~ inwards again settles this last path.

We expect the reader to be surprised by the great abundance of detail, though much which is to be handled by a mechanical theorem prover, has been omitted for such a simple example. In fact, we suspect that most resolution type theorem provers would have already been choked by the generation of garbage clauses when attempting the path 4 check.

The phenomen has occurred several times above, and we have observed it frequently elsewhere, that when say a disjunction P OR Q has to be proven, the context allows one to prove a stronger result, for instance P. The ability to recognize these situations simplifies the task for blind-search deductive components. Models can be employed for pinpointing essential parts of formulas. For example, when S1 is bound to x and the evaluation of (CAR S1) in an available environment yields an error then the prover can concentrate on proving atom(x) when it has to prove atom(x) OR car(x)=y.

4. Verification of the Substitution Functions

In this section, we will verify the substitution functions SUBSTSUPF1, SUBSTSUPF2, SUBSTAD1, SUBSTAD2 and SUBSTADP as defined in section 2. They will not be treated in as much detail as was the case for EQUAL in the former section. Rather, we will spell out only the most interesting paths through the code since we feel that making the deductions, although no trivial task, is not the most difficult. Accurate formulation of the output assertions and loop invariants is more challenging.

The proofs presented here for the functions SUBSTSUPF1 and SUBSTAD1 are basically transcriptions of the proofs constructed by the symbolic evaluator and the deduction program. Although more laborious we expect the proofs for SUBSTSUPF2 also to be in the realm of the currently available deductive component. Extensions to be made to the deduction complex to facilitate handling of the proofs for SUBSTAD2 are still conceivable. Mechanical proofs for SUBSTADP, the pointer reversal version, have not been attempted due to the practical infeasibility of formulating the general version of the loop invariants.

4.1 SUBSTSUPF1

We begin with the free space gobbler SUBSTSUPF1 and repeat its definition:

```
(SUBSTSUPF1(LAMBDA(S3)(COND
     ((EQUAL S2 S3)S1)
     ((ATOM S3)S3)
     (T(CONS(SUBSTSUPF1(CAR S3))
            (SUBSTSUPF1(CDR S3))))
))).
```

There are no preconditions, it is only required that S1, S2 and S3 have bindings on the alist, say
((S1.vs1) (S2.vs2) (S3.vs3)).
The output assertion, assuming that the value returned will be out, is:
replacedn(vs1,vs2,vs3,out),
where the predicate replacedn (replacement without destruction) is

defined as:

```
(x1)(x2)(x3)(ot){replacedn(x1,x2,x3,ot) <-->
   [ (equal(x2,x3) --> ot=x1) &
     {(~equal(x2,x3) & atom(x3)) --> ot=x3} &
     {(~equal(x2,x3) & ~atom(x3)) -->
      (~atom(ot) &
       replacedn(x1,x2,car(x3),car(ot)) &
       replacedn(x1,x2,cdr(x3),cdr(ot))) } ] }.
```

That the proofs for the verification conditions corresponding to the three distinct paths through the code are straightforward is a consequence of the ´isomorphism´ between the code of the function SUBSTSUPF1 and the definition of the predicate replacedn (as was the case with EQUAL and equal). We confine ourselves to the third and most interesting path.

As a consequence of not entering path 1 and 2, we have on the fact list (after simplification):
~equal(vs2,vs3) & ~atom(vs3).
The first recursive CAR-call will return a value out1 and will add to the fact list:
va=car(vs3) & replacedn(vs1,vs2,va,out1).
The second recursive CDR-call will return a value out2 and will add:
vb=cdr(vs3) & replacedn(vs1,vs2,vb,out2).
Finally the CONS-call will return with out3 and will add:
~atom(out3) & car(out3)=out1 & cdr(out3)=out2.
The value out3 will be returned by the toplevel SUBSTSUPF1 call and so we need to verify:
replacedn(vs1,vs2,vs3,out3).
Opening this formula once generates three subproblems which can be dealt with straightforwardly.

4.2 SUBSTSUPF2

The other support function SUBSTSUPF2 is more careful with free space consumption:

```
(SUBSTSUPF2(LAMBDA(S3)(PROG(X)(RETURN(COND
     ((EQUAL S2 S3)S1)
     ((ATOM S3)S3)
     ((EQ(CAR S3)(SETQ X(SUBSTSUPF2(CAR S3))))
      (COND((EQ(CDR S3)(SETQ X(SUBSTSUPF2(CDR S3))))S3)
           (T(CONS(CAR S3)X))))
     (T(CONS X(SUBSTSUPF2(CDR S3))))
))))).
```

We could state the output assertion of this function by again using replacedn, but this would not acknowledge its more austere behavior. A more subtle output assertion is:
replacedn2(vs1,vs2,vs3,out),
with replacedn2 defined as:

```
(x1)(x2)(x3)(ot){replacedn2(x1,x2,x3,ot) <-->
   [ (equal(x2,x3) --> ot=x1) &
     {(~equal(x2,x3) & atom(x3)) --> ot=x3} &
     {(~equal(x2,x3) & ~atom(x3) & ~occure(x2,x3)) -->
      ot=x3} &
     {(~equal(x2,x3) & ~atom(x3) & occure(x2,x3)) -->
      (~atom(ot) &
       replacedn2(x1,x2,car(x3),car(ot)) &
       replacedn2(x1,x2,cdr(x3),cdr(ot)) )}]},
```

where occure is defined as:

```
(x2)(x3){occure(x2,x3) <-->
          (occurecar(x2,x3) OR occurecdr(x2,x3))},
```

and occurecar and occurecdr are defined as:

```
(x2)(x3){occurecar(x2,x3) <-->
          [~atom(x3) &
           (equal(x2,car(x3)) OR occure(x2,car(x3))) ]}, and

(x2)(x3){occurecdr(x2,x3) <-->
          [~atom(x3) &
           (equal(x2,cdr(x3)) OR occure(x2,cdr(x3))) ]}.
```

The predicate replacedn2 is like replacedn, but specifies in addition that the ´output´ ot is identical to the ´input´ x3 when x2 does not occur somewhere inside the car or the cdr of x3.

There are five paths through the code. We will concentrate on path 3, where (EQ(CDR S3)...) evaluates to T, and path 5, on which (CONS X(...)) gets evaluated.

Path 3. First we describe the situation after evaluation of (EQ(CAR...)) and after assuming that its result was T. The alist will contain a binding for X:

((X.vx1) (S1.vs1) (S2.vs2) (S3.vs3)).

The fact list will be:

~equal(vs2,vs3) & ~atom(vs3) &
xa=car(vs3) & replacedn2(vs1,vs2,xa,vx1) & vx1=xa.

After evaluation of (EQ(CDR...)) and assuming again that its result is T we get another binding for X, say vx2, and to the fact list will have been added:

xd=cdr(vs3) & replacedn2(vs1,vs2,xd,vx2) & vx2=xd.

Since the value returned is vs3 we must infer:

replacedn2(vs1,vs2,vs3,vs3).

Opening this formula, using the definition of rplacedn2 yields four subproblems:

(1) equal(vs2,vs3) --> vs3=vs1, this is trivially solved since the negation of the premise belongs to the fact list;

(2) ~equal(vs2,vs3) & atom(vs3) --> vs3=vs3, this is obviously correct;

(3) ~equal(vs2,vs3) & ~atom(vs3) & ~occure(vs2,vs3) --> vs3=vs3, this subproblem is also trivial;

(4) ~equal(vs2,vs3) & ~atom(vs3) & occure(vs2,vs3) -->
[~atom(vs3) &
replacedn2(vs1,vs2,car(vs3),car(vs3)) &
replacedn2(vs1,vs2,cdr(vs3),cdr(vs3))],
again this leads to three trivially solvable subproblems.

Path 5. Just before entering CONS, the fact list will be (remember X is still bound to vx1):

~equal(vs2,vs3) & ~atom(vs3) &
xa=car(vs3) & replacedn2(vs1,vs2,xa,vx1) &
~(vx1=xa).

After CONS-ing X with the recursive call on the CDR, the fact list will have grown with:

xd=cdr(vs3) & replacedn2(vs1,vs2,xd,ot1) &
car(ot2)=vx1 & cdr(ot2)=ot1 & ~atom(ot2).

Since the value returned will be ot2, the parametrized output assertion will be:

replacedn2(vsl,vs2,vs3,ot2).
Opening this formula leads to four subproblems, of which the first two solve immediately since the fact list contains the negation of their premises. Therefore remain:

```
(3) ~equal(vs2,vs3) & ~atom(vs3) & ~occure(vs2,vs3) -->
    ot=vs3, and
(4) ~equal(vs2,vs3) & ~atom(vs3) & occure(vs2,vs3) -->
    [ ~atom(ot2) &
      replacedn2(vsl,vs2,car(vs3),car(ot2)) &
      replacedn2(vsl,vs2,cdr(vs3),cdr(ot2)) ].
```

Simple substitutions make the three consequences in (4) equal to formulas of the fact list and settle (4). Subproblem (3) is solved by deriving occure(vs2,vs3) from the fact list and thus squeezing the premise of (3). We do this with:

<u>LEMMA 5</u>.

```
[ replacedn2(vsl,vs2,xa,vxl) &
  ~(vxl=xa) &
  ~atom(vs3) &
  xa=cxr(vs3) { cxr is car or cdr } ] -->
occure(vs2,vs3).
```

<u>PROOF</u>. By expanding replacedn2 we distinguish four cases:

(1) equal(vs2,xa) and vxl=vsl. Thus certainly we get occurecar(vs2,xa) or occurecdr(vs2,xa), and therefore occure(vs2,vs3).

(2) ~equal(vs2,xa) & atom(xa) and vxl=xa. The preconditions of the lemma exclude this case.

(3) ~equal(vs2,xa) & ~atom(xa) & ~occure(vs2,xa) and xa=vxl. This case like (2) is also excluded by the preconditions.

(4) ~equal(vs2,xa) & ~atom(xa) & occure(vs2,xa) and [...]. Combining occure(vs2,xa) and xa=cxr(vs3) we get our conclusion by expanding occure(vs2,vs3). <<

Application of this theorem also settles path 5.

Although the output generated when SUBSTSUPF1 is used differs from the output generated when SUBSTSUPF2 is employed, the two must nevertheless be EQUAL. This corresponds with:

LEMMA 6.
{ replacedn(x1,x2,x3,ot1) & replacedn2(x1,x2,x3,ot2) } -->
equal(ot1,ot2).

PROOF. We expand replacedn and replacedn2 and consider the different cases.

(1) equal(x2,x3) yields ot1=x1 and ot2=x1 and thus certainly equal(ot1,ot1) (by opening equal).
(2) ~equal(x2,x3) & atom(x3) yields ot1=x3 and ot2=x3, and so we have the same argument as in case (1).
(3) ~equal(x2,x3) & ~atom(x3) & ~occure(x2,x3) yields
    I ~atom(ot1) &
      replacedn(x1,x2,car(x3),car(ot1)) &
      replacedn(x1,x2,cdr(x3),cdr(ot1)), and
    II x3=ot2.
    Application of car/cdr induction on equal, occure and replacedn gives equal(x3,ot1) and thus equal(ot2,ot1) as well.
(4) ~equal(x2,x3) & ~atom(x3) & occure(x2,x3) yields
    I ~atom(ot1) &
      replacedn(x1,x2,car(x3),car(ot1)) &
      replacedn(x1,x2,cdr(x3),cdr(ot1)), and
    II ~atom(ot2) &
      replacedn2(x1,x2,car(x3),car(ot2)) &
      replacedn2(x1,x2,cdr(x3),cdr(ot2)).
    By induction we infer:
    equal(car(ot1),car(ot2)) and
    equal(cdr(ot1),cdr(ot2)),
    and so also equal(ot1,ot2). <<

As a consequence of this lemma, we have:

THEOREM 3. *The output of* SUBST *with support function* SUBSTSUPF1 *is* EQUAL *to the output* of SUBST *when the support function* SUBSTSUPF2 *is used instead.*

4.3 SUBSTAD1

Now we switch to the support functions for the destructive substitution function SUBSTAD. The code for the recursive SUBSTAD1:

```
(SUBSTAD1(LAMBDA(S3)(PROG2
     (COND((ATOM(CAR S3))
           (COND((EQ LAT(CAR S3))(RPLACA S3 S1))))
          (T(SUBSTAD1(CAR S3))))
     (COND((ATOM(CDR S3))
           (COND((EQ LAT(CDR S3))(RPLACD S3 S1))))
          (T(SUBSTAD1(CDR S3))))
))).
```

The preconditions are:
- the binding of S3, say vs3, is not atomic;
- the binding of LAT, say lat, is atomic; and
- lat is not a leaf of the binding of S1, say vs1.

cf. Thm 4.

To simplify the proofs, we will also assume that vs1 does not share substructure with vs3. Consequently, lemma 4 will apply and therefore updating of the S1 binding will never occur (when vs1 does share structure we still can invoke lemma 2, since lat is not a leaf of vs1).

Since we assume the preconditions to hold, the fact list will contain (or these forms can be derived from the fact list):
atom(lat) & ~atom(vs3) & ~partof(lat,vs1).
The input alist is:
((S1.vs1) (LAT.lat) (S3.vs3)).
Assume the output alist to be:
((S1.vs1) (LAT.lat) (S3.nvs3)).
The output assertion to be verified will be:
replacedd(vs1,lat,vs3,nvs3),
with replacedd (replacement with potential destruction of vs3) defined as:

```
(x1)(x2)(x3)(ot){replacedd(x1,x2,x3,ot) <-->
[ eqa(x3,ot) &
  {atom(car(x3)) -->
   [(x2=car(x3) --> car(ot)=x1) &
    (~(x2=car(x3)) --> car(ot)=car(x3)) ]} &
  {~atom(car(x3)) --> replacedd(x1,x2,car(x3),car(ot)) } &
  {atom(cdr(x3)) -->
   [(x2=cdr(x3) --> cdr(ot)=x1) &
    (~(x2=cdr(x3)) --> cdr(ot)=cdr(x3)) ]} &
  {~atom(cdr(x3)) --> replacedd(x1,x2,cdr(x3),cdr(ot)) }]}.
```

We also postulate that the alist updating scheme using transf, as described in section 3.2, applies.

There are 9 different paths through the code, consisting of different combinations of the three distinct paths through the two top COND's. We will work our way through just one of the paths.

Initially the fact list contains:
atom(lat) & ~atom(vs3) & ~partof(lat,vs1).
Assuming that (ATOM(CAR S3)) yields T we get in addition:
xa=car(vs3) & atom(xa).
Assuming that (EQ LAT(CAR S3)) yields T we also get on the fact list:
lat=xa.
The subsequent RPLACA action will generate a new value, say nv1, and will add:
eqa(nv1,vs3) & car(nv1)=vs1 & cdr(nv1)=cdr(vs3).
The alist update scheme for RPLACA prescribes the generation of a new binding for S3, say ivs3, and the alist will change into:
((S1.vs1) (LAT.lat) (S3.ivs3)),
while the fact list grows with:
eqaupto(vs3,ivs3,vs3,nv1).
Assuming that (ATOM(CDR S3)) yields NIL we get on the fact list:
xd=cdr(ivs3) & ~atom(xd).
The next action concerns the recursive call on the CDR. Its parametrized and simplified input condition:
~atom(xd) & atom(lat) & ~partof(lat,vs1),
is trivially satisfied. The function will not be opened, but instead the fact list grows with:
replacedd(vs1,lat,xd,nxd) & transf(ivs3,jvs3,xd,nxd),
while the alist changes again into:

((S1.vs1) (LAT.lat) (S3.jvs3)).
The output assertion to be proven for this particular path will be:
replacedd(vs1,lat,vs3,jvs3).

Opening this formula produces five subproblems:

(1) eqa(vs3,jvs3).
By expanding the given eqaupto formula we derive eqa(vs3,ivs3). Opening the transf formula yields eqa(ivs3,jvs3). As a consequence of the transitivity of eqa, this case is closed.

```
(2) atom(car(vs3)) -->
    [ (lat=car(vs3) --> car(jvs3)=vs1) &
      (~(lat=car(vs3)) --> car(jvs3)=car(vs3)) ].
```

Since the premise of this implication holds as well as the premise of the first implication in the consequence, this problem reduces to: car(jvs3)=vs1. From the eqaupto formula we can infer that ivs3=nv1 and so we certainly have: vs1=car(nv1)=car(ivs3). Informally we can argue that since lat isn't a leaf of vs1, ~partof(lat,vs1), any replacement inside cdr(ivs3) will not affect vs1. More precisely we have to prove:

```
{xd=cdr(ivs3) & ~atom(xd) &
 replacedd(vs1,lat,xd,nxd) &
 transf(ivs3,jvs3,xd,nxd) &
 vs1=car(ivs3) &
 ~partof(lat,vs1)} -->
car(ivs3)=car(jvs3).
```

Loopfreeness allows us to infer: ~(xd=ivs3) as well as ~trl(ivs3,xd,nxd). Consequently we infer by expanding the transf formula:
transf(car(ivs3),car(jvs3),xd,nxd).
We rewrite our problem as:

```
{~atom(xd) &
 replacedd(vs1,lat,xd,nxd) &
 transf(car(ivs3),car(jvs3),xd,nxd) &
 vs1=car(ivs3) &
 ~partof(lat,vs1)} -->
car(ivs3)=car(jvs3).
```

Now we distinguish whether car(ivs3) (=vsl) is atomic or not.

Assume vsl is atomic. Since we can infer from the transf formula: eqa(car(ivs3),car(jvs3)), we can activate axiom 1 to obtain: car(ivs3)=car(jvs3).

Assume vsl is not atomic. Let us abbreviate car(car(ivs3)) by ia and car(car(jvs3)) by ja. We will show ia=ja and since the same argument will apply to cdr(car(ivs3)) and cdr(car(jvs3)) we can invoke axiom 2, because we also have eqa(car(ivs3),car(jvs3)). This line of reasoning leads us again to our goal.

Assume ia=xd, then loopfreeness and subsequently opening transf(car(ivs3),car(jvs3),xd,nxd) yields ja=nxd. By induction, we can prove xd=nxd, from replacedd(vsl,lat,xd,nxd) and ~partof(lat,vsl) (which gives ~partof(lat,xd)). And this gives ia=ja.

Assume ~(ia=xd) and trl(ia,xd,nxd). We now have reduced our problem to:

```
{~(ia=xd) & ~atom(xd) &
 tr2(ia,ja,xd,nxd) &
 ~partof(lat,ia) &
 replacedd(vsl,lat,xd,nxd)} -->
ia=ja,
```

which can be settled with car/cdr induction on xd and nxd.

Assume ~(ia=xd) and ~trl(ia,xd,nxd), we can infer:
transf(car(ia),car(ja),xd,nxd),
and thus our problem reduces to:

```
{~atom(xd) &
 replacedd(vsl,lat,xd,nxd) &
 transf(car(ia),car(ja),xd,nxd) &
 ~partof(lat,car(ia))} -->
car(ia)=car(ja),
```

and similarly for cdr(ia) and cdr(ja). This is of the same form as the problem we started with and so by car/cdr induction we obtain car(ia)=car(ja) as well as cdr(ia)=cdr(ja). Consequently we apply axiom 2 to obtain ia=ja.

(3) ~atom(car(vs3)) --> ...

The negation of the premise belongs to the fact list and so we can skip this case.

(4) atom(cdr(vs3)) --> [...]

We have cdr(vs3)=cdr(nvl)=cdr(ivs3)=xd. We also have ~atom(xd); thus the negation of the premise can be inferred from the fact list.

(5) ~atom(cdr(vs3)) -->
replacedd(vsl,lat,cdr(vs3),cdr(jvs3)).

This problem reduces after two substitutions to:
replacedd(vsl,lat,xd,xdn),
which is a member of the fact list.

Thus we have completed this path. The other paths can be verified in a similar way. But subproblem (5) is somewhat hairy for on this path there is CAR as well as CDR recursion. In that case cdr(vs3)=cdr(ivs3) can no longer be inferred due to the possibility of structure sharing and RPLACX operations on shared cells.

It is certainly to be expected that SUBSTAD and SUBST produce EQUAL results under SUBSTAD´s restrictions.

When we have dealt with the special case that the third argument S3 has an atomic binding then it remains to show:

<u>LEMMA 7.</u>

```
(x1)(x2)(x3)(ot1)(ot2)
[ {atom(x2) & ~atom(x3) &
   replacedn(x1,x2,x3,ot1) & replacedd(x1,x2,x3,ot2)} -->
  equal(ot1,ot2) ].
```

<u>PROOF</u>. By induction it is easy to see that ~atom(x3) implies ~atom(ot1) as well as ~atom(ot2). Therefore it suffices to show:
(1) equal(car(ot1),car(ot2)), and
(2) equal(cdr(ot1),cdr(ot2)).
The proof of (2) is analogous to the proof of (1), so we concentrate on (1) by digging into three cases:

(I) atom(car(x3)) & x2=car(x3).
Twice opening the replacedn-formula (since we have ~equal(x2,x3) and ~atom(x3) yielding replacedn(x1,x2,car(x3),car(ot1))) allows us to infer car(ot1)=x1. Once opening the replacedd-formula gives car(ot1)=x1, thus certainly: equal(car(ot1),car(ot2)).

(II) atom(car(x3)) & ~(x2=car(x3)).
We conclude car(ot2)=car(x3) as well as ~equal(x2,car(x3)) and so also car(ot1)=car(x3). Therefore we have again equal(car(ot1),car(ot2)).

(III) ~atom(car(x3))
We infer ~equal(x2,car(x3)) and thus we have:
replacedn(x1,x2,car(car(x3)),car(car(ot1))) and
replacedn(x1,x2,cdr(car(x3)),cdr(car(ot1))).
By car/cdr-induction, we get equal(car(ot1),car(ot2)). <<

As a consequence of this lemma, we have:

THEOREM 4. *The functions* SUBST *and* SUBSTAD *with support function* SUBSTAD1 *produce* EQUAL *results when the second argument is a literal atom and does not occur in the first argument.*

4.4 SUBSTAD2

We have to admit that the treatment of SUBSTAD1 as given above was slightly incorrect. Although it did not affect the result. The reason is that SUBSTAD does not use the value returned by SUBSTAD1, which is of no significance. Upon entry of SUBSTAD1 the alist is in fact:
((S3.vs3) (S1.vs1) (LAT.lat) (S3.vs3)),
where the first occurrence of S3 comes from SUBSTAD1 and the second from SUBSTAD. The output assertion of SUBSTAD1 did refer to the *second* occurrence of vs3. This more subtle treatment of the alist is essential for the half recursive half iterative support function SUBSTAD2. We first repeat its definition:

```
(SUBSTAD2(LAMBDA(S3)(PROG(HH)
AGAIN
     (COND((ATOM(SETQ HH(CAR S3)))
           (COND((EQ LAT HH)(RPLACA S3 S1))))
          (T(SUBSTAD2 HH)))
     (COND((ATOM(SETQ HH(CDR S3)))
           (COND((EQ LAT HH)(RPLACD S3 S1))))
          (T(SETQ S3 HH)
            (GO AGAIN)))
)))
```

Due to the assignment of the local S3 to its CDR before jumping back to AGAIN, the local S3 is not significant when SUBSTAD2 is exiting. The global S3 is the handle on the datastructure as a whole and enables a correct update of the calling environment after exiting.

