CADE-15

The 15th International Conference on Automated Deduction

July 5-10, 1998, Lindau, Germany

Workshop on

STRATEGIES IN AUTOMATED DEDUCTION

July 5

Proceedings

Bernhard Gramlich and Frank Pfenning

WORKSHOP Web Site — `http://www.logic.at/staff/gramlich/cade15-ws-strategies.html`

CADE-15 Web Site — `http://www.th-darmstadt.de/cade-15`

Preface

Strategies are almost ubiquitous in automated deduction and reasoning systems. They describe, guide, and combine computations and deductions in theorem proving environments, including automated theorem provers, proof checkers, and logical frameworks. Strategies are used for various purposes, such as

- proof search and proof planning,
- restriction of search spaces,
- specification of control components,
- combination of different proof techniques and computation paradigms,
- meta-level programming in reasoning systems.

Deterministic computations or deductions based on inference rules are often insufficient to capture every computation or proof development. A mechanism is needed, for instance, to sequentialize the search for different solutions, to check context conditions, to request user input to instantiate variables, to process subgoals in a particular order, etc. Strategies may involve iteration, case analysis, or various forms of deterministic and non-deterministic choices. One may want to program strategies, transform them, or prove properties on the computations or the proofs that they describe.

Topics of interest for the workshop include all aspects related to strategies in automated deduction, in particular:

- strategy concepts and features in first- and higher-order, deductive and inductive theorem proving systems and logical frameworks,
- proof planning,
- integration of strategies in existing proof systems,
- strategy languages,
- strategy analysis and properties of strategies,
- applications and case studies where strategies play a major role.

The workshop intends to foster the active exchange of ideas among researchers interested in strategy-related topics. It continues the work and discussion of last year's CADE-14 workshop on the same theme which were quite fruitful and promising.

The organizers of the workshop would like to thank the CADE-15 Programme and Conference Committees for their support.

Bernhard Gramlich and Frank Pfenning.

Table of Contents

Part I: Invited Talks

Part II: Regular Contributions

Part I

Invited Talks

Strategic Reflection*

Patrick Lincoln and José Meseguer

SRI International, Menlo Park, CA 94025, USA

Abstract. The Maude system incorporates reflective capabilities that enable quite sophisticated strategies to be expressed very conveniently. Maude's strategy languages have been used to control the execution of large and small rewriting systems, in several areas of application. Strategies have been used to encode winning strategies for games, and can also be used for more serious purposes, such as theorem-proving strategies, and strategies taking the place of model checkers or finite-state exploration tools in the analysis of finite-state systems. In particular, finite- or infinite-state systems, such as secure communication protocols, can be specified in rewriting logic and can be subjected to formal analysis through strategies at the metalevel. Communication protocols can be specified as concurrent object-oriented systems communicating through message passing, and can be executed as Maude programs. Nondeterministic behavior such as the order of messages arrival in such protocols can be controlled via strategies at the metalevel so that all possible behaviors of a system are explored. Previously-known and previously-unknown attacks or flaws in cryptographic and multicast protocols have been discovered through the use of Maude's strategy mechanisms. In this paper several examples of the successful use of this and other kinds of strategies are pointed out.

1 Introduction

This paper gives a snapshot of a continuing effort to bring deduction strategies within the field of logic, and to exploit these high-level strategies in various domains. In last year's CADE-14 Workshop on Strategies in Automated Deduction, Manuel Clavel presented work by himself and Jose Meseguer on Internal Strategies in a Reflective Logic [11], introducing the basic notions of internal strategy languages and reflective logics, and describing a particular strategy language and an interesting example. Here we summarize some recent developments along the same lines since that time.

Maude, the basis for this work, is a logical language based on rewriting logic [24, 29, 26]. It is therefore related to other rewriting logic languages such as Cafe [14], and ELAN [19]. The equational language OBJ [17] can be regarded as a functional sublanguage of Maude. Maude is supported by a high-performance execution engine built by Steven Eker and the rest of the Maude team [8, 7].

* This work supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grants CCR-9633363 and CCR-9509931.

The key characteristics of Maude can be summarized as follows:

- *Based on rewriting logic*. This makes it particularly well suited to express, in a declarative way, concurrent and state-changing aspects of systems. Programs are theories, and rewriting logic deduction exactly corresponds to concurrent computation.
- *Wide-spectrum.* Rewriting logic is a logical and semantic framework in which specification, rapid prototyping, and efficient parallel and distributed execution, as well as formal transformations from specifications to programs can be naturally supported [21].
- *Multiparadigm.* Since rewriting logic conservatively extends equational logic [22], an equational style of functional programming is naturally supported in a sublanguage. A declarative style of concurrent object-oriented programming is also supported with a simple logical semantics; however, the present beta version does not yet have special syntax for objects. Since rewriting logic also extends Horn logic with equality in a conservative way [22], Horn logic programming could also be supported and extended in a future implementation with basic facilities for unification.
- *Reflective.* Rewriting logic is reflective [9, 10, 5]. The design of Maude capitalizes on this fact to support a novel style of *metaprogramming* with very powerful module-combining and module-transforming operations that surpass those of traditional parameterized programming and can greatly advance software reusability and adaptability.
- *Internal Strategies.* The strategies controlling the rewriting process can be defined by rewrite rules and can be reasoned about inside the logic [10, 11, 5]. Therefore, instead of having a "Logic+Control" introduction of extra-logical features, in Maude "Control $\subseteq$ Logic."

Here we give a brief overview of some recent developments exploiting the meta-level strategy languages of Maude in several application areas. The work most closely related to ours is the work on strategies in the ELAN language [3, 30, 2, 1] which also uses strategies to control the execution of rewriting logic specifications. One distinctive feature of Maude in comparison to ELAN is the tight integration of strategies within Maude's reflective capabilities. In particular, this affords a clear linguistic separation between the given specification and the different strategies that can be used to control it.

2 Reflection for Strategic Computation

Rewriting logic reflection is an extraordinarily useful language feature, having several important uses, including metaprogramming and the reification of languages, but here we focus on its use in the definition of strategy languages. The reflective metalevel of Maude enables a modular and declarative treatment of proof tactics or *strategies.* Strategies can be formally specified by rewrite rules at the metalevel of the rewrite theory expressing any tool's inference system, and can be easily changed without modifying the underlying inference system. This approach lies in contrast to other approaches to the construction of theorem proving tools which deal with tactics as extralogical features outside the formal framework, or hardwire a limited set of tactics in the tool's code.

As a simple introductory example, one can use Maude's metalevel to program strategies of execution of naive programs and executable specifications. One can write a nieve specification of sorting associative lists of elements as a single rewrite rule as follows.

```
sort([X,E1,Y,E2,Z]) => sort([X,E2,Y,E1,Z]) if E2 < E1 .
```

Various sorting algorithms can be encoded at the metalevel by controlling how this rule is applied. In particular, bubble sort can be be captured by searching for matches of the above rewrite rule where `Y` is empty. Other sorting algorithms such as merge sort and quick sort can be encoded with slightly more elaborate strategies [5]. Maude's strategy language can also be used to encode winning strategies for games such as Nim [5].

Strategies can also be used for more serious purposes, to control the execution of large rewriting systems, such as theorem provers, and to take the place of a model checker [23] or more limited finite-state exploration tools [13] in the analysis of finite-state systems, Finite- or infinite-state systems, such as some communication protocols meant to provide fault tolerance, authentication, privacy, or other properties, can be specified in rewriting logic and can be subjected to formal analysis through strategies at the metalevel. Communication protocols can be specified as concurrent object-oriented systems communicating through message passing, and executed as Maude programs. Nondeterministic behavior such as the order of messages arrival, the actions of an unpredicable environment, the failure of components of the communication system, or the malicious behavior of an attacker can be controlled by strategies defined at the metalevel so that all possible behaviors of a system are explored. Even if systems are not known to be finite state, iterative-deepening depth-first search strategies can be used to explore all behaviors of a system up to a given length. This approach has been used fruitfully in the analysis of cryptographic protocols. Denker, Meseguer and Talcott [12] have specified the Needham-Schroeder public key cryptographic protocol plus an attacker object, and have put in evidence a previously-known attack. When the modification suggested by Lowe is made to the protocol, the Maude system has not been able to discover successful attacks. We have encoded several fault-tolerant communication protocols, and have explored their behavior using Maude. Ölveczky, Denker, García-Luna, Meseguer, Smith and Talcott have found a number of flaws and deadlocks in the informal specification of a reliable multicast protocol (see Appendix B in [7]). This last work is particularly interesting in that the interaction between protocol designer and formal analysis using Maude strategies was relatively tight. The anomalies Maude uncovered were pointed out to the designers, who adjusted their protocols. These changes were then fed back through the automated Maude system, and different anomalies were found. After a few interactions, a much higher-assurance protocol was the result; the design of a dynamic-topology generalization of the protocol is currently being specified and debugged in the same fashion. In the future this kind of tighter interaction between design and formal analysis will be even more important.

Maude strategies are also being found useful in an application motivated by recent research in active networks. In attempting to deeply analyze the behavior of several small active networks programs written in the PLAN language [18], Yaw Wang, Carl Gunter, and Jose Meseguer are finding that Maude's strategies allow the automatic parallelization of search. This ongoing work helps motivate continued exploration of ideas of Mobile Maude, and the strategies to control this kind of distributed computing engine.

Rewriting logic definitions of several languages have been developed experimentally: lambda calculus and (mini-) ML [20, 22], CCS [28, 22, 5, 4, 15] Tile Logic [15], the π-calculus [31], and sketches of UNITY [25] and Gamma [27]. Since Maude is efficiently executable, such language definitions in Maude may be considered to be interpreters for these programming languages, and Maude's strategy language enables reasoning about arbitrarily complicated properties of executions. A good example of an application of Maude strategies to concurrent languages is the rewriting logic executable specification of CCS developed by Bruni and Clavel [4, 5]. More generally, Bruni, Meseguer, and Montanari [4] have developed a general translation from specifications in Tile Logic to specifications in Rewriting Logic together with a general methodology to use Maude's strategies to execute the Tile Logic specifications in their Rewriting Logic translation and have illuminated this methodology with several process calculi examples [4].

Clavel has built an inductive theorem prover in Maude using the meta-level strategy language [5, 6]. An object is represented in Maude, and rules of inference, such as explicit induction [16] are defined at the meta-level [6]. Strategies for choosing and applying the rules of inference are thus meta-metalevel rules. Maude's extremely efficient implementation enables the effective computations of the result of such meta-metalevel rules on reasonably large examples in reasonable time. We have also developed a Church-Rosser checker for rewrite rules that utilizes higher-level meta reasoning [6]. With the strategy to guide the completion process given in [11], such a checker can be naturally extended to a Knuth-Bendix completion tool. A very positive experience about the inductive theorem prover and the Church-Rosser checker is that, thanks to the high-performance of Maude and the efficient implementation of its metalevel, their rewriting logic specifications — including strategies — are in fact formal tools, with competitive performance used in both CafeOBJ and Maude itself. We indeed plan to develop several other tools for Maude using this same method in the near future.

3 Continuing Work

Many researchers are now making interesting use of strategies in rewriting logic. The discussion and references here only represent a fraction of the ongoing work in the area. The growing list of application areas where these kinds of tactics or strategies have already had a positive impact in the broader research community demonstrate the wide spectrum nature of Rewriting Logic and the Maude programming language. Although much research and experimentation remains to be done, an important lesson here is the large amount of leverage that a power tool like an integrated metalevel and strategy language can bring to a system.

4 Acknowledgments

The authors would like to thank Steve Eker, Manuel Clavel, Francisco Duran, Grit Denker, Carolyn Talcott, Peter Olveczky, and Yaw Wang for enlightening discussions and for utilizing Maude's metalevel and strategies in novel and interesting ways.

References

1. P. Borovansky, C. Kirchner, and H. Kirchner. Strategies and rewriting in ELAN. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, 1997.
2. P. Borovansky, C. Kirchner, and H. Kirchner. Strategies of ELAN: meta-interpretation and partial evaluation. In *Proceedings of the International Workshop on Theory and Practice of Algebraic Specifications (Amsterdam, Holland)*, 1997.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
4. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Manuscript, March 1998, SRI International and C.S. Dept., Univ. of Pisa, 1998.
5. M. Clavel. Reflection in general logics, rewriting logic, and Maude. Ph.D. Thesis, University of Navarre, 1998.
6. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *in Proc. of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.
7. M. Clavel, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.
8. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
9. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.
10. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.
11. M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

12. G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.
13. D. L. Dill. The Murϕ verification system. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.
14. K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.
15. F. Gadducci and U. Montanari. The tile model. To appear in G. Plotkin, C. Stirling and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press. Also, TR-96-27, C.S. Dept., Univ. of Pisa, 1996.
16. J. Goguen. Principles of parameterized programming. In T. Biggerstaff and A. Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison-Wesley, 1989.
17. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.
18. M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A programming language for active networks. In *ICFP '98, International Conference on Functional Programming*, 1998.
19. C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In V. Saraswat and P. van Hentryck, editors, *Principles and Practice of Constraint Programming: The Newport Papers*, pages 133–160. MIT Press, 1995.
20. C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 1994. To appear.
21. P. Lincoln, N. Martí-Oliet, and J. Meseguer. Specification, transformation, and programming of concurrent systems in rewriting logic. In G. Blelloch, K. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*, pages 309–339. DIMACS Series, Vol. 18, American Mathematical Society, 1994.
22. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
23. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.
24. J. Meseguer. A logical theory of concurrent objects. In *ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 1990*, pages 101–115. ACM, 1990.
25. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
26. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
27. J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proc. CONCUR'96, Pisa, August 1996*, pages 331–372. Springer LNCS 1119, 1996.

28. J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.
29. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, pages 253–293. Springer LNCS 574, 1992. Also Technical Report SRI-CSL-91-08, SRI International, Computer Science Laboratory, November 1991.
30. P.-E. Moreau and H. Kirchner. Compilation techniques for associative-commutative normalisation. In *Proceedings of the International Workshop on Theory and Practice of Algebraic Specifications (Amsterdam, Holland)*, 1997.
31. P. Viry. Input/output for ELAN. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www1.elsevier.nl/mcs/tcs/pc/volume4.htm`.

Strategic Principles in the Design of Isabelle

Lawrence C. Paulson
lcp@cl.cam.ac.uk

Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, England

Abstract. Interactive proof assistants can support proof strategies, if the right primitives have been included. These include higher-order syntax, logical variables and a choice of search primitives. Such a system allows experimentation with different automatic proof methods, even for constructive logics, new variable-binding operators, etc. The built-in unification and search make proof procedures easy to implement, typically using tableau methods. Against subgoals that arise in practice, even straightforward heuristics turn out to be powerful.

1 Introduction

This paper describes the role of strategic principles in proof assistants, taking Isabelle as an example. I use 'strategic' in two senses: first, in the sense of proof strategies, and second, in the sense of decisions that have long-term consequences. Building a proof assistant is a major investment, and we cannot foresee at the outset precisely how it will be used. So we must identify those features that seem likely to be of potential benefit, and (if we are in the academic world) of scientific interest.

Key objectives are *automation* and *flexibility*. Why automation? Because Isabelle is interactive, and formal proofs are long; obvious formulas, at least, should be proved automatically. Why flexibility? Because users will have different problems and goals from those we have foreseen. They need not just a flexible syntax but control over deep properties of their formalism: it might be first-order or higher-order, constructive or classical, etc. A flexible system gives users a choice of tools to use for automating their chosen formalism.

Soundness is obviously necessary: a proof assistant should deliver correct results and should behave in a way that inspires trust. It should also be efficient in space and time, capable of supporting realistic proofs in a range of applications.

