Proceedings of the 4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)

held in conjunction with IJCAR 2001

Monday, 18th of June, 2001

Siena, Italy

Maria Paola Bonacina and **Bernhard Gramlich** (editors)

STRATEGIES 2001 Web Site:
`http://www.logic.at/strategies01/`

STRATEGIES Series Web Site:
`http://www.logic.at/strategies/`

IJCAR 2001 Web Site:
`http://www.dii.unisi.it/~ijcar/`

Preface

The inference rules at the heart of automated deduction systems are typically nondeterministic in nature. Thus, in order to obtain a mechanical procedure, inference rules need to be complemented by another component, usually called *strategy* or *search plan*, which is responsible for the *control* of the inference rules.

This is true, to a greater or lesser extent, across the whole spectrum of systems for automated reasoning. In fully automated theorem proving, the *search plan* selects the inference rule and the premises for the next step in the derivation, which involves prioritizing inferences rules (e.g., contraction-first search plans), heuristics to sort the database of existing formulae or to sort open subgoals, and criteria, also typically heuristic in nature, to prune the search space. Similar considerations apply to decision procedures, model building methods, rewriting engines, where the *rewriting strategy* selects redex and rule for the next step, and, at a higher level of abstraction, logical frameworks. In interactive systems, *tactics* define conditions for the application of inference rules, and in turn *tacticals* may control the application of tactics; it is still the control component of the proof assistant that is responsible for requesting the user to instantiate variables, or select lemmas.

All deduction paradigms (generation of consequences, subgoal-reduction, generation of instances, case analysis, enumeration) generate a very large space of choices, so that powerful strategies are fundamental for their effective realization. In contrast with this key role, much knowledge about strategies and search plans has remained hidden in the code of implemented systems: papers on deduction methods have traditionally emphasized the inference rules, leaving everything else to the implementation; system manuals focus on the interface, and system descriptions concentrate on experiments in order to advertise the system.

The series of workshops on "Strategies in Automated Deduction," or STRATEGIES for short, aims at redressing this situation by studying strategies in their own right, in all kinds of automated reasoning systems. The first workshop of this series was held in Townsville, Australia, in conjunction with the 14th Conference in Automated Deduction (CADE) in July 1997, followed by a second meeting during CADE-15 in Lindau, Germany, in July 1998, and a third one at the Federated Logic Conference (FLoC) in Trento, Italy, in July 1999. This year the workshop joins the first International Joint Conference in Automated Reasoning (IJCAR) in Siena, Italy. IJCAR merges CADE, FTP (the Workshop on First-order Theorem Proving) and TABLEAUX (the Conference on Automated Reasoning with Analytic Tableaux and Related Methods), providing an excellent context for a workshop on "Strategies in Automated Deduction," since strategies are ubiquitous in automated reasoning systems.

Topics of interest of the series and this year's workshop include:

- Models of the search space and languages or mathematical formalisms to define strategies and prove properties about them, including machine-independent evaluation and comparison of strategies;
- Analysis of the search space (e.g., regularities, symmetries, classification, stratification);
- Meta-level features (e.g., pre-processing, compilation, lemmatization, caching, usage of semantics or domain knowledge);

– Strategies in (existing) systems (e.g., implementation of the proof search model, flexibility and programmability of strategies, role of the user);
– Strategies devoted to specific theories including inductive theories, arithmetic, decidable theories, and combinations of theories;
– Applications and case studies in which strategies play a major role;

as well as all aspects related to *control* in automated theorem provers, automated model builders, decision procedures, interactive proof assistants, proof planners, and logical frameworks, in first-order (including propositional and purely equational as special cases), modal (e.g., temporal) and higher-order logics.

This technical report presents the invited paper and the eight regular papers featured at STRATEGIES 2001, as well as position papers from participants. We close by thanking all the people who made this workshop possible, including the members of the Program Committee, the referees, and Dieter Hutter and Fabio Massacci, Workshop Chair and General Chair, respectively, of IJCAR 2001.