The input alist is as given above. The output alist, after exiting from SUBSTAD2 will be:
((S1.vs1) (LAT.lat) (S3.nvs3)).
The preconditions are the same as for SUBSTAD1:
atom(lat) & ~atom(vs3) & ~partof(lat,vs1).
The output assertion is also the same:
replacedd(vs1,lat,vs3,nvs3).
The major difference with SUBSTAD1 is that we have to provide a loop invariant, since the body of SUBSTAD2 contains the label AGAIN. The loop assertion will refer to the current bindings of the variables and thus we must also give an alist at the label:
((HH.vhh) (S3.ls3) (S1.vs1) (LAT.lat) (S3.gs3)).
The value ls3 is the local value of S3, and gs3 is the global value of S3. The loop assertion will be:
atom(lat) & ~atom(ls3) & ~atom(vs3) & ~partof(lat,vs1) &
spine(vs1,lat,vs3,gs3,ls3).
Before giving the definition of spine and other support predicates, we sketch the situation at the label AGAIN, see figure 3.4.

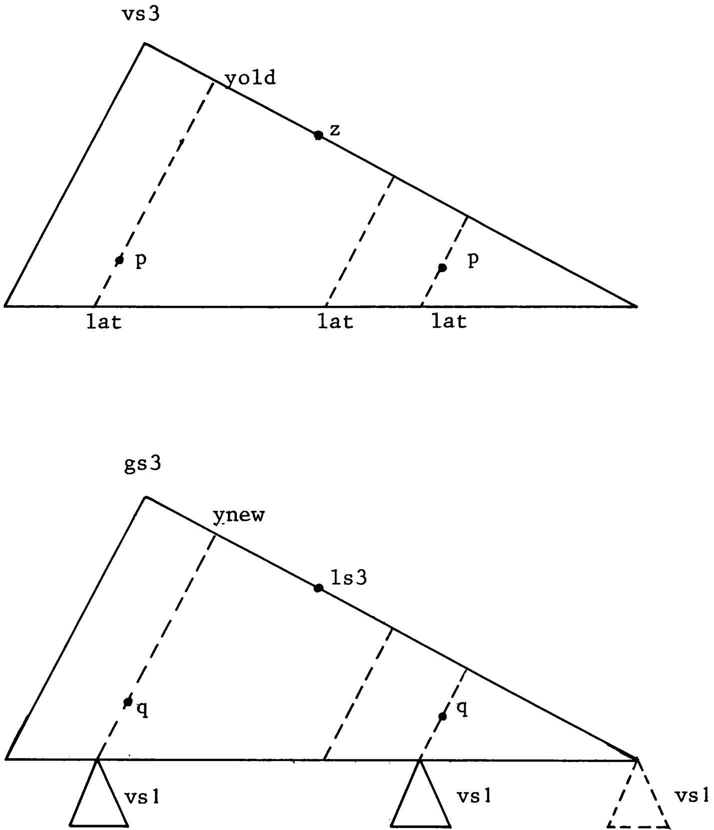


Fig. 3.4. The top triangle stands for the original binding vs3, the other triangle represents the situation at the label AGAIN. The global S3 binding is gs3, the local S3 binding is ls3. Structure sharing (p to the left of z is identical with p to the right of z) has caused the modification of the 'not yet visited' right-hand side part of z-ls3.

The binding of the S3 outside the body of SUBSTAD2 changed possibly due to RPLACX actions and is set to gs3. The local S3, bound to ls3, points to a cell on the spine of gs3, the current incarnation of the cdr-chain sprouting at vs3. Every leaf at the left of ls3 has already been investigated and appropriate modifications have been made. So for a non-atomic side tree at car(yold), hanging off yold on the spine, there is a corresponding side tree car(ynew) with ynew on the spine of vs3 above ls3, for which we have:
replacedd(vsl,lat,car(yold),car(ynew)).
The subtree hanging at ls3 need not be identical with the corresponding subtree hanging at z (of course we have eqa(z,ls3)), since structure sharing may have led to side effects in a subtree of z. For instance, the replacement of the subtree p, occurring twice in vs3 at the lefthand side of z, by q, simultaneously affects the occurrence at the righthand side of z. Although ls3 is on the spine of gs3, there is not necessarily a corresponding cell z on the spine of vs3. Structure sharing may have caused the replacement of the right most leaf of vs3 by a pointer to vsl, see fig. 3.4. So ls3 may eventually reside on the spine of vsl for which there is no corresponding cell on the spine of vs3 (hence the precondition ~partof(lat,vsl), to avoid cycles). The loop invariant is expressed by the predicates spine, spinel, spine2, sidetree, onspine and sidefct.

The predicate spine distinguishes between the special case that xl=x3, which only holds upon entrance of SUBSTAD2, and otherwise gives the responsability to spinel for describing the situation. When the latter holds we are assured that xl resides somewhere on the spine below xg.

```
(xl)(xa)(x3)(xg)(xl)
          { spine(xl,xa,x3,xg,xl) <-->
[(xl=x3 --> xg=x3) &
 (~(xl=x3) --> spinel(xl,xa,x3,xg,xl,x3,xg))]}.
```

The predicate spinel slides along the cdr-chains of x3-xg. It expresses that the car´s are properly investigated and possibly updated, using sidetree. When the bottom of the x3-spine is hit, structure sharing has caused replacement of the right most leaf of x3 and control resides somewhere on the spine of xl, which is expressed with onspine. When the xl-cell on the spine of gs3 is reached instead, sidefct is used to describe the remainder of x3-xg still to be

investigated, possibly modified as a consequence of structure sharing.

```
(xl)(xa)(x3)(xg)(xl)(xo)(xn)
         { spinel(xl,xa,x3,xg,xl,xo,xn) <-->
[eqa(x3,xg) &
 sidetree(xl,xa,car(x3),car(xg)) &
 {atom(cdr(x3)) -->
  [cdr(x3)=xa & cdr(xg)=xl & onspine(xl,xl)]} &
 {~atom(cdr(x3)) -->
  [{xl=cdr(xg) -->
    sidefct(xo,xn,cdr(x3),xo,xn,cdr(x3),xl)} &
   {~(xl=cdr(xg)) -->
    spinel(xl,xa,cdr(x3),cdr(xg),xl,xo,xn)}]}]}.
```

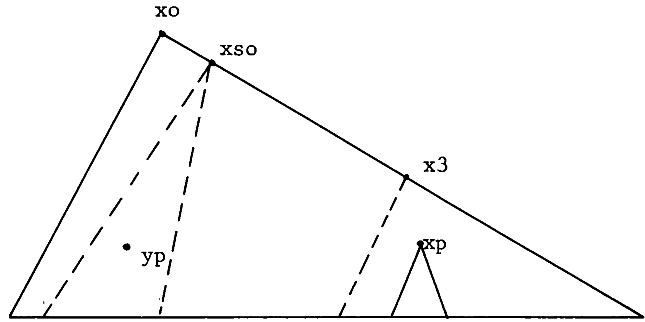
The predicate sidetree simply separates whether the car of an already visited spine element was originally atomic or not and describes in each case the possibly updated result.

```
(xl)(xa)(x3a)(xga)
         { sidetree(xl,xa,x3a,xga) <-->
[{~atom(x3a) --> replacedd(xl,xa,x3a,xga)} &
 {atom(x3a) --> [{x3a=xa --> xga=xl} &
                 {~(x3a=xa) --> xga=x3a}]}]}.
```

The predicate onspine says only that xl is somewhere on the spine of xl.

```
(xl)(xl){ onspine(xl,xl) <-->
[xl=xl OR {~atom(xl) & onspine(cdr(xl),xl)}]}.
```

The predicate sidefct is used to describe the fact that xp-xq which is a part of the not yet visited subtree x3-xl of the original-current incarnation xo-xn is unchanged unless structure sharing has led to side effects (the predicates tr1 and tr2 that occur in the body of sidefct have been defined above in section 3), see fig. 3.5. which depicts the parameters of sidefct.



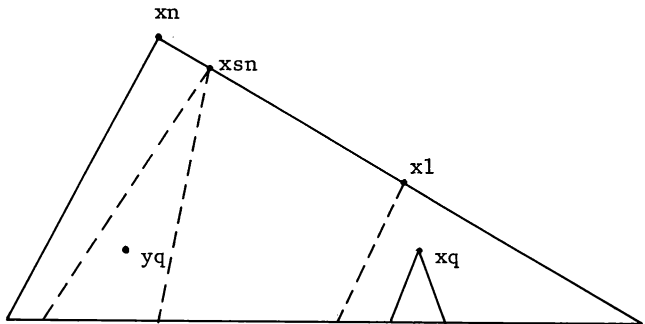


Fig. 3.5. The top triangle again is the original binding, the other one is the current binding. The predicate sidefct is used to express that when xp inside x3 corresponds with yp inside xso (xso-xsn has already been visited) then xq corresponds with yq.

```
(xo)(xn)(x3)(xso)(xsn)(xp)(xq)
         { sidefct(xo,xn,x3,xso,xsn,xp,xq) <-->
[eqa(xp,xq) &
 {xso=x3 -->
  [{atom(car(xp)) --> car(xp)=car(xq)} &
   {~atom(car(xp)) -->
    sidefct(xo,xn,x3,xo,xn,car(xp),car(xq))} &
   {atom(cdr(xp)) --> cdr(xp)=cdr(xq)} &
   {~atom(cdr(xp)) -->
    sidefct(xo,xn,x3,xo,xn,cdr(xp),cdr(xq))}]} &
 {~(xso=x3) -->
  [{car(xso)=xp --> car(xsn)=xq} &
   {~(car(xso)=xp) -->
    [{trl(xp,car(xso),car(xsn)) -->
      tr2(xp,xq,car(xso),car(xsn))} &
     {~trl(xp,car(xso),car(xsn)) -->
      sidefct(xo,xn,x3,cdr(xso),cdr(xsn),xp,xq)}]}]}]}.
```

Verifying SUBSTAD2 requires the following checks:
(I) deducing the loop invariant when control reaches the label AGAIN upon entering the function;
(II) deducing the output assertion for six paths, the combinations of the three paths generated by the first COND and the two paths generated when (ATOM(SETQ HH(CDR S3))) yields T;
(III) deducing the loop invariant for the three paths when the same test (ATOM ...) yields NIL.

We will describe only one check from each category.

(I)
When we sink into the label AGAIN, we have as alist:
((HH.nil) (S3.vs3) (Sl.vsl) (LAT.lat) (S3.vs3)).
The fact list is:
atom(lat) & ~atom(vs3) & ~partof(lat,vsl).
The parametrized loop assertion to be checked is:
atom(lat) & ~atom(vs3) & ~atom(vs3) & ~partof(lat,vsl) &
spine(vsl,lat,vs3,vs3,vs3).
Only the last term needs attention. Opening spine gives two subproblems:
{vs3=vs3 --> vs3=vs3} &
{~(vs3=vs3) --> ...},

which are obvious. This settles (I).

(II)

We will work our way along the path where SUBSTAD2 is recursively called on the CAR and where the atomic CDR is replaced. We start at label AGAIN with its alist:

((HH.vhh) (S3.ls3) (S1.vsl) (LAT.lat) (S3.gs3)),

and fact list:

atom(lat) & ~atom(ls3) & ~atom(vs3) & ~partof(lat,vsl) &
spine(vsl,lat,vs3,gs3,ls3).

After the first ATOM test is assumed to return with NIL, the alist has been replaced by:

((HH.vhhl) (S3.ls3) ...),

and on the fact list has been stored:

vhhl=car(ls3) & ~atom(vhhl).

The preconditions for the recursive call on HH are satisfied and thus the fact list will grow with:

replacedd(vsl,lat,vhhl,vhh2).

The alist updating scheme wakes up and generates the alist:

((HH.ivh) (S3.ils3) (S1.vsl) (LAT.lat) (S3.igs3)),

and adds to the fact list:

transf(vhhl,ivh,vhhl,vhh2) &
transf(ls3,ils3,vhhl,vhh2) &
transf(gs3,igs3,vhhl,vhh2).

We assume the second ATOM test to yield T and so the alist becomes:

((HH.vhh3) ...),

while the fact list has grown with:

vhh3=cdr(ils3) & atom(vhh3).

To complicate matters, we assume that replacement will occur, so we get on the fact list:

vhh3=lat &
eqa(ils3,jls3) & car(ils3)=car(jls3) & cdr(jls3)=vsl.

The RPLACD alist update scheme transforms the alist into:

((HH.vhh3) (S3.lls3) (S1.vsl) (LAT.lat) (S3.lgs3)),

and adds to the fact list:

eqaupto(ils3,lls3,ils3,jls3) &
eqaupto(igs3,lgs3,ils3,jls3).

From this fact list, we have to infer:
replacedd(vs1,lat,vs3,lgs3).

Expansion of replacedd gives five problems.

(1) eqa(vs3,lgs3).
Opening up the given spine formula and subsequently the resulting spinel formula allows the inference:
vs3=gs3 OR eqa(vs3,gs3).
Opening transf(gs3,igs3,vhh1,vhh2) gives:
eqa(gs3,igs3).
Opening eqaupto(igs3,lgs3,ils3,jls3) gives:
eqa(igs3,lgs3).
The transitivity of eqa settles (1).

(2) atom(car(vs3)) -->
    [(lat=car(vs3) --> car(lgs3)=vs1) &
     (~(lat=car(vs3)) --> car(lgs3)=car(vs3))].

We distinguish between vs3=ls3 and ~(vs3=ls3). The first assumption solves (2) immediately since we have vhh1=car(ls3) and ~atom(vhh1), which renders the premise of (2) false. We proceed with the assumption ~(vs3=ls3). Opening the spine formula under this assumption, spinel gives among other forms the formula:
sidetree(vs1,lat,car(vs3),car(gs3)).
Opening this formula gives:
{~atom(car(vs3)) --> ... } &
{atom(car(vs3)) -->
 [{lat=car(vs3) --> car(gs3)=vs1} &
  {~(lat=car(vs3)) --> car(gs3)=car(vs3)}]}.
Consequently it remains to show that:
car(gs3)=car(lgs3),
whether the car(vs3) was replaced or not. In both cases, we can infer from transf(gs3,igs3,vhh1,vhh2):
car(gs3)=car(igs3).
Similarly, we can infer from eqaupto(igs3,lgs3,ils3,jls3):
car(igs3)=car(lgs3).
Combining them settles (2).

(3) ~atom(car(vs3)) -->
replacedd(vsl,lat,car(vs3),car(lgs3)).

In contrast with (2), both cases ls3=vs3 and ~(ls3=vs3) should be considered. Assuming the latter, we infer as under (2), by opening spine, spinel and sidetree:
~atom(car(vs3)) --> replacedd(vsl,lat,car(vs3),car(gs3)).
As under (2), we infer car(gs3)=car(igs3)=car(lgs3), which yields (3).

Assuming the former, ls3=vs3, gives gs3=vs3. From replacedd(vsl,lat,vhhl,vhh2), we get:
replacedd(vsl,lat,car(vs3),vhh2).
From transf(gs3,igs3,vhhl,vhh2), we get:
car(igs3)=vhh2,
so we infer:
replacedd(vsl,lat,car(vs3),car(igs3)).
From eqaupto(igs3,lgs3,ils3,jls3), we obtain:
car(igs3)=car(lgs3),
and thus (3) is obtained again.

(4) atom(cdr(vs3)) -->
[(lat=cdr(vs3) --> cdr(lgs3)=vsl) &
(~(lat=cdr(vs3)) --> cdr(lgs3)=cdr(vs3))].

Again we distinguish between vs3=ls3 and ~(vs3=ls3). By induction ls3 resides on the spine sprouting at gs3. So according to the premise of (4), we can exclude ~(vs3=ls3). Therefore gs3=vs3 and so also according to transf(gs3,igs3,vhhl,vhh2), we have:
cdr(vs3)=cdr(gs3)=cdr(igs3)=cdr(ils3)=vhh3=lat.
Thus the premise of (4) and the premise of the first implication in its consequence are fulfilled. We have cdr(jls3)=vsl and with ils3=igs3 and eqaupto(igs3,lgs3,ils3,jls3) we get:
cdr(jls3)=cdr(lgs3),
which yields:
cdr(lgs3)=vsl,
resolving (4).

(5) ~atom(cdr(vs3)) -->
replacedd(vsl,lat,cdr(vs3),cdr(lgs3)).

This time the assumption vs3=ls3 flounders in atom(vhh3), since this assumption would again lead to:
cdr(vs3)=cdr(gs3)=cdr(igs3)=cdr(ils3)=vhh3.
Thus we continue with ~(vs3=ls3). Therefore by opening spine we infer:
spinel(vsl,lat,vs3,gs3,ls3,vs3,gs3).
Naming this formula Sd we have to prove: Sd --> (5).

Opening this formula Sd we infer, with the premise of (5), among other forms:
{ls3=cdr(gs3) -->
 sidefct(vs3,gs3,cdr(vs3),vs3,gs3,cdr(vs3),ls3)} and
{~(ls3=cdr(gs3)) -->
 spinel(vsl,lat,cdr(vs3),cdr(gs3),ls3,vs3,gs3)}.
Assuming the premise of the latter, we derive the formula:
spinel(vsl,lat,cdr(vs3),cdr(gs3),ls3,vs3,gs3)},
from which we still need to derive (5); cdr-induction settles this case. Thus we continue with the assumption ls3=cdr(gs3). The sidefct formula expresses the fact that if cdr(vs3) (or a substructure of it) occurred in car(vs3), it has already been investigated (so if it contained lat in leaf positions those leaf positions will have been replaced already by pointers to vsl.) Whenever the cell cdr(vs3), corresponding with ls3, has already been visited we have:
replacedd(vsl,lat,cdr(vs3),cdr(gs3)),
and also: cdr(gs3)=cdr(lgs3), since all lat leaves will have disappeared.
Otherwise we get [cadr(x) stands for car(cdr(x))]:
~atom(cadr(vs3)) & replacedd(vsl,lat,cadr(vs3),cadr(lgs3)) &
atom(cddr(vs3)) & cddr(vs3)=lat & cddr(lgs3)=vsl,
and therefore we again have:
replacedd(vsl,lat,cdr(vs3),cdr(lgs3)).

This finishes one of the paths of category (II).

(III)
We will have to show that the loop invariant holds when control reaches the (GO AGAIN) instruction. We will use the same path as followed under (II) until the ATOM test on the CDR of S3. Initially we have as alist:

((HH.vhh) (S3.ls3) (S1.vsl) (LAT.lat) (S3.gs3)),
and as fact list:
atom(lat) & ~atom(ls3) & ~atom(vs3) & ~partof(lat,vsl) &
spine(vsl,lat,vs3,gs3,ls3).
When the ATOM test on the CDR of S3 is assumed to yield NIL we have as alist:
((HH.vhh3) (S3.ils3) (S1.vsl) (LAT.lat) (S3.igs3)),
while the fact list has grown with:
vhh1=car(ls3) & ~atom(vhh1) &
replacedd(vsl,lat,vhh1,vhh2) &
transf(vhh1,ivh,vhh1,vhh2) &
transf(ls3,ils3,vhh1,vhh2) &
transf(gs3,igs3,vhh1,vhh2) &
vhh3=cdr(ils3) & ~atom(vhh3).
The next action causes the local S3 binding to be replaced and resulting in the alist:
((HH.vhh3) (S3.vhh3) (S1.vsl) (LAT.lat) (S3.igs3)).
According to this alist, we have to verify the loop invariant:
atom(lat) & ~atom(vhh3) & ~atom(vs3) & ~partof(lat,vsl) &
spine(vsl,lat,vs3,igs3,vhh3).
The first four subproblems are trivially solved because they belong to the fact list. Expanding the spine formula to be proven reduces the problem to (since we have ~(vhh3=vs3), otherwise, as a consequence of eqa(vs3,igs3) and vhh3 on the spine of igs3, we would have a cycle):
spinel(vsl,lat,vs3,igs3,vhh3,vs3,igs3),
on which cdr induction will be applied. Expanding this spinel formula gives the subproblems:

```
eqa(vs3,igs3) &
sidetree(vsl,lat,car(vs3),car(igs3)) &
{atom(cdr(vs3)) -->
 [cdr(vs3)=lat &
  cdr(igs3)=vsl &
  onspine(vsl,vhh3)]} &
{~atom(cdr(vs3)) -->
 [{vhh3=cdr(igs3) -->
   sidefct(vs3,igs3,cdr(vs3),igs3,cdr(vs3),vhh3)} &
  {~(vhh3=cdr(igs3)) -->
   spinel(vsl,lat,cdr(vs3),cdr(igs3),vhh3,vs3,igs3)}]}.
```

We will not treat these problems in as great detail as hitherto has been our practice. The general strategy should now be clear: it is a combination of subproblem recognition, case reasoning, expansion of recursive definitions and application of car/cdr induction.

(1) eqa(vs3,igs3)
This problem solves by considering vs3=ls3 as well as its negation and in each case expanding the given spine-formula and the transf formula containing igs3.

(2) sidetree(vsl,lat,car(vs3),car(igs3))
Again the two cases, vs3=ls3 and its negation, have to be considered. The first case is fairly easy and only requires expanding the sidetree formula. The other case is more laborious and requires expansion of the given spine formula and the resulting spinel formula yielding the formula:
sidetree(vsl,lat,car(vs3),car(gs3)).
As before, it subsequently can be shown that we have car(igs3)=car(gs3), which settles this subproblem.

```
(3) {atom(cdr(vs3)) -->
     [cdr(vs3)=lat &
      cdr(igs3)=vsl &
      onspine(vsl,vhh3)]}
```

The premise of this subproblem cannot be satisfied whether vs3=ls3 or not. The first case gives:
cdr(vs3)=cdr(igs3)=cdr(ils3)=vhh3,
and so the negation of the premise is reached. The second case, ~(vs3=ls3) cannot occur, because ls3 is on the spine of gs3, and cannot have passed the atomic cdr(vs3) (which is not affected by any actions on car(vs3)).

```
(4) {~atom(cdr(vs3)) -->
     [{vhh3=cdr(igs3) -->
        sidefct(vs3,igs3,cdr(vs3),igs3,cdr(vs3),vhh3)} &
      {~(vhh3=cdr(igs3)) -->
        spinel(vsl,lat,cdr(vs3),cdr(igs3),vhh3,vs3,igs3)}]}.
```

The first case vs3=ls3 satisfies the premise because

non-atomicity of cdr(vs3) is preserved with respect to actions on car(vs3) and we have ~atom(vhh3). This case will lead to satisfaction of vhh3=cdr(igs3) and by induction we can handle the sidefct formula. The second case, ~(vs3=ls3), also satisfies the premise but leads to satisfaction of ~(vhh3=cdr(igs3)). The resulting problem, the spinel formula, is solved by cdr induction.

This settles this path from category (III).

Since SUBSTAD1 has the same output assertion as SUBSTAD2, we have:

THEOREM 5. *The output of the function* SUBSTAD *when using the support function* SUBSTAD1 *is* EQUAL *to the output when the support function* SUBSTAD2 *is used instead.*

4.5 SUBSTADP

The disparity between amount of code and amount of ad hoc definitions is even greater for the pointer reversal support function SUBSTADP. The code contains three labels, so in addition to the input and output assertion we have to formulate three loop invariants. We shall limit our selves here to the first label only since that will take up enough space on its own. Moreover, as argued in the sequel, we consider the input configuration only as a special case.