The desire for automation and flexibility in Isabelle has resulted in the inclusion of three basic features:

- higher-order syntax
- logical variables and unification
- search primitives based on lazy lists

These have interesting implications, especially when combined in one system, as we shall see. Automation and flexibility have also affected the design of Isabelle's more specialized tactics, such as the rewriter.

PVS [7] provides automation in the form of special-purpose decision procedures. These are valuable, but the great majority of theorems cannot be proved using decision procedures alone. The rest of the system must provide sufficient automation to deal with more general problem domains.

2 Higher-Order Syntax

Most provers support quantifiers, and a few go further. A *higher-order* syntax allows users to define new variable-binding constructs such as $\mathsf{least}\ n\ P(n)$. A higher-order syntax will typically be based upon some form of typed λ-calculus.

Most research still seems to be devoted to first-order systems and, if I may be provocative, this is a curious state of affairs. First-order logic is not expressive: consider how powerful resolution provers are, and how few applications they have. Adding set theory to first-order logic yields an expressive system [14, 16], but formalizing concepts such as set comprehension, $\{x \in A \mid P(x)\}$, and general union, $\bigcup_{x\in A} B(x)$, requires a higher-order syntax. Approaches based on first-order syntax are not attractive—we might have a clumsy language of combinators or be forced to define an auxiliary function every time we want to write a comprehension.

In spite of its strengths, set theory is unfashionable. Higher-order logic has an established track record, especially for hardware verification; it is the basis of interactive verification tools such as PVS and HOL-Light [2]. But it is important to remember that higher-order syntax does not force us to use higher-order logic. It merely allows us to define variable-binding constructs, and that provides flexibility.

3 Logical Variables and Unification

By *logical variables* I mean that certain variables in formulas can be automatically instantiated using unification. The automated reasoning community is well aware of their benefits:

- *don't know* parts of a goal can be left unspecified until later
- unification can complete those parts automatically
- proof procedures (such as free-variable tableaux) can exploit unification

Logical variables have a further advantage that is relevant to Isabelle. They allow inference rules to be represented declaratively, as a generalization of Horn clauses. For example, Isabelle/HOL defines the bounded quantifier $\forall_{x\in A} P(x)$ to be $\forall x\,[x \in A \to P(x)]$ and derives the rule

$$\frac{\forall_{x\in A} P(x) \quad a \in A}{P(a)}$$

A declarative representation supports many operations on rules. They can be displayed. They can be used forwards (to derive theorems from theorems) or backwards (to derive subgoals from goals). Proof tools can classify and transform rules automatically. The cost of applying rules is independent of the computational effort expended to prove them in the first place.

Other systems represent rules procedurally, as code to transform formulas or subgoals. The best they can do is to exploit higher-order quantification, expressing the rule as a formula:

$$\forall P\, \forall a\, \forall A\, [\forall_{x \in A}\, P(x) \wedge a \in A \rightarrow P(a)]$$

This form is declarative, but it only works in higher-order logic.

Despite the advantages of logical variables, many new tools lack them. Unification is hard to provide in the presence of higher-order syntax. Isabelle performs *higher-order unification*, using Huet's procedure [3]. It is curious that such an important and useful procedure has received so little attention. Perhaps higher-order unification's theoretical limitations are to blame. It is semidecidable and sometimes yields infinitely many unifiers. But Isabelle seldom encounters these hard cases. *Pattern unification* is a refinement that handles only the easy cases: there is a fairly simple decision procedure, which yields at most one unifier. Miller introduced pattern unification in his logic programming language L_λ [4]. Isabelle uses pattern unification [5] to reduce the number of calls to Huet's procedure. One could base a proof assistant on pattern unification alone: Nipkow has found that it covers 97% of unifications in Isabelle.

The HOL family of provers [2] follows a strict philosophy intended to ensure soundness. In part, it demands that all proof steps should go through a trusted kernel of minimal size. Isabelle follows a similar philosophy (otherwise you get a tool in which users prove 0=1 every few months), but allows the kernel to contain a sophisticated unification algorithm.

The difficulty of combining higher-order syntax with logical variables reminds us that that distinct goals (flexibility and automation) may be hard to achieve simultaneously.

4 Search Primitives Based on Lazy Lists

Many automated reasoning systems employ search, but there are many different strategies. A programmer could hardly cope with a version of Prolog that used anything other than depth-first search. Stickel's PTTP [18] brought depth-first iterative deepening into fashion. Some Isabelle reasoning tactics use best-first search. Flexibility requires giving the user a choice of search strategies.

A *lazy list* is a list (possibly infinite) whose elements are computed upon demand rather than all at once. Lazy lists provide a uniform interface to an arbitrary search strategy. (Most functional programmers know this; for examples see my ML book [9, Chap. 5].) The idea is to implement your search strategy so that it yields up its results as a lazy list. The recipient of the list, just by looking through the elements, incrementally invokes the search strategy. Isabelle uses lazy lists to return, in a prescribed order, possibly infinitely many higher-order unifiers.

In Isabelle, proof strategies are called *tactics*. An Isabelle tactic can generate a lazy list of results. The simplest tactics consist of the application of a single rule to a goal. Users can combine tactics using simple control structures such as THEN and ORELSE. The latter tactical introduces choice, thereby defining a search space. They can explore this search space using primitives such as REPEAT, DEPTH_FIRST, BEST_FIRST, ITER_DEEPEN, etc. Nor is this menu fixed: Isabelle is coded in ML, and users can code their own search strategies. For example, Norbert Völker has implemented the A^* search strategy [17], which is a form of best-first search.

5 The Benefits

The applications of strategic design choices can't be predicted precisely. But it is obvious that a system that combines search strategies and unification will have some potential for automatic proof search. Add interaction and higher-order syntax, and you have an ideal test-bed for experimentation with different proof strategies.

In early work I focussed on intuitionistic logic, independently discovering Dyckhoff's techniques [1] (which he went on to prove complete). Turning to classical logic, it was not hard to build simple tableau provers. But such provers are practically useless, since users don't work in pure first-order logic: they use set theory and other complex notions of their own. Further experimentation led to Isabelle's *classical reasoner* [10]. Its tactics, such as `fast_tac`, achieve the necessary integration—users can give it any reasonable inference rules, represented declaratively as discussed in Sect. 3. The details do not concern us here; let me just stress that its heuristics are elementary by modern standards, and that its power comes from the combination of higher-order syntax, unification and search primitives.

Recently I have built a new tactic, `blast_tac` [11]. It finds proofs using its own generic tableau prover, which is coded directly in ML for speed. Because it cuts corners, it converts the resulting proof into tactics for verification using Isabelle's unification and search primitives. Perhaps one could add such a tactic to a proof assistant that lacked logical variables, but the job would be harder.

Though it cannot prove challenge problems that feature in theorem prover competitions, `blast_tac` is valuable against subgoals that arise in practice [6]. The majority of systems entered in such competitions could not even parse, let alone prove, these subgoals, since they typically involve complex variable binding. Often the proof involves concepts (such as inductively defined relations) that cannot be expressed in first-order logic. Consider this example, formulated in ZF set theory [8]:

$$C \neq \emptyset \to \bigcap_{x \in C} [A(x) \cap B(x)] = (\bigcap_{x \in C} A(x)) \cap (\bigcap_{x \in C} B(x))$$

Back in 1988, proving it required manually replacing $C \neq \emptyset$ by $\neg(C \subseteq \emptyset)$, then applying two repetitive tactics. Now, it takes `blast_tac` half a second. I suspect it is still a hard challenge for other set theory provers.

Isabelle supports higher-order logic as well as set theory, and the same classical reasoner works for both. Many set theory problems become easier to prove in higher-order logic, but perhaps they are still challenging. Some examples appear in my `blast_tac` paper [12].

I have also implemented model elimination (MESON) on Isabelle's proof engine. Compared with PTTP it is absurdly slow, but it crushes `blast_tac` on harder first-order challenge problems. Unfortunately, one seldom encounters pure predicate logic tasks that can be given to the MESON tactic. Because it is generic, `blast_tac` finds uses throughout the Isabelle case studies. This paradox illustrates the irrelevance of traditional challenge problems to real-world verification.

6 Conclusions

Isabelle's combination of features—higher-order syntax, logical variables and search primitives—is ideal for trying out new proof strategies. It is less ideal for producing high-performance tools. Implementing `blast_tac` required a direct resort to ML. The same thing happened with the simplifier: early versions ran as logic programs upon Isabelle's proof engine, but it too is now coded in ML for speed. We can expect most proof procedures to be expressed ultimately as algorithms and implemented as straight code.

Of course, the final implementation should retain the new ideas discovered during experimentation. Isabelle's simplifier has unusual features, such as congruence rules [13, Sect. 4.5]. These express contextual rewriting, for instance, given $A \wedge B$, to assume A to be true while rewriting B. Congruence rules were discovered back when the simplifier ran on Isabelle's proof engine.

Beginning users do not conduct research on automatic proof, but even they benefit from Isabelle's combination of features. Higher-order syntax is indispensable in any general-purpose specification language. Logical variables arise naturally, and can be left to be instantiated automatically. Even search can arise naturally. Giving Isabelle a list of rules makes it search for the first one that is applicable, and joining two tactics using THEN can cause a bit of search to find a route through both tactics. Over time, users learn to undertake more adventurous searches.

Acknowledgement. Isabelle is the outcome of many years' collaboration with Tobias Nipkow and his group at the Technical University of Munich. Isabelle has been supported by numerous grants, including the EPSRC grant GR/K57381 'Mechanizing Temporal Reasoning' and the ESPRIT working group 21900 'Types.' Tim Leonard made detailed suggestions on this paper; James Margetson and Norbert Völker also commented.

References

1. Dyckhoff, R., Contraction-free sequent calculi for intuitionistic logic, *Journal of Symbolic Logic* **57**, 3 (1992), 795–807

2. Gordon, M., From LCF to HOL: a short history, In Plotkin et al. [15], In press
3. Huet, G. P., A unification algorithm for typed λ-calculus, *Theoretical Comput. Sci.* **1** (1975), 27–57
4. Miller, D., A logic programming language with lambda-abstraction, function variables, and simple unification, *Journal of Logic and Computation* **1**, 4 (1991), 497–536
5. Nipkow, T., Functional unification of higher-order patterns, In *Eighth Annual Symposium on Logic in Computer Science* (1993), M. Vardi, Ed., IEEE Computer Society Press, pp. 64–74
6. Nipkow, T., Verified lexical analysis, In *Theorem Proving in Higher Order Logics: TPHOLs '98* (1998), J. Grundy M. Newey, Eds., LNCS, pp. ??–??, invited lecture
7. Owre, S., Rushby, J. M., Shankar, N., Srivas, M. K., A tutorial on using PVS for hardware verification, In *Theorem Provers in Circuit Design: Theory, Practice, and Experience* (1995), R. Kumar, Ed., LNCS 901, Springer, pp. 258–279
8. Paulson, L. C., Isabelle: The next seven hundred theorem provers (system abstract), In *9th International Conference on Automated Deduction* (1988), E. Lusk R. Overbeek, Eds., LNCS 310, Springer, pp. 772–773
9. Paulson, L. C., *ML for the Working Programmer*, 2nd ed., Cambridge University Press, 1996
10. Paulson, L. C., Generic automatic proof tools, In *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, R. Veroff, Ed. MIT Press, 1997, ch. 3
11. Paulson, L. C., A generic tableau prover and its integration with Isabelle, Tech. Rep. 441, Computer Laboratory, University of Cambridge, Jan. 1998
12. Paulson, L. C., A generic tableau prover and its integration with Isabelle (extended abstract), In *CADE-15 Workshop on Integration of Deduction Systems* (1998), in press
13. Paulson, L. C., Tool support for logics of programs, In *Mathematical Methods in Program Development: Summer School Marktoberdorf 1996*, M. Broy, Ed., NATO ASI Series F. Springer, Published 1997, pp. 461–498
14. Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *Journal of Automated Reasoning* **17**, 3 (Dec. 1996), 291–323
15. Plotkin, G., Stirling, C., Tofte, M., Eds., *Essays in Honor of Robin Milner*, MIT Press, 1998, In press
16. Quaife, A., Automated deduction in von Neumann-Bernays-Gödel set theory, *Journal of Automated Reasoning* **8**, 1 (1992), 91–147
17. Rich, E., Knight, K., *Artificial Intelligence*, 2nd ed., McGraw-Hill, 1991
18. Stickel, M. E., A Prolog technology theorem prover: Implementation by an extended Prolog compiler, *Journal of Automated Reasoning* **4**, 4 (1988), 353–380

Part II

Regular Contributions

Strategic Issues Related to the Development and Use of TPS

Peter B. Andrews

Department of Mathematical Sciences, Carnegie Mellon University,
Pittsburgh, PA 15213, U.S.A.
Peter.Andrews@cmu.edu
http://www.cs.cmu.edu/~andrews/tps.html

TPS [5] is an automated theorem proving system for classical type theory which has been under development at Carnegie Mellon University for many years.[1] The methods of proof search used by TPS are based on expansion trees and matings (using ideas described in [1] and [3]), but TPS proofs are presented in natural deduction style, and TPS can operate in a mixture of interactive and automatic modes [6].

Many strategic decisions were involved in the development of TPS, and many more are made (by setting various flags) when one uses TPS to search for a proof of a theorem. Other strategic questions arise when one considers how more diverse methods of proof search might be integrated with those already implemented in TPS. Relevant work [8] on integrating TPS into the ΩMEGA system [7] is already being done. It is also valuable to try to understand what the strategies used for other methods of proof search mean from the perspective of expansion proofs and matings.

Ultimately, we wish to understand how fundamental ideas about strategies manifest themselves in a variety of theorem-proving contexts.

References

1. Peter B. Andrews. Theorem Proving via General Matings. *Journal of the ACM*, 28:193–214, 1981.
2. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.
3. Peter B. Andrews. On Connections and Higher-Order Logic. *Journal of Automated Reasoning*, 5:257–291, 1989.
4. Peter B. Andrews and Matthew Bishop. On Sets, Types, Fixed Points, and Checkerboards. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Theorem Proving with Analytic Tableaux and Related Methods. 5th International Workshop, TABLEAUX '96, Terrasini, Italy, May 15-17, 1996*, Lecture Notes in Artificial Intelligence 1071, pages 1–15. Springer, 1996.
5. Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[1] Work on TPS is supported by the National Science Foundation under grant CCR-9624683.

6. Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: An Interactive and Automatic Tool for Proving Theorems of Type Theory. In *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop, HUG '93, Vancouver, B.C., Canada, August 11-13, 1993. Proceedings*, Lecture Notes in Computer Science 780, pages 366–370. Springer-Verlag, 1994.
7. C. Benzmuller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a Mathematical Assistant. In William McCune, editor, *Automated Deduction - CADE-14; 14th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 1249, pages 252–255. Springer, 1997.
8. Christoph Benzmuller and Volker Sorge. Integrating TPS with ΩMEGA. Technical report, Universitat des Saarlandes, 1998.
9. Matthew Bishop and Peter B. Andrews. Selectively Instantiating Definitions. In Claude and Helene Kirchner, editors, *Automated Deduction - CADE-15; 15th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 345–360. Springer, 1998.

Strategy analysis: from sequential to parallel strategies

(Position paper)

Maria Paola Bonacina ⋆

Dept. of Computer Science – University of Iowa – Iowa City, IA 52242-1419, USA
E-mail: bonacina@cs.uiowa.edu

The growth of methods for mechanical theorem proving poses the problem of evaluating their efficiency. The evaluation of an approach for the automated solution of a problem is made usually of at least two components: algorithm analysis (machine-independent), and performance evaluation (machine-dependent). In the tradition of theorem proving, only the second component has been available. Research in *strategy analysis* aims at reducing this hole, by providing mathematical tools for some machine-independent evaluation of theorem-proving strategies.