May 2001

Maria Paola Bonacina and Bernhard Gramlich
Program Co-Chairs
STRATEGIES 2001

Organization

Organizers

Maria Paola Bonacina, The University of Iowa, U.S.A.
Bernhard Gramlich, Technische Universität Wien, Austria

Program Committee

Alessandro Armando, Università degli Studi di Genova, Italy
armando@dist.unige.it

Maria Paola Bonacina (**Co-chair**), The University of Iowa, U.S.A.
bonacina@cs.uiowa.edu

Gilles Dowek, INRIA, France and ICASE/NASA-Langley, U.S.A.
Gilles.Dowek@inria.fr

Ruben Gamboa, AnythingOvernight.com, Inc., Texas, U.S.A.
ruben@acm.org

Bernhard Gramlich (**Co-chair**), Technische Universität Wien, Austria
gramlich@logic.at

Bernd Löchner, Universität Kaiserslautern, Germany
loechner@informatik.uni-kl.de

Christophe Ringeissen, LORIA-INRIA, Nancy, France
Christophe.Ringeissen@loria.fr

Additional referees

Claudio Castellini   Luigi Liquori
Silvio Ranise   Sorin Stratulat

Table of Contents

Mechanical Software Verification: High Level Control Aspects from a User's Perspective

(Invited paper)

Wolfgang Goerigk

Christian-Albrechts-Universität zu Kiel, Germany*

Abstract. We present lessons learned from using mechanical theorem proving for proof support in software verification, with trusted execution of programs in mind. We will use two realistic running examples, compiler verification, which is central if we want to prove that we can trust a piece of executable software, and an industrial project in which we proved the correctness of a safety critical expert system using (verified) runtime result verification. We will emphasize the role of partial program correctness and its preservation. And we will comment on high level control aspects, in particular on what we can and what we will not be able to prove for a concrete piece of executable software.

1 Introduction

Our paper is about software verification, *i.e.,* about proving properties of programs which we want to guarantee in order to trust in program execution. But how can we mechanically verify executable machine programs? Will we ever be able to prove non-trivial correctness properties of realistic executable machine programs and furthermore check the proofs using theorem provers? Or should we better transfer such attempts into Wittgenstein's domain of logical scepticism, arguing that no way of reasoning will ever lead towards convincing solutions – hence just give up? Our answer is no. There are solutions. But since we definitely will not succeed to verify large programs semantically on machine code level (at least not in an engineering style), we have to modularize the problem. Hence, our paper is about source level verification of programs and compiler verification.

In spite of the problems mentioned above, there is a high risk either to formulate theorems which are simply not true, or (practically) not provable, or useless. Therefore, a major goal will be to show what we can and what we cannot expect to be able to prove for a realistic piece of executable software. We will emphasize the importance of partial program correctness and its preservation, i.e., of a correctness guarantee for regularly computed non-erroneous results also of machine executables. This is often sufficient, very practical, and sometimes surprisingly easy to prove, even rigorously, formally and mechanically.

We do not believe that the entire work necessary for a convincing software correctness proof can be "push button", *i.e.,* fully mechanical and automatic. We have to

* Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, D-24098 Kiel, Germany. Email: wg@informatik.uni-kiel.de

think about what to prove and how to prove it, tasks which are very similar to those proposed to successfully manage software engineering processes. We have to analyze requirements, think about the architecture and the design of proofs, and finally we have to implement them using a particular theorem prover. Additional work is necessary to produce a consistent and complete proof documentation in order to convince our customers and help them to believe both in the formalization and in the proof.

If we do not want to verify programs on machine code level, we have to trust in compilers. Their executable implementations have to guarantee that correctness properties of source programs are preserved, so that we can trust in the correctness of the generated target programs. But note that this opens up a similar question for the compiler and its implementation, which is a piece of (executable) software that we want to trust. In order to avoid circuli vitiosi, *i.e.,* never ending dependencies on unsafe executable software (of uncertain pedigree, [21]), we need for principle reasons some manual proof checking. Fortunately, we can push this effort entirely into compiler implementation correctness proofs and rigorously verify compiler executables to preserve adequate and reasonable correctness properties of source programs[1] [17].