As stated earlier the pointer reversal technique does not use a stack when descending down trees. Instead, a cell that has to be descended, say down the CAR-pointer, will be appropriately marked. After the CAR-pointer has been saved, it will be replaced by a pointer to the parent of its cell. Figure 3.6 depicts an example.

•

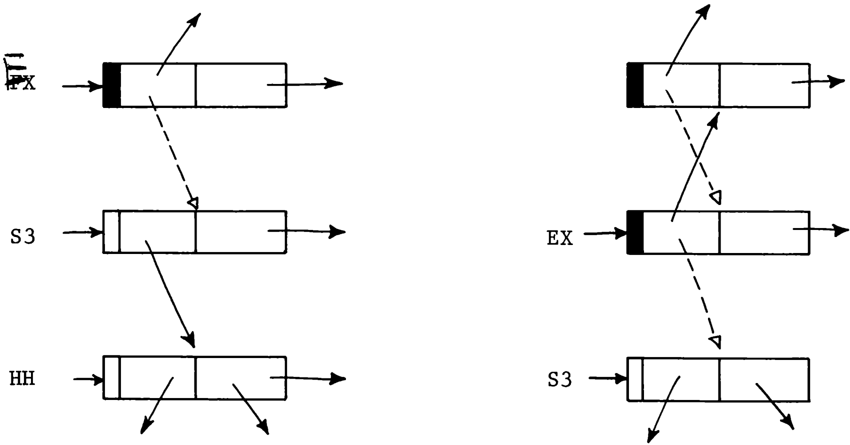


Fig. 3.6. Example of pointer reversal. The righthand side gives the situation after S3 goes down one level in the CAR-direction. Observe that the mark-bit is set.

A typical configuration at label L2 is given in figure 3.7. The left part depicts the non-atomic third argument vs3, in which atomic leaves identical with lat have to be (destructively) replaced by vs1. The right part shows the configuration midway during this process. A cell on the spine of g2, corresponding to a cell on the spine of the former vs3, has a CAR pointing to the special atom $, indicating the end of the present reversed pointer chain. A terminal of vs3 identical with lat, to the left of the spine-cell pointing to $, has been visited and has been replaced by vs1. A similar occurrence of lat at the right hand side is not yet affected.

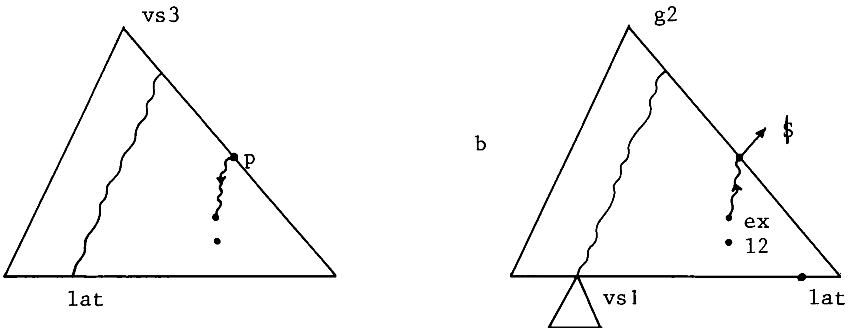


Fig. 3.7. The lefthand side depicts the original binding, the righthand side is a typical configuration at label L2.

This typical situation is by no means the general situation. In the configuration in fig. 3.7b, both the cell containing the pointer to $ as well as the cell 12 which points to that part of the tree under the reversed pointer chain that is still to be investigated, have corresponding cells in the original tree vs3, respectively the cells p and q in fig. 3.7a. Fig. 3.8 shows a configuration where there is no corresponding cell in vs3 for 12. The three occurrences of the leaf lat in vs3 are the consequence of structure sharing in vs3. Therefore visiting the leftmost occurrence of lat leads to its replacement by vs1 in all these occurences. Consequently, the reversed pointer chain will temporarily descend into vs1 when reaching the place of the former second (and third) occurrence of lat, and thus temporarily modify the S1-binding. (By the way, this will also happen when vs1 shares structure with the original vs3.)

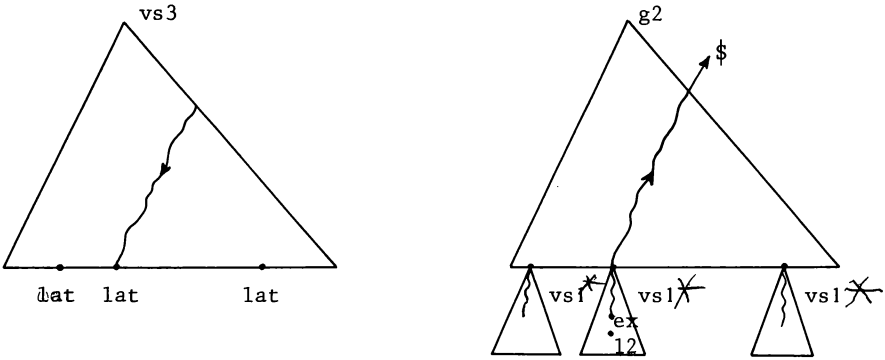


Fig. 3.8. Example of a situation in which there is no corresponding cell in vs3 for 12.

Figure 3.9 shows a configuration where the $-cell is also outside the realm of vs3. Structure sharing here has caused the replacement of the right most CDR, accidentally pointing to lat, by a pointer to vsl. Since we also assume that vsl is non-atomic the cell pointing to $ will ultimately reside on the spine of vsl.

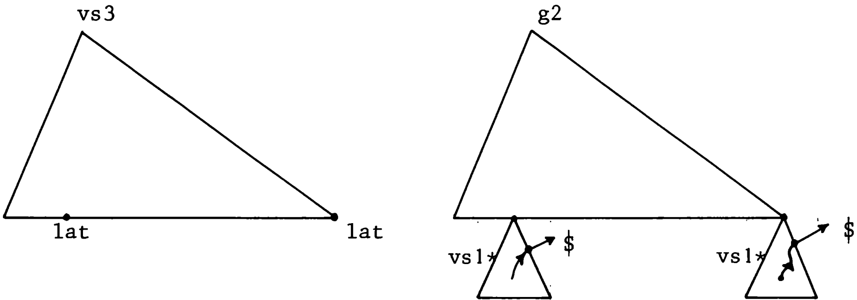
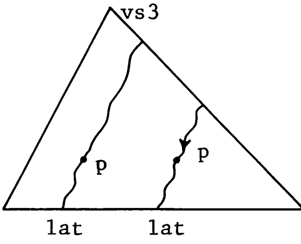
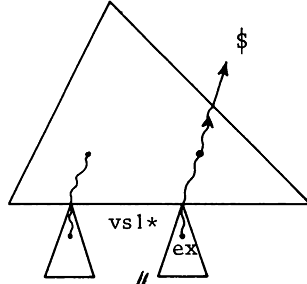


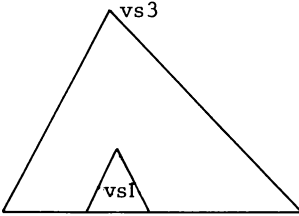
Fig. 3.9. Example of a configuration in which the $-cell is outside the realm of vs3; vsl is temporarily modified into vs1*.

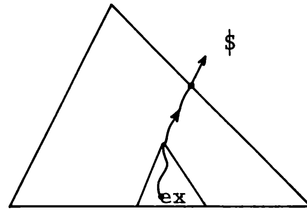
We have depicted in figure 3.10 other atypical positions of the $-cell and the ex-cell as they depend on how vsl is related to vs3.



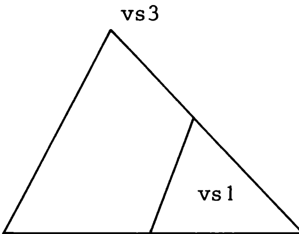


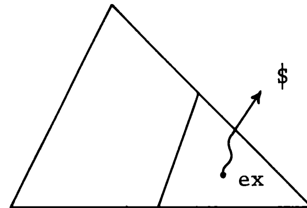
{vs1 ∩ vs3 = ∅} & {ex in vs1*} & {ex not in cell belonging to vs3}





{vs1 ∩ vs3 ≠ ∅} & {ex in vs1*} & {ex in cell belonging to vs3}





{spine of vs1 on spine of vs3} & {$-cell in vs1*} &
{ex in vs1*} & {ex in cell belonging to vs3}

Fig. 3.10. Examples of other atypical positions of the $-cell and the ex-cell.

We are concerned with only the simplest case, i.e. we will assume that vs1 is atomic. This assumption makes the typical case of fig. 3.7 also the general case. Even after this drastic simplification, we get an inflated number of definitions due to the complications arising from potential structure sharing in vs3. In order to give the definitions for the other cases we would need a staggering amount of additional definitions which only obscure what is actually the matter. For the same reason, we omit specifying the loop invariants for L4 and L5 and rather concentrate on what happens at the L2 label when a jump is made to it from the nearest GO-instruction.

We begin with some definitions of test-predicates that are repeatedly used in the simplified loop invariant.

```
(ex)(x){ onichain(ex,x) <-->
[~atom(ex) &
 {ex=x OR [markb(ex) & onichain(car(ex),x)] OR
          [~markb(ex) & onichain(cdr(ex),x)]}]}.
```

The predicate onichain formalizes the notion that the second argument resides on the inverted pointer chain sprouting at the first argument. This predicate assumes the availability of an elementary predicate markb (the counterpart of the function MARKB), which expresses whether the mark-bit that is supposed to be associated with each non-atomic cell is off or on. This mark-bit (which may be the one employed by the garbage collector) is used here to indicate whether the car- or the cdr-part of a cell contains an inverted pointer (when mark(y) holds, it is the car which contains the inverted pointer). Next we define a predicate that expresses whether a cell of a tree, that at first sight has not yet been treated, has in fact already been visited as a result of structure sharing. This visited predicate uses the additional predicates visited2 and partv.

```
(tp2)(ex)(x){ visited(tp2,ex,x) <-->
[{~(car(tp2)=$) -->
  [partv(car(tp2),ex,x) OR visited(cdr(tp2),ex,x)]} &
 {car(tp2)=$ --> visited2(ex,ex,x)}]}.

(ex)(chel)(x){ visited2(ex,chel,x) <-->
[{~markb(chel) &
  [partv(car(chel),ex,x) OR visited2(ex,cdr(chel),x)]} OR
 {markb(chel) & ~(car(chel)=$) &
  visited2(ex,car(chel),x)}]}.
```

The body of the partv definition uses existential quantifiers, as will

do other predicates in the sequel, because the 'next' cell of the inverted pointer chain is 'invisible' and we know only that there exists a next cell.

```
(trc)(ex)(x){ partv(trc,ex,x) <-->
[~atom(trc) &
 {~(trc=x) -->
  [{~onichain(ex,trc) -->
    [partv(car(trc),ex,x) OR partv(cdr(trc),ex,x)]} &
   {onichain(ex,trc) -->
    [{markb(trc) -->
      [partv(cdr(trc),ex,x) OR
       (E icel){onichain(ex,icel) & partv(icel,ex,c) &
               [markb(icel) --> car(icel)=trc] &
               [~markb(icel) --> cdr(icel)=trc]}]} &
     {~markb(trc) -->
      [partv(car(trc),ex,x) OR
       (E icel){onichain(ex,icel) & partv(icel,ex,c) &
               [markb(icel) --> car(icel)=trc] &
               [~markb(icel) --> cdr(icel)=trc]}]}]}]}]}.
```

The visited-predicate will be used to test cells which are encountered 'under' and at the 'right hand side' of the inverted pointer chain. The cell to be investigated, x, is systematically compared with cells at the left hand side of the inverted pointer chain. The comparison is made by checking whether x is a part of subtrees hanging off the spine above the $-cell (the partv predicate has to be used instead of partof since structure sharing may lead to a virtual appearance of the reversed pointer chain to the left of chain), or is a part of a subtree hanging off the left hand side of the inverted pointer chain, which is accomplished by the visited2-predicate.

These tools enable us to express the simplified and therefore incomplete loop invariant at label L2. We assume that at the entrance of the function SUBSTADP, S3 is bound to vs3 and S1 is bound to vs1 and we assume that the local alist is:
((EX.ex)(HH.hh2)(S3.13)(S1.vs1*)(LAT.lat)(S3.g3)),
and the fact list:

```
~atom(13) & atom(lat) & ~atom(g3) & ~partof(lat,vs1) &
atom($) &
{ex=$ --> [vs1*=vs1 & spine(vs1,lat,vs3,g3,13)]} &
{~(ex=$) -->
 [{atom(vs1) -->
   [vs1*=vs1 & 1b2at1(vs1,lat,g3,ex,13,vs3,g3)]} &
  {~atom(vs1) --> ...}]}.
```

So we have to specify the predicate 1b2at1 which describes how the

structures hanging at g3 and at ex relate to the original tree vs3. We will also have to specify the other support predicates lb2at2, lb2at3, lb2at4 and lb2at5.

```
(vsl)(lat)(g3)(ex)(13)(ol)(nw)
     { lb2atl(vsl,lat,g3,ex,13,ol,nw) <-->
[eqa(ol,nw) &
 {car(nw)=$ --> lb2at2(vsl,lat,g3,ex,13,ol,nw)} &
 {~(car(nw)=$) -->
  [lb2atl(vsl,lat,g3,ex,13,cdr(ol),cdr(nw)) &
   {atom(car(ol)) -->
    [{car(ol)=lat --> car(nw)=vsl} &
     {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
   {~atom(car(ol)) -->
     lb2at3(vsl,lat,ex,13,car(ol),car(nw))}]}]}.
```

The last two arguments of lb2atl lie respectively on the spine of vs3 and on the spine of its current incarnation g3. It recursively invokes itself over the cdr´s of ol and nw until the $-cell on the spine is reached. In the meantime it asserts with the predicate lb2at3 that subtrees hanging off the spine above the $-cell have been visited and that proper replacements have been made. Whenever the $-cell is reached the responsability for describing the situation is handed over to lb2at2 which assumes that its last two arguments are eqa and that the last argument lies on the inverted pointer chain.

```
(vsl)(lat)(g3)(ex)(13)(ol)(nw)
     { lb2at2(vsl,lat,g3,ex,13,ol,nw) <-->
[eqa(ol,nw) &
 {visited(g3,ex,nw) --> lb2at5(vsl,lat,ex,13,ol,nw)} &
 {~visited(g3,ex,nw) -->
  [{ex=nw -->
    [{markb(nw) -->
      [lb2at4(vsl,lat,g3,ex,13,car(ol),13) &
       {atom(cdr(ol)) --> cdr(ol)=cdr(nw)} &
       {~atom(cdr(ol)) -->
        lb2at4(vsl,lat,g3,ex,13,cdr(ol),cdr(nw))}]} &
     {~markb(nw) -->
      [lb2at4(vsl,lat,g3,ex,13,cdr(ol),13) &
       {atom(car(ol)) -->
        [{car(ol)=lat --> car(nw)=vsl} &
         {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
       {~atom(car(ol)) -->
        replacedd(vsl,lat,car(ol),car(nw))}]}]} &
   {~(ex=nw) -->
    [{markb(nw) -->
      [{atom(cdr(ol)) --> cdr(ol)=cdr(nw)} &
       {~atom(cdr(ol)) -->
        lb2at4(vsl,lat,g3,ex,13,cdr(ol),cdr(nw))} &
       (E icel){onichain(ex,icel) &
                lb2at2(vsl,lat,g3,ex,13,car(ol),icel) &
                [markb(icel) --> car(icel)=nw] &
                [~markb(icel) --> cdr(icel)=nw]}]} &
     {~markb(nw) -->
      [{atom(car(ol)) -->
        [{car(ol)=lat --> car(nw)=vsl} &
         {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
       {~atom(car(ol)) -->
        lb2a3(vsl,lat,ex,13,car(ol),car(nw))} &
       (E icel){onichain(ex,icel) &
                lb2at2(vsl,lat,g3,ex,13,cdr(ol),icel) &
                [markb(icel) --> car(icel)=nw] &
                [~markb(icel) --> cdr(icel)=nw]}]}]}]}]}.
```

The main distinction inside lb2at2 depends on whether the nw-cell - which is on the inverted pointer chain - has already been visited, i.e. whether there is an occurrence of nw to the left of the inverted pointer chain. If so the predicate lb2at5 takes over. Otherwise it checks whether the end of the inverted pointer chain has been reached (ex=nw). In either case a distinction must be made depending on whether the inverted pointer resides in the car- or cdr-part of nw, which is checked with the markb-predicate. The next predicate describes the situation that nw has been visited already.

```
(vsl)(lat)(ex)(13)(ol)(nw)
     { lb2at3(vsl,lat,ex,13,ol,nw) <-->
[eqa(ol,nw) &
 {onichain(ex,nw) --> lb2at5(vsl,lat,ex,13,ol,nw)} &
 {~onichain(ex,nw) -->
  [{atom(car(ol)) -->
    [{car(ol)=lat --> car(nw)=vsl} &
     {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
   {~atom(car(ol)) -->
     lb2at3(vsl,lat,ex,13,car(ol),car(nw))} &
   {atom(cdr(ol)) -->
    [{cdr(ol)=lat --> cdr(nw)=vsl} &
     {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
   {~atom(cdr(ol)) -->
     lb2at3(vsl,lat,ex,13,cdr(ol),cdr(nw))}]}]}.
```

The main distinction in lb2at3 depends on whether nw - which is already visited - is residing on the inverted pointer chain, as a consequence of structure sharing. If so the predicate lb2at5 will describe the situation. The next predicate pertains to those situations where the last argument nw lies ´under´ 13 or to the right of the inverted pointer chain.

```
(vsl)(lat)(g3)(ex)(13)(ol)(nw)
     { lb2at4(vsl,lat,g3,ex,13,ol,nw) <-->
[eqa(ol,nw) &
 {visited(g3,ex,nw) --> lb2at3(vsl,lat,ex,13,ol,nw)} &
 {~visited(g3,ex,nw) -->
  [{onichain(ex,nw) --> lb2at2(vsl,lat,g3,ex,13,ol,nw)} &
   {~onichain(ex,nw) -->
    [{atom(car(ol)) --> car(ol)=car(nw)} &
     {~atom(car(ol)) -->
      lb2at4(vsl,lat,g3,ex,13,car(ol),car(nw))} &
     {atom(cdr(ol)) --> cdr(ol)=cdr(nw)} &
     {~atom(cdr(ol)) -->
      lb2at4(vsl,lat,g3,ex,13,cdr(ol),cdr(nw))}]}]}]}.
```

Although the argument nw lies ´under´ 13 or to the right of the inverted pointer chain, we still have to check whether nw has already been visited as a consequence of structure sharing. If so we can back up to predicate lb2at3. Otherwise we have to deal with the question whether nw lies on the inverted pointer chain or not. Finally, the predicate lb2at5 applies to the case that nw is on the inverted chain and has also been visited.

```
(vsl)(lat)(ex)(13)(ol)(nw)
     { lb2at5(vsl,lat,ex,13,ol,nw) <-->
[eqa(ol,nw) &
 {ex=nw -->
  [{markb(nw) -->
    [replacedd(vsl,lat,car(ol),13) &
     {atom(cdr(ol)) -->
      [{cdr(ol)=lat --> cdr(nw)=vsl} &
       {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
     {~atom(cdr(ol)) -->
      replacedd(vsl,lat,cdr(ol),cdr(nw))}]} &
   {~markb(nw) -->
     {atom(car(ol)) -->
      [{car(ol)=lat --> car(nw)=vsl} &
       {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
     {~atom(car(ol)) -->
      replacedd(vsl,lat,car(ol),car(nw))} &
     replacedd(vsl,lat,cdr(ol),13)]}]} &
 {~(ex=nw) -->
  [{markb(nw) -->
    [{atom(cdr(ol)) -->
      [{cdr(ol)=lat --> cdr(nw)=vsl} &
       {~(cdr(ol)=lat) --> cdr(nw)=cdr(ol)}]} &
     {~atom(cdr(ol)) -->
      lb2at3(vsl,lat,ex,13,cdr(ol),cdr(nw))} &
     (E icel){onichain(ex,icel) &
              lb2at5(vsl,lat,ex,13,car(ol),icel) &
              [markb(icel) --> car(icel)=nw] &
              [~markb(icel) --> cdr(icel)=nw]}]} &
   {~markb(nw) -->
    [{atom(car(ol)) -->
      [{car(ol)=lat --> car(nw)=vsl} &
       {~(car(ol)=lat) --> car(nw)=car(ol)}]} &
     {~atom(car(ol)) -->
      lb2at3(vsl,lat,ex,13,car(ol),car(nw))} &
     (E icel){onichain(ex,icel) &
              lb2at5(vsl,lat,ex,13,cdr(ol),icel) &
              [markb(icel) --> car(icel)=nw] &
              [~markb(icel) --> cdr(icel)=nw]}]}]}]}.
```

Verification activities at label L2 amount to checking the instantiated loop invariant (see also the source code of SUBSTADP in section 2):

-- when control reaches L2 after entering SUBSTADP;

-- when a deeper level of the tree is explored by following a CAR-branch; and

-- when a deeper level of the tree is explored by following a CDR-branch (in fact there are again two distinct cases here: one in which the inverted pointer chain has been extended, and one in which this extention was not necessary as the consequence of being in a spine-cell).

We concentrate on the first two, because the third task would require a specification of the L4 loop invariant.

The first task is rather trivial. The binding of EX has just been set to $ and therefore the problem reduces to showing:
vsl=vsl & spine(vsl,lat,vs3,vs3,vs3).
For the second subproblem, we can refer to the foregoing section where the same problem was solved for SUBSTAD2.

The second task amounts to showing that, starting with the alist and loop invariant at L2 and updating the alist and fact list on the basis of the actions encountered on the path to the nearest (GO L2) instruction, the thusly obtained alist and fact list can be proven to comply with the loop invariant.

We start with the alist:
((EX.ex)(HH.hh2)(S3.13)(S1.vsl*)(LAT.lat)(S3.g3)).
The fact list contains the loop invariant, which we may simplify assuming that vsl is atomic, and will be:
atom(vsl) & ~atom(vs3) & atom(lat) & ~atom(g3) &
~partof(lat,vsl) & ~atom(13) & atom($) &
{ex=$ --> [vsl*=vsl & spine(vsl,lat,vs3,g3,13)]} &
{~(ex=$) --> [vsl*=vsl & lb2atl(vsl,lat,g3,ex,13,vs3,g3)]}.

The execution of (SETQ HH(CAR S3)) changes the HH-binding, to say hh3, and will add to the fact list:
hh3=car(13).

The next instruction is a COND-ition of which we will only pursue the branch where the first test is assumed to yield T. The alist remains the same while the fact list grows as a consequence of this assumption with:
~atom(hh3).

The (MARK S3 1) instruction adds to the fact list:
markb(13).