In earlier work [4] (also presented in [1]), an approach which begins to address issues in strategy evaluation was proposed. It comprises *a model for the representation of the search space and search process*, a notion of *complexity of search* in infinite spaces, and a measure of this complexity, termed *bounded search spaces*. These tools were applied to *contraction-based strategies*, that is, forward-reasoning strategies with eager contraction (those originated from the term-rewriting and Knuth-Bendix paradigm on one hand, the resolution-paramodulation paradigm on the other). Strategies of different contraction power were compared, showing that a strategy with higher contraction power causes a larger reduction of the bounded search spaces during the derivation. This was the first theoretical result to justify in analytical terms the advantage of contraction observed in experiments.

In recent work [2, 3], this methodology was extended from sequential search to *parallel search*. Parallel search means that deductive processes search in parallel the space of the problem: each executes a strategy, develops a derivation, builds a set of data, and communicates with the others; the parallel search succeeds as soon as one of the processes does. Approaches to parallel search differ in how they differentiate and combine the activities of the processes. In *distributed search*, the strategy *subdivides the search space* among the processes, with communication to

⋆ Supported in part by the National Science Foundation with grant CCR-97-01508.

preserve completeness: *distributed-search contraction-based strategies* are the object of the analysis.

In a strategy with distributed search the modelling problem becomes more complicated, because not only contraction, but also *subdivision* and *communication* modify the search space. Furthermore, many processes are active in parallel. In a *parallel marked search graph* [2], the structure of the graph represents the search space of all the possible inferences, while the *marking of vertices and arcs* (one per process) represents the search, including: selections by the search plan, deletions by contraction, communication (clauses may be received in addition to being generated), and subdivision (an arc may be *allowed* or *forbidden* to a process by the *subdivision function* of the search plan).

The next objective was to refine the complexity measure in such a way to capture both benefits (e.g., the subdivision) and costs (e.g., communication) of distributed search. A clause occurs in the bounded search space of bound j with multiplicity equal to the number of its *ancestor-graphs* whose *distance* is smaller or equal to j. Clauses made unreachable by contraction have infinite distance, and are excluded from all bounded search spaces. In [3], the notion of *ancestor-graph* is refined to distinguish between *allowed* and *forbidden* ancestor-graphs, in such a way to take both subdivision (which forbids ancestor-graphs) and communication (which may re-allow them) into account. The *bounded search spaces for process* p_k only count ancestor-graphs allowed to p_k, so that they measure both the advantage of subdivision and the cost of communication.

These tools were applied to identify and analyze issues in the parallelization of contraction-based strategies. It is not obvious that the parallelization of a contraction-based strategy is contraction-based, because of the interactions of eager contraction with subdivision and communication. Once sufficient conditions for a parallel strategy to be contraction-based had been identified, the impact of communication delays on eager contraction was analyzed: two patterns of behaviour – *late contraction* and *contraction undone* – where communication delays cause eager contraction to fail, were discovered. While in a sequential derivation the bounded search spaces decrease monotonically (expansion visits the search space, contraction prunes it), in a parallel derivation they may oscillate *non-monotonically*, because contraction and subdivision reduce them, whereas communication undoes some of the subdivision and causes failures of eager contraction. A strength of this analysis is to capture the cost of communication in terms of bounded search spaces, without the possibility to recur to the classical notion of time complexity. The next step was to consider all the parallel processes together, by introducing *parallel bounded search spaces*. If different processes are allowed to visit the same ancestor-graphs, there is an *overlap*, which may be caused by imperfect subdivision or communication. Sufficient conditions to minimize the overlap in the presence of local eager contraction were given.

The last phase was to compare a contraction-based strategy $\mathcal{C}$ with its parallelization $\mathcal{C}'$. The analysis of the evolution of their bounded search spaces shows that: all redundant clauses excluded by $\mathcal{C}$ are excluded also by $\mathcal{C}'$; if $\mathcal{C}'$ has instantaneous communication and minimizes the overlap, its parallel bounded search spaces are smaller than those of $\mathcal{C}$. This result is significant as a limit theorem, similar to other theoretical results obtained under an ideal assumption: it explains the nature of the problem, indicating in overlap and communication-contraction node its essential aspects, and it represents a limit that concrete strategies may approximate. Since instantaneous communication is necessary, it is also a negative result on the parallelizability of contraction-based strategies, which contributes to explain – analytically, not empirically – the difficulty with obtaining generalized improvements by parallelism in theorem proving.

References

1. M. P. Bonacina. Machine-independent evaluation of theorem-proving strategies (position paper). In B. Gramlich and H. Kirchner, editors, *Proc. of the Workshop on Strategies in Automated Deduction, CADE-14*, pages 37–39, 1997.
2. M. P. Bonacina. On the representation of parallel search in theorem proving (extended abstract). In *Proc. of FTP-97*, number 97-50 in Technical Report of RISC, pages 22–28. Johannes Kepler Universität, 1997.
3. M. P. Bonacina. Distributed contraction-based strategies: model and analysis. Technical report, Department of Computer Science, University of Iowa, April 1998.
4. M. P. Bonacina and J. Hsiang. On the modelling of search in theorem proving – towards a theory of strategy analysis. *Information and Computation*, forthcoming, 1998.

Using Rewriting and Strategies for Describing the B Predicate Prover (Extended Abstract)

Horatiu Cirstea and Claude Kirchner

LORIA & INRIA
615, rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
email: {Horatiu.Cirstea,Claude.Kirchner}@loria.fr

Abstract. The framework of computational systems has been already used for describing several computational logics. In this paper is described the way a propositional prover and a predicate prover are implemented in ELAN, the system developed in Nancy for describing and executing computational systems. The inference rules for the provers are described by conditional rewrite rules and their application is controlled by strategies. We show how different strategies using the same set of rewrite rules can yield different proof methods.

1 Introduction

We start from the idea that logic programming in a broad sense, and theorem proving can be uniformly described in a logical framework. The rewriting logic ([MOM96]), a particular case of general logics ([Mes92]), provide such a logical framework in which many other logics can be represented by describing the syntax of formulas, the set of axioms, the set of deduction rules and the proof calculus.

The representation of a theorem prover like the predicate prover in this formalism allows to describe in a simple and expressive way the inference rules of the prover allowing on one hand the user to better understand its behaviour and on the other hand to prove properties of the rules. Furthermore, by making the formalisation executable we allow either to directly use the prover or to replay proofs done by another one.

In order to describe a computational version of a certain logic we use computational systems that can express the proof calculus of the given logic. A computational system ([KKV95]) is a combination of a rewrite theory and a strategy describing the intended set of computations. Computational systems are simple to understand as they are based on rewriting logic and first-order tools.

These ideas are implemented in the language ELAN ([BKK^{+}96]) which allows to describe computational systems. In the environment provided by this language a logic can be specified by describing its syntax and inference rules. The inference rules of the logic are described by conditional rewrite rules possibly containing local assignments. The application of these rules is controlled by strategies at two levels: the first one consists of defining expressions built over the alphabet of rule labels and the second one consists of using strategies in the local assignments of rule definitions.

We use ELAN as a logical framework for implementing the propositional calculus and the first-order predicate calculus whose syntax and inference rules are presented by J.-R. Abrial in [Abr96,Abr97].

The implementation of these rules in ELAN is done in a very natural way, most of the rewrite rules being syntactical transformations of the initial inference rules. ELAN provides a powerful definition of strategies used to guide the application of the rewrite rules and a great flexibility in developing new strategies. The description of the strategy is conceptually easy and the structure of the denoted proof terms can be straightforwardly derived from the computations. Thus, proofs of conjectures in the specified logics can be obtained by analysing the trace (provided in the ELAN environment) of the computations performed during the deduction process.

The section 2 presents the basic notions concerning the rewriting logic and the computational systems. The way these ideas are implemented in ELAN is then briefly presented. Section 3 presents some representative rules of the propositional and predicate prover and the strategies that guide their application. The last section of the paper contains the conclusions and gives further perspectives for this work. [1]

2 General Setting

This section presents the main concepts of rewriting logic that is fully described in [Mes92,MOM96]. The computational systems are then defined as rewrite theories in rewriting logic together with strategies that guide the computations. This provides the semantical foundation of the ELAN system as an environment in which computational systems can be described and executed.

2.1 Rewriting Logic

We just briefly recall the basic notions that are consistent with [JK91,DJ90] to which the reader is referred for a more detailed presentation. The definitions below are given in the one-sorted case, the many-sorted and order-sorted cases can be given a similar treatment.

We consider a set $\mathcal{F} = \bigcup_n \mathcal{F}_n$ of ranked function symbols, where $\mathcal{F}_n$ is the subset of functions of arity n, a set $\mathcal{X}$ of variables and the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$

[1] The implementations of the propositional and predicate prover can be tried at http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/Applications/PropProver and http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/Applications/Prover

built on $\mathcal{F}$ using the variables in $\mathcal{X}$. $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$ denotes the free algebra of terms modulo E, where E is a set of equalities.

In rewriting logic, proofs are first-order objects and are represented by proof terms. In order to build proof terms the rules are labelled with a set $\mathcal{L}$ of ranked label symbols. A *proof term* is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$, function symbols from $\mathcal{F}$, label symbols from $\mathcal{L}$ and the concatenation operator ";". Thus, a proof term is an element of the algebra $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{;\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, \mathcal{X})/E)$.

A theory in this logic is a generalisation of the notion of theory from [Mes89], the axioms in the rewriting logic being identified by labels. A *labelled rewrite theory* is a 4-tuple $\mathcal{R}$=($\mathcal{F}$,E,$\mathcal{L}$,R) where $\mathcal{F}$ is a ranked alphabet of function symbols, E is the set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, $\mathcal{L}$ is the set of labels and R a set of labelled rewrite rules of the form l *: lhs* $\rightarrow$ *rhs* where $l \in \mathcal{L}$ and *lhs, rhs* $\in \mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying *Var(rhs)* $\subseteq$ *Var(lhs)*. The arity of l is the number of variables in *Var(lhs)*.

2.2 Computational Systems

The rewriting logic is proposed in [MOM96] as a logical framework in which other logics, like equational logic and Horn logic, can be represented and as a semantic framework for the specification of languages and systems. The operational semantics for rewrite theories are presented in [KKV95] where it is shown how computations in a rewrite theory can mirror computations in various logical systems.

The equivalence relation generated by E and the set of equational axioms (presented in [KKV95]) on the set of proof terms induces an equivalence on computations by considering that two computations are equivalent if their proof terms are equivalent.

A *strategy* is a subset of the set of proof terms $\mathcal{PT}$ of the current rewrite theory and is used to describe the computations one is interested in. The result of applying a strategy on a term is the set (possibly empty) of all terms that can be derived from the initial term using the strategy. We formally express this by giving a functional representation to the application of a strategy $\mathcal{S}$ on a term t:

$$\mathcal{S}(t) = \{[t']_E | \exists \pi \in \mathcal{S}, [t]_E \overset{\pi}{\Rightarrow} [t']_E\}$$

A *computational system* is defined as a labelled rewrite theory $\mathcal{R}$=($\mathcal{F}$,E,$\mathcal{L}$,R) together with a strategy $\mathcal{S}$. [KKV95] gives examples of languages and systems designed in this formalism.

2.3 The ELAN Environment

ELAN is a language for designing and executing computational systems. In ELAN, a logic can be expressed by specifying its syntax and its inference rules. The syntax can be described using mixfix operators and the inference rules are described by conditional rewrite rules. In order to guide the application of the rewrite rules strategies are introduced. A full description of the language and its implementation is given in [BKK98] and a survey of several examples that have been developed using ELAN is presented in [BKK+96].

All rewrite rules are working on equivalence classes induced by the set of equations E that in the case of ELAN is restricted to associativity and commutativity axioms, for the symbols defined to be associative-commutative.

A labelled rewrite rule in ELAN is defined as a pair of terms built on functional symbols and local variables and additionally it can be applied under some conditions and it can use some local assignments. Several rules may have the same label, the resulting ambiguities being handled by the strategies. The rule label is optional. In this case it is the responsibility of the designer to provide a confluent and terminating set of unlabelled rewrite rules.

The application of the labelled rewrite rules is controlled by user-defined strategies while the no-named rules are applied according to a default normalisation strategy. The normalisation strategy consists in applying no-named rules at any position of a term until the normal form is reached, this strategy being applied after each reduction produced by a labelled rewrite rule.

The application of a rewrite rule in ELAN can yield several results due to the equational(associative-commutative) matching and to the **where** clauses that can return as well several results. Applying consecutively two rules is represented by the concatenation operator "**;**" and the non-determinism is handled by two basic strategy operators: **dont care choose**(**dc**) that returns at most one result un-deterministically chosen from the set of possible results of the application of the rule and **dont know choose**(**dk**) that returns all the possible results. The strategy **repeat*** apply sub-strategies in a loop until none of them is applicable.

The syntactic definitions and the rewrite rules can be implemented in ELAN in a modular way. Each module represents a computational system that can be parameterised and combined with other modules in order to build the whole system

3 The Theorem Prover

Using the framework presented in the previous section we map the rules of the predicate prover described by J.-R. Abrial [Abr96,Abr97] in a computational system whose computations describe proofs in the initial proof calculus.

We just briefly describe the format of inference rules and the way these rules are used, a more detailed presentation can be found in the B-Book [Abr96].

The conjectures to be proved are described by sequents consisting of a number of hypotheses **H** and a goal **P** and they are represented in the form: $\mathbf{H} \vdash \mathbf{P}$.

We use the notation presented in [Abr97] for describing the inference rules:

	Antecedents	Consequent
RuleName	$S_1, S_2, ...S_n$	S

where *RuleName* is the name that uniquely identifies the rule and S_i are the antecedents that can be sequents of the form $\mathbf{H} \vdash \mathbf{P}$ or side-conditions written in English. Such a rule represents the fact that the proof of the consequent can be reduced to the proof of his antecedents. The rules that have no sequents in the antecedents represent axioms. A sequent can be proved by an axiom having it as consequent.

The theorem prover is described by a signature providing the syntax, an inference system defining the possible inferences and a search plan that selects at each step of the reduction the inference rules that can be applied. In our approach the inference rules are expressed using conditional rewrite rules and the search plan by a strategy that guides the application of the rewrite rules. Depending on the strategies used for applying the inference rules the reduction process can yield different proofs, i.e. different computations.

A sequent of the form $\mathbf{H} \vdash \mathbf{P}$ is proved to be valid when it is transformed using the inference rules in a set of valid sequents, this proof corresponding in ELAN in finding a derivation $(\mathbf{H} \; |{-} \; \mathbf{P}) \stackrel{*}{\Rightarrow} \mathbf{\$}$ (where **$** represents the valid sequent) using the rewrite rules guided by the specified strategy.

In the development of the theorem prover we can distinguish 3 steps: the implementation of a propositional prover, its extention to a predicate prover and finally the introduction of the notion of equality. Each of the provers extends the previous one by adding a certain number of syntactic elements and inference rules. Since the strategies used in the three cases are quite similar we insist on the rules and strategies used in the propositional prover, a more detailed description of its extensions is presented in [CK97].

3.1 The Propositional Prover

We present now the syntax and the inference rules of the first level of the prover - the propositional prover. It is also described the strategy used to apply these rules. Each of the described rules will be presented in the form proposed by J.-R. Abrial in [Abr97] and by its implementation in ELAN.