We want to demonstrate the proposed modularization in source level and compiler correctness. For that we use two examples which fit together. In section 2 we sketch an industrial project on the verification of a safety-critical expert system. Its correctness proof has actually been checked by the ACL2 theorem prover [24]. Safety can be proved by a technique which we call (verified) runtime result verification [13]. In section 3 we informally collect requirements for realistic correct compilation. Section 4 will yield a mathematical framework which enables us to precisely define compiler correctness by preservation of source program correctness. In particular, we will sketch a mechanical proof of preservation of partial correctness for a toy compiler in ACL2 (section 5). The role of toy examples is crucial in order to finally succeed, and we will give some comments in section 6. We will end up with some lessons learned and some wishes for theorem provers used in software verification.

2 Verified Runtime Result Verification

The effort of proving the correctness of large software systems seems often not justifiable. Heuristics and programming tricks are necessary in order to to solve complex problems successfully and efficiently. Mathematical induction then often fails because the algorithms to be verified get too complex and tricky. This applies in particular to knowledge-based systems. However, if such systems are safety- or mission-critical, we rely on their correctness.

In [13] we propose a *checker-based* approach to software verification which exploits the idea of *runtime result checking* [4] for verification. It is applicable if partial correctness of the application suffices. Partial correctness can be proved by a-posteriori runtime result verification. Let us sketch an industrial software verification project [3] in which we exploit the technique of verified runtime result verification (checker-based

[1] The work reported here has been supported by the Deutsche Forschungsgemeinschaft (DFG) in the *Verifix* and *VerComp* projects on *Correct Compilers* and *Techniques for Compiler Implementation Verification*

program verification) in order to prove the correctness (soundness) of a safety-critical expert system. The system is called Relais Master [1, 25], and it is an expert system for computing test-plans for relay assembly groups that control railway systems. It is used in the engineering phase of such devices in order to (more automatically) provide support for hardware-in-the-loop tests. The test-plans are generated from circuit descriptions. Later, they are to be automatically executed by a *test-roboter* in order to check the correct electrical behavior of the device.

Although digital circuits get more frequently used today, at least many German railway systems are still controlled by relay blocks. In order to guarantee a sufficiently high level of safety, these devices need regular maintenance, and in particular, a re-certification of every single physical device is required within regular time intervals.

2.1 The Relais Master – A Safety Critical AI Tool

The Relais Master automates the time-consuming and error prone manual test-plan construction. The system has been developed by the German company DTK (Hamburg) in cooperation with the Laboratory for Artificial Intelligence (LKI) at the University of Hamburg, Germany. After scanning the circuit, the engineer transforms it into an internal graph representation, and the system then generates a set of measurements between terminal contacts of the device (the test-plan). Each measurement is augmented with information about the required *state* of relay contacts (*open* or *closed*). The test roboter will later pneumatically switch the relays. We have the following requirements for the generated test-plan:

- It should be *complete* for the circuit, *i.e.,* it should contain every test necessary to detect any combination of any number of defects in the relay group. (*soundness*)
- It should be "good" in the sense that it does not require too many relay switching operations, because the maintenance interval is mainly triggered by the estimated number of switches. (*quality*)

The first property is safety-relevant, whereas, fortunately, the second is not, although it is as important for practical usability and makes the use of AI techniques adequate for the problem. But for safety we only need soundness. In fact, this observation is crucial to our *checker-based* approach: Soundness can quite easily be checked for a given test-plan, whereas the test-plan generation is complicated and has to assure quality as well.

The input of the Relais Master is a circuit description which actually is manually constructed from a scanned circuit plan. Figure 1 shows a very small example and the Ascii representation of the internal object-oriented Relais Master data structure is a Lisp-s-expression of the form:

$$
\begin{array}{l}
\texttt{(((}R_1\ \texttt{released ((}c_1\ \texttt{closed) (}c_2\ \texttt{open)))}\\
\quad\texttt{(}R_2\ \texttt{released ((}c_3\ \texttt{closed) (}c_4\ \texttt{closed)))}\\
\quad\texttt{(}R_3\ \texttt{released ((}c_5\ \texttt{open))) (}R_4\ \texttt{released ((}c_6\ \texttt{open)))}\\
\quad t_1\ t_2\ t_3\ t_4\texttt{)}
\end{array}
$$