The (RPLACA S3 EX) instruction modifies the alist into:
((EX.ex2)(HH.hh3)(S3.14)(S1.vs1*)(LAT.lat)(S3.g4)),
while the fact list grows with:
eqa(13,14) & car(14)=ex & cdr(14)=cdr(13) &
eqaupto(ex,ex2,13,14) &
eqaupto(g3,g4,13,14).
The EX-binding has to be updated since we may have the special case that the CAR as well as the CDR of the original cell corresponding to ex, were both pointing to 13. The g3-binding of S3 has to be updated whenever 13 can be reached from g3, which is the case when ex is equal to $. Observe that these two updatings cannot both be effective, thus we have:
~(ex=ex2) --> g3=g4 and
~(g3=g4) --> ex=ex2=$.
The hh3-binding need not be replaced, because the cycle freeness of vs3 and therefore of 13 precludes that the cell 13 occurs again in hh3 (= car(13)).

Subsequently there are two assignment instructions:
(SETQ EX S3) and
(SETQ S3 HH),
which only affect the alist, producing:
((EX.14)(HH.hh3)(S3.hh3)(S1.vs1*)(LAT.lat)(S3.g4)).
Finally we get to the jump: (GO L2).

The jump instruction leads to the following *piece de resistance*:
atom(vs1) & ~atom(vs3) & atom(lat) & ~atom(g4) &
atom($) & ~atom(hh3) & ~partof(lat,vs1) &
{14=$ --> ...} &
{~(14=$) --> [vs1*=vs1 & lb2at1(vs1,lat,g4,14,hh3,vs3,g4)]}.

The consequence of 14=$ has not been specified since the condition cannot be fulfilled (we have atom($), ~atom(13) and eqa(13,14)).

This problem reduces to showing:
lb2atl(vsl,lat,g4,14,hh3,vs3,g4). We distinguish between two cases:

Case 1: ex=$.
Let us first assume in addition that 13=vs3. Expansion of the lb2atl-formula leads to the subproblems:

```
eqa(vs3,g4) &
{car(g4)=$ --> lb2at2(vsl,lat,g4,14,hh3,vs3,g4)} &
{~(car(g4)=$) --> ...}.
```

The spine-formula in the fact list gives:
vs3=13=g3.
One of the given eqaupto-formula allows the inference:
14=g4.
Since we also have in the fact list: eqa(13,14), we have dealt with the first subproblem.

Having available 14=g4 and having in the fact list car(14)=ex, allows us to infer with the assumption ex=$ that the premise in the second subproblem holds and thus we can dismiss immediately the third subproblem {car(g4)=$ --> ...}. Therefore the second subproblem reduces to the lb2at2-formula. Expanding this formula yields the subproblems:

```
eqa(vs3,g4) &
{visited(g4,14,g4) --> ...} &
{~visited(g4,14,g4) -->
 [{14=g4 -->
   [{markb(g4) -->
     [lb2at4(vsl,lat,g4,14,hh3,car(vs3),hh3) &
      {atom(cdr(vs3)) --> cdr(vs3)=cdr(g4)} &
      {~atom(cdr(vs3)) -->
       lb2at4(vsl,lat,g4,14,hh3,cdr(vs3),cdr(g4)}]} &
    {~markb(g4) --> ...}]} &
  {~(14=g4) --> ...}]}.
```

The first subproblem has already been solved. The visited-formula is false since g4 cannot be a part of subtrees hanging off the spine above the $-cell, since it is itself the $-cell a fact that can easily be confirmed by expansion of the visited- and visited2- predicate. Thus we are left with the third subproblem, made up out of two

alternatives which themselves contain two alternatives. Since we obviously still have l4=g4 we can dismiss the second alternative. The premise made up by the markb(g4) formula cannot be decided as yet. The RPLACX-triggered updatings should be extended such that when a non-atomic cell x is modified into y, the fact markb(x) <--> markb(y) is in addition added to the fact list (the mark-bit is not affected by updatings of the car/cdr sections of a cell). This extension to the RPLACX updating mechanism, allows the inference that markb(g4) holds because we have in the fact list markb(l3) and thus since we still have l4=g4 we can infer markb(l4). We therefore obtain the next problem simplification:

```
lb2at4(vsl,lat,g4,l4,hh3,car(vs3),hh3) &
{atom(cdr(vs3)) --> cdr(vs3)=cdr(g4)} &
{~atom(cdr(vs3)) -->
 lb2at4(vsl,lat,g4,l4,hh3,cdr(vs3),cdr(g4)}.
```

The second subproblem is easy since we have l3=vs3, l4=g4 and in the fact list the formula cdr(l4)=cdr(l3) allowing us to infer the consequence.

The third subproblem which leads to expansion of the lb2at4-formula, requires an inductive argument and is similar to the first subproblem to which we turn now.

Expansion of lb2at4(vsl,lat,g4,hh3,car(vs3),hh3) produces the reduction:

```
eqa(car(vs3),hh3) &
{visited(g4,l4,hh3) --> ...} &
{~visited(g4,l4,hh3) -->
 [{onichain(l4,hh3) --> ...} &
  {~onichain(l4,hh3) -->
   [{atom(caar(vs3)) --> caar(vs3)=car(hh3)} &
    {~atom(caar(vs3)) -->
     lb2at4(vsl,lat,g4,l4,hh3,caar(vs3),car(hh3))} &
    {atom(cdar(vs3)) --> ...} &
    {~atom(cdar(vs3)) --> ...}]}]}.
```

The eqa-formula is trivial since we have hh3=car(l3) in the fact list and we assume vs3=l3.

The premise with the visited-formula can not hold since we just entered vs3 according to our assumption; otherwise one should investigate both cases, a course which will lead to inductive arguments.

The premise with the onichain-formula cannot hold either; 14 is the beginning as well as the end of the inverted pointer chain and thus hh3 cannot be on the inverted pointer chain.

Therefore we focus our attention on the four formulas following the ~onichain(14,hh3)-premise. As a result of hh3=car(vs3) we obtain the consequences of the first and third implication. The consequences of the second and fourth formulas are handled by inductive arguments.

Without the assumption that 13=vs3 we have essentially the same line of reasoning only with more recourse to inductive reasoning to treat the more general situations.

Case 2: ~(ex=$)
As is case 1 we have to show:
lb2atl(vsl,lat,g4,14,hh3,vs3,g4).
Expansion of this formula leads to:

```
eqa(vs3,g4) &
{car(g4)=$ --> ...} &
{~(car(g4)=$) -->
 [lb2atl(vsl,lat,g4,14,hh3,cdr(vs3),cdr(g4)) &
  {atom(car(vs3)) -->
   [{car(vs3)=lat --> car(g4)=vsl} &
    {~car(vs3)=lat --> car(g4)=car(vs3)}]} &
  {~atom(car(vs3)) -->
   lb2at3(vsl,lat,g4,14,car(vs3),car(g4))}]}.
```

The first subproblem, the eqa-formula, requires an induction argument. Let us assume that we had eqa(vs3,g3). From the eqaupto-formula containing g3 and g4, we obtain eqa(g3,g4) giving us with our assumption eqa(vs3,g4). In a similar way one should be able to show the invariant:
eqa(vs3,´binding 2nd occurrence of S3 on alist´),

for all paths starting at the label L2 and ending in L2. This property certainly holds the first time L2 is entered (since the 2nd occurrence of S3 will have the binding vs3). Thus we are done with the first subproblem.

The second subproblem {car(g4)=$ --> ...} leads to the expansion of the predicate lb2at2, which we handled already for case 1, and which we here take for granted.

The third subproblem requires, assuming ~(car(g4)=$), us to show:

```
lb2atl(vsl,lat,g4,l4,hh3,cdr(vs3),cdr(g4)) &
{atom(car(vs3)) -->
 [{car(vs3)=lat --> car(g4)=vsl} &
  {~car(vs3)=lat --> car(g4)=car(vs3)}]} &
{~atom(car(vs3)) -->
 lb2at3(vsl,lat,g4,l4,car(vs3),car(g4))}.
```

The first subproblem can be dismissed by our reliance on cdr induction.

Let us assume the premiss of the second subproblem: atom(car(vs3)). Expansion of the lb2atl-formula in the fact list yields:

```
atom(car(vs3)) -->
[{car(vs3)=lat --> car(g4)=vsl} &
 {~car(vs3)=lat --> car(g4)=car(vs3)}].
```

Expansion of the eqaupto-formula containing g3 and g4 gives car(g3)=car(g4) and completes the second subproblem.

Assuming the premise of the third subproblem instead, ~atom(car(vs3)), gives the same argument.

In a similar way, the proofs can be given:
-- when the loop invariants have been specified for the other labels,
-- when the path is followed from label L4 to the RETURN-statement,
-- when the restriction atom(vsl) is dropped, and finally

-- when the restriction ~partof(vs1,vs3) is dropped (see again fig. 3.10).

After all these laborious proofs - admittedly infeasible without machine support - we can state:

<u>THEOREM 6</u>. *The output of the function* SUBSTAD *when using the support function* SUBSTADP *is* EQUAL *to the output when the support function* SUBSTAD1 *(or* SUBSTAD2*) is used instead.*

Consequently, no matter which support function is used, the outputs of SUBST and SUBSTAD are EQUAL provided the second argument is a literal atom and does not occur in the first argument.

5. Implementation Results

We give here time and space consumption comparisons of SUBST, SUBSTAD with the pointer reversal support function SUBSTADP and SUBSTAD with the half recursive half iterative support function SUBSTAD2. Subsequently, unification algorithms with different substitution functions will be compared.

To account for the COPYing property of SUBST we measured not only the time and space needed by SUBSTAD but also a COPY operation followed by a SUBSTAD operation. Two different objects were used:
-- a list of the form (A B C D E A B C D E A ...) of length 2560, and
-- a balanced tree of depth 10 with at its leaves (A B C D E).
In case of the balanced tree it mattered for the space consumption whether A or E was replaced by a SUBST operation as can be seen from table 2.

Operation	time	space
SUBST	77	2565
COPY+SUBSTADP	38	2570
COPY+SUBSTAD2	38	2570
SUBSTADP	10	9
SUBSTAD2	9	9

Table 1. Time and space measurements for the different substitution functions and in combination with COPY on a list of length 2560.

Operation	time	space
SUBST on A	164	2056
SUBST on E	185	6152
COPY+SUBSTADP	104	6153
COPY+SUBSTAD2	91	6153
SUBSTADP	34	9
SUBSTAD2	19	9

Table 2. Time and space measurements for the different substitution functions and in combination with COPY on a balanced tree of depth 10 with at its leaves (A B C D E).
Two values are given for the space consumption of SUBST; the ´on A´ value pertains to the replacement of A, the ´on E´ value to the replacement of E.

We can conclude from these measurements:

-- The half recursive half iterative SUBSTAD2 is to be preferred to the pointer reversal version SUBSTADP since it is faster while having equal space consumption.
-- When one need not worry about preservation of the original S-expression, the SUBSTAD versions are to be preferred to SUBST, and the savings are ´gigantic´.
-- When the original S-expression must be maintained and a COPY needs to be performed, it is still better to use the SUBSTAD functions over SUBST with respect to speed, but space consumption may deteriorate.

To investigate how the last mentioned trade-off behaves in practice, we outfitted a unification algorithm with SUBST as well as with SUBSTAD and exercised them on 13 unifiable strings. After compilation of these unification algorithms we ´hand-optimized´ the version using SUBSTAD2. Table 3 contains the measurements for the three different unification algorithms.

Provided that the used strings are representative for unifiable strings we may conclude that the unification SUBSTAD2 algorithms out-perform the unification SUBST algorithm. Of course it is also possible to hand-optimize the unification version with SUBST. It should be noted however, that on non-unifiable strings the optimized-SUBSTAD2 version will have a zero space consumption (since COPY-ied structure can be recovered) while COPY-ied structure by SUBST cannot be recognized.

object	unification with					
	SUBST		SUBSTAD2		optimized SUBSTAD2	
	time	space	time	space	time	space
01	31	465	18	465	15	315
02	10	315	8	315	9	165
03	18	345	11	405	11	315
04	14	345	12	345	8	255
05	15	405	12	345	11	255
06	12	345	13	390	11	300
07	18	495	13	390	9	300
08	12	375	13	405	8	195
09	15	375	14	405	8	195
010	22	450	15	465	11	225
011	26	450	14	465	14	225
012	25	675	15	405	16	165
013	23	540	16	405	14	165
total	241	5580	174	5205	145	3075

Table 3. Time and space consumption of a unification algorithm implemented with SUBST, SUBSTAD2 and a hand optimized unification algorithm with SUBSTAD2. To improve the time measurements, all unifications were repeated 15 times (as can be seen from the space usages which all have 15 as divisor). Due to the loop overhead we may expect that the relative improvement of the SUBSTAD algorithms with respect to the SUBST algorithm is even better than shown.

Finally, we report the measurements of unifying (xl ... xn) and (a ... a) for n=100, 200 up to n=800 with the unification algorithms employing SUBST and SUBSTAD2 and the hand-optimized unification algorithm with SUBSTAD2. Table 4 contains the results. As to be expected, the unification algorithms with SUBSTAD2 have a better performance than those with SUBST.

object length	unification with					
	SUBST		SUBSTAD2		optimized SUBSTAD2	
	time	space	time	space	time	space
100	.5	413	.1	413	.1	213
200	2.1	813	.4	813	.4	413
300	4.7	1213	.8	1213	.8	613
400	8.3	1613	1.5	1613	1.5	813
500	13.1	2013	2.3	2013	2.3	1013
600	18.9	2413	3.2	2413	3.2	1213
700	25.7	2813	4.4	2813	4.4	1413
800	33.5	3213	5.8	3213	5.7	1613

Table 4. Time and space consumption for the same unification algorithms as in table 3. In contrast with table 1, 2 and 3, the time units in this table are seconds instead of milliseconds. Apparently the time complexity of all versions is quadratic.

6. Conclusions

It turns out that the substitution function SUBSTAD is a worthwhile addition to the LISP repertoire. A simple unification algorithm could be modified such that it takes advantage of SUBSTAD and has a better performance than the version that uses the function SUBST. Whether a similar improvement can be obtained for a linear unification algorithm [65] is an interesting issue to be investigated.

The attempt to give correctness proofs for several versions of SUBSTAD revealed that the method of symbolic execution - although theoretically adequate - flounders in some cases in a practical problem: the formal description of input/output statements as well as loop invariants lead to a proliferation of ad hoc definitions to unmanageable amounts. We suspect that this disadvantage holds for all currently available verification techniques. If so, verification specialists may be well-advised to give more attention to the practical implications of their theories, rather than devote all their energy to esoteric refinements, or the design of logics that become an end in themselves.

We feel that the bottle-neck lies in the necessity to specify in state-description terms what a function is supposed to do. Whether a function is recursive or not is not even explicitly expressible in its specification. Somehow people feel closer to a definition of a function in procedural terms, such as "the terminals equal to lat will be *replaced* by vsl" and "the tree will be *visited* from left to right". Proving correctness of a function ´reduces´ then to showing that the function *behaves* according to these expectations rather than that input/output description pairs conform to a certain relation.

The technique we have developed for describing evolving states using an alist, a fact list and predictates like eqaupto and transf that capture specific side effects, may be of interest to other areas of A.I. The alist can be considered a collection of individual concepts, where the bindings are the actual extensions. A new situation differs primarily in that some concepts have different extensions which is reflected in fresh facts. Outdated facts do not

have to be deleted but merely become invisible since they contain arguments that no longer reside on the alist.

The old frame problem [58] linked with the usage of the predicate calculus for state descriptions has evaporated. There is no need for unwieldy axioms to express that, when P(x,...,z,s1) holds in situation s1 and additional conditions are fulfilled, the fact P(x,...,z,s2) can be inferred in s2. Instead we have a different frame problem. A fact may seem obsolete (since an argument has been removed from the alist) while an analogous fact can be inferred for a newly introduced extension. We have encountered this in lemma 1-4 where particular circumstances allow us to equate old and new binding. Since updatings and the recognition of identities are object centered, and may affect many facts simultaneously, this frame problem appears to be less obstructive than the original one; but more thinking and/or experimenting is needed to validate this ~~suggestion.~~ conjecture

To end this section on the positive side: although program verification cannot as yet be promoted as a tool for wide distribution, it pays off to have a second look at one's program from a verification perspective. After all, writing this chapter forced us to rethink the conditions in which the function SUBSTAD is applicable. The specification we published five years ago, turned out to be too liberal!

APPENDIX

The following are generalizations of formerly given recursive definitions to arbitrary data objects, thus possibly containing cycles. It should be borne in mind that all arguments are made up of only a finite number of cells.

```
(d)(e){ partof(d,e) <--> partof1(d,e,0) }

(d)(e)(V){ partof1(d,e,V) <-->
[ partofcar(d,e,V) OR partofcdr(d,e,V) ] }

(d)(e)(V){ partofcar(d,e,V) <-->
[ ~atom(e) & ~(e in V) &
  { d=car(e) OR partof1(d,car(e),{e} U V) } ] }

(d)(e)(V){ partofcdr(d,e,V) <-->
[ ~atom(e) & ~(e in V) &
  { d=cdr(e) OR partof1(d,cdr(e),{e} U V) } ] }

(d)(e){ compatible(d,e) <--> compatible1(d,e,0,0) }

(d)(e)(V)(W){ compatible1(d,e,V,W) <-->
[ atom(d) OR atom(e) OR (d in V) OR (e in W) OR
  { eqa(d,e) & d=e } OR
  { ~eqa(d,e) &
    compatible1(d,car(e),V,{e} U W) &
    compatible1(d,cdr(e),V,{e} U W) &
    compatible1(car(d),e,{d} U V,W) &
    compatible1(cdr(d),e,{d} U V,W) } ] }

(y1)(y2)(x1)(x2){ eqaupto(y1,y2,x1,x2) <-->
                  eqaupto1(y1,y2,x1,x2,0) }

(y1)(y2)(x1)(x2)(V){ eqaupto1(y1,y2,x1,x2,V) <-->
[ eqa(y1,y2) &
  { (y1 in V) --> y1=y2 } &
  { ~(y1 in V) -->
    [ { y1=x1 --> y2=x2 } &
      { [ ~(y1=x1) & ~atom(y1) ] -->
        [ eqaupto1(car(y1),car(y2),x1,x2,{y1} U V) &
          eqaupto1(cdr(y1),cdr(y2),x1,x2,{y1} U V) ]}]}]}

(y1)(y2)(x1)(x2){ transf(y1,y2,x1,x2) <-->
                  transf1(y1,y2,x1,x2,0) }
```

```
(y1)(y2)(x1)(x2)(V){ transf1(y1,y2,x1,x2,V) <-->
[ { (y1 in V) --> y1=y2 } &
  { ~(y1 in V) -->
    [ eqa(y1,y2) &
      { x1=y1 --> y2=x2 } &
      { [ ~atom(y1) & ~(x1=y1) & tr1(y1,x1,x2,0) ] -->
        tr2(y1,y2,x1,x2,0) } &
      { [ ~atom(y1) & ~(x1=y1) & ~tr1(y1,x1,x2,0) ] -->
        [ transf1(car(y1),car(y2),x1,x2,{y1} U V) &
          transf1(cdr(y1),cdr(y2),x1,x2,{y1} U V) ]}]}]}

(y1)(x1)(x2)(V){ tr1(y1,x1,x2,V) <-->
[ ~(x1 in V) & ~atom(x1) & eqa(x1,x2) &
  { y1=x1 OR
    tr1(y1,car(x1),car(x2),{x1} U V) OR
    tr1(y1,cdr(x1),cdr(x2),{x1} U V) }]}

(y1)(y2)(x1)(x2)(V){ tr2(y1,y2,x1,x2,V) <-->
[ ~(x1 in V) &
  { y1=x1 --> y2=x2 } &
  { ~(y1=x1) -->
    [ { tr1(y1,car(x1),car(x2),{x1} U V) -->
        tr2(y1,y2,car(x1),car(x2),{x1} U V) } &
      { tr1(y1,cdr(x1),cdr(x2),{x1} U V) -->
        tr2(y1,y2,cdr(x1),cdr(x2),{x1} U V) }]}]}
```

The generalization of equal to arbitrary data objects deviates from the pattern given above. We use <x.y> to indicate the ordered pair formed from x and y; P1 and P2 stand respectively for selectors on ordered pairs.

```
(e1)(e2){ equal(e1,e2) <--> equal1(e1,e2,0,0) }

(e1)(e2)(V)(W){ equal1(e1,e2,V,W) <-->
[ e1=e2 OR
  { [ eqt(e1,V) --> eqck(e1,e2,V,W) ] &
    [ ~eqt(e1,V) -->
      { ~eqt(e2,W) & ~atom(e1) & ~atom(e2) &
        equal1(car(e1),car(e2),<e1.V>,<e2.W>) &
        equal1(cdr(e1),cdr(e2),<e1.V>,<e2.W>) }]}]}

(e1)(V){ eqt(e1,V) <-->
[ ~(V=0) &
  { e1=P1(V) OR eqt(e1,P2(V))}]}

(e1)(e2)(V)(W){ eqck(e1,e2,V,W) <-->
[ ~(V=0) &
 { e1=P1(V) --> e2=P1(W) } &
 { ~(e1=P1(V)) --> eqck(e1,e2,P2(V),P2(W))}]}.
```

TWO THEOREM PROVER PREPROCESSORS

1. Motivation

Several schools can be distinguished within the Automatic Theorem Proving community. The method employed divides them. The two leaders are "resolution" and "natural deduction". Resolution has been more thoroughly investigated and is more attractive from a theoretical point of view since completeness can be easily verified. Natural deduction, however, seems closer to the method employed by human beings, and what goes on in a natural deduction proof can be readily interpreted. Moreover, recent results of natural deduction are more impressive than those obtained by resolution.

For many years yet another school, the proceduralists, have been beating the drum pervasively. In order to achieve real deductive power, they combine deductive rules with advice as to how and when they should be used. This technique depends upon an explicit reference to the characteristics of a specific domain. Consequently, a theorem prover equipped with these peppers is no longer general while its specificness does not reside in a replaceable component. They even developed a special language for this approach: QA4 [72] (QA3 [37], its predecessor, is a resolution type theorem prover). In fact, it was immediately obsolete and was absorbed into QLISP/INTERLISP. Until now, theorem provers have not been implemented in these languages, as far as we know. Moreover, these languages are geared rather toward plan generation and automatic programming than theorem proving. They will have to find a cure against the allurement to write adhoc deductive procedures to prevent that this school may disappear from the deductive scene.

Contrary to what one may expect as a consequence of its deep roots into the past, the predicate calculus (PC) has no standard setting lending it to deductive tasks. While the syntax for PC-formulas is certainly standard, axioms and derivation rules range from several axiom schemata, standing for an infinite number of axioms, and only modus ponens, to no axioms and many derivation rules. Dependent on the aims of the investigator/user, an arbitrary selection

in this range may be made.

Resolution, which uses a heavily restricted subset of the PC, has a minimal amount of connectives, and no quantifiers while only one derivation rule is employed (considering factoring a part of resolution). Consequently, implementing resolution-based theorem provers required - in the beginning - minimal programs. Furthermore a translator is required to transform PC-formulas into conjunctive-normal-form (CNF), the current champion of unnatural knowledge representation. Unnaturalness is in itself no disadvantage since one can get accustomed to it; what counts is the near impossibility to set up a cooperation between a resolution theorem prover and non-syntactic heuristics, knowledge sources and models.

Another extreme is natural deduction without axioms, employing many derivation rules and working with the full PC. An immediate advantage of this technique is that it does not require the translation to CNF. Thereby making the object under consideration more familiar, permitting construction of humanlike deductive operations. A disadvantage is that such a theorem prover becomes opaque. With many derivation rules, the issue of control becomes paramount. One assignment of priorities to the derivation rules may be very effective for solving some problems while it is singularly unsuccessful for solving others.