The inference rules of the prover are applied on sequents having the form **Seq** ::= **Pred** |− **Pred** with **Pred** a predicate in the form:

Pred ::= **Pred** <=> **Pred** | **Pred** | => **Pred** | **Pred** && **Pred** |
Pred or Pred | ^**Pred** | **Prop**

The operator "**::**" is provided for describing rules with several antecedents thus, the syntax of **Seq** is extended with the clause **Seq** ::= **Seq :: Seq**. Intuitively, the rule describing the behaviour of this operator specifies that a sequent of this form is valid if all its components are valid.

We should point out that the logical operators **or**, **&&**, <=> are declared in ELAN as being associative-commutative so the matching concerning these operators is done modulo associativity- commutativity. The constants **TRUE** and **FALSE** identify the true and respectively the false predicate and **Hvide** an empty set of predicates. This symbol is used in order to represent sequents of the form $\vdash$ **P** in our syntax, i.e. sequents with no hypotheses.

For each of the binary connectors (**AND, OR, EQV, IMP**) a collection of four inference rules is defined together with another two ones for **NOT** and additionally a set of axioms that allow to stop the reduction process is provided. A detailed presentation of these rules can be found in [Abr97] and [Abr96].

We start by presenting a rule with two antecedents, the rules with only one antecedent are handled in a similar way.

AND4	$H \vdash Q$ $H \vdash P$	$H \vdash P \wedge Q$

For the case where the rule contains more than one antecedent the reduction strategy (*dedstrat*) is applied recursively on the antecedents and the combination of the two results is provided as final result. The strategy used is presented later on in this section. The representation of the rule **AND4** in ELAN is (with `H, P, Q : Pred; S1, S2 : Seq`):

```
[AND4] H |-  (P && Q)         =>         S1 :: S2
                        where S1 := (dedstrat) H |- Q
                        where S2 := (dedstrat) H |- P   end
```

In the current implementation the strategy *dedstrat* used in the local assignments is the strategy applied on the initial sequent but we can use some other strategy built by the user for proving the antecedents.

The other rules for the conjunction, disjunction, implication and equivalence are represented in the same way except one rule for the implication that transforms the sequent to be proved in a sequent with an additional hypothesis:

IMP4	$H,P \vdash Q$	$H \vdash P \Rightarrow Q$

If we use a strategy where the **IMP4** rule is applied after none of the other rules is applicable then we can prove that only simple propositions can be added to the hypotheses, where by simple proposition we understand a proposition containing no binary logical connectors. In order to make this property explicit in our system we implement the **IMP4** rule by the following rules (with `Ps : Prop; H, Q : Pred`):

```
[IMP4a] H |-  (Ps |=> Q)      =>         H && Ps |- Q              end
[IMP4b] H |-  (^Ps |=> Q)     =>         H && ^Ps |- Q             end
```

By specifying that *Ps* is an object of **Prop** we prevent the rule to be applied on sequents of the form $\mathbf{H} \vdash \mathbf{P} \Rightarrow \mathbf{Q}$ when **P** is not a simple proposition. For example, if we consider the sequent $\mathbf{H} \vdash (\mathbf{A} \wedge \mathbf{B}) \Rightarrow (\mathbf{A} \wedge \mathbf{B})$ a strategy trying to apply the **IMP4** rule at the beginning would give as result the sequent $\mathbf{H} \wedge (\mathbf{A} \wedge \mathbf{B}) \vdash (\mathbf{A} \wedge \mathbf{B})$ that cannot be proved using the rules of the prover. Using the rules **IMP4a** and **IMP4b** we are sure that the hypotheses part consists of a conjunction of simple propositions independently of the strategy used.

Since in ELAN the no-named rules are efficiently implemented we can improve the performance of the prover by replacing the labelled rule that reflects that the predicates (`P && P`) and `P` are equivalent with a no-named rule that is used in the normalising strategy.

```
[] P && P    =>     P                 end
```

The number of rules applied in order to prove a sequent depends as well on the strategy used to guide the application of these rules.

Two strategies are proposed for applying the rules presented above. In the first approach the strategy is described by a loop in which at each step we try to apply one of the rules presented above in the following order: one of the rules **AND, OR, IMP1, IMP2, IMP3, EQV, NOT**, one of the rules **AXM**, the rule **IMP4** and the auxiliary rule **TB**.

```
[] dedstrat  =>
      repeat*( dc( SetAON, SetAXM, SetIMP, SetTB ) ); dc(FR)      end
```

The rule **FR** is applied at the end of the strategy in order to check if the result is a valid sequent. If this is not the case we backtrack to the last choice point in the strategy *repeat* and we try the next applicable set of rules. If none of the rules from this level leads to a valid sequent we backtrack again and we continue like this until we obtain a proof for the input sequent or when all the possibilities have been tried without success.

For reasons of modularity we group together the similar rules by using a strategy **dont care choose (dc)** that tries to apply each of the enumerated rules in the specified order. The set **AON** below is an example of using this method:

```
[] SetAON  =>    dc(AND1,AND2,AND3,AND4,OR1,OR2,OR3,OR4,
           IMP1,IMP2,IMP3,EQV1,EQV2,EQV3,EQV4,NOT1,NOT2)         end
```

A second approach consists in trying to apply the strategy **SetAON** repeatedly and as soon as none of the rules used by the strategy is applicable we start to do the same thing for the next strategy **SetAXM**. We proceed in the same manner for the rest of strategies and after **SetTB** is no more applicable we re-iterate the process from the beginning. The reduction process stops when none of the rules is applicable on the current sequent.

This strategy is represented in ELAN in the following way:

```
[] SetAON  => repeat*( dc(AND1,AND2,AND3,AND4,OR1,OR2,OR3,OR4,
                IMP1,IMP2,IMP3,EQV1,EQV2,EQV3,EQV4,NOT1,NOT2)) end
. . .
[] SetSEQ    =>  SetAON; SetAXM; SetIMP; SetTB                 end
[] dedstrat  =>  repeat*( dc(RuleSEQ) )                        end
```

The strategy *repeat*()* used to build the loop returns the same sequent as the one received as input if none of the rules tried inside it is applicable. Thus, we have to introduce a new rule **RuleSEQ** that verifies if anything new has been achieved by executing one time the loop (if the sequent received as input has been changed) and if not it returns the input without continuing the loop.

```
[RuleSEQ] H |- B              =>    S
              where S := (SetSEQ) H |- B
              if neq_Seq(S, H |- B)            end
```

The difference between the two approaches is that while for the first one, after a rule is applied successfully we try to use rules starting with the ones with high priority, the second strategy applies the rules from the same level of priority (rules grouped in a strategy Set*op_name*) until none of them is applicable and then continues with the rules from the next level.

The problem that can arise in the second case is that applying **IMP4a** or **IMP4b** sooner in the reduction process can lead to a longer proof path. We give a simple example that illustrates the behaviour of the two strategies. We use the sequent $\mathbf{Hvide} \vdash \mathbf{A} \Rightarrow (\mathbf{B} \Rightarrow \mathbf{A})$. The rules used for the first strategy are:

```
'IMP4a':        (Hvide && A) |- (C |=> A)
'AXM4':         $
```

and for the second one:

```
'IMP4a':        (Hvide && A) |- (C |=> A)
'IMP4a':        ((Hvide && A) && C) |- A
'AXM3':         $
```

The environment ELAN allows to get the trace of the computations executed and to obtain statistics about the application of the rewrite rules. We show this by providing the output obtained when we try to prove the sequent `(Hvide |- (A |=> (B |=> C)) |=> (A |=> B) |=> A |=> C`.

For the first strategy the following output is obtained using the ELAN interpretor:

```
[] start with term :
   Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
  $
 Statistics:
 total time     (0.176+0.000)=0.176 sec (main+subprocesses)
 average speed  159 inf/sec
                0 nonamed rules applied,     0 tried
               28   named rules applied,   913 tried
 named rules
          applied   tried    rule for symbol
                4      13    TB:Seq
                9       9    FR:Seq
                4      25    IMP3:Seq
                2      19    IMP4a:Seq
                4      17    IMP4b:Seq
                2      21    AXM1:Seq
                1      26    AXM5:Seq
                2      28    AXM4b:Seq
```

The application of the second strategy on the same sequent yields the following result:

```
[] start with term :
   Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
  $
```

```
 Statistics:
 total time     (0.360+0.000)=0.360 sec (main+subprocesses)
 average speed  136 inf/sec
               0 nonamed rules applied,     0 tried
              49   named rules applied,   725 tried
 named rules
          applied   tried    rule for symbol
                4      19    TB:Seq
                9       9    FR:Seq
                4      19    IMP3:Seq
                1      16    AXM4:Seq
                3      33    STOP:Seq
                5      24    IMP4a:Seq
                4      19    IMP4b:Seq
                1      19    AXM1:Seq
                2      18    AXM2:Seq
                1      16    AXM5:Seq
               15      24    RuleSEQab:Seq
```

We notice that the main difference between the two consists in the number of rules **IMP4a** applied, the rule **RuleSEQ** consisting practically in a test.

Another possibility for guiding the rewrite rules would be to use the implicit normalising strategy, in this case the rules have to be unlabelled. The advantage of this approach is the efficient implementation of the unlabelled rules in ELAN but a great disadvantage is the lack of control in applying the rewrite rules. The statistics for such a strategy on the same input sequent are:

```
[] start with term :
    Hvide|-((A|=>(B|=>C))|=>((A|=>B)|=>(A|=>C)))
[] result term:
   $
 Statistics:
 total time     (0.052+0.000)=0.052 sec (main+subprocesses)
 average speed  596 inf/sec
              31 nonamed rules applied,   381 tried
               0   named rules applied,     0 tried
```

This normalisation strategy is not appropriate for the predicate prover where multiple instantiations for universally quantified predicates should be obtained by strategies that returns all possible results.

We should point out that not only the way the strategy operators are applied on the deductions rules is important but the order of these rules in the guiding strategy can be crucial in some cases. For instance a strategy that would try to apply the axioms only when none of the other main rules is applicable:

```
[] dedstrat  =>
      repeat*( dc( SetAON, SetIMP, SetAXM, SetTB ) ); dc(FR)      end
```

yields the use of 42 rules for the first strategy and 44 for the second one.

We can obtain all the possible proofs for a sequent by replacing the **dont care choose**(**dc**) operator with a **dont know choose**(**dk**) one that generates all the possible matchings (due especially to associative-commutative operators) of rewrite rule on the input sequent and applies the rules in all the possible orders. For example, the rule **AND4** applied on $\mathbf{H} \vdash \mathbf{A} \wedge \mathbf{B} \wedge \mathbf{C}$ matches **P** on **A** and **Q** on $\mathbf{B} \wedge \mathbf{C}$ or **P** on $\mathbf{B} \wedge \mathbf{C}$ and **Q** on **A** and so on.

3.2 The Predicate Prover

The propositional prover is simple but in the same time very restrictive since we cannot reason about quantified predicates.

The syntax of the prover is extended with quantified predicates and we add a set of inference rules that transform the structure of the sequents received as input until they are proved or they are in the form **(H $\vdash$ FALSE)** and a rule (called **INS**) that tries to instantiate the quantified variables from sequents of this form with proper values in order to discover contradictions in the hypotheses and thus, to prove the respective sequents. We just present the rule **INS** and the way it is applied, the other rules are used in a similar way as the ones of the propositional prover.

In the **INS** rule we are looking for instantiations that transform the quantified predicates in simple predicates (or at least in predicates with a smaller number of quantified variables) that can lead further on to contradictions.

INS	Determine the instantiations P_1,P_2,...,P_n H $\vdash$ $P_1 \Rightarrow (P_2 \Rightarrow ...(P_n \Rightarrow$ FALSE$)...)$	H $\vdash$ FALSE

The search of proper instantiations is done by matching the universally quantified predicates on the already instantiated predicates from the hypotheses. Since for the matching problem we use a combined matching algorithm that combines algorithms for empty, commutative and associative-commutative theories we can obtain more than one result, each of them corresponding to different solutions of the matching algorithm.

In order to explore all the solutions of the matching algorithm we have to use a **dont know choose** strategy that allows to backtrack to a new instantiation if the application of the strategy on the previous one has failed.

```
[] SetINS  =>  dk(INS)        end
```

The main strategy looks similar to the one for the propositional prover and is containing the sets of rules for dealing with quantified predicates.

```
[] dedstrat  =>
     repeat*( dc( SetALLXST, SetAON, SetAXM, SetIMP, SetINS, SetTB ) );
     dc(FR)                                                           end
```

This strategy works like described in section 3.1 only that for the rule **INS** all its results will be considered when backtracking at its level.

4 Conclusions

We have shown how computational systems can be used as a logical framework for representing the first-order predicate prover proposed by J.-R. Abrial. The inference rules describing the two logics are naturally represented by conditional rewrite rules, the mapping of an inference rule from the form proposed by J.-R. Abrial in a rewrite rule in the system ELAN being in most of the cases a syntactical transformation.

Due to the semantics of ELAN that is based on many-sorted rewriting logic and to the possibility to define associative-commutative operators, we have been able to represent rules in a fairly intuitive way. By assigning types to the variables involved in the rewrite rules we have implicitly specified conditions on the structure of the terms that can be transformed by each rule thus, taking advantage of the well known benefit of typing objects. The definition of some operators as being associative-commutative has permitted to eliminate inference rules simulating these properties in the approach proposed by J.-R. Abrial. Some additional no-named rules have been introduced in order to improve the efficiency of the prover.

The application of all labelled rewrite rules is guided by strategies. Two approaches for defining them have been proposed and the results yielded by the two strategies have been compared. The effects of changing the order of the application of the rules have been presented and the use of an implicit normalising strategy has been considered. Although the differences between the proofs obtained with the different strategies for a simple sequent are not significant, we have to point out that small changes in the guiding strategy can influence significantly the efficiency of the computational system in the case of more complicated conjectures.

One of the advantages of representing the prover in rewriting logic is that we are then able to use standard rewrite tools for proving properties of the system like the termination for the propositional prover ([CK97]).

Among the perspectives open by this work, one concerns the implementation of interfaces for integers as well as for sets. The current syntax of the prover can be easily extended to include new operators on integers and sets and new inference rules describing their properties have to be introduced. Proper strategies should be designed for guiding the application of these new rules.

Acknowledgements We sincerely thank Thierry Lecompte for helpful discussions on the Atelier B predicate prover and Peter Borovanský and Pierre-Etienne Moreau for their useful hints and suggestions concerning the implementation in ELAN .

References

[Abr96] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996. ISBN 0-521-49619-5.

[Abr97] J.-R. Abrial. Le prouveur de prédicat. Technical report, 1997.

[BKK⁺96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[BKK98] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[CK97] Horatiu Cirstea and Claude Kirchner. Theorem proving using computational systems: The case of the B predicate prover. In *Workshop CCL'97*, Schlob Dagstuhl, Germany, September 1997.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. 1990. Also as: Research report 478, LRI.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. Cambridge (MA, USA), 1991.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. 1995.

[Mes89] J. Meseguer. General logics. In *Proc. Logic Colloquium '87*. North Holland, 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MOM96] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

Actions and plans in ELAN

Hubert Dubois and Hélène Kirchner
LORIA-UHP & CNRS
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France

Abstract. ELAN is a declarative language based on rewriting logic. The ELAN language is based on labelled conditional rewrite rules and on strategies for controlling their application. ELAN provides a strategy language to control labelled rules. In this paper we show how to use the ELAN strategy language for planning. We describe how to encode situations and actions and take advantage of ELAN strategies to build plans. We give an example of our approach by describing an elevator controller.