$$
\begin{array}{l}
\texttt{((}t_1\ c_1\texttt{) (}t_2\ c_6\texttt{) (}t_3\ c_2\ c_4\ t_4\texttt{) (}t_4\ t_3\ c_2\ c_4\texttt{) (}c_1\ t_1\ c_3\texttt{) (}c_3\ c_1\ c_5\ c_2\ c_4\texttt{)}\\
\quad\texttt{(}c_5\ c_6\ c_3\ c_2\ c_4\texttt{) (}c_6\ c_5\ t_2\texttt{) (}c_2\ t_3\ t_4\ c_4\ c_3\ c_5\texttt{) (}c_4\ t_3\ t_4\ c_2\ c_3\ c_5\texttt{))}
\end{array}
$$

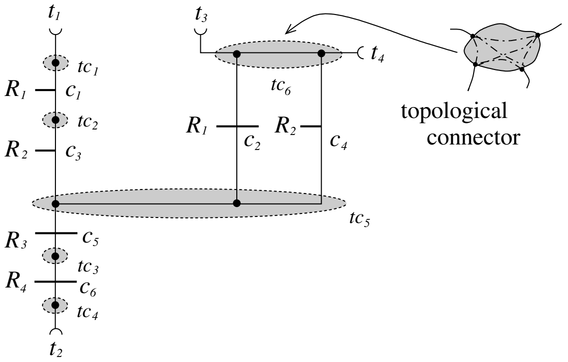


Fig. 1. A sample circuit. Relais R_1 has two contacts, c_1 and c_2, and R_2 has contacts c_3 and c_4. If R_1 is not excited (released), contact c_1 is *closed* (*i.e.,* is to conduct), whereas c_2 is *open* (*i.e.,* is to isolate). Topological connectors are abstractions from wiring and soldering, for n ports, at least $n-1$ connections have to conduct.

```
((con t3 t4 () (R1 R2 R3 R4)) (con t3 t1 () (R1 R2 R3 R4))
 (iso t2 t1 (R3) (R1 R2 R4)) (con t3 t2 (R3 R4) (R1 R2))
 (iso t3 t2 (R2 R3 R4) (R1)) (iso t1 t2 (R2 R3 R4) (R1))
 (con t3 t2 (R1 R2 R3 R4) ()) (iso t1 t2 (R1 R3 R4) (R2))
 (iso t2 t3 (R1 R4) (R2 R3))))
```

The first sub-list contains the contacts and terminals, the second is the graph structure of the circuit (every single node is followed by the list of direct neighbors), and the third list is the set T of measurements (conduction or isolation between two terminals, *e.g.* t_3 and t_4, with a certain state, *i.e.,* a set of excited and a set of released relays).

In the following, C will refer to the circuit description (the first two sublists), and T will refer to the third sublist, *i.e.,* the test-plan. C represents the input of the Relais Master, and T its output. C and T together form the input of the correctness predicate $check_Q$.

2.2 The Correctness Requirement

For simplicity, we just consider relay groups that contain terminals, connectors (wires, soldering points etc.) and relay contacts as *elements*. Diodes, resistors and capacitors are left out. The generated test-plan consists of *conduction* and *isolation* tests. Note however, that relay contacts are part of relays, so can not be switched independently, and the test-roboter can only measure between terminals (outside connectors). Since later the certification is supposed to be performed automatically, the generated test-plan and hence the Relais Master is *safety critical*. We need the following guarantee:

> *Every* defect of any relay contact, connector or terminal of the physical device will be detected by at least one measurement of the test-plan.

The combination of the generated tests must assure, if successful, that every single element is individually tested to work properly, *i.e.,* to conduct (terminals, connectors and relay contacts) and isolate (relay contacts) if supposed to. In that case, we call the test-plan T *complete for* the circuit C. The crucial point is, that this is a *partial* *correctness* requirement for the Relais Master program (R for short): If it successfully generates a test-plan T for a given circuit C, then we want T to be *correct*. Let $P(C)$ characterize regular circuit descriptions, and let $Q(T)$ define T to be *complete for* C. Then we may formalize the correctness requirement for R by

$$\{ P(C) \} \; T \texttt{ := R (} C \texttt{)} \; \{ Q(T) \} \,.$$

Partial correctness requirements are typical for tools used in the construction of safety critical applications. We do not want to prove that the tool never fails. But we want to guarantee that any given result is correct, *i.e.,* that the tool is partially correct w.r.t. the correctness requirement for its results.