If the proceduralists are confronted with serious troubles, the resolution and natural deduction schools also have inherent problems. Both have in common the view that finding a proof is regarded as a search problem, where objects in the search space are respectively clauses and predicate calculus formulas. They differ only in the kind of operators used. Resolution people have been busy ever since 1965 refining the resolution rule in order to limit the generation of redundant clauses. A recent addition to the field is the connection graph [46], a data representation for clauses where the label on an edge represents the substitution of unifiable literals. This representation is attractive because it requires no search for unifiable literals. The question as to which pair of unifiable literals to resolve upon remains wide open. Resolution as well as

natural deduction have a certain "flatness" in common which is the consequence of their uniform data representation.

In contrast, we advance the thesis that deduction consists of distinct operations, each requiring another optimal representation for the objects on which they work. (This observation is after all not far-fetched; one only has to look at the different subdisciplines of mathematics to stumble on a wealth of formalisms.) From this perspective, one can even justify resolution with its CNF. Whenever one runs out of deductive ´high order´ operations on a problem one may, as a last resort effort, submit it to the search mechanism of a - refined - resolution prover.

Resolution as well as natural deduction can be critized for not being sensitive to ´obvious´ peculiarities of problems to be solved. They do not know the difference between an axiom, a theorem, a definition, or a recursive definition. They do not know when to ignore the fact that a formula is an equation or make explicit use of such a fact. They are not flexible with respect to the decision to continue proving something instead of trying to find a counterexample. They cannot juggle with several interpretations of a set of formulas to guide decisions. They cannot recognize that a proof in fact allows the assertion of a stronger conclusion than the conjecture started off with. (When an assumption is introduced and it can be recognized that the subsequent proof does not depend on it one can prevent repeating the proof under the negated assumption). They are ignorant of other theories and thus cannot attempt adapting proofs by analogical reasoning (with the notable exception of a first attempt in [45]).

Summarized: the deduction rules tool box is supposed to do too much work and its tools are clumsy, while useful data - models, other theories, distinctions between formulas or even the proof sequence of already proven theorems - are inaccessible.

Consequently, we envision a different ´architecture´ for more powerful theorem provers. They should be arranged as cooperating deductive ´specialists´, each one embodying sound deductive power and when productive - in the sense of a production system - able to ensure

a positive contribution to a solution. No single one needs to be complete. The issue of completeness, constantly popping up in the automatic deduction literature and drawing an undue amount of attention, also becomes unimportant. When for example, a simple resolution specialist belongs to the community all discussions about completeness become unnecessary.

Of course another problem - well known in production systems circles - comes up when there are many deductive specialists around: how to decide cheaply which specialist is applicable, and when more than one is applicable, which one to give control to? When we are getting used to supplying more information than just axioms and a conjucture to a community of deductive specialists, as argued above, this problem might be alleviated in a non adhoc way.

We will describe in this chapter two deductive specialists:
-- INSTANCE, which is able to decide whether a conjecture is a special case, an alphabetic variant and/or an and/or-connective permutation variant of an already accepted axiom, theorem, lemma, intermediate result, etc.
-- INSURER, which is able to recognize when a conjecture can be rewritten into independent, easier to handle subproblems.
These specialists - which can be considered as preprocessors for conventional theorem provers - were implemented - together with a simple applier of definitions - and integrated with a connection graph, resolution-based theorem prover. The augmented power of this deduction complex with respect to the sole theorem prover will be shown by examples.

Some of this work has been reported in [15]. In the mean time, it has turned out that the cooperation between the two preprocessors could be increased, making one of them still more effective, and maintaining their algorithmic, always halting nature. The cooperation has become so tight that we have the following apparent paradox: although the output format of INSURER is the input format of INSTANCE, INSTANCE has processing responsibilities deep down, inside INSURER.

The next section is devoted to the definition of INSTANCE and a description of some of its properties. In section 3, INSURER is defined and its properties explored. Section 4 describes the structure of the supervisor of the theorem prover COGITO, containing INSTANCE, INSURER, a definition opener and a connection graph resolution component, with which the examples to be discussed in section 5 have been handled.

2. Compressed Mini-Scope and INSTANCE

As discussed in the former section, distinct deductive specialists may require distinct data representations for the objects on which they operate. Here we want to describe another data representation, Compressed Mini-Scope (CMS), which allows the support of operations like "for symmetry reasons it is sufficient to consider only ..." and "since A is a special case/alphabetic variant of B we conclude that ...".

While our intention is to treat the full PC, we will first illustrate what is at stake with the propositional calculus.

Let us start with a propositional calculus formula P_0, which stands for a conjecture. We can transform P_0 with a well known recipe into an equivalent formula P_1 such that P_1 is in CNF, say:
$P_1 = \&R_i$, with $R_i = OR(R_{ij})$.
(To simplify the notation we use the ´Einstein´ convention, thus for instance $\&R_i$ abbreviates $\&(R_1,\ldots,R_n)$; & stands for *and*.)

P_1 may be simplified with the following rules:

(1) if $R_{ix}=R_{iy}$ (with x unequal y) then drop R_{iy} from R_i (whenever only one element of the disjunction remains then drop the or-connective);
(2) if $R_{ix}={\sim}R_{ij}$ then drop R_i from P_1;
(3) if each element R_{ix} in R_i has a corresponding element R_{jy} in R_j with $R_{ix}=R_{jy}$ then drop R_j from P_1;
(4) if ${\sim}R_i=R_j$ then P_2:=FALSE;
(5) if no R_i remains then P_2:=TRUE.

Whenever the resulting formula P_2 is a conjunction while P_0 is not then we have decomposed the problem P_0 into subproblems which are in general easier to solve than P_0. Rule 1 simplifies subproblems. Rule 2 removes tautologies. Rule 3 takes care for removing identical subproblems. Its generalization to the PC removes subproblems that disappear on symmetry type of arguments. Rule 4 is a watch dog against non-sensical problems. Rule 5 makes these rules a special case theorem prover.

The generalization of the translator and the rules 1-5 to PC input is the topic of the next section. (The translator INSURER applies the rules while transforming to CMS - the analogue for CNF - instead of applying those rules afterwards.) The rules 1-4 presuppose a simple test to check whether propositional constants are related in such a way that a rule may ´fire´. For example rule 1 requires only a test for the identity of R_{ix} and R_{iy}. The generalization of these tests to the PC is the topic of this section and yields the definition of the INSTANCE algorithm.

Roughly speaking, a CMS-formula is a closed mini-scope PC-formula (see [85]), with the additional properties that
(a) no two arguments of an *and* or *or* subformula are in an INSTANCE relationship,
(b) no two arguments of such a subformula are in a half-negated INSTANCE relationship (two formula´s F and G are in a half-negated INSTANCE relation when the negation of F is in an INSTANCE relationship with G), and
(c) each quantifier has a unique variable.

Apart from the necessity to be more precise about mini-scope, we have to face the complication that the INSTANCE relationship is mentioned in the definition of CMS while the INSTANCE algorithm definition presupposes that its two arguments will come from the CMS domain. Recursion will be the way out of this paradox.

Let us first define *mini-scope*. It does not contain the connectives --> and <-->, and is generated by the following pseudo-BNF rules {comment is added inside curly brackets}:

<miniscope> :=: &<topAform>* | <topAform> | TRUE | FALSE.

{The * indicates a repetition of at least two elements, so a mini- scope formula is a conjunction with <topAform> arguments or a <topAform> formula.}

<topAform> :=: OR<topOform>* | <topOform>.

<topOform> :=: <form with var *nil*>.

{*nil* occupies here an argument position of <form with var ...> and shows that a mini-scope formula does not have free variables.}

<form with var X> :=: <literalform X> | (Y)<uqbody X, Y> | (E Y)<eqbody X, Y>.

{X, Y, Z stand for sets of free variables, e.g. if X= (x1,x2) then <literalform X> can be P(a,x1,x2). <uqbody ...> (respectively <eqbody>) stands for the body of a universal (existential) quantified formula.}

<literalform X> :=: <atomicform X> | ~<atomicform X>.

{Since we assume that the reader knows already about atomic forms, we will not expand them. Observe that the negation sign can occur only in front of atomic forms.}

<uqbody X Y> :=: <literalform X U Y> | (E Z)<eqbody X U Y,Z>
| OR<form with var Y U Z_i>, with U Z_i = X.

<eqbody X Y> :=: <literalform X U Y> | (Z)<uqbody X U Y,Z>
| &<form with var Y U Z_i>, with U Z_i = X.

Comment: The double parameters of <uqbody ...> (and <eqbody ...>) are introduced to ensure that all arguments of a disjunctive (conjunctive) body of <uq ...> (<eq ...>) will contain the variables in X, preventing the push of a quantifier to the right by factoring an argument out of an and/or-connective.

The definition above is more rigorous than [68] and [85] and seems to deviate on some points. Instead of meticulously describing the differences and their inaccuracies, we believe that the authors had the above definition in mind.

We still have to refine this mini-scope subset to *CMS*. Let the AND/OR-level of a formula be the maximum number of AND/OR-connectives one may encounter by going from the toplevel to a literal terminal. Suppose CMS has been defined up to level n-1 and INSTANCE has already been defined on this CMS subset, then an n-AND/OR-level mini-scope formula is in CMS iff no arguments of its nth-level AND/OR-connective is in the INSTANCE- or half negated INSTANCE- relationship (which is defined since those are at most of level n-1).

Before continuing with the definition of INSTANCE, we will present examples to clarify the direction in which we are going.
(1) (x){A(a) & A(x)} is not mini-scope.
(2) A(a) & (x)A(x) is not CMS because A(a) is in the INSTANCE relationship with (x)A(x).
(3) ~A(a) & (x)A(x) is not CMS because ~A(a) is in the half-negated INSTANCE relationship with (x)A(x).
(4) (x)A(x) is in CMS.

Remark: example (2) may be rewritten to (x)A(x) and example (3) rewrites to FALSE.

And now INSTANCE. Its input consists of:
(1) a CMS formula T to be investigated for being a special case and/or alphabetic variant of:
(2) another CMS formula K.
The output of INSTANCE is Y or nil whether or not T is an ´instance´ of K respectively. Being an ´instance´ is in fact defined in the sequel in a procedural way by an algorithm with as main characteristic that it is stronger than implication. After its specification we will show that INSTANCE(T,K) implies: |- K --> T.

The first action of INSTANCE is to Skolemize K and to anti-Skolemize T, thus existential quantifiers in K and universal quantifiers in T are removed by replacing the associated variables by fresh functions, while universal quantifiers in K and existential quantifiers in T remain. Those functions have variables dependent on the preceding universal and existential quantifiers respectively.

Example: Consider the formula Z: (x)(E y)P(x,y). In case Z is the second argument K of INSTANCE, the Skolemization of Z yields (x)P(x,F(x)), with F a fresh unary Skolem function. In case Z is the first argument T of INSTANCE then anti-Skolemization of Z yields (E y)P(f,y) where f is a Skolem constant since there are no preceding existential quantifiers to "(x)".

<u>LEMMA 1</u>. *If* S1(K) *is the result of the Skolemization of* K *and* S2(T) *the result of the anti-Skolemization of* T *then*:
|- S1(K) --> S2(T) *iff* |- K --> T.

<u>PROOF</u>. Obviously we have:
(1) |- K --> T iff |- ~K OR T.
According to lemma 42A, page 275 of [28], we have:
(2) |- ~K iff |- ~S1(K).
In analogy with the construction of the Skolem-normal-form SN1(F) for any first order PC-formula F, one may construct the anti-Skolem-normal form SN2(F), which consists of a sequence of universal quantifiers (possibly empty) over individual variables and/or function variables, followed by a sequence of existential quantifiers (possibly empty) over inidividual variables, followed by a quantifier free matrix and for which we have:
(3) |- F <--> SN2(F).
This construction is obvious when F is quantifier free. We induct on the number of quantifiers in F. By using standard transformations, we can transform F in an equivalent formula G, not containing the equivalence- and implication- connective any more. By working ´not´ inwards we get an equivalent formula H. When H has a leading connective ´and´ or ´or´ we treat each argument separately (possibly after renaming variables, such that no variable is bound more than once). Thus we obtain say &(SN2(H1),...,SN2(Hk)). By pushing the quantifiers to the left and reordering quantifiers, which is allowed since SN2(Hi) and SN2(Hj) still do not have variables in common, we obtain a formula of the required format.

Suppose H is of the form (i)I(i). Let c be a fresh constant and J(c):= SN2(I(c)). Then we have by induction:

|- J(c) <--> I(c),

and by generalization:

|- (z){ J(z) <--> I(z) }.

Let K:= SN2(H) and suppose that not |- K <--> H. So there is an interpretation with:

(I) VAL(K) = true and VAL(H) = false or

(II) VAL(K) = false and VAL(K) = true.

Assume case I.

K= (fl)...(fk)(Ei)...M(i,fl(i,...),...,fk(i,...),...).

In this interpretation thus:

VAL(M(i,fl(i,...),...)) = true,

while VAL(I(i)) = false.

For i we have |- J(i) <--> I(i), and

J(i)= (gl)...(gk)...M(i,gl(...),...).

We are free to construct for each gj lambda expressions gj such that in this interpretation:

M(i,gl(...),...)=M(i,fl(i,...),...).

Consequently, VAL(I(i)) = true. Contradiction.

Case II is analogous to case I.

Since (3) has been dealt with, we can derive from it:

(4) |- F iff |- S2(F).

From left to right, this is obvious since |- SN2(F) --> S2(F) and together with (3), we obtain |- S2(F).

From right to left, we apply generalization on |- S2(F) to obtain |- SN2(F), and again with (3), we obtain |- F.

Combining (4) with (2) yields:

(5) |- ~K OR T iff |- ~S1(K) OR S2(T).

Combining (1) with (5) produces:

|- S1(K) --> S2(T) iff |- K --> T. <<

The next action of INSTANCE, after anti-Skolemizing and Skolemizing its two arguments, is to call the recursive support function INS2 with:
INS2(S2(T), S1(K), nil, nil),
where the 3rd and 4th argument stand for respectively the set of free variables in S2(T) and S1(K), and since we start off with closed formulas, they will be empty at the top level of INS2.

The output of INS2 is:
-- NO, signifying that S2(T) is not an instance with respect to the INS2 procedure of S1(K) and will cause INSTANCE to return with nil, or
-- a non-empty list of substitutions, where each substitution ss allows to infer:
|- S1(K) --> S2(T),
and will cause INSTANCE to return with Y.

At the top level call of INS2, it is sufficient that INS2 returns with only one substitution in order to allow INSTANCE to report success. The reason for letting INS2 produce possibly more than one substitution is that generating more than one substitution on a lower level may may prevent INS2 returning with NO on a higher level. An example where more than one substitution will be returned by INS2 is:
S2(T) = (E x)A(x,x),
S1(K) = (y)(A(p,y) & A(q,y)) with the two substitutions:
(x <-- p, y <-- p) and (x <-- q, y <-- q).

We now define the support function INS2. To simplify the notation we represent S2(T) and S1(K) respectively by ST and SK. VARST and VARSK are respectively the set of variables in ST and SK. We also need the following notations:
X.tt stands for performing the substitution tt on X. {A substitution is of the form ((x_1 <-- s_1) ... (x_n <-- s_n)) with x_i not in s_j and all x_i different; nil is the empty substitution}.
tt+ss stands for concatenation of the substitutions tt and ss.tt (a variable may not occur in ss as well as in tt at the left hand side, and no left hand side variable occurs in any right hand

side).

P*tt is the formula obtained by:

- -- removing each quantifier in P for which there exists a replacement prescription in tt for its accompanying variable; and
- -- subsequently performing P1.tt on the formula P1 obtained thus.

A unifier has to be understood as a most general unifier in the sense of [71]. We have generalized the unification algorithm slightly to allow also matching of formulas. For example &(A(x),B(b,x)) and &(A(a),B(y,z)) will have the unifier:
(x <-- a, y <-- b, z <-- a).

```
INS2(ST, SK, VARST, VARSK) :=
if ST  and  SK are unifiable, with respect to the free variables of ST
   and SK as given by VARST and VARSK, with unifier ss
   then {ss}
   else
if SK = (x)Form(x)
   then INS2(ST, Form(x), VARST, VARSK U {x})
   else
if ST = (E x)Form(x)
   then INS2(Form(x), SK, VARST U {x}, VARSK)
   else
if SK = OR(K1, ..., Kn)
   then [ UU:= INSORK(nil, ST, (K1, ..., Kn));
          if UU = nil then NO else UU]
{Where INSORK is a recursive function defined as:
INSORK(ss, STST, (KKj, ..., KKn)) :=
    if j = n+1, thus all Ki have been treated already
    then {ss}
    else [ tt := INS2(STST, KKj, VARST, VARSK);
           if tt = NO then return nil
           else return(U Ut) ]
           with Ut=INSORK(t+ss,STST*t,(KKj+1.t,...,KKn.t))
           for t in tt}
   else
if ST = &(T1, ..., Tn)
   then [ UU := nil;
          INSANDT(nil, (T1, ..., Tn), SK)
          {which works like INSORK in the former case};
          if UU = nil then NO else UU]
   else
if SK = &(K1, ..., Kn)
   then [ UU := nil;
```

```
          for each K_i do
             {U_2 := INS2(ST, K_i, VARST, VARSK);
             if U_2 unequal NO then UU := UU U U_2};
          if UU = nil then NO else UU]
   else
if ST = OR(T_1, ..., T_n)
   then [ UU := nil;
          for each T_i do
             {U_2 := INS2(T_i, SK, VARST, VARSK);
             if U_2 unequal NO then UU := UU U U_2};
          if UU = nil then NO else UU]
   else NO.
```

To prove the soundness of INSTANCE, we first have to prove an already announced property of INS2.

LEMMA 2. *If* ss *is a substitution in* INS2(ST, SK, VARST, VARSK) *then*

|- (Vk)SK --> (E Vt)ST,

where Vt *from* VARST *and* Vk *from* VARSK *are respectively the free variables of* ST *and* SK.

PROOF. By case reasoning and induction on the length of the formulas.

case 1: If SK and ST are unifiable with substitution ss then SK*ss = ST*ss and thus obviously

(1) |- SK*ss --> ST*ss.

As a consequence of:

(2) |- (Vk)SK --> SK*ss,

(3) |- ST*ss --> (E Vt)ST and

(4) |- { ((Vk)SK --> SK*ss) -->
        [(ST*ss --> (E Vt)ST) -->
          {(SK*ss --> ST*ss) --> ((Vk)SK --> (E Vt)ST)}]},

we conclude by deleting left hand sides of implications in (4), using m.p. with (1), (2) and (3) as premisses:

(Vk)SK --> (E Vt)ST.

case 2: Let ss in INS2(ST, Form(x), VARST, VARSK U {x}). Thus we have:

|- (Vk)(x)Form(x) --> (E Vt)ST,

and thus:

|- (Vk)SK --> (E Vt)ST.

<u>case 3:</u> This case with ST = (E x)Form(x) runs parallel to the former case.

<u>case 4:</u> Assume ss in INSORK(nil,ST,($K_1,\ldots,K_n$)). We have to show:
|- (Vk)OR($K_1,\ldots,K_n$) --> (E Vt)ST.
We proceed by induction on the INSORK calls. Thus assume that K_1, ..., K_{j-1} have been dealt with already and that ss has been obtained thus far with: STST = ST*ss, KK_i = K_i*ss and
(1) |- (Vk)OR($K_1,\ldots,K_{j-1}$) --> (E Vt)ST.
Let tt in INS2(STST, KK_j, VARST, VARSK), thus we have:
(2) |- (Vk)KK_j --> (E Vt)STST.
Since we have:
(3) |- (Vk)K_j --> (Vk)KK_j,
(4) |- (E Vt)STST --> (E Vt)ST and
(5) |- {((Vk)K_j --> (Vk)KK_j) -->
[((E Vt)STST --> (E Vt)ST) -->
{((Vk)KK_j --> (E Vt)STST) -->
((Vk)K_j --> (E Vt)ST)}]},
we get from (2), (3), (4) and (5):
(6) (Vk)K_j --> (E Vt)ST.
Combining (1) and (6) yields:
(Vk)OR($K_1,\ldots,K_j$),
and thus we are done with the induction step. The base case is obvious since we have:
|- FALSE --> ST.

<u>case 5:</u> This case with ST = &($T_1,\ldots,T_n$) runs parallel again to the former case.

<u>case 6:</u> Let ss in INS2(ST, &($K_1,\ldots,K_n$), VARST, VARSK), thus there is a K_i with ss in INS2(ST, K_i, VARST, VARSK). So we have:
|- (Vk)K_i --> ST.

Since obviously:
|- (Vk)&(K_1,...,K_n) --> (Vk)Ki,
we are done.

<u>case 7:</u> This case with ST = OR(T_1,...,T_n) runs parallel again to case 6.

Now we have dealt with all cases of the INSTANCE algorithm thus confirming the induction step. The base of the induction was dealt with in case 1. <<

Combining lemma 1 and lemma 2, we get:

<u>THEOREM</u>. *If* T *and* K *are closed, compressed mini-scope, predicate calculus formulas while* INSTANCE(T, K) *holds then* |- K --> T, *thus* INSTANCE *is sound.*

<u>PROOF</u>. According to lemma 1, we have:
|- K --> T iff |- S1(K) --> S2(T).
Since (Vk)S1(K) = S1(K) and (E Vt)S2(T) = S2(T) because S1(K) as well as S2(T) are closed formulas, application of lemma 2 gives the required result. <<

As a consequence of the transitivity of implication:
{(P --> Q) & (Q --> R)} |- (P --> R)
one may wonder whether INSTANCE(K,L) and INSTANCE(L,M) implies INSTANCE(K,M) for arbitrary K, L and M. As yet we have not been able to prove it or find a counter-example. A preliminary investigation suggests that substitutions ss and tt obtained by:
ss in INS2(S2(K), S1(L), nil, nil) and
tt in INS2(S2(L), S1(M), nil, nil)
might be used to construct a substitution rr in INS2(S2(K), S1(M), nil, nil), leading to the transitivity of INSTANCE.

Remark: The definition of CMS given in the beginning of this section mentions INSTANCE. As long as a different INSTANCE procedure satisfies our theorem we can accept another delineation of a subset of the PC claiming the name CMS. A non-interesting example would be

INSTANCE(T,K) iff T=K. Since all results of the next section refer only to the theorem in this section they immediately generalize to other - stronger - versions of INSTANCE.

3. INSURER

INSURER is the theorem prover preprocessor which expects for its input a problem, specified as a closed predicate calculus formula, and tries to rewrite it in an equivalent formula with the leading connective *and* [while refraining itself of being funny and e.g. rewriting P into P & P]. If the output does have such a leading connective then its arguments can be seen as subproblems, which can be attacked one at the time. When all of them can be solved the original problem has been dealt with. Our concern here focuses on the question of whether INSURER gives a maximal decomposition into independent subproblems.

As we have already stated in [15], the set of rewrite rules that make up INSURER is - coincidentally - a subset of the rewrite rules that make up a PC-CNF translator. Since we will eventually be arguing that this PC-CNF translator remedies deficiencies of the one described in [19], [54] and [53], we will first describe the PC-CNF translator and discuss its properties.