1 Introduction

Among numerous contributions in the area of logic and changes, rewriting logic has been proposed in [MOM94] as a logic of concurrent action and change that solves the frame problem and unifies a number of previous logics of change including linear logic and Horn logic with equality. ELAN is a declarative language based on rewriting logic [Mes92] and its logical foundations are detailed in [KKV95,BKK96].
In ELAN a rewrite rule may be labelled, may have conditions and may introduce local variables useful to compute intermediate results. A computation may have several results which introduces the need to deal with non-determinism and backtracking.
One of the main originality of the language is to provide strategy constructions to specify whether a function call returns several, at least one or at most one result. This declarative handling of non-determinism is part of a strategy language allowing the programmer to specify the control on rules application. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, more complex strategies can be expressed. In addition the user can introduce new strategy operators and define them by rewrite rules. Evaluation of strategy application is itself based on rewriting.
ELAN has been used as an environment for specifying and prototyping constraint solvers, theorem provers and deduction systems in general. It also provides a framework for experimenting their combination.
The current version of ELAN includes an interpreter and a compiler written in C++ and Java, a library of standard ELAN modules, a user manual and examples of applications. Among those, let us mention for

instance the design of rules and strategies for constraint satisfaction problems [Cas97], theorem proving tools in first-order logic with equality [KM95,CK97], the combination of unification algorithms and of decision procedures in various equational theories [Rin97]. More information on the system can be found on the WEB site[1].

We open in this yet prospective paper a new application area for the ELAN language. We want to experiment its ability to model planning problems. A correspondence is drawn between actions and strategies, where primitive actions correspond to primal strategy (i.e. labelled rewrite rules) in ELAN, and where more complex actions are expressed using the full power of the strategy language. Then plans are a specific form of proof terms generated by the execution of a complex action on an initial situation. We mention in conclusion a few points for further research perspectives.

2 ELAN rules and strategies

An ELAN program describes a set of operators with possibly structural axioms (like associativity and commutativity properties), a set of conditional rewrite rules that can be named by labels, a set of strategy operators and a set of strategy rules.

In ELAN, rules are labelled conditional rewrite rules with local variable assignments

$$[\mathsf{lab}] : l \Rightarrow r \textbf{ if } v \textbf{ where } y := (S)u$$

where lab is the label, l and r the respective left and right-hand sides, v the condition and $y := (S)u$ a local assignment, giving to the local variable y the results of the strategy S applied to the term u. Any sequence of **where** and **if** is allowed but their order is relevant for the evaluation. For applying such a rule on a term t, say at top position, first l is matched against t, then the expressions introduced by **where** and **if** are instantiated with the matching substitution and evaluated in order. Instantiations of local variables (such as y) after **where** extend the matching substitution. When every condition is satisfied, the replacement by the instantiated right-hand side is performed.

Unlabelled rules are applied with a leftmost-innermost strategy by the interpreter. On the contrary, labelled rules are used in user-defined strategies and applied at top position in the term to be reduced. Instantiations of local variables after **where** also invoke ELAN strategies, built from a few constructors. Let us explain now how to express a strategy.

- A labelled rule is a primal strategy. The result of applying a rule labelled lab on a term t returns a set of terms.
- **first-one**$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns its first result. This strategy is said deterministic since it returns at most one result.
- **first**$(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns all its results. This strategy is said non-deterministic since it may fail or return more than one one result.

[1] http://www.loria.fr/equipes/protheo/PROJECTS/ELAN/elan.html.

- $\mathbf{dk}(S_1, \ldots, S_n)$ chooses all strategies given in the list of arguments and for each of them returns all its results. This strategy is again non-deterministic.
- $S_1; S_2$ is the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails. Its results are all results of S_1 on which S_2 is applied and gives some results.
- **id** is the identity that does nothing but does not fail.
- **fail** is the strategy that always fails.

The strategy language and its semantics are described in [BKK98], thanks to an application operator

$$[\_](\_) : (Strategy, Term) \quad SetOfTerms$$

whose interpretation is given by labelled rewrite rules.
ELAN also gives the possibility to the user to define recursive and parameterised strategies with rewrite rules [BKK96,BKK97]. Given a set of strategy symbols, rewrite rules on user-defined strategies are of the form $[\mathsf{lab}]\ S_1 \Rightarrow S_2$ where lab is a label, S_1 and S_2 are strategy terms built from all previously introduced strategy symbols. Such strategy rewrite rules are called *implicit* since they do not involve explicitly the application operator @. However, it is sometimes useful to express a strategy rule depending on the argument on which it is applied. Such rules are also allowed and are called *explicit* strategy rules. They are of the form:

$$[\mathsf{lab}][S_1](t) \Rightarrow t'$$

where S_1 is a strategy term as before, t is a term and t' is built on function and strategy symbols and possibly the strategy application operator. From the evaluation point of view, these rules are just added to the strategy interpreter.

3 Situations, actions and changes

In order to model action and change, we use a formalism very similar to the situation calculus. This first-order language is devoted to represent dynamically changing worlds. First introduced at the end of the 60' by Hayes and McCarthy in [MH69], the situation calculus was rediscovered in the earlier 90' in some recent works like [GLR91].
A *situation* is a first-order term representing a possible world history, which records changes to the world resulting from actions. Here a situation is defined by a pair of a state and a sequence of *primitive actions*. The initial situation S_0 is characterised by an initial state and an empty sequence of actions. The situation calculus introduces a reserved binary function symbol do, that builds from an action α and a situation s, a new situation $do(\alpha, s)$. For instance, consider an elevator controller. For the action $\alpha = up(5)$ (we want to go up to floor 5), $do(up(5), s)$ is the situation resulting from the application of $up(5)$ on the situation s.
A world dynamically changes during the execution. For every action, some preconditions characterise when this action can be performed. For instance, we only can move a block c from the place A to the place B if we

carry the block c. This precondition may depend on the current situation. So-called *fluents* are used to test or evaluate changes. Relational fluents return a boolean value, while others are called functional fluents.
For example, the relational fluent *is_carrying(robot,p,s)* asks if in the situation s the robot *robot* is carrying the object p; the results can be *true* or *false*. The functional fluent *location(robot,s)* indicates the location of the robot *robot* in the situation s; the results are not boolean.
The situation calculus introduces *frame axioms* in order to specify the unchanged fluents, during the application of a given action. The problem of defining these axioms is the huge number of them, although an action only changes a few of them. A solution to the frame problem has been described by Levesque and al. in [LRL$^+$97] using a single axiom called *successor state axiom*.
Therefore a problem in the situation calculus is described by a set of axioms to define the initial situation, a successor state axiom for every fluent, axioms describing the primitives actions and a set of preconditions for every action.
Beyond primitives actions, more complex actions such as procedures, conditional actions, loops can be expressed. Such extensions are provided for instance in the language GOLOG [LRL$^+$97]. So, an action is either a primitive action as before, or a complex action of the form:

- $[\mathbf{a_1}, ..., \mathbf{a_n}]$ for the sequence of actions $a_1, ..., a_n$.
- $?(\mathbf{p})$, where p is a *condition*. We call condition a relational fluent or an expression like $And(p_1, p_2)$, $Or(p_1, p_2)$, $Not(p)$ or $Some(v, p)$ where p, p_1 and p_2 are conditions and v is a variable that occurs in p and which is instantiated by a value taken from a set computed by a functional fluent. The action $?(\mathbf{p})$ stands for the test of the condition p.
- $\mathbf{a_1}\#\mathbf{a_2}$ for the non-deterministic choice between two actions a_1 and a_2.
- $\mathbf{if}(\mathbf{p}, \mathbf{a_1}, \mathbf{a_2})$ is a test of the condition p. If this test leads to *true*, then the action a_1 is performed, otherwise, a_2 is executed.
- $\mathbf{star}(\mathbf{a})$ repeats the execution of the action a ad infinitum.
- $\mathbf{while}(\mathbf{p}, \mathbf{a})$ executes the action a as long as the condition p is *true*.
- $\mathbf{pi}(\mathbf{v}, \mathbf{a})$, where v is a variable and a an action using v, assigns a value taken from a given set to the variable v and then instantiates this value by the action a.
- a procedure call, where a procedure is denoted by $\mathbf{proc}(\mathbf{name}, \mathbf{body})$ where *name* is the identifier associated to the procedure name and the body of the procedure is the second argument of *proc*.

4 ELAN, actions and plans

When looking at the language of actions, it is natural to draw an analogy between primitive actions and primal strategy: applying an action is like applying in ELAN a labelled rewrite rule on a term representing a state (or situation). More complex actions need the full expressivity of the strategy language.
In a situation calculus application, thanks to the predicate *do*, the primitive actions are memorised in a situation. For instance, the situation

do(close,do(open,do(turnoff(5),do(up(5),S_0)))) indicates that, from the initial situation S_0, we went up to the floor 5, then we removed the floor 5 from the list of floors, opened the doors and finally close them. A *plan* is a succession of actions applied to the initial situation.
In a similar way, in ELAN at every step of rewriting, it is possible to record the label of the applied strategy. This correspond to the notion of proof term in rewriting logic initially defined in [MOM93] and extended to built-in strategies of ELAN in [BKK97].
Then plans are a specific form of proof terms generated by the execution of a complex action on an initial state. These proof terms are obtained by concatenation of primitive actions. Since in general we do not want to keep all the information involved in a rewriting step (i.e. the rule, the application position, the substitution), we use a restricted notion of proof terms that only memorises the names of primitive actions and their parameters.
Our ELAN term to reduce (the request) is then a pair composed of an object designing the current state and a proof term corresponding to the actions that have been executed to reach this state.

4.1 Specification of the situation

Let us consider the example of an elevator controller. The problem can be described by two components: first, the list of all floors to be served, and second, the current floor. In ELAN, we introduce a structure

```
<Liste-of-floors: L & Current-floor: CF>
```

with two attributes, list of floors and current floor, and a constructor `<@ & @>` which takes two arguments of respective sorts `list[int]` and `int`, and builds a result of sort `state`.

```
sorts int, list[int];
<@ & @> : (Liste-of-floors:list[int] Current-floor:int) state
```

This automatically provides two selectors:

```
@.Liste-of-floors : (state) list[int]
@.Current-floor   : (state) int
```

and some rules to extract and replace the attributes:

```
<L & C>.Liste-of-floors => L
<L & C>.Current-floor   => C
<L & C>[.Liste-of-floors <- L1] => <L1 & C>
<L & C>[.Current-floor <- C1]   => <L & C1>
```

Once the state is defined, we define a situation by a pair made of a state and a strategy term to represent the sequence of actions that have been performed to reach this state. We use the notation `s * P` to denote this pair. P is of sort `proof` standing for `state -> state` which is the sort of the strategies on the sort `state`. A term of the form `<L & CF> * P` is of sort `situation`.
The initial situation is given by the user at the beginning of the rewrite process. It describes the initial values of the list `L` and the current floor `CF`. For instance, if we start on floor 4 and the 3 and 5 are switched on, the initial state is `<3,5,nil & 4>`, and the initial sequence of actions is the empty list `nil`. The initial situation corresponds to the term `<3,5,nil & 4> * nil`.

4.2 The tests

In order to test if some condition is satisfied in a given state, a few rules are added. These rules do not modify the state when they apply, but fail when the test is not true.
In the elevator controller, we define `?(on(n))` and `?(current_floor(n))`, where `n` is an integer, for a given state `<L & CF>`. `?(on(n))` checks whether the floor `n` occurs in the list `L` of the floors not yet served; `?(current_floor(n))` checks whether the current floor `CF` is equal to `n`.

```
strategies for state
        n,CF    : int;
        L       : list[int];
explicit
[] [?(on(n))] <L & CF>                => <L & CF>
                if belongs(n,L)
end
[.] [?(current_floor(CF))] <L & CF>   => <L & CF>
end
end
```

Other rewrite rules model functional fluents depending on the current state. In our example, there are two functional fluents defined by rewriting rules: `next_floor` and `Set_Values`.

- `next_floor` computes which floor will be served next by considering the current state.
- `Set_Values` also depends on the current state. This set is composed of all possible values for variables that have to be instantiated. For each state, `Set_Values` is the union of the list of floors to be served and the current floor.

More complex tests can use connectors like `And`, `Or`, `Not` or `Some`. Such compound conditions are decomposed into primitive ones, using again rewrite rules, as shown below. These strategy definitions are given in a module parameterised by a general sort `X`, which introduces sorts `atom` and `condition`. They are applied with a pre-defined strategy in the strategy interpreter, which is denoted by the label `[.]`.

```
strategies for X
        x,y             : atom;
        T               : X;
        C,C1,C2         : condition;
explicit
[.] [?(Some(x,C))] T    => [?(sub_C(x,y,C))] T
        where y := (listExtract) elem(Set_Values(T))
end
implicit
[.] ?(And(C1,C2))        => ?(C1) ; ?(C2)
end
[.] ?(Or(C1,C2))         => ?(C1) # ?(C2)
end
[.] ?(Not(C))            => first(?(C) ; fail , id)
end
end
```

4.3 The actions

The strategy language is used to define primitive actions including their preconditions directly thanks to the `if` construction in the rule definition of a strategy in ELAN.
The strategy operators are the names of the primitive actions. The signature of strategies corresponding to primitive actions in our elevator controller example is given below. Note that strategies may have a parameter (`up` for instance).

```
stratop global
        turnoff(@)              : (int) <state>;
        open                    : <state>;
        close                   : <state>;
        up(@)                   : (int) <state>;
        down(@)                 : (int) <state>;
end
```

The possible primitive actions are the opening/closure of the doors by `open` and `close`, the fact of going up/down to a floor `N` by the actions `up(N)` and `down(N)` and the removal of a floor `N` from the list of floors by `turnoff(N)`. The rewrite semantics of these actions is given by a set of rewrite rules given below. Notice that two primitive actions have preconditions, namely `up` and `down`: we can only go up if the current floor is below the next floor (and the precondition is the opposite for `down`). Note that each application of a primitive action generates the concatenation of the corresponding strategy in the proof term.

```
strategies for state
        N,CF    : int;
        L,L1    : list[int];
        P       : list[proof];
explicit
[.] [turnoff(N)] <L & CF> * P => <remove(N,L) & CF> * turnoff(N),P
end
[.] [open] <L & CF> * P       => <L & CF> * open,P
end
[.] [close] <L & CF> * P      => <L & CF> * close,P
end
[.] [up(N)] <L & CF> * P      => <L & N> * up(N),P
        if CF < N
end
[.] [down(N)] <L & CF> * P    => <L & N> * down(N),P
        if CF > N
end
end
```

More complex action definitions use the full strategy language of ELAN. For instance, we easily define by simple rules on strategies, the non-deterministic choice between two actions `S1#S2`, the conditional `if(p,S1,S2)`, the loop `star(S)`, the iteration `while(p,S)` of `S` until condition `p` is false, the assignment `pi(v,S)` of a variable `v` that occurs in a strategy `S`.

```
strategies for X
        x,y             : atom;
        T               : X;
        C               : condition;
        S,S1,S2         : <X->X>;

explicit
[.] [pi(x,S)] T         => [sub_S(x,y,S)] T
        where y := (listExtract) elem(Set_Values(T))
end

implicit
[.] S1 # S2              => dk(S1,S2)
end
[.] if(C,S1,S2)          => first(?(C) ; S1 , S2)
end
[.] star(S)              => first(S ; star(S) , id)
end
[.] while(C,S)           => star(?(C) ; S)
end
end
```

The procedure call can be expressed by an implicit or explicit strategy rule of one of the following forms:

```
[.]  proc_name    => proc
end
[.] [proc_name]s  => s'
end
```

For instance, in the example of the elevator controller, we give the next procedures, among which `control` is the main one.

```
explicit
[.] [serve_a_floor] s   => s1
               where s1 := [serve(next_floor(s))] s
end

implicit
[.] go_floor(N) => ?(current_floor(N)) # up(N) # down(N)
end
[.] serve(N)    => go_floor(N) ; turnoff(N) ; open ; close
end
[.] park        => if(current_floor(0) , open , down(0) ; open)
end
[.] control     => while(Some(v,on(v)),serve_a_floor) ; park
end
end
```

4.4 The evaluation process

To execute this program, we give a request of the form `[control] S0` where `S0` is the initial term, for instance `[control] <3,5,nil & 4> * nil`. We get the following result:

```
<nil&0>*open,down(0),close,open,turnoff(3),down(3),close,open,
turnoff(5),up(5),nil
```

where the plan must be read from right to left.