2.3 Runtime Verification of Results - Getting the Idea

Let us assume a transformational program `y := `π`(x)` (or π for short) to be specified by pre- and post-conditions $P(x)$ and $Q(y)$ for inputs x and outputs y. Instead of proving the partial correctness (cf. [19, 22])

$$\{ P(x) \} \; y \texttt{ := } \pi \texttt{ (} x \texttt{)} \; \{ Q(y) \}$$

of the program π itself – *i.e.,* if P holds for x, and if π successfully terminates on x with result y, then Q holds for y – the idea is to modify the program and add a *checker predicate* which rejects incorrect results. That is to say, we construct a sufficient algorithmic formulation $check_Q$ of the post-condition Q and prove

$$check_Q\,(x,y) \;=\; true \;\Longrightarrow\; Q\,(y)\,.$$

We call this property the *checker correctness*. It is now a simple exercise to prove that the modified program π', which additionally checks the post-condition and rejects any incorrect result, is partially correct with respect to P and Q:

$$\{ P(x) \} \; y := \; \pi\,(x)\,; \; \textbf{if} \neg \, check_Q\,(x,y) \; \textbf{then abort fi} \; \{ Q(y) \}$$

Of course we additionally have to guarantee, that running π does not corrupt $check_Q$. In our case, we run a separate checker program on ASCII representations of x and y, not modifying the expert system at all.

2.4 The Checker and its Verification

In order to exploit the runtime result verification approach we just have to construct (and to verify) a predicate $check_Q$ on circuits C and test-plans T which checks T to be complete for C, *i.e.,* which guarantees that the postcondition $Q(T)$ holds. Thus, in order to verify the Relais Master expert system it remains to prove such a $check_Q$ to be *sufficient* to guarantee Q, *i.e.,*

$$check_Q\,(C,T) \;=\; true \;\Longrightarrow\; Q\,(T)\,.$$

We prove that the checker program returns $true$ only if the measures in T are sufficient to guarantee that every contact c_i in C is individually tested both to conduct, if it should be $closed$, and not to conduct, if it should be $open$. Terminals and single connections of topological connectors can be seen as special $closed$ contacts. Note, that for safety it is again sufficient to prove *partial correctness* of the checker, *i.e.,* the checker might fail even if Q holds. We have to prove, that the test-plan T is complete for the circuit C, if the checker returns *true* for C and T.

We cannot go into much detail of the program or the proof. The checker is a Lisp program. Inputs are s-expressions representing C and T (as in the above example). C is a graph with nodes for every element, T is a set of either conduction or isolation tests between two terminals in a given state of the relay contacts.

A successful conduction test between t_1 and t_2 guarantees *one* conducting path (we do not know which), whereas a successful isolation test guarantees *every* path to isolate. In both cases, however, this might well be due to a defect. The checker proceeds in five steps:

1. The circuit representation C is transformed into a handier internal graph representation.
2. Each test in T is replaced by the set of all paths between t_1 and t_2 in C together with its type.
3. Every contact of each path in a test is augmented with the corresponding should-be-state (open or closed).
4. Assuming all tests to succeed, we get true logical propositions about physical conduction of (sets of) paths in the concrete device; simple logical transformations give us true propositions about individually tested single elements.
5. Finally, we check that indeed every circuit element is individually tested. If not, the checker aborts.