The translator consists of the following steps:

1- Eliminate ´if ... then´ and ´if and only´.
   Replace A --> B by OR(~A, B) and
           A <--> B by &(OR(~A, B), OR(A, ~B)).

2- Move ´not´ inwards.
   Replace ~(x)A by (E x)~A,
           ~(E x)A by (x)~A,
           ~(OR(A_1,...,A_n)) by &(~A_1,...,~A_n),
           ~&(A_1,...,A_n) by OR(~A_1,...,~A_n) and
           ~~A by A.

3- Push quantifiers to the right.
   Let (Qx) be (x) or (E x), and let XX be & or OR.
   3.1 Excise parts without free variables.
       Replace (Qx)XX(A_1,...,A_i,...,A_n) by
               XX{A_i,(Qx)(A_1,...,A_{i-1},A_{i+1},...,A_n)}
       when x not free in A_i.
   3.2 Straighten out ´and´s and ´or´s.
       Replace XX(A_1,...,A_i,XX(B_1,...,B_k),A_{i+1},...,A_n) by

$XX(A_1,\ldots,A_i,B_1,\ldots,B_k,A_{i+1},\ldots,A_n)$.

3.3 Distribute ´and´ over ´or´ in the context of (x) or vice versa in the context of (E x).
Replace
$(x)OR(A_1,\ldots,A_i,\&(B_1,\ldots,B_k),A_{i+1},\ldots,A_n)$ by
$(x)\&(OR(B_1,A_1,\ldots,A_n),\ldots,OR(B_k,A_1,\ldots,A_n))$ and
$(E\ x)\&(A_1,\ldots,A_i,OR(B_1,\ldots,B_k),A_{i+1},\ldots,A_n)$ by
$(E\ x)OR(\&(B_1,A_1,\ldots,A_n),\ldots,\&(B_k,A_1,\ldots,A_n))$.

3.4 Eliminate redundant forms.
Replace $\&(A_1,\ldots,A_i,\ldots,A_j,\ldots,A_n)$ by
$\&(A_1,\ldots,A_i,\ldots,A_{j-1},A_{j+1},\ldots,A_n)$ when
$INSTANCE(A_j,A_i)$ holds.
Replace $OR(A_1,\ldots,A_i,\ldots,A_j,\ldots,A_n)$ by
$OR(A_1,\ldots,A_i,\ldots,A_{j-1},A_{j+1},\ldots,A_n)$ when
$INSTANCE(A_i,A_j)$ holds.

3.5 Try to collapse forms.
Replace $\&(A_1,\ldots,A_i,\ldots,A_j,\ldots,A_n)$ by FALSE when
INSTANCE(movenotinwards($\tilde{}A_i$),A_j) holds.
{movenotinwards performs step 2}
Replace $OR(A_1,\ldots,A_i,\ldots,A_j,\ldots,A_n)$ by TRUE when
INSTANCE(movenotinwards($\tilde{}A_i$),A_j) holds.

3.6 Try further collapsing.
Replace $\&(A_1,\ldots,A_i,TRUE,A_{i+1},\ldots,A_n)$ by
$\&(A_1,\ldots,A_i,A_{i+1},\ldots,A_n)$.
Replace $\&(A_1,\ldots,A_i,FALSE,A_{i+1},\ldots,A_n)$ by FALSE.
Replace $OR(A_1,\ldots,A_i,TRUE,A_{i+1},\ldots,A_n)$ by TRUE.
Replace $OR(A_1,\ldots,A_i,FALSE,A_{i+1},\ldots,A_n)$ by
$OR(A_1,\ldots,A_i,A_{i+1},\ldots,A_n)$.

3.7 Distribute quantifiers over connectives.
Replace $(x)\&(A_1,\ldots,A_n)$ by $\&((x)A_1,\ldots,(x)A_n)$ and
Replace $(E\ x)OR(A_1,\ldots,A_n)$ by $OR((E\ x)A_1,\ldots,(E\ x)A_n)$.

4- Eliminate existential quantifiers by Skolem functions.
Pick out the leftmost well-formed part of the form (E y)B(y) and replace it by $B(f(x_1,\ldots,x_n))$ where:

a) $x_1,\ldots,x_n$ are the free variables of (E y)B(y) which are universally quantified to the left of (E y)B(y); and
b) f is a ´fresh´-n-ary function constant.

5- Eliminate universal quantifiers.

6- Distribute ´and´ over ´or´.

6.1 Attempt the rewrite rules 3.2, 3.4, 3.5 and 3.6.
6.2 Replace $OR(A_1,\ldots,A_i,\&(B_1,\ldots,B_k),A_{i+1},\ldots,A_n)$ by $\&(OR(B_1,A_1,\ldots,A_n),\ldots,OR(B_k,A_1,\ldots,A_n))$.

7- Do nothing.

The following rules should also be observed:

-- If step(i) can be reapplied it has precedence over step(i+1) (, which motivates the void step 7).

-- A sequence of universal (or existential) quantifiers should get special attention in step 3.1. Since quantifiers may be permuted, using the rule (x)(y)A <--> (y)(x)A, and similarly for existential quantifiers, rule 3.1 is allowed to look beyond the right-most quantifier.

-- While rule 2 and 4 must be applied top-down, rule 3 and 6 must be applied bottom up.

-- When distributing in step 3.3 or step 6.2, each bound variable inside formulas getting a multiple occurrence should be renamed to preserve the requirement that each quantifier has a unique variable.

-- When a conjunction or disjunction as a consequence of application of rule 3.4 or 3.6 winds up with one argument, the connective will be deleted.

The main difference between this PC-CNF translator and the ones described in [19], [53] and [54] is the incorporation of INSTANCE. The translator in [19] is based on first producing prenex normal form, which is done with rule 1 and 2, followed by pushing quantifiers to the *left*. Consequently, Skolem functions may be introduced with an unnecessary number of arguments, e.g. (x){A(x)OR(E y)B(y)} will be transformed into (x)(E y){A(x) OR B(y)} leading to the CNF A(x) OR B(f(x)). Instead, it can be transformed into (x)A(x)OR(E y)B(y) leading to the simpler form: A(x) OR B(g). The translator in [53] lacks rule 3.3 and its preparatory step 3.2, while the translator in [54] lacks 3.2, 3.3 and 3.7. Consequently, these translators may be forced to generate Skolem functions with too many arguments.

Remark: Since we have not been able to show the transitivity of INSTANCE, the generation of canonical output may be endangered by a rash application of rule 3.4. If ever a triple (K,L,M) surfaced with INSTANCE(K,L), INSTANCE(L,M) but not INSTANCE(K,M), application of rule 3.4 on &(K,L,M) may end up with &(K,M) while in fact it can be replaced by M, by the application of 3.4 such that the intermediate result &(L,M) is obtained.

In [53], theorem 1.5.1, is shown that the translator preserves unsatisfiability. Combining this result with application of the soundness theorem in the former section on the rules containing INSTANCE-related rewriting, leads also to preservation of unsatisfiability by the translator detailed here.

Now we define INSURER simply as a subset of the PC-CNF translator rules by omitting rule 4 and 5 concerning the elimination of existential and universal quantifiers. INSURER produces closed PC formulas and since all transformations concern equivalences we have:

LEMMA 1. *If* Q := INSURER(P) *then* |- P <--> Q.

While the input format of INSURER is the unrestricted PC the next observation concerns its output format.

LEMMA 2. INSURER *maps* PC *into compressed mini-scope.*

PROOF. To facilitate the proof, we introduce a stepping stone; we define a subset of the PC which we will show to encompass the format produced by rule 1-3:
<out3> :=: &<top2Aform>* | <top2Aform> | TRUE | FALSE;
<top2Aform> :=: OR <top2Oform>* | <top0form>;
<top2Oform> :=: &<top2Aform>* | <top0form>.
An immediate feature of the subset <out3> is that the connectives --> and <--> may not occur. This is being taken care of by rule 1. The next feature concerns the occurrence of ~; the negation sign may occur only in front of literal formulas. This is handled by rule 2 and the subsequent rules do not affect this property. Another characteristic of <out3> is that conjunctions (disjunctions) do not contain terms

which are themselves conjunctions (disjunctions). This is prevented by rule 3.2. Minimal scope of quantifiers is checked by rule 3.1, 3.3 and 3.7. These features taken together make up <out3> and therefore the output of rule 3 conforms to the format prescribed by <out3>.

CMS is partially defined in an operational way by reference to INSTANCE (which cuts away redundancies, as recognized by INSTANCE, in conjunctions and disjunctions). Since INSTANCE is incorporated at the proper places in rule 3 we are assured that, in addition to the requirements of <out3>, conjunctions and disjunctions in <top0form> components of rule 3's output are not INSTANCE-redundant.

Finally the 'and' over 'or' distribution in rule 6 takes care that the <miniscope> format is reached as well as that the additional INSTANCE-testing is performed. <<

INSURER turns out to be a special case theorem prover. It can at least recognize ground tautologies.

LEMMA 3. *If* Q *is a valid* PC *formula without quantifiers then* INSURER(Q) = TRUE.

PROOF. Since the output of INSURER is in mini-scope INSURER(Q) is TRUE, FALSE or of the form <top0form>, OR<top0form>* or &<topAform>*. The case FALSE is prohibited by lemma 1. The case <top0form> contradicts the validity assumption because the value false may be assigned to <top0form>.
Suppose the output is of the form OR<top0form>*, thus like OR($O_1,\dots,O_k$). If all O_i are positive (or all are preceded by a negation sign) then we have a contradiction since we can assign all O_i the value false (true). Thus we can rewrite the disjunction as follows:
{OR($P_1,\dots,P_1$)} OR {OR($\neg N_1,\dots,\neg N_m$)}. No P_i can be equal to a N_j since that would have been recognized in rule 6.1 of INSURER. Again we get a contradiction with validity by assigning all P_i the value false and all N_j the value true. Consequently OR<top0form>* is ruled out.
Applying the former cases on every argument of &<topAform>* eliminates

this case as well.
Thus INSURER(Q) can only be TRUE. <<

A generalization to monadic predicate calculus turns out not to hold. A counter-example is:
(x)Q(x) <--> [(y){Q(y) OR P(y)}&(z){Q(z) OR ~P(z)}],
which is valid and will be translated into:
(x)Q(x) OR (y){~Q(y) OR P(y)} OR (z){~Q(z) OR ~P(z)},
instead of TRUE.

The role on INSURER may be clarified by observing that a PC-CNF translator mostly works in a resolution environment where its input is among other formulas a *negated* conjecture. In contrast, INSURER's input will be mostly a *non-negated* conjecture.

INSURER is an independent subproblem recognizer since INSURER is "strongly motivated" to rewrite its input into an equivalent (lemma 1) formula which is a conjunction. In case the output is a conjunction, each of the two arguments are independent to each other with respect to the implicative testing on INSTANCE. We wrote "strongly motivated" since it is not possible to prove that INSURER gives a maximal conjunctive decomposition. Even an atomic formula P can be equivalently rewritten into the conjunction (P OR Q)&(P OR ~Q) for an arbitrary Q. This conjunction, however, is an example of case reasoning because it can be rewritten as:
(Q --> P)&(~Q --> P), embodying: in order to prove P it is sufficient that Q as well as its negation implies P.

We are going to show that when a non-conjunctive output (or output component) Q of INSURER can be equivalently rewritten into a conjunction $\&R_i$, then $\&R_i$ embodies case reasoning on Q. Since one cannot expect that INSURER takes the initiative to case reasoning we will conclude that INSURER gives maximal decompositions. First we deal with supporting lemmas.

<u>LEMMA 4</u>. *Let* Q *and* R *be quantifier free and in* CMS, *while* Q *in* <topAform>, *thus not a conjunction, and* R *a conjunction, thus* R = $\&R_i$,

and suppose |- Q <--> &(Q OR R_i),

assume Q = OR(Q_1,...,Q_q) *(possibly with* q=1*), and*

R_i = OR(R_{i1},...,R_{ir}) (possibly r=1),

then for each R_i *there is a* R_i^* *defined as:*

R_{ix}^* := *if there is a* Q_y *with* Q_y = R_{ix} *then* FALSE *else* R_{ix},

R_i^* := OR(R_{ix}^*), *while deleting those* R_{iy}^* *that are equal to* FALSE *such that* |- Q <--> &(Q OR R_i^*) *and* |- ~&R_i^*.

PROOF. Applying the rule A OR B --> A OR A OR B gives:
Q OR R_i^* --> Q OR R_i.
Applying the rule A OR A OR B --> A OR B gives:
Q OR R_i --> Q OR R_i^*.
Consequently: |- Q <--> &(Q OR R_i^*).
Suppose |- ~&R_i^* does not hold. Consequently there is an assignment to all R_i^* (possibly when R_i^* is a disjunction by giving assignments to its constituents) such that VAL(~&R_i^*) = false, thus VAL(&R_i^*) = true. So for all R_i^*, we get VAL(R_i^*) = true. As a consequence of the definition of the R_i^*'s we are free to give an arbitrary assignment to Q. Letting VAL(Q) = false gives a contradiction with:
|- Q <--> &(Q OR R_i^*). <<

LEMMA 5.
If 1) Q *is in* CMS *and in* <topAform>, *thus not a conjunction,*
2) R *is in* CMS *and a conjunction, thus* R = &R_i *with* R_i *in* <topAform> *and*
3) |- Q <--> &R_i,

then &R_i *is directly or indirectly an example of case reasoning on* Q.

PROOF. From $|- Q <--> \&R_i$ it is easy to see that:

(1) $|- Q <--> \&(Q \text{ OR } R_i)$.

Assume that $Q = Q_1$ OR ... OR Q_q (possibly q=1) and

$R_i = R_{i1}$ OR ... OR R_{ir} (possibly r=1).

Define R_{ix0} := if there is a Q_y with INSTANCE(Q_y,R_{ix}) and INSTANCE(R_{ix},Q_y) then FALSE else R_{ix}.

R_{i0} := OR(R_{ix}) while deleting those R_{ix} equal FALSE.

We still have:

(2) $|- Q <--> \&(Q \text{ OR } R_{i0})$.

Now we have two possibilities:

(I) $|- \sim\&R_{i0}$, thus we are dealing with an immediate example of case reasoning, or

(II) not $|- \sim\&R_{i0}$. From (2) and by defining Q_0 := Q we get:

(3) $|- Q_0 <--> \&(Q_0 \text{ OR } R_{i0})$,

and from (3):

(4) $|- \&R_{i0} --> Q_0$.

By lemma 4, we can conclude that (3) and also (4) should contain at least one quantifier.

We will construct a terminating sequence of $\{Q_n\}$ and $\{R_{in}\}$ fulfilling (1), ultimately leading to case (I), by stripping away quantifiers.

Assume that $\{Q_n\}$ and $\{R_{in}\}$ have already been constructed.

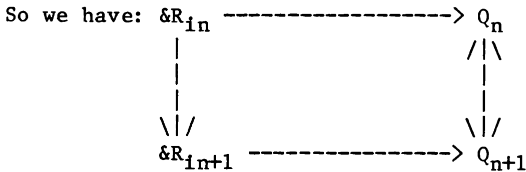
Construction rule 1. If there is a R_{jn} such that:

R_{jn} = (x)R_{j1n} OR R_{j2n} OR ... OR R_{jrn}, and we have a derivation of Q_n in which the full power of the universal quantifier is not used (i.e. the derivation tree for Q_n can be modified such that all occurrences of R_{jn} on leaf positions can be replaced by instantiations for x of R_{jn}), we define:

$Q_{n+1} := Q_n$,

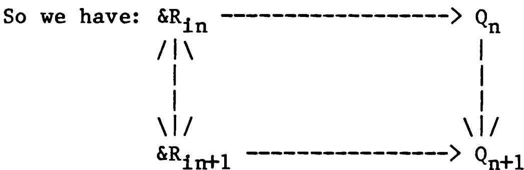
R_{in+1} := R_{in} for i not equal j,

R_{jn+1} := &{R_{j1n}(uk) OR R_{j2n} OR ... OR R_{jrn}}, where {uk} is the finite set of required instances for (x)R_{j1n}(x) to derive Q_n.



Example of applicability of construction rule 1:
Q(a,b) <--> [{Q(a,b) OR Q(a,a)} &
{Q(a,b) OR (x)(~Q(x,x) OR Q(x,b))}].
R_{10} = Q(a,a); R_{20} = (x)(~Q(x,x) OR Q(x,b)); the finite set of necessary instances is here {a}.

Construction rule 2. If Q_n is of the form:
(x)Q_{1n} OR Q_{2n} OR ... OR Q_{qn} then weaken Q_n by instantiating x with a new constant c and thus define:
Q_{n+1} := Q_{1n}(c) OR Q_{2n} OR ... OR Q_{qn}, and
R_{in+1} := R_{in}.



Example of applicability of construction rule 2:
(x)Q(x) <--> [(y){Q(y) OR P(y)}&(z){Q(z) OR ~P(z)}]
replacing (x)Q(x) by Q(c).

Construction rule 3. If Q_n is of the form:
(E x)Q_{1n}(x) OR Q_{2n} OR ... OR Q_{qn} and we have a derivation of (4) which allows us to strengthen Q_n by instantiating x by a constant c then

define:

R_{in+1} := R_{in} and

Q_{n+1} := Q_{1n}(c) OR Q_{2n} OR ... OR Q_{qn}.

Q_{n+1} may not be in <topAform> anymore, since Q_{1n}(c) can be a conjunction. If so, we obtain as many examples of (1) as there are terms in the conjunction by distributing ´and´ over ´or´.

```
So we have: &R_in   ------------------> Q_n
            /|\                        /|\
             |                          |
             |                          |
            \|/                         |
            &R_in+1 ----------------> Q_n+1
```

Example of applicability of construction rule 3: Begin with:

(E x)Q(a,x) <--> [{(E x)Q(a,x) OR (y)(P(y) OR Q(a,y))} &
{(E x)Q(a,x) OR (E z)~P(z)}].

One obtains:

Q_0 = (E x)Q(a,x),

R_{10} = (y){P(y) OR Q(a,y)} and

R_{20} = (E z)~P(z).

By application of construction rule 4 (see next rule), one obtains for instance:

Q_1 = Q_0, R_{11} = R_{10} and R_{21} = ~P(c). Since &(R_{11},R_{21}) not only allows us to derive Q_1 but also Q(a,c), application of construction rule 3 leads to Q_2 = Q(a,c).

Construction rule 4. If there is a R_{jn} such that:

R_{jn} = (E x)R_{j1n}(x) OR R_{j2n} OR ... OR R_{jrn} then strengthen R_{jn} by instantiating x with a new constant c and thus define:

Q_{n+1} := Q_n,

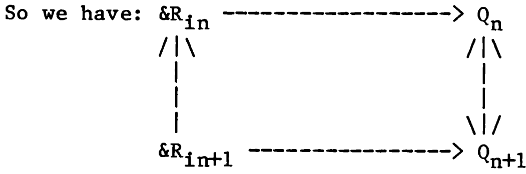
R_{in+1} := R_{in} for i not equal j, and

R_{jn+1} := R_{j1n}(c) OR R_{j2n} OR ... OR R_{jrn}.

As with construction rule 3, R_{jn+1} may not be in <topAform> when

$R_{j1n}(c)$ is a conjunction. By distributing 'and' over 'or' we simply get more $\{R_{in+1}\}$'s than $\{R_{in}\}$'s.

So we have:



For an example of applicability of construction rule 4, see the example at the former rule above.

When a construction rule was applicable, we go back to check whether the condition of case (I) holds, leading to an example of case reasoning.

It remains to show that at least one construction rule applies. We immediately can rule out the cases that Q_n contains a universal quantifier or $\&R_{in}$ an existential quantifier, since applicability of construction rule 2 respectively rule 4 depends only on these syntactic features. So we are left with Q_n containing an existential quantifier and/or $\&R_{in}$ containing a universal quantifier. Assume Q_n contains an existential quantifier. Thus we have:
(5) $|- \&R_{in} \dashrightarrow \{(E\ x)Q_{1n}(x)$ OR Q_{2n} OR ... OR $Q_{qn}\}$.
A derivation tree for (5) can be modified into a derivation tree for:
$\&R_{in} \dashrightarrow \{Q_{1n}(c)$ OR Q_{2n} OR ... OR $Q_{qn}\}$,
where c is fresh constant, by propagating modifications from the root to the leaves, unless a leaf of the tree is of the form $(E\ y)R_{jn}(j)$. This, however, we have already ruled out since it would lead to applicability of rule 4.

The case that $\&R_{in}$ contains a universal quantifier leads to a similar contradiction.

This process halts because we start with a finite number of quantifiers and we strip off a quantifier each time a construction rule applies. <<

THEOREM. *Let* X *be the output of* INSURER, *while it is not a conjunction or else an arbitrary member of the conjunction.* INSURER *produces a maximal conjunctive decomposition in the sense that there is no equivalent conjunctive decomposition of* X *when we disregard case reasoning on* X.

PROOF. Apply lemma 5. <<

Remark: It still may occur that when the output of INSURER is say $Q_1 \& Q_2$, Q_2 can be decomposed while using Q_1.

Example: Q_1 = (x)(x=1 OR x=2 OR ... OR x=N),

Q_2 = (x)P(x).

$Q_1 \& Q_2$ = INSURER($Q_1 \& Q_2$), however we have also:

|- $Q_1 \& Q_2$ <--> Q_1&P(1)&...&P(N).

4. Interplay between INSURER and INSTANCE

INSURER, INSTANCE, a connection graph resolution-based contradiction recogniser, a PC-CNF-translator and a simple definition opener were imbedded in a 'fixed' regime. Input for the prover consists of axioms, supporting theorems (proof sequence is not taken into account), definitions (again without sequence), and the conjecture. For the next description, we should remember that activation of the connection graph component should be postponed at all costs.

Roughly, a supervisor triggers the following activities:

step 1: If the conjecture is an INSTANCE of an axiom, a theorem or an already proven theorem (see step 2) then return with success.

step 2: If the conjecture, using INSURER, decomposes into the sub-problems C_1, ..., C_n

then for each C_i go (recursively) to step 1

if the value returned for treating C_i is succesful

then add C_i to the collection of already proven theorems

else quit with failure;

return with success.

step 3: If the conjecture contains a predicate defined in one of the definitions (non-recursive) then substitute for each occurrence in the conjecture the instantiated body of the definition and go to step 1.

step 4: Translate the axioms, supporting theorems and the negation of the conjecture into conjunctive normal form, call the contradiction resolution type recognizer and return the value which rsesults from this call (a resource parameter ensures termination).

This is a simple-minded supervisor and made only to demonstrate the effectiveness of INSTANCE and INSURER. The deduction complex used for program verification, chapter 3, has a somewhat more sophisticated controller. Much remains to be desired. An attractive alternative would be to implement the supervisor as a multi-process scheduler. The

overall structure of the cooperating specialists would be more transparant, facilitating the addition of a new specialist, and opening up the way to parallel processing which, but for the lack of available languages like QLISP, INTERLISP and MAGMA-LISP, would have been possible.