5 Conclusion

This research is yet at a very preliminary stage. Our ultimate goal is to take advantage of this formulation of planning in rewriting logic to attack some planning and scheduling problems. Detecting inconsistent states, non-terminating or ambiguous plans should be possible in this formalisation. In addition it may be worth emphasising that, thanks to the concurrent nature of rewriting logic, plans are endowed with a notion of parallelism and we thus get for free a concurrent semantics for the execution of plans.

References

[BKK96] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.

[BKK97] P. Borovansky, C. Kirchner, and H. Kirchner. Strategies and rewriting in ELAN. In *Proc. of the CADE-14 workshop: Strategies in Automated Deduction*, Townsville, Australia, 1997. Report CRIN 97-R-126.

[BKK98] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[Cas97] Carlos Castro. Constraint Manipulation using Rewrite Rules and Strategies. In Alice Drewery, Geert-Jan M. Kruijff, and Richard Zuber, editors, *Proceedings of the Second ESSLLI Student Session, 9th European Summer School in Logic, Language and Information, ESSLLI'97*, pages 45–56, Aix-en-Provence, France, August 1997.

[CK97] Horatiu Cirstea and Claude Kirchner. *Theorem proving using computational systems: The case of the B Predicate prover.* http://www.loria.fr/˜cirstea/Papers/PredicateProver.ps, 1997.

[GLR91] M. Gelfond, V. Lifschitz, and A. Rabinov. What Are the Limitations of the Situation Calculus ? In R.S. Boyer, editor, *Automated Reasoning. Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, chapter 8, pages 167–179. Kluwer Academic Publishers, Dordrecht, 1991.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[LRL+97] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG : A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MH69] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[MOM93] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[MOM94] Narciso Marti-Oliet and Jose Meseguer. Action and change in rewriting logic. Technical Report SRI-CSL-94-07, SRI, April 1994.

[Rin97] Christophe Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

An agent architecture that combines backward and forward reasoning

R.A. Kowalski, F. Sadri, F.Toni
Imperial College of Science, Technology and Medicine,
Department of Computing
180 Queen's Gate, London SW7 2BZ, UK
{rak,fs,ft}@doc.ic.ac.uk

Abstract. We present a proof procedure that combines backward reasoning with logic programs and forward reasoning with integrity constraints. We illustrate the use of the proof procedure as the inference engine underlying agents. Roughly speaking, the backward reasoning component of the proof procedure is responsible for the deliberative behaviour of agents, whereas the forward reasoning component is responsible for their reactive behaviour.

1 Introduction

Recent years have seen a shift in the treatment of agents in artificial intelligence. In conventional approaches, intelligent agents reason and plan within a closed environment, producing a complete plan before they start executing it. In more recent, and arguably more realistic approaches, intelligent agents reason and plan while reacting to and interacting with the external environment (e.g., see [15]). The agent architecture of [6–8] combines rational, deliberative behaviour and reactivity to the external environment. At the heart of this approach lies a proof procedure, which is executed within an observe-think-act cycle, and is responsible for both the thinking and the reacting components. In this paper we present the proof procedure and discuss the benefits of the combination of forward and backward reasoning that it allows. The proof procedure we present is a version of the procedure of [10], which, in turn, is strongly related to the procedures of [4, 2, 3].

The proof procedures of [4, 2, 3] compute "abductive explanations" for given queries, understood as "observations", with respect to an underlying "abductive logic program". An *abductive logic program* consists of a logic program, a pool of candidate *hypotheses*, i.e. atoms whose predicates (called the *abducible* predicates) are amongst the undefined predicates in the program, and a set of *integrity constraints*, i.e. sentences that must be satisfied by every chosen set of hypotheses in the given pool. *(Abductive) explanations* for queries (or observations) are sets of hypotheses Δ such that the logic program together with Δ "entails" the given queries, and such that the logic program together with Δ "satisfies" the integrity constraints. Various notions of entailment and satisfaction can be adopted (see [5]).

The proof procedure of [10] (henceforth referred to as SLP) is a generalisation of [4]. SLPcan be used to compute explanations for observations in abductive logic programming as well as to optimise queries in deductive databases (*semantic query optimisation*) and to compute answers to queries in *constraint logic programming.* SLP uses logic programs written as sets of iff-definitions (rather than if-clauses as in conventional logic programming) and integrity constraints written as (closed) implications. Iff-definitions are used *backward* to *unfold* atoms in queries replacing the atoms by their definition. Integrity constraints are used *forward* to *propagate* atoms in queries, adding additional (reduced) integrity constraints to the queries. The proof procedure employs additional operations to manipulate equality and to perform logical simplification. Operations are seen as rewrite rule and are used to reduce a given query to a disjunction of answers. Answers consist of conjunctions of *suspended atoms*, i.e. atoms whose definition is not available (hypotheses) or which have multiple definitions. Therefore, suspension is used to control (explosive) non-determinism. For example, SLP assumes iff-definitions for the predicate $>$ such that an atom $X > 10$ is suspended and returned as part of an answer, rather than being replaced by (infinitely many) answers $X = 11$, $X = 12$, $\ldots$.

The remainder of the paper is organised as follows. Section 2 defines some preliminary notions. Section 3 gives the proof procedure. Section 4 revises the agent architecture of [6–8]. Section 5 gives examples of the use of the proof procedure within the chosen agent architecture. Section 6 concludes and discusses future work.

2 Preliminaries

An (if-)**clause** is a sentence of the form

$A \leftarrow L_1, \ldots, L_n,$

with A atom, $L_1, \ldots, L_n$ (positive or negative) literals, and $n \geq 0$. All variables in A, $L_1, \ldots, L_n$ are implicitly universally quantified, with scope the entire clause. If $A = p(t)$, for some vector of terms t, the clause is said to *define* p.

A **logic program** is a set of clauses.

An **iff-definition** is a sentence of the form

$A \leftrightarrow D_1 \vee \ldots \vee D_k$

with A atom, $D_1, \ldots, D_k$ conjunctions of (positive or negative) literals, and $k \geq 0$. All variables in A are implicitly universally quantified, with scope the entire iff-definition. All variables in $D_1, \ldots, D_k$ but not in A are implicitly existentially quantified, with scope the disjunct in which they occur. A is called the *head* and $D_1, \ldots, D_k$ the *body* of the iff-definition. If $A = p(t)$, for some vector of terms t, $A \leftrightarrow D_1 \vee \ldots \vee D_k$ is said to *define* p.

An **(iff-)theory** is a set of iff-definitions. The **completion**, $Comp(P)$ [1] of a logic program P consists of Clark's equality theory (CET) [1], together with an iff-theory consisting of exactly one iff-definition for every predicate different from $=$. Any such iff-definition is of the form

$p(X) \leftrightarrow [X = t_1 \wedge D_1] \vee \ldots \vee [X = t_k \wedge D_k]$
with X a vector of variables, each t_i a vector of terms, and $p(t_i) \leftarrow D_i$, for $i = 1, \ldots, n$, all clauses defining p in P, or of the form
$p(X) \leftrightarrow false$
with X a vector of variables and $false$ a special predicate, if p is not defined in P.

In an iff-theory (one not necessarily resulting from the completion of a logic program), predicates may be defined by no iff-definition, by exactly one iff-definition, or by multiple iff-definitions. For example, the predicate $=$ may be defined by the (possibly infinite) set of iff-definitions $t = t \leftrightarrow true$, for all variable-free terms t, or by the single iff-definition $X = X \leftrightarrow true$ ($true$ is a special predicate). Undefined predicated correspond to the abducible predicates in abductive logic programs. Undefined predicates are distinguished from predicates defined by iff-definitions with $false$ in the body.

The proof procedure defined in the next section uses iff-theories not necessarily resulting from the completion of a logic program. In order to avoid inconcistencies, we assume that any given iff-theory satisfies the restriction that the heads of no two different iff-definitions unify.

Integrity constraints are implications of the form
$L_1, \ldots, L_n \rightarrow A$
with $L_1, \ldots, L_n$ (positive or negative) literals, $n \geq 0$, and A atom, possibly $false$. All variables in A, $L_1, \ldots, L_n$ are implicitly universally quantified, with scope the entire implication. A is called the *conclusion* and $L_1, \ldots, L_n$ are called the *conditions* of the integrity constraint.

Integrity constraints can be used to express "properties" of theories. For example, the integrity constraints
$X < Y \wedge Y < Z \rightarrow X < Z$
$X < Y \wedge Y < X \rightarrow false$
express the transitive and non-symmetric properties of any correct iff-theory defining the predicate $<$. Given the sentence $3 < Y \wedge Y < 1$, the first integrity constraints can be used to add $3 < 1$, and then the second, together with the information $1 < 3$, derivable from any correct iff-theory for $<$, can be used to reduce the original sentence to $false$.

An **initial query** is a conjunction of literals, with all variables free. The proof procedure that we will describe in the next section is a rewriting procedure, consisting of a number of inference rules, each of which replaces a formula by one which is equivalent to it in a given iff-theory, for all variable-free instances of the free variables. We will assume that the given iff-theory contains the (possibly infinitely many) iff-definitions $t = t \leftrightarrow true$, for all variable-free terms t, and some iff-definition for arithmetic predicates such as $<$ and $plus$.

3 The proof procedure

The basic inference rules are:
• **Unfolding**: given an atom A and an iff-definition

$H \leftrightarrow D_1 \vee \ldots \vee D_n$
such that A is an instance of H via a substitution θ (i.e. $A = H\theta$)
A is replaced by $(D_1 \vee \ldots \vee D_n)\theta$.

• **Propagation**: given an atom $p(t)$ and an integrity constraint
$L_1 \wedge \ldots p(s) \ldots \wedge L_n \rightarrow C$,
a new integrity constraint $L_1 \wedge \ldots t = s \ldots \wedge L_n \rightarrow C$ is added.

• **Logical simplification**:
$[B \vee C] \wedge E$ is replaced by $[B \wedge E] \vee [C \wedge E]$ (**splitting**)
not $A \wedge B \rightarrow C$ is replaced by $B \rightarrow C \vee A$ (**negation elimination**)
$B \wedge false$ is replaced by $false$
$B \vee false$ is replaced by B, and so on.

• **Equality rewriting**: applies equality rewrite rules (see [4]) simulating the unification algorithm [11].

Unfolding is backward reasoning with iff-definitions, whereas propagation is forward reasoning with integrity constraints. Note that unfolding does not instantiate the unfolded atom A, and therefore differs from backward reasoning in ordinary logic programming. Note also that propagation is a form of resolution. The proof procedure described here is a version of SLP in the sense that SLP adopts full resolution as propagation.

Given an iff-theory T, a set of integrity constraints I and an initial query Q, a **derivation** for Q is a sequence of formulae, $F_1 = Q \wedge I, \ldots, F_n$, such that each F_{i+1} in the sequence is obtained from the previous formula F_i by application of one of the inference rules: unfolding (applied to atoms in bodies of integrity constraints and to atoms that are conjuncts in F_i), propagation (applied to atoms that are conjuncts in F_i), equality rewriting (for equalities) or logical simplification. [1] We assume that negative literals *not* A occurring as conjuncts in any F_i in a derivation are rewritten as integrity constraints $A \rightarrow false$. Note that integrity constraints occurring in a derivation might be integrity constraints in I as well as implications derived from I or from the rewriting of a negative literal by possibly several applications of the inference rules.

Let $F_1 = Q \wedge I, \ldots, F_n = D \vee Rest$ be a derivation such that $D \neq \texttt{false}$, no inference step can be applied to D which has not already been applied earlier in the derivation and all atoms in D are **suspended**, i.e. no atoms in D can be further unfolded. Then, the set of all atoms and *denials* (integrity constraints with $false$ as conclusion) in D is a **computed answer** to Q.

Note that atoms may be suspended because there is no iff-definition for the predicate of the atoms (corresponding to failure in ordinary logic programming and to the assumption of hypotheses in abductive logic programming) or because there are multiple iff-definitions for the predicate of the atoms (corresponding to suspension in concurrent logic languages [14, 13]). Note also that atoms of undefined predicates are always suspended.

[1] To simplify the treatment of quantifiers, both iff-definitions in T and integrity constraints in I are assumed to be range-restricted, i.e. all variables in the head/conclusion must appear in at least one atom in the body/conditions, and this atom must not be an equality between two variables.

Example 1. Let T be the iff-theory

$has(X,Y) \leftrightarrow buys(X,Y) \vee steals(X,Y)$

$honest(x) \leftrightarrow x = tom$

and I consist of the integrity constraint

$honest(x) \wedge steals(x,y) \rightarrow false.$

The predicates *buys* and *steals* have no iff-definitions. A possible derivation for $Q{=}has(tom, computer)$ is

$F_1 = Q \wedge I$

$F_2 = Q \wedge [x = tom \wedge steals(x,y) \rightarrow false]$ (by unfolding)

$F_3 = Q \wedge [steals(tom,y) \rightarrow false]$ (by equality rewriting)

$F_4 = [buys(tom, computer) \vee steals(tom, computer)] \wedge$ (by unfolding)
$[steals(tom,y) \rightarrow false]$

$F_5 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee$ (by splitting)
$[steals(tom, computer) \wedge [steals(tom,y) \rightarrow false]]$

$F_6 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee$ (by propagation)
$[steals(tom, computer) \wedge [y = computer \rightarrow false]]$

$F_7 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee$
$[steals(tom, computer) \wedge false]]$ (by equality rewriting)

$F_8 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee false$
(by logical simplification)

$F_9 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]]$
(by logical simplification)

from which the answer $buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]$ can be extracted.

Example 2. Let T be the iff-theory $p \leftrightarrow p$, I be empty and Q be p. The only possible derivation is

$F_1 = Q \wedge I = p$

$F_2 = p.$

No answer can be extracted from F_2 since p is not suspended.

4 The agent architecture

Every agent can be thought of as an iff-theory together with a set of integrity constraints and an initial query. Atoms in answers are *actions* to be executed as well as *observations* to be performed. Both updates and queries to the agent are treated uniformly as observations. Selected actions can be thought of as outputs into the environment and observations as inputs from the environment. ¿From every agent's viewpoint, the environment contains all the other agents. The basic engine of an agent is the proof procedure given above, executed via the following cycle:

To cycle at time T,

(i) observe any input at time T,
(ii) record any such input,
(iii) resume the proof procedure by propagating the inputs,
(iv) continue applying the proof procedure,
using for steps (iii) and (iv) a total of R units of time,
(v) select an atomic action which can be executed at time $T+R+2$,
(vi) execute the selected action at time $T+R+2$ and record the result,
(vii) cycle at time $T+R+3$.

The cycle starts at time T by observing and recording any inputs from the environment (steps (i) and (ii)). Steps (i) and (ii) are assumed to take one unit of time each. Then, the proof procedure is applied for R units of time (steps (iii) and (iv)), both assimilating the input and working on any remaining goals. The amount of resources R available in steps (iii) and (iv) is bounded by some predefined amount n. By decreasing n the agent is more *reactive*, by increasing n the agent is more *rational*. Propagation is applied first (step (iii)), in order to allow for an appropriate reaction to the inputs. Finally, an action is selected and executed, taking care to record the result (steps (v) and (vi)). Steps (v) and (vi) conjoined are assumed to take one unit of time.