The checker is written in the subset of applicative Common Lisp supported by the Boyer/Moore theorem prover ACL2 [24]. More details on the checker and on the correctness proof can be found in [3, 2]. The crucial part (step 4) can be seen as a problem specific tautology or model checking [7, 23]: Successful hardware-in-the-loop test according to the generated set T of measurements will guarantee a circuit dependent set of logical formulas expressing conduction or isolation of paths between terminal contacts to be true, and our checker program basically checks the particular formula

$$\bigwedge_{l=1}^{p} t_l \;\wedge\; \bigwedge_{j=1}^{k} tc_j \;\wedge\; \bigwedge_{i=1}^{n} (c_i^{closed} \wedge \neg\, c_i^{open})$$

(for all p terminals, k topological connectors and all n relay contacts) to be a logical consequence of that set of formulas: Every terminal, topological connector and relay contact conducts if closed, and every relay contact isolates if open. If so, the set T is *complete* for the given circuit, *i.e.,* the (finite) set T of *test cases* is sufficient for certification of the devices.

Runtime result verification [13, 35] is strongly related to program checking [4], and our case study is an application in the field of testing safety critical devices. However, our main focus is on verification of the checker program that guarantees correctness of the test case generator (the Relais Master expert system). In contrast to *e.g.* [20], where test case generation is based on model checking, our approach does not rely on particular techniques used in the expert system itself. Classical verification and also model checking might well find their limits for complex (and large) applications. We strongly believe that (verified) runtime result verification scales up in certain mission critical domains – not only for safety but also for security. The approach compares to Necula's proof-carrying code and certifying compilers [32, 33], but we check *sufficient* and not only necessary conditions for correctness.

Compared to the Relais Master expert system, the checker is a recursive correctness predicate of moderate size written in ACL2 Lisp. Its (partial) correctness is proved and the proof is mechanically checked by ACL2. However, in order to guarantee that we can trust test plans generated by execution of the checked Relais Master, we have to prove safety, *i.e.,* partial correctness, for the checker executable as well. And this brings us to the second running example: compiler verification.

3 Trusted Execution of Programs

Compilers are sequential transformational programs. A compiler takes a (syntactical representation of) a *source program* $\pi \in \mathbf{SL}$ as input and sometimes it successfully terminates and returns a *target program* $m \in \mathbf{TL}$. If so, we hope that m has something to do with π. Actually, we want a guarantee that m is a *correct implementation* of π. But this needs further explanation and we will go into some detail on the precise meaning of correct implementation later (section 4). A practical compiler, however, will fail in most cases. Actually, it will fail on nearly every source program (with the precise mathematical meaning of *nearly every*). There are usually infinitely many and arbitrarily large source programs, whereas the compiler will run on a finite machine with hard resource restrictions.

Thus, with trusted execution on a real machine in mind, we can not hope to be able to prove that a compiler (executable) will succeed on every proper source program. But we can prove that if it succeeds and returns a machine program m, then m is a correct implementation of π. The intuition behind this kind of requirements is crucial to many other transformational programs as well, like proof checkers, model or equivalence checkers, batch runs that move our money from one to another account and so on. We accept failures, but we do not want to see incorrect non-erroneous results [36]. This is in fact what we often depend on. Not more. In our every day experience using programs we observe them to fail with a segmentation fault or bus error, for instance due to lack of memory, a programmer's error, a compiler bug, or a misuse of an optimizing compiler under wrong assumptions. Although very annoying, we all live with software errors, but we all hope that the application programmers, compiler constructors, operating system designers and even hardware engineers have been sensible enough to detect and signal any such runtime error. Undetected errors might have harmful consequences, in particular if they are intentional, in which case we would call them a *virus* or *Trojan Horse* [9].

As a compiler constructor, we can not relieve the application programmer from the burden to prove applications correct. We cannot guarantee correctness of implementations for incorrect source programs. Actually we have to construct the compiler without any knowledge about the intended meaning of source programs. On the other hand, a compiler can not preserve every property of the source program. We have to negotiate on a contract between user, language designer and implementor that involves the concrete description of the concrete language to be implemented, the concrete description of the target machine, definitions of both which are sufficiently mathematically exact so that user and implementor can agree on them without misunderstanding. Once such a contract has been negotiated, however, all bets are off for properties of programs not mentioned in that contract.

Some Requirements

Let us collect some requirements for a realistic notion of correct implementation: It should

– handle resource limitations of the target machine,
– handle non-determinism of the source program semantics,
– allow for optimizations
– allow for trusted execution also of non-terminating reactive programs,
– and support full recursion and dynamic data types of source languages.