5. Implementation results

Our first example looks terribly simple but a straight forward treatment, after direct translation into CNF by the resolution component, had not yet found a contradiction after generating 35 clauses. It consists only of the following:

definition:

```
    (s)(t){SETEQ(s,t) <-->
            (x)(ESTI(x,s) <--> ESTI(x,t))},
```

and conjecture:

```
    (u)(v){SETEQ(u,v) <--> SETEQ(v,u)}.
```

INSURER immediately recognizes that the conjecture reduces to two subproblems:

```
    (u1)(v1){~SETEQ(u1,v1) OR SETEQ(v1,u1)} &
    (u2)(v2){~SETEQ(v2,u2) OR SETEQ(u2,v2)}
```

INSTANCE will recognize that the second subproblem is an alphabetic variant of the first subproblem so we have only to bother about the first one. The definition opener will recognize its applicability and will rewrite the first subproblem into:

```
    (u1)(v1){~(x1)(ESTI(x1,u1) <--> ESTI(x1,v1)) OR
              (x2)(ESTI(x2,v1) <--> ESTI(x2,u1))}.
```

Again INSURER will be invoked for this formula. To appreciate its result, we will zoom in on its actions. First <--> is removed and ~ is moved inwards resulting in:

```
    (u1)(v1)[OR(E x3){~ESTI(x3,u1) & ESTI(x3,v1)}
               (E x4){ ESTI(x4,u1) & ~ESTI(x4,v1)}
               (&(y1){~ESTI(y1,v1) OR ESTI(y1,u1)}
                 (y2){ ESTI(y2,v1) OR ~ESTI(y2,u1)})].
```

The structure of this formula is:

```
    (u1)(v1)[OR O1
                O2
                (& A1 A2)]
```

Since the body is a disjunction, the universal quantifiers cannot be distributed. The third argument of the disjunction is a conjunction however, allowing the production of a conjunction as body of the quantifiers. So we obtain the following:

```
    (u1)(v1)[&{OR A1 O1 O2}
              {OR A2 O1 O2}]
```

Before pushing the quantifiers to the right, the disjunctions are scrutinized since simplifications may be possible after the distribution. The first disjunction is in fact:

```
[OR(y1){~ESTI(y1,v1) OR ESTI(y1,u1)}
   (E x3){~ESTI(x3,u1) & ~ESTI(x3,v1)}]
   (E x4){ ESTI(x4,u1) & ~ESTI(x4,v1)}].
```

INSTANCE will find that the negation of the first argument is an alphabetic variant of the second argument, so this disjunction collapses to TRUE. The second disjunction collapses in a similar way. Thus the whole formula collapses to TRUE and we are finished. Notice that the resolution component remained sound asleep.

The next example comes from group theory, see box 1. The axioms (1-5) do not constitute a minimal characterization of a group. A subset of a group is represented by a predicate-variable. SUBGR, which expresses the property of a subgroup, is therefore a 2nd order predicate. Equality of subsets is expressed by SETEQ in (7). The notion of a right-coset is defined by (8); COSET(g,xx,HH) should be read as "xx is the right-coset with respect to the subgroup HH and the group element g". SETEQ and COSET are like SUBGR 2nd order predicates. Theorem (9) says that the element g belongs to the subgroup HH iff HH is equal to the g-HH-coset.

```
+----------------------------------------------------------+
| (1)  (x)(y)(z) x(yz)=(xy)z                               |
| (2)  (x) xe=x                                            |
| (3)  (x) ex=x                                            |
| (4)  (x) xI(x)=e                                         |
| (5)  (x) I(x)x=e                                         |
| (6)  (H) (SUBGR(H) <-->                                  |
|              [& (E x) H(x)                               |
|                 (x)(y) {H(x)&H(y) --> H(xy)}             |
|                 (x) {H(x) --> H(I(x))}])                 |
| (7)  (H1)(H2) (SETEQ(H1,H2) <-->                         |
|                 (x)(H1(x) <--> H2(x)))                   |
| (8)  (g)(xx)(H) (COSET(g,xx,H) <-->                      |
|              [& SUBGR(H)                                 |
|                 (x){xx(x) <-->                           |
|                     (E y)(H(y) & x=yg)}])                |
| (9)  (g)(xx)(H) (COSET(g,xx,H) -->                       |
|                 [H(g) <--> SETEQ(xx,H)])                 |
+----------------------------------------------------------+
```

Box 1. Axioms - not minimal - (1-5), definitions (6-8) and a theorem (9) from group theory.

Direct translation of (1-8) and the negation of (9) into conjunctive normal form yields 39 clauses with altogether 109 literals. INSURER however recognizes that (9) can be decomposed into:

(10) (g)(H) (H(g) OR (xx) {OR ~COSET(g,xx,H)
~SETEQ(xx,H)}) and

(11) (g)(H) (~H(g) OR (xx) {OR ~COSET(g,xx,H)
SETEQ(xx,H)}).

Working on (10) the definitions of COSET, SETEQ and SUBGR are respectively substituted. The result is negated and together with (1-5) translated into conjunctive normal form yielding 14 clauses with 23 literals. After removing COSET and SETEQ in (11), it turns out that INSURER applies again splitting up (11) into two subproblems. Each one ends up with 13 clauses and 20 literals. Although the resolution component is not able to handle these three subproblems, the chance of finding a solution has increased by an ´infinite´ amount when compared to the non-decomposed situation.

INSURER also can handle the sorted predicate calculus that was described in [14]. The same coset example formulated in sorted predicate calculus - without decomposition - yields 28 clauses with 61 literals. INSURER also finds here three subproblems each having 12 clauses with respectively 16, 14 and 14 literals. A significant reduction again, although the connection graph resolution component, in the mean time extended with paramodulation facilities, still cannot handle them. (Instead of relying on paramodulation, we consider adding an equality specialist to the deductive community.)

The next example was taken from [37] and was already worked on as reported in [14], see box 2. It was originally used in [37] for illustrating automatic programming. A simple sorting algorithm was generated by adding an ´answer-predicate´ to the negated conjecture and submitting all the formulas to the QA3 resolution theorem prover. C. Green admits that the axioms are ´tuned´ for the algorithm generation. The conjecture contains for instance the function ´sort´ which is not referred to by the other axioms. In fact one can prove from the axioms the expression (7) with ´R(cdr(x),sort(cdr(x)))´ replaced by:
´(E z)R(cdr(x),z)´, from which (7) can be inferred.

The main predicate is Sd which expresses that its argument, a list, is sorted. The expression R(x,y) signifies that the list y is a sorted permutation of the list x; Equal(x,y) signifies that the list x is identical with the list y, the empty list is indicated by nil. The function merge corresponds with merging a list with a sorted list such that a sorted list is the result. The function cons corresponds to adding an element in front to a list. The functions car and cdr respectively produces the first element and the remainder of a list.

```
+--------------------------------------------------------+
| (1)  (x)(y) (Sd(y) --> Sd(merge(x,y)))                  |
| (2)  (x)(y)(u) {(Sd(y) & Same(x,y)) -->                 |
|                Same(cons(u,x),merge(u,y))}              |
| (3)  (x) (Equal(x,nil) --> R(x,nil))                    |
| (4)  (x) (~Equal(x,nil) -->                             |
|           Equal(x,cons(car(x),cdr(x))))                 |
| (5)  (x)(u)(v) ((Equal(x,u) & Same(u,v)) -->            |
|                Same(x,v))                               |
| (6)  (x)(y) (R(x,y) <--> (Same(x,y) & Sd(y)))           |
| (7)  (x)(E y) (&(Equal(x,nil) --> R(x,y))               |
|                ((~Equal(x,nil) & R(cdr(x),sort(cdr(x))))|
|                  --> R(x,y)))                           |
+--------------------------------------------------------+
```

Box 2. These formulas were used by Green to generate a sorting algorithm (with an answer predicate), see [37]. Axioms (1-5), definition (6) and conjecture (7).

INSURER will decompose the conjecture in two subproblems. When INSTANCE would not have been incorporated in INSURER eight subproblems would have been found of which six are redundant. Subsequently INSTANCE recognizes that one of the subproblems is an instance of axiom(3). The remaining subproblem was solved as well with as without definition substitution (by adding the definition to the axioms). In both cases a contradiction was found more easily than in the non-decomposed case, see table 1.

program and strategy	input + generated clauses	g-penetrance
QA3	286	0.091
resolution only	38 (25)	0.579 (0.680)
+ INSURER and INSTANCE	28 (17)	0.785 (0.882)
+ definition substitution	20 (12)	0.800 (0.917)

Table 1 shows the effectiveness of INSURER and INSTANCE. The numbers between brackets refer to values obtained when the sorted predicate calculus is used [14]. The g-penetrance is defined as #(clauses in proof)/ #(input+generated clauses). The QA3 values were taken from [37].

Our final example consists of only one formula:

[{(E x1)(y1)P(x1)<-->P(y1)}<-->{(E x2)Q(x2)<-->(y2)P(y2)}]
<-->
[{(E x3)(y3)Q(x2)<-->Q(y3)}<-->{(E x4)P(x4)<-->(y4)Q(y4)}].

P. Andrews posed this problem at the Fourth Workshop on Automated Deduction, Austin, February 1979. He added that he was willing to send the first 500 clauses for free. Resolution theorem provers are drowned as a result of the many clauses generated by the PC-CNF translator as a consequence of 7 equivalences which each time double the length of the formula. INSURER, heavily invoking INSTANCE resulted in 169 succesful instance recognitions, reducing the formula to TRUE.

6. What next

The results of the preceding section suggest to us that a deductive ´architecture´ built up from deductive specialists is promising. Certainly it is advisable to pursue this road first with the restriction that the deductive components are algorithmic and thus always halting. Examples are: -- model evaluator to decide whether a subgoal is hopeless (since nót true in a model) [36], -- equality substitution simplifier which replaces complex terms by equal but less complex terms, -- an if-then-else recognizer which can split a problem into two subproblems of lesser complexity, etc. At a certain point, this algorithmic restriction should be abandoned. Then the realm of search is entered again, no longer on the modus ponens level but with operators of greater scope: -- check whether it is worthwhile to introduce an abbreviation for a recurring expression; -- apply key theorem aa at bb; -- try to adapt the proof for a similar result in a less general theory; -- try to prove a more general result which can be expressed more concisely (and which is not falsified by any available model); -- resort to induction in a specific context; -- try to reinterpret the theory under consideration into other available theories; etc.

Somehow the phenomena must be dealt with that at a certain stage in a theory, some previous result will be applied ´automatically´ when they can be applied. Thus when a theory becomes activated, some theorems become active in a ´compiled format´, as an additional derivation rule. At the same time, we doubt that this ´compilation´ is an all-or-nothing matter; a theorem can gradually reach the status of being applied automatically (while this process still always remains backtrackable).

Frequently it has been stressed that something should be done with a newly found proof, that it should be the input for some kind of a learning component. Somehow, nobody has ever designed a procedure that could do something useful with the many mechanical proofs that have been generated in the last decades. But even when we refrain from starting a learning process we still need a description of the proof (and also of its associated theorem) in order to use it as a guideline

for setting up a proof in an analogy type of reasoning. We suspect that the lack of a greater variety of deductive operators which hamper proving interesting theorems, is also responsible for the impossibility to make sense of obtained proofs.

When a larger collection of operators in a theory is available, an obvious step would be to assign them priorities, automatically on the basis of performance, or initially by ´Acts of Gods´ - hence by programmers. Then it would be possible to generate - recursively, and thereby introducing another dimension in which search is performed - skeleton proofs, to be refined in the next level of recursion. Sacerdoti in [73] has obtained convincing results with this technique in the realm of plan generation.

Yet there is still a fair chance that the problem of mechanically proving of difficult mathematical conjectures can advantageously be replaced by another problem: how to generate automatically (with respect to a given collection of definitions, axioms, lemmas, theorems, models and similar theories) an interesting conjecture or concept to be defined. This capability, at least to some extent, might be essential for generating intermediate stepping stones for a really difficult theorem.

CONCLUSION

A.I. has been expanding vigorously in the last 20 years, and the number of publications continues to increase. The field has become so large that a tendency has emerged to split it up into different sections Computational Linguistics, Deduction, Cognitive Science and Vision. A hidden motivation for this fragmentation may be a desire to escape from the name ´Artificial Intelligence´ which arouses strong feelings in some circles. In spite of this centrifugal force the field still (1981) manages to organize conferences where all sections come together.

It is customarily pointed out that substantial progress in all sections of A.I. awaits the capability of storing large amounts of knowledge to be used for intelligent activities. This position is certainly correct, but the snag is that before a lot of knowledge can be amassed, profound insight into the activities to be supported is required, otherwise the knowledge cannot be structured in such a way that relevant facts will be found quickly. Thus we have a real chicken and egg situation.

The substance of this thesis concerns algorithms for Search, Program Verification and Deduction. These algorithms perform well without support from massive knowledge. We believe that more such algorithms can be developed. Nevertheless work in the realm of permanent and temporal knowledge representation is to be recommended. In particular, it is recommended that the main source of inspiration for knowledge representation should not be the generalization of lexicon structures, but the support of knowledge-intensive algorithms. Giving heuristic functions, as they are used in the A^*-algorithm, a firm footing in general knowledge representation schemes, is an obvious example of the work to be done.

Chapter two deals with the generalization of the uni-directional A^*-algorithm to the bi-directional case. A uni-directional theorem says that a shortest path will be found (without exhaustive searching) provided the heuristic has certain properties. This theorem has been generalized to the bi-directional algorithm, as well as the so called

´optimality´ theorem.

The results we reported about bi-directional heuristic search suggest there is still room for improvement. Shorter solution paths were found in comparison with uni-directional search, but at higher computational costs. The potential advantage of working simultaneously in both directions, as we as humans frequently do, has not yet been formally clearified. Recently we initiated a new bi-directional project to attack this problem anew.

The main results of chapter three (substitution functions coded in LISP), concern automatic verification of code with side effects. The method developed ensures correct description of side effects for a subset of nasty LISP functions, which includes our newly introduced substitution function SUBSTAD. The verification of several versions, some of which were done completely automatically, reveals that the formal description of some functions is at present practically intractable. For instance, we estimate that the formal description of loop invariants for a particular version of a support function for SUBSTAD, requires several magnitudes more text than code (bearing in mind that making up formal descriptions is certainly as difficult as programming). This imbalance suggests that the expressive power of computer languages has currently outgrown the expressive power of state-description languages.

Although we agree with De Millo et al [23] that the present verification tools do not lend themselves to practical use, we do not share their conviction that the whole bussiness should be abandoned. Verifiers will probably always run into resource limitations, but it is premature to assume that they will never be able to use mechanisms similar to those that enable humans to circumvent, without sacrificing preciseness, some of these limitations.

Chapter four deals with algorithmic deductive modules and its theoretical results concern obvious requirements for these modules. It is reassuring to observe that when the one-way pattern matcher INSTANCE reports success, one of the arguments can be inferred from the other, which makes INSTANCE sound. It is likewise nice to know

that the subproblem recognizer gives maximal problem decompositions. But still more important are the results of the implementation of the modules. A deduction complex made up of a simple supervisor for these modules, together with a definition applier and a (connection graph resolution) refutation constructor, could solve problems which were distinctly beyond the capability of the sole refutation machine. The setup is structurally similar to the Hearsay [50] architecture, in which separate specialist modules - here even located in different machines, thus allowing parallel processing - were also cooperating. We intend to develop other deductive modules and to give more attention to elaborate supervisors as a means of pushing the deductive limitations further away.

The relative ease with which fairly complicated problems can be programmed, and the reasonable performance on a wide range of problem instances of such programs, suggest that more attention should be given to real-life application of A.I. A few years ago we managed in two months time to program a natural language input processor for a nice fragment of Dutch. This was no ad hoc program, but one which used the special ATN language [90,91,92] that supports a wide range of natural languages. With such tools, the development of commercial products becomes feasible. The industry/ software houses should jump at these opportunities [40].

Although we have studied quite disparate topics in A.I. the method we employed has been consistently the same. A quick and superficial literature study quided by fresh intuitions was translated as rapidly as possible into a running program. Study of the results and the *behavior* of the program then led to improvements, generalizations and/or complete revision. The literature was subsequently studied more carefully and some theory eventually developed. Finally, experiments were performed using when possible problems from the literature.

This method is time consuming, and not the way to present Flashy Grand Theories. In fact, we shy away from F.G.T. because there have been too many of them in the past lending to A.I. an exotic albeit questionable reputation. We recommend this method as a way to study A.I. ´seriously´.

SAMENVATTING (in Dutch)

Aangezien Kunstmatige Intelligentie in Nederland niet erg bekend is en omdat het vak snel verandert, begint dit proefschrift met een tamelijk lange inleiding waarin een overzicht gegeven wordt van het gehele gebied. Uiteraard pretenderen we geenszins dat naar volledigheid gestreefd; daarvoor is het gebied te omvangrijk, terwijl er geen duidelijke hoofdstroom is. Beoefenaars hebben wijduiteenlopende achtergronden: wiskunde, psychologie en linguistiek. De positionering van K.I. ten opzichte van de overige wetenschappen wordt er niet eenvoudiger door. Een van de conclusies uit de inleiding is, dat de samenhang die K.I. gedurende de afgelopen 25 jaar ontegenzeggelijk getoond heeft, niet zozeer veroorzaakt wordt door een onveranderd onderwerp van interesse, maar door een onorthodoxe methodologie: het maken van theorieen die geimplementeerd kunnen worden, zodat niet alleen uitkomsten, maar in het bijzonder het *gedrag* van programma´s en zo de daarmee corresponderende processen, bestudeerd kunnen worden. Uiteraard ´beperkt´ men zich daarbij tot intelligente processen, maar die zijn zeer divers en bij lange na niet geinventariseerd, zodat dit niet een echte beperking genoemd kan worden.

Een bekende karakterisering van K.I. is dat ze nooit een succes kan boeken. Elk behaald resultaat wordt meteen gekleineerd als een onbetekenend speciaal geval van een nog ongebegrepen proces waarin de ´echte´ intelligentie zich verschuilt. In deze zin hebben we ons met echte K.I. bezig gehouden, want we kunnen niet pretenderen de bestudeerde vraagstukken definitief te hebben verhelderd.

De afgelopen jaren hebben we ons in feite met verschillende aandachtsgebieden uit de K.I. bezig gehouden, die men, zoals we in sectie 1.2 hebben beargumenteerd, toch als samenhangend kan beschouwen.

Hoofdstuk 2 heeft betrekking op de generalisatie van het z.g.n. A^*-algorithme. Het A^*-algorithme kan vanuit een gegeven toestand en met gegeven operatoren een pad construeren naar een verlangde toestand. Onder zekere voorwaarden is er de garantie dat, zonder de

brute kracht van het nagaan van alle mogelijkheden, toch een kortste pad gevonden zal worden. Het gegeneraliseerde algorithme construeert een oplossing door beurtelings vanaf twee kanten aan een pad te werken. Onder gelijke voorwaarden geldt weer dat een gevonden oplossing een gegarandeerd kortste pad is. Bovendien geldt dat de pad-componenten elkaar ontmoeten in het ´midden´ van de zoekruimte, daarmee een bekend nadeel van Pohl´s generalisatie vermijdend [66]. Een implementatie van het algorithme wijst uit, dat op een verzameling van 320 problemen er kortere paden dan met het A^*-algorithme worden gevonden, terwijl wanneer de beperking wordt opgelegd, dat er per probleem niet meer dan 1000 toestanden worden bezocht, er ook meer oplossingen worden gevonden.

In hoofdstuk 3 worden een aantal substitutie functies bestudeerd. Deze zijn van belang omdat ze een centrale rol spelen bij z.g.n. unificatie-algorithmen, die op hun beurt weer de cruciale onderdelen zijn van programma´s voor patroonherkenning en het automatisch bewijzen van stellingen. Omdat de meeste tijd doorgaans wordt doorgebracht in het unificatie-algorithme, is het de moeite waard om efficientie daar tot het uiterste op te voeren. Enkele bekende unificatie-algorithmen, geschreven in LISP, werden aanzienlijk versneld door een niet-copieerende substitutie functie toe te voegen aan het repertoire van LISP en de unificatie-algorithmen enigszins te herschrijven. Het niet-copieren van deze substitutie functie is tot stand gebracht met behulp van zijeffecten van enkele standaard functies. Tot op heden heeft men weinig aandacht geschonken aan het automatische verifieren van programma´s waarin zijeffecten optreden, want het verifieren van gewone programma´s is al lastig genoeg. Het was dan ook een uitdaging om de door ons geintroduceerde substitutie functie op dit punt nader te bestuderen. Een theorie werd ontwikkeld over de semantiek van LISP, waarin voor een ruime klasse van functies met zijeffecten voorzieningen zijn aangebracht. De theorie kon verder worden ontwikkeld dan een soortgelijke die voor PASCAL werd gemaakt [52,81]. Aangezien programma-verificatie met de hand een zeer bewerkelijke zaak is, ontwikkelden we een z.g.n. symbolische evaluator, die met symbolische invoer alle paden van een programma ´doorrekend´. Met behulp van het deductieprogramma, dat in het volgende hoofdstuk gedeeltelijk beschreven wordt, kan vervolgens

gecontroleerd worden of de symbolische uitvoer voldoet aan de functiespecificatie, waarmee de verificatie voltooid is. We slaagden erin enkele versies met deze techniek te verifieren, maar moeten anderzijds rapporteren dat er een versie was die zich aan verificatie onttrok, doordat specificatie een hoeveelheid tekst vereist die naar schatting een meer dan honderdvoudige omvang zou krijgen dan de tekst van de code zelf.

Het vierde en laatste hoofdstuk beschrijft een tweetal algorithmische deductiemodulen. Het is algemeen bekend dat deductie essentieel onoplosbaar is in die zin dat er geen algorithme kan bestaan dat kan beslissen of een vermoeden al dan niet een theorema is. De zoekruimte is zodanig, dat elk betrouwbaar programma er oneindig lang in kan ronddolen zonder een beslissing te kunnen nemen. Belangrijker is echter dat elk programma dat slechts gebruik maakt van brute kracht, zelfs als te voren bekend is dat een vermoeden in feite te bewijzen is, vastloopt in tijd- en ruimtebeperkingen alvorens het juiste antwoord te vinden. Dat er iets mis is met de brute kracht programma's blijkt ook uit het feit, dat oplossingen die ze wel vinden voor ons niet inzichtelijk zijn. Dit heeft ons er toe gebracht om 'deductieve trucs', via introspectie verkregen, die een duidelijk afgebakende werking hebben, in algorithmen onder te brengen. We kiezen daarmee voor een deductie 'familie' (in principe uitbreidbaar hoewel we ons voorshands verre houden van leerproblemen) bestaand uit algorithmische modulen die een probleem met een hogere prioriteit kunnen aanpakken dan een algemene niet-algorithmische zoekmethode die ook deel uitmaakt van de familie. Een van de modulen tracht een probleem equivalent te herschrijven in een aantal deelproblemen, die in principe eenvoudiger oplosbaar zijn. Een ander moduul kan herkennen dat een probleem een alfabetische variant en/of een speciaal geval is van een ander probleem of een al bekende formule.

Eigenschappen van deze modulen worden weer (met de hand) bewezen. Deze modulen werden geintegreerd te samen met een eenvoudige definitie-specialist en een klassieke stellingenbewijzer. Experimenten met o.a. voorbeelden uit de literatuur hebben aangetoond dat dit deductiecomplex inderdaad krachtiger is dan de klassieke zoekcomponent.