Selected actions will be amongst the suspended atoms in an answer extracted from a disjunct in a derivation. The disjunct represents an *intention*, i.e. a (possibly partial) plan executed in stages. We can assume that actions from the same disjunct (intention) are selected at different iterations of the cycle. Failure of a selected plan is obtained via logical simplification, after having propagated $false$ into the selected disjunct.

5 Examples

The event calculus [9] is a logic programming formalism for temporal reasoning. A simplified version of the event calculus can be expressed via the iff-definitions:

$$\begin{aligned} &holds\_at(P,T_2) \leftrightarrow happens(E,T_1) \wedge T_1 < T_2 \wedge \\ &\qquad initiates(E,P) \wedge not\, broken(T_1,P,T_2) \\ &broken(T_1,P,T_2) \leftrightarrow happens(E,T) \wedge terminates(E,P) \wedge T_1 < T < T_2. \end{aligned}$$

The first iff-definition states that a property P holds at some time T_2 if and only if it is initiated by an event E occurring at some earlier time T_1 and is not broken (i.e. P persists) from T_1 to T_2. The second iff-definition states that a property P is broken (i.e. does not persist) from a time T_1 to a later time T_2 if an event E that terminates P happens at a time T between T_1 and T_2.

There is no iff-definition for the predicate *happens*. Atoms of this predicate will always be suspended and be generated as part of answers (plans). Atoms of this predicate can also be given as inputsi, or be part of initial queries.

The following integrity constraint

$happens(E,T) \wedge precondition(E,T,P) \wedge not\, holds\_at(P,T) \rightarrow false$

can be used to express that an event E cannot happen at a time T if a precondition P of E does not hold at time T.

The predicates *precondition*, *initiates* and *terminates* are defined by application-specific iff-definitions, e.g.

$$precondition(carry\_umbrella,T,P) \leftrightarrow P = own\_umbrella \vee$$
$$P = borrowed\_umbrella$$

$initiates(rain,raining) \leftrightarrow true$

$terminates(sun,raining) \leftrightarrow true.$

Additional integrity constraints might be given to represent reactive behaviour of intelligent agents, e.g.

$happens(rain,T) \rightarrow happens(carry\_umbrella,T+1)$

and to prevent concurrent execution of actions (events):

$happens(E,T) \wedge happens(E',T) \rightarrow E = E'.$

An answer computed by the proof procedure can be interpreted as sets of actions to be performed to achieve given goals (queries) and/or to react to given events. For example, given the query

$happens(rain,5)$

the proof procedure computes the answer $happens(carry\_umbrella,6)$.

Within cycle, the query is an input at time 5, and the computed answer is a selected action at time 6, matching the time within the action.

6 Conclusions

We have presented a version of the proof procedure of [10]. The procedure combines backward reasoning with iff-definitions and forward reasoning with integrity constraints. We have illustrated the application of the proof procedure within the agent architecture of [6–8]. Here, backward reasoning with iff-definitions corresponds to reduction of goals to sub-goals and forward reasoning with integrity constraints corresponds to reactivity to inputs. Agents' plans of action are generated via a combination of the two reasoning modes. The execution of (possibly partial) plans is achieved via a particular execution of the proof procedure, where processing of inputs from the external environment in interleaved with application of the procedure.

Acknowledgements

This reasearch was supported by the UK EPSRC project "Logic-based multi-agent systems".

References

1. K.L. Clark, Negation as failure. *Logic and Data Bases*, Gallaire and Minker eds., Plenum, New York(1978) 293–322
2. Console, L., Theseider Dupré, D., Torasso, P. On the relationship between abduction and deduction. *Journal of Logic and Computation* 2(5) (1991) 661–690
3. M. Denecker, D. De Schreye, SLDNFA: an abductive procedure for normal abductive programs. Proceedings of *International Conference and Symposium on Logic Programming* (1992), 686–700
4. T.H. Fung, R.A. Kowalski, The iff procedure for abductive logic programming. *Journal of Logic Programming* 33(2):151–165 (1997)
5. A.C. Kakas, R.A. Kowalski, F. Toni, The role of abduction in logic programming. *Handbook of logic in Artificial Intelligence and Logic Programming* 5, 235–324, D. Gabbay, C. Hogger, and A. Robinson eds., Oxford University Press (1998)
6. R.A. Kowalski, Using meta-logic to reconcile reactive with rational agents. In *Meta-logic and Logic Programmimg*, K.Apt and F. Turini (eds.), MIT Press (1995), 227–242
7. R.A. Kowalski, F. Sadri, Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic in Databases*, San Miniato, Italy, Springer-Verlag LNCS 1154 (1996)
8. R.A. Kowalski, F. Sadri, An agent architecture that unifies rationality with reactivity. Imperial College Technical Report (1997)
9. Kowalski, R.A. and Sergot, M.: "A logic-based calculus of events", *New Generation Computing* **4**, 1986, 67–95.
10. R.A. Kowalski, F. Toni, G. Wetzel, Executing suspended logic programs. To appear in *Fundamentae Informaticae* (1998)
11. A. Martelli, U. Montanari, An efficient unification algorithm, *ACM Trans. on Prog. Lang. and Systems* 4(2), 258–282, 1982
12. F. Sadri and F. Toni, Active behaviour in deductive databases, Imperial College Technical Report (1996)
13. Saraswat, V. A. and Rinard, M.: "Concurrent Constraint Programming", *Proc. of the* 17^{th} *ACM Symp. on the Principles of Programming Languages*, 1990, 232–245.
14. Shapiro, E.: "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys* **21 (3)**, 1989, 413–510.
15. Y. Shoham, Agent-oriented programming. *Artificial Intelligence* 60(1) (1993), 51–92

Proof Planning with Multiple Strategies

Erica Melis*

Universität des Saarlandes, FB Informatik
D-66041 Saarbrücken, Germany
melis@cs.uni-sb.de

Abstract. This paper addresses the integration of several planning strategies as a way to cope with otherwise intractable search spaces in proof planning. It discusses why it is reasonable to employ several refinement strategies and shows that it can be even *necessary* in order to find a proof plan at all. Motivated by our experiences in proof planning, the paper introduces new refinement strategies. Since choosing among strategies requires additional control some exemplary control knowledge that can be employed in proof planning is presented.

1 Introduction

Proof planning is planning in mathematical domains. Proof planning can be a particularly complex task with prohibitively large search spaces for which pure backward or forward planning cannot find a solution. Consider, for instance, the following relatively simple example from algebra. Let the formula to be proved[1]

$$f(x * a) = x * f(a) \tag{1}$$

and the proof assumptions, among others,

$$f_2(x * Y) = x * f_2(Y), \tag{2}$$

$$f(f_1(x)) = f_2(x), \tag{3}$$

$$\exists z(f_1(z) = y). \tag{4}$$

Backward proof planning would try to find an operator that reduces the goal (1) to simpler subgoals, e.g., by applying one of the assumptions. This attempt fails because none of the lhs or rhs of the assumptions matches the lhs or the rhs of (1). A lemma had to be invented to enable such a goal reduction. The search space for a lemma is, however, potentially infinite. On the other hand, pure forward planning would yield a potentially infinite search space because the assumptions can be combined to implications, conjunctions, etc. to new derived assumptions rather arbitrarily. Hence forward planning has to be strongly directed by search

* This work was supported by the Deutsche Forschungsgemeinschaft, SFB 378

[1] Actually, "formula" is simplified for sequent, see below.
* denotes the multiplication function and f, f_1, f_2 other functions.

heuristics. The gap between (1) and the assumptions (2), (3), and (4) is, however, too large in order to apply any available directing heuristics. This is due to the fact that in order to close the gap, differences would have to be enlarged and reduced subsequently. In summary, pure backward or forward refinement does not yield a proof plan for this example.

The intractable search in proof planning is just one reason to develop a multi-strategy proof planner, however, the most important from a computational point of view. In addition, multi-strategy planning appears to be desirable (i) from a resource-adaptive planning point of view and (ii) from a cognitive modeling point of view:

i In order to adapt a planning process to, say, time resources for the planning itself, different planning strategies can be used dependent on the resources available. For instance, the expansion of an abstract operator is realized if there is enough time only.
ii As for the cognitive adequacy of multistrategy proof planning, there is empirical evidence that people do use several strategies to solve problems.

Contribution of the Paper

In the following we shall show why proof planning needs to employ several refinement strategies, and thereby we generalize combinations of, say, case-based planning and generative planning.

In order to cope with the search space we employ several refinement strategies in a multistrategy-planning framework that was introduced for a different purpose. Furthermore, the paper briefly introduces new refinement strategies that are needed in proof planning. This is done mainly to give an idea on how novel strategies can evolve. and provide first suggestions for the control of strategy choices. We think that these new ideas can be beneficial for planning tasks in other realistic domains with large search spaces too.

2 Proof Planning

Proof planning as introduced by [3] uses operators that encapsulate mathematical methods, such as induction, diagonalization, refutation. The resulting proof plans are abstract representations of calculus proofs because the steps in a plan are more complex than calculus steps such as resolution.

In terms of AI-planning the basic approach to proof planning is essentially that of classical state-space planning in artificial intelligence [5]. A proof planning state is a set of sequents $\Delta \vdash F$[2] rather than literals, where Δ is a set of predicate logic formulae and F is a formula. In the proof planner OMEGA [2], a planning state is a set of sequents[3] that is divided into open *goals* and *assumptions*.

[2] meaning F is derivable from Δ

[3] In the following, sequents are abbreviated by their formula if Δ is not particularly relevant or by a name such as L1.

A *partial plan* π is a tuple $(T, \prec, B, Aux)$, where T is a set of steps that are instantiated operators. T contains the `start` and `finish` step t_0 and t_∞; $\prec$ is a partial order over T; B is a set of binding and prohibited binding constraints on variables; The auxiliary constraints Aux are Contiguity constraints $(t_i \lhd t_j)$ between two steps that demands that no step intervene between t_i and t_j and Abstraction constraints represented by $\mathrm{abs}(g_i, g_j)$ for goals/assumptions g_i, g_j. The abstraction constraints express that g_j is an abstraction of g_i.

A proof planning problem is defined by an initial state specified by the proof assumptions, the open goal g given by the theorem to be proved, and a set of operators. As usual, a proof planning problem defines a first partial plan π_0 with $T = \{t_0, t_\infty\}$ and $t_0 \prec t_\infty$, where t_0 provides the assumptions and t_∞ satisfies g.

Partial plans can be refined by refinement strategies[4] that introduce steps, constraints, and auxiliary constraints into a partial plan until a solution is found, i.e., a sequence of steps that transforms the initial state into a final state, i.e., a state with no open goals. The solution satisfies the constraints in the partial plan. In the simplest version of proof planning, precondition satisfaction, the planner searches for a step `M` whose application proves a goal g and introduces `M` into the plan. The subgoals needed for the application of `M` replace g in the planning state. The planner continues to search for operators applicable to a subgoal and terminates when a solution is found.

More elaborate versions of proof planning can already employ the refinement strategies expansion of abstract operators (also known as task reduction [19]) and precondition abstraction (also known as state abstraction [17]) with the following **fixed control**:

- When a solution is found, each abstract operator is expanded.
- When an operator is expanded, the previously ignored preconditions become open goals.

Eventually, planning and recursive expansion of operators yield a calculus-level proof, e.g., a Natural Deduction (ND)-proof.

In OMEGA, operators are data structures with the slots *premises* and *conclusions, application-conditions*, and *proof schema*. *Premises* are (annotated) sequents that are used by an operator to **logically derive** the *conclusions*, and *conclusions* are (annotated) sequents which the operator is designed to prove. Roughly, the annotations indicate the dynamic **planning behavior** of the operator. When an instantiated operator is applied, a $\ominus$ *conclusion* is deleted from the state as a goal, a $\oplus$*premise* is added as a new subgoal, a $\ominus$ *premise* is deleted as an assumption, and a $\oplus$*conclusion* is added as an assumption. For instance, when the step `ApplyEquation`(Δ, F, rhs, lhs) is introduced into a partial plan, the goal L2 is replaced by a subgoal L1 if an equational assumption (L0) from the planning state can be used to rewrite L2, i.e., if there is a substitution σ that unifies one side of the equation with a term t_2 in L2. The tactic `equ` in the

[4] We do not formally define refinement strategies here because this distracts from the main purpose of the paper and needs more space. For a definition see [9].

line-justification of L2 applies the calculus rules necessary to derive the formula $F(t_2)$ from $F(t_1)$.

operator: `ApplyEquation`(Δ, F, rhs, lhs)	
premises	L0, $\oplus$ L1
conclusions	$\ominus$ L2
appl-cond	$\exists\sigma(t_1 = \sigma(lhs)\ \&\ t_2 = \sigma(rhs))$
proof schema	L0. $\quad \vdash lhs = rhs \quad$ () L1. $\Delta \quad \vdash F(t_1) \quad$ (OPEN) L2. $\Delta \quad \vdash F(t_2) \quad$ (equ;L0,L1)

Compared with AI-planning proof planning is difficult because typically the search space is large or even potentially infinite which is due to the following facts:

1. Typically, the solutions are deep.
2. A potentially infinite branching of the search space can occur for the instantiation of existentially quantified variables.
3. A potentially infinite branching of the search space can occur when searching for a lemma to be invented. In principle, this search cannot be avoided because the cut rule, i.e., the backwardly applied rule $\frac{\Gamma,B\vdash A \quad \Gamma\vdash B}{\Gamma\vdash A}$, is not avoidable in general, and infinitely many lemmata B can exist. For a theoretical analysis see [12].

As mentioned above, proof planning in OMEGA originally employed the strategies backward state-space (BSS), forward state-space (FSS), expansion of operators (HTN), and precondition abstraction (PA) with a fix control. There are at least two reasons for improvement. First, these refinement strategies turn out to be insufficient for many even relatively simple problems, e.g., the problem sketched in the introduction. Secondly, a reasonable decision on which refinement to choose may well depend on the time resources available for the planning or on the particular domain rather than being fixed in the planner. Hence, below we describe a framework for planning with multiple strategies that allows for a more flexible control and then we define new refinement strategies. First, however, let's motivate these strategies by returning to the introductory example.

3 Example, continued

Recall the situation in which the planning state consists of the goal (1) and the assumptions (2), (3), and (4). Since the gap between the goal and the assumptions is too big for a lemma speculation, a possible strategy is to search for an intermediate state and the island step that yields this state, i.e., a step that is not yet connected to other steps in the plan by precondition satisfaction. Introducing an island step into the partial plan and in particular knowing its associated intermediate state, reduces the search space because the problem is reduced to two subproblems as described in [11]. Moreover, the smaller gaps may provide enough information for determining a lemma or at least a schematic version of a lemma[5] whose meta-variables can be instantiated in due course. In the following we exemplify strategies that are based on a proper language abstraction of the goal and assumptions and yield island steps and intermediate states:

¿From an abstract point of view, the goal (1): $f(\boxed{x*}a) = \boxed{x*}f(a)$ requires to move the expression in the box from a position wrt. $f(a)$ to another position wrt. $f(a)$. An abstraction *abs* of the goal that captures this situation is

$$@(x * @) = x * @, \tag{5}$$

where @ stands for any expression. Similarly, applying the same kind of abstraction to the assumption (2) yields $abs(2) : @(x * @) = x * @$.