We often think of source programs to serve as specifications of their machine implementations and thus require that the target program should at least be as defined as the source program and hence be able to compute every source program result. That is to say, we require the classical implementation correctness notion of specification refinement. On the other hand, we often do not want to see incorrect results computed by target program executions. That is to say every target program result should be correct with respect to the source program semantics. If we summarize these two requirements, the target program should

– at least be able to compute every result which the source program specifies
– at most compute correct results with respect to the source program semantics

The careful reader will already have realized, that in general these two requirements contradict. They only meet in the rare cases where we can guarantee the target program to be defined if and only if the source program is, and in case of non-deterministic source program semantics we even would guarantee the target program to be able to compute every source program result. Implementations are often more deterministic and less defined than source programs. It depends on the application domain of user programs if we prefer a notion of correct implementation which corresponds to the first or to the second intuitive requirement.

For the every day programming situation a requirement corresponding to the second item above is the adequate correctness notion. It does not require well-definedness proofs for source programs and guarantees a well-placed trust in target computations even if source programs have not been verified.

> Thesis: We can *trust machine program execution* if we can guarantee partial correctness [19] for the machine program with sufficient mathematical rigor.

Note that in this context *to be undefined* means not to return any non-erroneous result at all, *i.e.,* to abort with a runtime error or not to terminate. Our intuitive compiler correctness requirement is to guarantee partial correctness of the generated machine program. But we did not yet say what kind of correctness property we expect for the source program in order to be able to give such a guarantee. This question actually opens up a deep and subtle discussion on source program properties which we want to preserve or need for instance for optimization (see [31, 17]). As long as we do not really depend on sophisticated optimizations, or on a termination guarantee for the target program, partial source program correctness is sufficient, which means that we require a correct compiler to *preserve partial program correctness* [12, 30].

There is a subtle difference to specification refinement (like in VDM [22]) or to former work on compiler verification using ACL2 resp. its predecessor Nqthm ([27, 8, 28]). J Moore proves in [28], that every non-erroneous result of (the Piton machine on) π will also be computed by m (on the FM9001), that m is more defined than π. This allows for optimizations, but trusted execution of m requires total correctness of π.

4 Compiler Correctness for Transformational Programs

Let us model the semantics of transformational programs by partial relations (multivalued partial functions) $f \subseteq D_i \times D_o$ between input domains D_i and output domains D_o. Thus f is an element $f \in (D_i \rightharpoonup D_o) =_{\text{def}} \mathbf{2}^{D_i \times D_o}$. Data in D_i and D_o are considered *regular* or *non-erroneous* input/output data or states. In order to handle irregular data, *i.e.,* finite and infinite errors, we assume that every data domain D is *extended* by an individually associated disjoint non-empty error set Ω, *i.e.,* $D^\Omega =_{\text{def}} D \ \dot{\cup}\ \Omega$ and $D \cap \Omega = \emptyset$. For every transformational program semantics f we assume an *extended* version

$$f \in ({D_i}^\Omega \rightharpoonup {D_o}^\Omega)$$

which we denote with the same symbol f. Errors are final. No computation will ever recover from an error [2]. Thus, we require (the extended) f to be *error strict*, *i.e.,* f to be total on Ω_i and $f(\Omega_i) \subseteq \Omega_o$.

However, we have to respect a further phenomenon. Errors are of essentially different types. They are either unavoidable and we have to accept them, or they are unacceptable and thus to be avoided. The implementation has to guarantee that acceptable errors are signaled and hence do not lead to unexpected incorrect results, whereas the absence of unacceptable errors has to proved by the user, for instance if she/he wants to use an optimizing compiler which omits array boundary checks. We will allow unacceptable errors to cause unpredictable (or chaotic) consequences. In order to model this phenomenon, we partition Ω in a non-empty set $A \subseteq \Omega$ of *acceptable* and a non-empty set $U =_{\text{def}} \Omega \setminus A$ of *unacceptable* or *chaotic* errors. So we require

$$\Omega_i = A_i \ \dot{\cup}\ U_i \quad \text{and} \quad \Omega_o = A_o \ \dot{\cup}\ U_o$$