Als saillant detail zij tenslotte vermeld dat de niet-copieerende substitutie-functie in de stellingbewijzer gebruikt wordt, die op zijn beurt gebruikt is om de correctheid van die substitutie-functie aan te tonen.

REFERENCES

[1] ABELSON, R.P., *The Structure of Belief Systems*, in R.C. Schank & K.M. Colby (Ed), Computer Models of Thought and Language, pp 251-286, Freeman & Co, San Fransisco, 1973.

[2] ALLEN, J.F. & C.R. PERRAULT, *Plans, Inference and Indirect Speech Acts*, Proceedings of the 17th Annual Meeting of the Association of Computational Linguistics, pp 85-87, August 1979.

[3] BARSTOW. D.R., *The Role of Knowledge and Deduction in Program Synthesis*, IJCAI6, pp 37-43, Tokyo, 1979.

[4] BOYER, R.S. & J.S. MOORE, *A Computational Logic*, Academic Press Inc., NY, 1979.

[5] BRACHMAN, R.J., *A Structural Paradigm for Representing Knowledge*, BBN Report no 3605, May 1978.

[6] BRESNAN, J., *A Realistic Transformational*, in M. Halle et al (Ed), Linguistic Theory and Psychological Reality, pp 1-59, MIT Press, 1978.

[7] BROWN, J.S. & R.R. BURTON, *Multiple Representations of Knowledge for Tutorial Reasoning*, in D.G. Bobrow & A. Collins (Ed), Representation and Understanding, Academic Press, 1975.

[8] CAMPBELL, D., *Blind Variation and Selective Survival as a General Strategy in Knowledge-Processes*, in M. Yovits and S. Cameron (Ed.), Self-Organizing Systems, pp 205-231, Pergamon Press, NY, 1960.

[9] CARBONELL, J.G., *Subjective Understanding: Computer Models of Belief Systems*, Ph.D.Th., Yale University, 1979.

[10] CARBONELL, J.G., *Computer Models of Human Personality Traits*, IJCAI6, pp 121-123, Tokyo, 1979.

[11] CARBONELL, J.G., *The Counter Planning Process: Reasoning under Adversity*, IJCAI6, pp 124-130, Tokyo, 1979.

[12] CHAMPEAUX, D. de, *Solutions and their Problems*, in E. Morlet and D. Ribbens (Ed.), International Computing Symposium, North-Holland, 1977, pp 119-127.

[13] CHAMPEAUX, D. de, *SUBSTAD: For Fast Substitution in LISP with an Application on Unification*, Information Processing Letters, vol 7, no 1, January 1978, pp 58-62.

[14] CHAMPEAUX, D. de, *A Theorem Prover Dating a Semantic Network*, Proceedings of AISB/GI Conference, Hamburg 1978, pp 82-92.

[15] CHAMPEAUX, D. de, *Sub-Problem Finder and Instance Checker, Two Cooperating Preprocessors for Theorem Provers*, IJCAI6, pp 191-196, Tokyo, 1979.

[16] CHAMPEAUX, D. de, *Bi-directional Heuristic Search Again*, submitted to the JACM.

[17] CHAMPEAUX, D. de, & L. SINT, *An Improved Bi-directional Heuristic Search Algorithm*, JACM, vol 24, no 2, April 1977, pp 177-191. A slightly different version appeared first in IJCAI4, pp 309-314, Tbilisi, 1975.

[18] CHAMPEAUX, D. de, & L. SINT, *An Optimality Theorem for a Bi-directional Heuristic Search Algorithm*, The Computer Journal, vol 20, no 2, pp 148-150.

[19] CHANG, C.L. & R.C. LEE, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.

[20] CHANG, C.L. & J.R. SLAGLE, *An Admissible and Optimal Algorithm for Searching AND/OR Graphs*, Artificial Intelligence, vol 2, pp 117-128, 1971.

[21] CLARK, D.W. & G.C. GREEN, *An Empirical Study of List Structure in LISP*, CACM, vol 20, no 2, 1977, pp 78-87.

[22] DARRINGER, J.A. & J.C. KING, *Applications of Symbolic Execution to Program Testing*, IBM Report RC 6965, January 1977.

[23] DE MILLO R.A. et al, *Social Processes and Proofs of Theorems and Programs*, CACM, vol 22, no 5, May 1979, pp 271-280.

[24] DORAN, J. & D. MICHIE, *Experiments with the Graph Traverser Program*, Proceedings of the Royal Society (A), 294, pp 235-259, 1966.

[25] DOYLE, J., *Truth Maintenance Systems for Problem Solving*, A.I. Laboratory, MIT, AI-TR-419, January 1978.

[26] DRESHER, B.E. & N. HORNSTEIN, *On Some Supposed Contributions of Artificial Intelligence to the Scientific Study of Language*, Cognition, vol 4, pp 321-398, 1976.

[27] DRESHER, B.E. & N. HORNSTEIN, *Reply to Winograd*, Cognition, vol 5, pp 379-392, 1977.

[28] ENDERTON, H.B., *A Mathematical Introduction to Logic*, Academic Press, 1972.

[29] ERNST, G. & A. NEWELL, *G.P.S.: A Case Study in Generality and Problem Solving*, ACM Monograph Series, Academic Press, NY, 1969.

[30] FAUGHT, W.S., *Motivation and Intensionality in a Computer Simulation Model of Paranoia*, Basel: Birkhaeuser Verlag, 1978.

[31] FAUGHT, W.S., *Modelling Intentional Behavior Generation*, IJCAI6, pp 263-265, Tokyo, 1979.

[32] FAUGHT, W.S., K.M. COLBY & R.C. PARKISON, *Inferences, Affects and Intentions in a Model of Paranoia*, Cognitive Psychology, no 9, pp 153-187, 1977.

[33] FIKES, R.E., P.E. HART & N.J. NILSSON, *Learning and Executing Generalized Robot Plans*, Artificial Intelligence, vol 3, no 4, pp 251-288, 1972.

[34] FIRSCHEIN, O. et al, *Forecasting and Assessing the Impact of Artificial Intelligence on Society*, IJCAI3, pp 105-2120, Stanford, 1973.

[35] FLECK, J., *Artificial Intelligence. A Case Study in Scientific Development*, AISB, 35, pp 3-6, October 1979.

[36] GELERNTER, H., *Realization of a Geometry Theorem Proving Machine*, Proceedings of an International Conference on Information Processing, Paris: UNESCO House, 1959, pp 273-282, reprinted in E.A. Feigenbaum & J. Feldman (Ed.), Computers and Thought, McGraw-Hill, NY, 1963, pp 134-152.

[37] GREEN, C., *The Application of Theorem-Proving to Question-Answering Systems*, Stanford Artificial Intelligence Project Memo AI-96, June 1969.

[38] HAREL, D., *Proving the Correctness of Regular Deterministic Programs: A Unifying Survey Using Dynamic Logic*, IBM Report 7557, March 1979.

[39] HAREL, D., *First Order Dynamic Logic*, Springer Verlag, 1979.

[40] HARRIS, L.R., *Status Report on the ROBOT Natural Language Query Processor*, Sigart Newsletter, no 66, pp 3-4, August 1978.

[41] HART, P., N. NILSSON, and B. RAPHAEL, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Trans. Sys. Sci. Cybernetics, vol SSC-4, no 2, pp 100-107, July 1968.

[42] HAYES, P.J., *A Representation for Robot Plans*, IJCAI4, pp 181-188, Tbilisi, 1975.

[43] JOHNSON-LAIRD, P.N., *Procedural Semantics*, Cognition, vol 5, no 3, September 1977, pp 189-214.

[44] KING, J.C., *Symbolic Execution and Program Testing*, CACM, vol 19, no 7, July 1976, pp 385-394.

[45] KLING, R.E., *A Paradigm for Reasoning by Analogy*, Artificial Intelligence, vol 2, pp 147-178, 1971.

[46] KOWALSKI, R., *A Proof Procedure Using Connection Graphs*, JACM 22, no 4, October 1975, pp 572-595.

[47] LANGLEY, P., *Rediscovering Physics with BACON3*, IJCAI6, pp 505-507, Tokyo, 1979.

[48] LEE, S., W.P. de ROEVER & S.L. GERHART, *The Evolution of List Copying Algorithms*, 6th Annual ACM Symposium on Principles of Programming Languages, pp 53-67, San Antonio, 1979.

[49] LENAT, D.B., *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*, Memo AIM-286, Stanford AI Lab, 1976.

[50] LESSER, V.R. & L.D. ERMAN, *A Retrospective Vieuw of the HEARSAY-II Architecture*, IJCAI5, pp 790-800, Cambridge, 1977.

[51] LONDON, P.E., *Dependency Networks as a Representation for Modelling in General Problem Solvers*, Department of Computer Science, University of Maryland, TR-698, 1978.

[52] LUCKHAM, D. & N. SUZUKI, *Automatic Program Verification V*, Memo AIM-278, Stanford AI Lab, 1976.

[53] LOVELAND, D.W., *Automated Theorem Proving. A Logical Basis*, North-Holland, 1978.

[54] MANNA, Z., *Introduction to Mathematical Theory of Computation*, McGraw-Hill, NY, 1972.

[55] MARR, D., *Visual Information Processing*, IJCAI6, pp 1108-1126, Tokyo, 1979.

[56] McCALLA, G. et al, *Investigations into Planning and Executing in an Independent and Continuously Changing Microworld*, AI Memo 78-2, Department of Computer Science, University of Toronto, July 1978.

[57] McCALLA, G. & P.F. SCHNEIDER, *The Execution of Plans in an Independent Dynamic Microworld*, IJCAI6, pp 553-555, Tokyo, 1979.

[58] McCARTHY, J. & P.J. HAYES, *Some Philosophical Problems from the Standpoint of Artificial Intelligence*, in B. Meltzer & D. Michie (Ed), Machine Intelligence 4, Elsevier NY, 1969, pp 463-502.

[59] MILLER, G.A., *Semantic Relations among Words*, in M. Halle et al (Ed), Linguistic Theory and Psychological Reality, pp 1-59, MIT Press, 1978.

[60] MOORE, E., *The Shortest Path Through a Maze*, Proc. Intern. Symp. Theory Switching, Part II, April 2-5, 1957.

[61] NEVINS, A.J., *A Human Oriented Logic for Automatic Theorem-Proving*, JACM 21, no 4, October 1974, pp 606-621.

[62] NEWELL, N. & H.A. SIMON, *G.P.S. A Program that Simulates Human Thought*, Lernende Automaten, R. Oldenbourg KG, Munich, 1961, reprinted in Computers and Thought, E.A. Feigenbaum & J. Feldman (Ed), McGraw-Hill, NY, 1963, pp 279-296.

[63] NILSSON, N.J., *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, NY, 1971.

[64] *Ontwikkelingsplan Informatica*, Centrale Informatica Commissie Universiteit van Amsterdam, Januari 1979.

[65] PATERSON, M.S. & M.N. WEGMAN, *Linear Unification*, Proceedings Eighth Annual ACM Symposium on Theory of Computing, Hershey, Penn., May 1976, pp 181-186; and in JCSS, 16, 1978, pp 158-167.

[66] POHL, I., *Bidirectional Heuristic Search in Path Problems*, Stanford University, California, 1969.

[67] POHL, I., *First Results on the Effect of Error in Heuristic Search*, in B. Meltzer and D. Michie (Ed.), Machine Intelligence 5, pp 219-236, American Elsevier Publishing Company, NY, 1970.

[68] QUINE, W.O., *Methods of Logic*, Holt N.Y., 1950.

[69] RAPHAEL, B., *The Thinking Computer*, Freeman & Co, San Fransisco, 1976.

[70] RATHENAU, G.W. et al, *Rapport van de Adviesgroep Micro-Electronica*, Staatsuitgeverij, Den Haag, 1980.

[71] ROBINSON, J.A., *A Machine-Oriented Logic Based on the Resolution Principle*, JACM, vol 12, no 1, Jan 1965, pp 23-41.

[72] RULIFSON, J.F., J.A. DERKSEN, R.J. WALDINGER, *QA4: A Procedural Calculus for Intuitive Reasoning*, Stanford Research Institute, Technical Note 73, November 1972.

[73] SACERDOTI, E.D., *Planning in a Hierarchy of Abstraction Spaces*, IJCAI3, Stanford, pp 412-422, 1973.

[74] SACERDOTI, E.D., *A Structure for Plans and Behavior*, Artificial Intelligence Series, Elsevier Computer Science Library, 1977.

[75] SAMUEL, A.L., *Some Studies in Machine Learning Using the Game of Checkers*, IBM J.R.D., (3) pp 211-229, reprinted in E.A. Feigenbaum & J. Feldman (Ed), Computers and Thought, pp 71-108, McGraw-Hill NY, 1963.

[76] SCHANK, R.C., *SAM A Story Understander*, Research Report no 43, Yale University, August 1975.

[77] SCHORR, H. & W.M. WAIT, *An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures*, CACM, vol 10, pp 501-506, 1967.

[78] SLOMAN, A., *The Computer Revolution in Philosophy*, Harvester Press, Sussex, 1978.

[79] *Special Issue on Non-Monotonic Logic*, Artificial Intelligence, vol 13, no 1 & 2, April 1980.

[80] *Struktuurplan Informatica (W.O.)*, Academische Raad, December 1974.

[81] SUZUKI, N., *Automatic Verification of Programs with Complex Data Structures*, Memo AIM-279, Stanford AI Lab, 1976.

[82] TATE, A., *Interacting Goals and their Use*, IJCAI4, pp 215-218, Tbilisi, 1975.

[83] THOMPSON, A.M., *Network Truth-Maintenance for Deduction and Modelling*, IJCAI6, pp 877-879, Tokyo, 1979.

[84] TOPOR, R.W., *The correctness of the Schorr-Waite List Marking Algorithm*, Acta Informatica, vol 11, pp 211-221, 1979.

[85] WANG, H., *Toward Mechanical Mathematics*, IBM Journal, Jan 1960, pp 2-22.

[86] WANNER, E. & M. MARATSOS, *An ATN Approach to Comprehension*, in M. Halle et al (Ed), Linguistic Theory and Psychological Reality, pp 60-118, MIT Press, Cambridge MA, 1978.

[87] WARD, M.R., L. ROSSOL & S.W. HOLLAND, *CONSIGHT: A Practical Vision-Based Robot Guidance System*, Computer Science Department, General Motors Research Laboratories, Warren Michigan, 1979.

[88] WINOGRAD, T., *On Some Contested Suppositions of Generative Linguists about the Scientific Study of Language*, Cognition, 5, pp 151-179, 1977.

[89] WINSTON, P.H., *Learning Structural Descriptions from Examples*, AI-TR-231, AI Lab MIT, 1970.

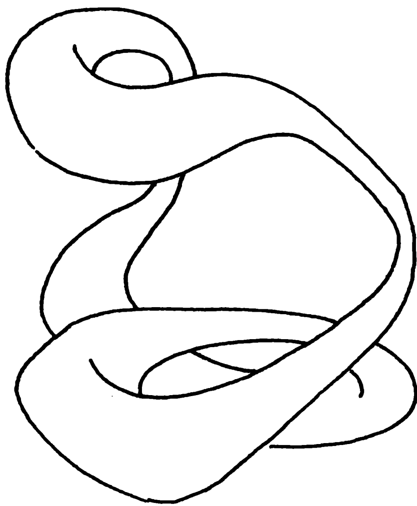
[90] WOODS, W.A., *Procedural Semantics for a Question- Answering Machine*, Proc. AFIPS 1968 Fall Joint Computer Conference, vol 38, Part I, MDI Publications, pp 457-471.

[91] WOODS, W.A., *Transition Network Grammars for Natural Language Analysis*, CACM, vol 13, no 10, pp 591-606, October 1970.

[92] WOODS, W.A. et al, *The Lunar Sciences Natural Language Information System*, BBN Report no 2378, NTIS N72-28984, June 1972.

[93] ZIEGLER, J.F. & W.A. LANFORD, *The Effect of Cosmic Rays on Computer Memories*, IBM Report RC 7648, October 1979.

DUMMIES ARE NICE, TOO!



ERRATA (March 19, 1981)

page	line(s)	replace	by
2.7	in top figure	the capital S at the top of fig 2.4	a small s
			add a small t at the bottom of fig 2.4 as in fig 2.5
2.25	in subscriptions of fig 2.7 & 2,8	Uni-directonial	Uni-directional
		Bi-directonial	Bi-directional
2.32	in subscription of fig 2.13	all occurrences of δ	d
		unidirectional	uni-directional
		bidirectional	bi-directional
		.are	.
		. (a)	.
		f-values	F-values
		. f=m	.
		f=	F=
2.33	in subscription of fig 2.33	Bidirectional	Bi-directional
		all occurrences of δ	d
		f=	F=
		2 2 (	2x2x(
2.34	in subscription of fig 2.15	f-values	F-values
		all occurrences of δ	d
		f=	F=
3.25	in fig	the capital S_1	s small s_1
3.58	top of fig	FX	EX
3.60	in fig 3.8	the three occurrences of vs1	vs1*
3.61	under top two triangles	ex not in	ex not in cell
	under middle triangles	} {	} & {
3.80	last line middle paragraph	suggestion	conjecture
6.3	3th line from bottom	heeft	hebben

STELLINGEN

behorende bij het proefschrift

ALGORITHMS IN ARTIFICIAL INTELLIGENCE

van

D.M.G. de Champeaux de Laboulaye

29 april 1981

1.
Het algorithme beschreven in Solutions and their Problems, D. de Champeaux, International Computing Symposium 1977, (Ed.) E. Morlet & D. Ribbens, North-Holland, 1977, bldz. 119-127, bedoeld voor het behandelen van problemen waarbij reductieoperatoren ter beschikking staan, die aanleiding geven tot de genererinq van afhankelijke deelproblemen, is bestand tegen de noodzaak om een oplossingspad te construeren waarop een deelprobleem N-1 maal (voor willekeurige N) opgelost en ongedaan gemaakt moet worden, alvorens een totale oplossing bereikt kan worden.

2.
De z.g.n. 'sorted predicate calculus', als beschreven in A Theorem Prover Dating a Semantic Network, D. de Champeaux, Proceedings of the AISB/GI Conference on A.I., Hamburg, 1978, bldz. 82-92, maakt het mogelijk om een probleem compact te formuleren zodat, met een aangepast unificatie-algorithme, een korter bewijs mogelijk wordt (en daarmee de kans kleiner om tegen de grens van een beperkt computergeheugen aan te lopen) dan wanneer de gewone predicatencalculus wordt gebruikt.

3.
Met het programma beschreven in Een Gedemocratiseerde Besluitvormingsmethode met een Komputer voor Lokaliseringsproblemen in de Planologie, B. Erwich & D. de Champeaux, Stedebouw & Volkshuisvesting, no 12, december 1973, bldz. 473-482, kan de positiebepaling van objecten in een twee-dimensionale ruimte zodanig plaats vinden dat met verschillende visies van belanghebbenden rekening wordt gehouden, terwijl het daarmee samenhangende besluitvormingsproces doorzichtig is.

4.
De lineariteit van het unificatie-algorithme van M.S. Paterson en M.N. Wegman, ACM Sigact, 1976, bldz. 181-186, is misleidend omdat
- van de veronderstelling wordt uitgegaan dat gelijksoortige structuur binnen en tussen de argumenten te voren al onderkend is, en
- een gevonden substitutie van de gedaante $((x_1.v_1) \ldots (x_n.v_n))$ slechts voldoet aan de eigenschap: x_i komt niet voor in v_j voor $i<=j$ (in plaats van de eigenschap: x_i komt niet voor in v_j).

5.
Het voorstel tot standaardisatie van het invoerformaat van deductie programma's/ deductieve modulen, Overbeek, R.A. & E.L. Lusk, Data Structures and Control Architecture for Implementation of Theorem-Proving Programs, 5th Conference on Automatic Deduction, (Ed.) W. Bibel & R. Kowalski, Springer-Verlag, 1980, bldz. 232-246, is voorbarig, aangezien de bijdrage van de context waarin deductie plaats vindt nog niet voldoende is onderzocht.

6.
Het belang van waarheidscondities voor het proces van het begrijpen wordt overschat.

7.
Het begrip 'natuurlijk' dat in de wiskunde veelvuldig gehanteerd wordt, vindt daarin z'n meest tegennatuurlijke toepassing.

8.
Algemenere toepasbaarheid, met veel moeite in een computerprogramma ingebouwd, blijkt bij gebruik juist m.b.t. tot een andere dimensie tekort te schieten.

9.
De beste algorithmen vindt men in kookboeken.

10.
Aan de wettelijke verplichting van een vakgroep om een onderzoeksplan op te stellen, dienen concrete consequenties te worden verbonden.

11.
Het huidige systeem van drempelkriteria waarmee woningen al dan niet tot de vrije sector behoren, heeft tot gevolg dat geringe verschillen in woongenot aanleiding kunnen geven tot exorbitante verschillen in woonlasten.

12.
De kwaliteit van de besluitvorming op het gebied van de ruimtelijke ordening vindt een triest dieptepunt in de beslissing om 500 ha weiland in de Houtrakpolder ten westen van Amsterdam te bedelven onder opgespoten zand - daarbij Ruigoord tot een macaber surrealistisch spookdorp makend - terwijl er in de afgelopen 15 jaar zelfs nog geen begin is gemaakt met industriele vestiging.

13.
Een 'lijnen' stad, waarin een knooppunt op de lijn te voet te bereiken is vanuit elk punt van de bebouwing, terwijl langs de verzonken lijn openbaar transport plaats vindt, en reisafstanden verkleind worden doordat een grotere dichtheid gerealiseerd kan worden (met al het priveautoverkeer geelimineerd), verdient nadere bestudering door stadsontwerpers.

14.
De moderne onleefbaarheid wordt niet zozeer veroorzaakt door grote tegenslagen, maar veeleer door de veelvuldigheid van op zichzelf weinig betekenende ergernissen.

15.
Iemand die zich opwindt over de zeehonden van de Waddenzee is voornamelijk bezorgd voor zichzelf. En terecht.

16.
Het verdient aanbeveling om kentekenbewijs copie deel drie te vervangen door deel vier.

17.
De overdaad aan regelknoppen van de hedendaagse Hi-Fi-apparatuur moet de gebruikers de illusie geven althans iets te kunnen beheersen.

18.
In het licht van de volgende citaten:

> Een complicatie bij het storten, c.q. ingraven is, dat het gevaar bestaat voor het binnendringen van giftige of schadelijke stoffen in de bodem. Hiertegen zal te allen tijde gewaakt moeten worden, zowel uit bodemkundig oogpunt als ter bescherming van de kwaliteit van het grondwater. ...
> Verontreiniging van de bodem dreigt voorts door infiltratie van industrieel afvalwater en olie en door langzame besmetting met kleine doses schadelijke stoffen (bv. chemische plantenziektenbestrijdingsmiddelen, motordampen, industriele afvalprodukten, enz.). De zorg voor de bodem vereist een zo groot mogelijke bescherming, ook tegen deze invloeden.

uit de Tweede Nota van de Ruimtelijke Ordening in Nederland, 1966, bldz. 72, doen recente uitspraken van autoriteiten over chemische verontreinigingen denken aan struisvogelpolitiek.

19.
Als een volgende stelling, geformuleerd als deze, waar is, dan is dit niet de laatste stelling.