That is, the abstract problem has the goal (5) and the assumptions $abs(2)$, $abs(3)$, and $abs(4)$. In order to satisfy the goal (5) the abstract step `ApplyEquation`$(abs(2))$ is introduced. This yields the abstract precondition $@(x* @) = @(x * @)$ which is an instantiation of $z = z$ and can therefore be reduced to *true* by the step `Elementary`. *true* is satisfied in any proof planning state, so there is no open goal anymore. A section on control describes knowledge as to why this particular abstraction is chosen.

The lhs of Figure 1 depicts the described situation. Rectangles represent steps of the plan, e.g. s_a for `ApplyEquation`$(abs(2))$, and ovals indicate goals/assumptions. $abs(g)$ denotes the abstraction of the goal g and therefore an abstraction relation $\text{abs}(g, abs(g))$, depicted by a dashed line, holds between g and $abs(g)$. Abstracting the language of the current problem and continuing by planning at the abstract level is called *abstract-planning*.

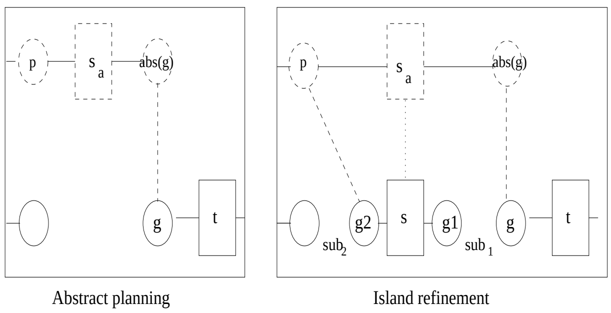

Fig. 1. Abstract Planning and Island Refinement

The rhs of Figure 1 depicts a refinement strategy, called *island-refinement*, that takes the abstract steps and goals and maps them back to the ground level. This island refinement produces island steps at the ground level. In the example, the step s_a, `ApplyEquation`$(abs(2))$, is mapped to the island step s

[5] Schematic means that some sub-expressions may be represented by a meta-variable.

which is ApplyEquation(2). The abstract step Elementary is mapped back to an Elementary step at the ground level. For each of the island steps pre- and postconditions are computed, some with meta-variables.

In other words, by abstract planning and island refinement we find that at **some point** ApplyEquation(2) and Elementary have to be applied at the ground level and corresponding approximate intermediate states are computed: ApplyEquation(2) has the precondition $g_2 : F_0 = f_2(x * Y)$[6] (in the rhs of the figure), and the postcondition

$$g_1 : \qquad F_0 = x * f_2(Y). \tag{6}$$

for a meta-variable F_0.[7]

That is, now the ground-level plan contains a gap between the goal (1) and (6). This gap constitutes the subproblem sub_1 which can be solved by FSS that is guided by a difference reduction heuristics described in the control section and that prefers the step ApplyEquation(3) from (6). This step yields the new assumption $F_0 = x * f(f_1(Y))$.

The planning for the subproblem sub_1 can be completed by instantiating the meta-variable F_0 to $f(x * a)$ and by a step ReplaceTerm that replaces $f_1(Y)$ by a and that speculates the lemma $\exists Y. f_1(Y) = a$ which can be proved from (4). Similarly, the subproblem sub_2 can be solved.

The example has shown the necessity to combine different strategies to solve the given problem. It has illustrated abstract planning and island refinement.

4 Realizing Multiple-Strategy Proof Planning

In order to flexibly control the choices of strategies, we use a planner that can call different refinement strategies. This is similar to the framework in [9] that was developed for a different purpose. All strategies refine one and the same proof plan data structure [2] in OMEGA.

Table 1 shows the top-level procedure of the planner. Termination check is as usual. A strategy is selected according to the control knowledge. The control-knowledge is represented by control-rules in OMEGA. The selected strategy refines the partial plan π. Case-based refinement is realized in OMEGA by analogy-driven proof plan construction [13].

A multi-strategy planner may have refinement strategies of different granularity and completeness at its disposal. For instance, FSS and BSS introduce single steps into a partial plan, whereas HTN and CBP introduce whole subplans. Theoretically, some of the strategies may be complete while others are not. Practically, however, such a distinction does not really matter in planning

[6] The precondition is simplified because the lhs really is $F_0(x * F_2)$ for meta-variables F_0, F_2 because of the corresponding abstract precondition p.

[7] Actually, the postcondition of ApplyEquation$(f_2(x*Y) = x*f_2(Y))$ is $F_0 = F_1(f_2(x* Y))$ and the precondition $F_0 = F_1(f_2(x * Y))$ for meta-variables F_0, F_1. The meta-variable F_1 can, however, be instantiated to the identity function because the rhs of the abstract postcondition is $x * @$.

Planning Algorithm `PLAN`(π)
Parameters: `sol` procedure for picking solutions

Termination Check: If `sol`(π) returns a solution, return it and terminate. If it returns *fail*, fail. Otherwise continue.
Refinement: Pick one of the following refinement strategies and refine(π): (*Not a backtrack point*)

- forward-state-space refinement (FSS)
- backward-state-space refinement (BSS)
- expansion of abstract operators (HTN)
- case-based refinement (CBP)
- abstract-refinement,
- island-refinement,
- refinement-by-user

Recursive Invocation: Call `PLAN` on the refined plan.

Table 1. Outline for multistrategy planning

with potentially infinite search spaces. Here, most importantly, each strategy can contribute to finding a plan at all.

Note that a multi-strategy-planning that has the strategy CBP at its disposal is a generalization of case-based planning approaches that combine case-based and generative planning, e.g. [20, 7]. These approaches use the fix control that first case-based planning is chosen and then the remaining open subgoals are closed by generative planning.

The new strategies abstract-refinement and island-refinement are described in the following.

4.1 New Strategies

Table 2 provides an outline of abstract-refinement for a partial plan π. In this strategy, the construction of an abstracted problem P_{abs} involves abstracting a goal g, the assumptions of the current problem, and the operators by an abstraction mapping abs. This construction yields the abstract problem P_{abs} and an Abstraction constraint abs$(g, abs(g))$. Then PLAN calls the planner for P_{abs} with the abstracted operators. `insert` involves at least adding to π the constraints and steps of the partial plan π_{abs}.

Recall the example, where the goal (1) is picked and abstracted to (5). The abstracted problem P_{abs} has an initial state that includes the goal (5) and the abstracted proof assumptions $abs(2)$ etc. Planning for P_{abs} inserts the steps `ApplyEquation`($abs(2)$) and `Elementary`. `insert` adds the auxiliary constraints abs((1), (5)),
(`ApplyEquation`($abs(2)$) $\lhd\, t$),
(`Elementary` $\lhd$ `ApplyEquation`($abs(2)$)),
and the order constraint (`Elementary` $\prec$ t) into π for the ground step t whose postcondition is g.

Algorithm abstract-refinement(π)/* Returns refinements of π*/
Parameters: `insert` procedure for refining π by inserting the abstract plan.

Goal selection: Pick a goal g. (*Not a backtrack point.*)
Abstraction: Choose abstraction mapping abs and construct the abstract problem P_{abs} with goal(P_{abs}) $= abs(g)$ and assumptions(P_{abs}) $= abs$(assumptions). (*Backtrack point.*)
PLAN(π_{abs}) for P_{abs}.
Insertion `insert` π_{abs} and auxiliary Abstraction constraint into π.

Table 2. Backward Abstract-Refinement

The abstract steps introduced by abstract-refinement have to be refined by island-refinement in order to obtain a plan consisting of ground steps. Island-refinement replaces an abstract step s_a by a ground step s and abstract goals/assumptions by ground goals/assumptions. Thereby it computes a subproblem that yields an approximate intermediate state. Approximate means the goals and assumptions may contain meta-variables.

For the abstract step s_a a mapping mb is chosen that maps s_a to a ground-level operator s such that $mb(post(s_a)) = g$ for an open ground goal g. mb also maps goals and assumptions. Even though mb is restricted by the requirement $mb(post(s_a)) = g$, in general different mappings mb are possible. Therefore the search for mb is reduced by allowing for meta-variables for terms and formulas in goals (e.g., F_0). Thereby binding decisions are postponed.

Algorithm island-refinement(π)/* Returns refinements of π*/
Parameters: `introduce-subproblem` procedure

Step selection: Pick abstract step s_a for which $abs(g) = post(s_a)$ for an open ground-level goal g. (*Not a backtrack point*)
Island instantiation: Choose mapping back to ground level mb so that $mb(post(s_a)) = g$
Replace s_a in π by step s with $s = mb(s_a)$. For any ground step t this involves

- replacing $s_a \lhd t$ by $s \prec t$ and $t \lhd s_a$ by $t \prec s$
  $s_a \prec t$ by $s \prec t$ and $t \prec s_a$ by $t \prec s$
- introducing auxiliary constraints: for each precondition pre_k of s_a introduce the constraint abs$(pre_k(s)), pre_k)$.

(*Backtrack point; all mb can be considered*)
Intermediate state construction:
`introduce-subproblem` defines a problem P_{sub} with the goal g and assumptions $post(s)$ and computes an approximate intermediate state.

Table 3. Island-Refinement

For island-refinement in the rhs of Figure 1 the abstract step s_a is picked. The dotted line indicates the mapping back of the step s_a. The ground step s is introduced into the plan π, $s_a \lhd t$ is replaced by $s \prec t$. For $pre(s) = \{p\}$, the auxiliary constraint abs(g_2, p) is introduced.

4.2 Control Knowledge in Proof Planning

In OMEGA, the planning is controlled by the interpretation of control rules similar to those in Prodigy [14]. Control-rules contain meta-predicates that can describe the planning history, the state, resources, constraints. Currently, we distinguish the following classes of control-rules that correspond to different kinds of decisions of the planner.

- *strategy*
- *operator*
- *goal*
- *abstraction*
- *mapping-back*

In order to give an idea of what the control knowledge looks like, we verbally describe some control-rules. Note that our work on devising strategy control-rules is just a beginning. Currently we investigate the learning of such control-rules.

In what follows, we present control-rules applicable to a class of problems that includes the example.
strategy control knowledge:

- BSS is the default refinement strategy.
- If BSS and focused FSS fail, then prefer abstract-refinement and mark current goal with `fss`.
- If there is an abstract step in the plan, then prefer island-refinement.
- If a goal of a subproblem P (introduced by island-refinement) is marked with `fss`, then prefer FSS over BSS in the planning for P.
- Expand an operator if there is sufficient time only.

abstraction control knowledge:

- If the current goal (essentially) is an equational formula, where the lhs and rhs differ in the position of a common expression only, then prefer an abstraction of the parts of the formula that do not belong to this expression. In the example, the original $f(a)$ that does not belong to the common expression $x*$ is abstracted.
- If the current goal (essentially) is an equational formula and if the lhs and rhs differ in the occurring function symbols, then prefer the abstraction to maximal_symbol_multisets described in [1]. This abstraction takes the multisets of symbols occurring in a formula and removes those symbols that define others.
  In subproblem sub_1 in the example, f_2 is a maximal symbol in (6) and f is a maximal symbol in (1). The multisets of maximal symbols in the goal (1) and in the assumption (6) are $\{f,*\}$ and $\{f_2,*\}$, respectively. The next *operator* control-rule that aims at difference reduction uses this abstraction to decide which step to introduce by FSS on (6).

operator control knowledge:
A control-rule applicable in FSS requires: If the goal is (essentially) equational and if the goal and an assumption have the maximal-symbol-multisets S_g and S_a, respectively, then prefer ApplyEquation(e) for an equation e that introduces a (large) subset of $S_g \setminus S_a$ or removes a (large) subset of $S_a \setminus S_g$. For instance, in FSS planning for subproblem sub_1 with the assumption (6) and goal (1), ApplyEquation(3) "removes" f_2 from the assumption and "introduces" f.

5 Evaluation and Conclusion

Our evaluation is a qualitative one rather than quantitative Because of a potentially infinite search space the default BSS proof planning may fail to find a plan. Hence, the availability and integration of several refinement strategies primarily is a question of solvability rather than of run time. A qualitative evaluation shows that planning with multiple strategies is useful because problems can be solved in that way that cannot be solved otherwise. Preliminary quantitative results show that multiple-strategy proof planning has quite some impact on the number of solvable problems. Run time experiments would not make much sense.

Apart from the reduction of the search space, the integration of several planning strategies in one framework naturally allows for a flexible integration of user interaction and of case-based planning too. Moreover, the framework helps to explicate, where which kind of control knowledge is needed. Allowing for a flexible use of different refinement strategies adds a choice point to the general planning algorithm. This requires control knowledge on when to choose which strategy in addition to the control knowledge used within the refinement strategies. For proof planning we have presented some exemplary control knowledge.

The problem of potentially infinite search spaces exists beyond proof planning. Therefore, the need of and our approach to multi-strategy planning is applicable and useful for other realistic planning domains as well.

5.1 Related Work

Our work builds on ideas from planning, problem solving, and theorem proving. In planning, a unifying general planning framework was introduced by [8, 9] in order to compare and evaluate different strategies that are the basis for different planners. [18] empirically substantiated the need for different search heuristics in BSS for efficient planning in *different domains*. [4] argue for combining HTN and operator-based planning strategies for representational reasons.

While our approach to island-refinement is based on a proper language abstraction of the goal and assumptions, island search does not necessarily involve abstraction, see e.g., [16] on stepping stones.

There is a variety of approaches to abstraction. Problem abstraction to guide problem solving and planning has, for instance, been addressed in [17, 6, 10]. More specifically, abstraction in theorem proving is addressed, e.g., in Plaisted's classical paper [15]; Hutter and Autexier [1] propose concrete abstractions for equational theorem proving, e.g., maximal_symbols_multiset used in our example above.

References

1. S. Autexier and D. Hutter. Equational proof-planning by dynamic abstraction. In M.P. Bonacina and U. Furbach, editors, *Proceedings of FTP97*, 1997.
2. C. Benzmueller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. Siekmann, and V. Sorge. OMEGA: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of CADE-14*, pages 252–255, 1997.
3. A. Bundy. The use of explicit plans to guide inductive proofs. In E. Lusk and R. Overbeek, editors, *Proceedings of CADE-9*, pages 111–120, 1988.
4. T.A. Estlin, S.A. Chien, and X. Wang. An argument for a hybrid htn/operator-based approac to planning. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning. 4th European Conference on Planning, ECP'97*, volume 1348 of *Lecture Notes in Artificial Intelligence*, pages 182–194. Springer, 1997.
5. R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
6. F. Giunchiglia and T. Walsh. A theory of abstraction. *Artificial Intelligence*, 57:323–390, 1992.
7. S. Kambhampati and J.A. Hendler. A validation based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–253, 1992.
8. S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence, special issue on Planning and Scheduling*, 76:167–238, 1995.
9. S. Kambhampati and B. Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 61–78. 1996.
10. C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.
11. R.E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.
12. G. Kreisel. Mathematical logic. In T. Saaty, editor, *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons, 1965.
13. E. Melis. A model of analogy-driven proof-plan construction. In *Proceedings of IJCAI-95*, pages 182–189, Montreal, 1995.
14. S. Minton, C. Knoblock, D. Koukka, Y. Gil, R. Joseph, and J. Carbonell. *PRODIGY 2.0: The Manual and Tutorial*. School of Computer Science, Carnegie Mellon University, Pittsburgh, 1989. CMU-CS-89-146.
15. D. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16:47–108, 1981.
16. D. Ruby and D. Kibler. Learning subgoal sequences for planning. In *Proceedings of IJCAI-89*, pages 609–614, 1989.
17. E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.
18. P. Stone, M.M. Veloso, and J. Blythe. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning*, 1994.

19. A. Tate. Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893. Morgan Kaufmann, 1977.
20. M.M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer, Berlin, New York, 1994.