[2] Exceptions are not errors in our sense. We think of exceptions as special cases of non-local regular control flow.

and a *strong* error strictness of f, namely that f is total on Ω_i and

$$f\,(A_i) \;\subseteq\; A_o \quad \text{and} \quad f\,(U_i) \;\subseteq\; U_o\,.$$

The error sets Ω are supposed to contain a particular standard error element $\perp$ which is to model infinite computation (divergence). We additionally require $(d, \perp_o) \in f$ whenever there is a non-terminating (infinite) computation of f starting with $d \in {D_i}^{\Omega}$.

Correct Implementation

Let f_s be source and f_t target program semantics, and let $\rho_i \in (\,D_i^{s\,\Omega} \rightharpoonup D_i^{t\,\Omega}\,)$ and $\rho_o \in (\,D_o^{s\,\Omega} \rightharpoonup D_o^{t\,\Omega}\,)$ be data representation relations between source and target input and output data. We require both relations, ρ_i and ρ_o, and their inverses ${\rho_i}^{-1}$ and ${\rho_o}^{-1}$, to be strongly error strict (in both directions).

$$\begin{tikzcd}
D_i^{s\,\Omega} \arrow[r, "f_s", harpoon] \arrow[d, "\rho_i"', harpoon] & D_o^{s\,\Omega} \arrow[d, "\rho_o", harpoon] \\
D_i^{t\,\Omega} \arrow[r, "f_t"', harpoon] & D_o^{t\,\Omega}
\end{tikzcd}$$

Fig. 2. Source and target program semantics f_s, f_t and data representations $\rho_i\,, \rho_o$

Definition 1 (Correct implementation or refinement). *f_t is said to be a* correct implementation *or* refinement *of f_s* relative *ρ_i and ρ_o and associated error sets iff*

$$(\,\rho_i\,;\,f_t\,)\,(d) \;\subseteq\; (\,f_s\,;\,\rho_o)\,(d) \,\cup\, A_o^t \tag{1}$$

holds for all $d \in D_i^s$ with $f_s\,(d) \subseteq D_o^s \cup A_o^s$ (that is with $f_s\,(d) \cap U_o^s = \emptyset$).

For any target program computation, the outcome d'' in $D_o^{t\,\Omega}$ is either an acceptable error output in A_o^t, or there exists a source program computation that either ends in a (regular) d' corresponding to d'' or that ends with an unacceptable (chaotic) error output in U_o^s. That is to say: If f_t is a correct implementation of f_s, then f_t either returns a correct result, or an acceptable error, or, if f_s can compute an unacceptable error, f_t may (chaotically) return any result.

If f_t correctly implements f_s relative ρ_i and ρ_o, we will write $\rho_i\,;\,f_t \sqsupseteq f_s\,;\,\rho_o$ or even shorter just $f_t \mathrel{\boldsymbol{\sqsupseteq}} f_s$ (with a boldface $\boldsymbol{\sqsupseteq}$). We indicate this diagram commutativity by the diagram in Figure 3 and we can easily prove that we can compose correct implementation diagrams both vertically and horizontally. Correct implementation is compositional, which is a very important fact for practical software engineering and in particular for compiler construction and bootstrapping (see [17] for details).

It is an important observation, that we can exactly characterize correct implementation by *preservation of relative correctness* [37], which generalizes Floyd's and Hoare's notions of partial respectively total program correctness. Let $f \in (\,{D_i}^{\Omega} \rightharpoonup {D_o}^{\Omega}\,)$ be

$$
\begin{tikzcd}
f_s \in \mathrm{Sem}_s \arrow[d, "\sqcap\!|"'] & : & D_i^{s\,\Omega} \arrow[r, "f_s"] \arrow[d, "\rho_i"'] \arrow[dr, phantom, "\sqsupseteq"] & D_o^{s\,\Omega} \arrow[d, "\rho_o"] \\
f_t \in \mathrm{Sem}_t & : & D_i^{t\,\Omega} \arrow[r, "f_t"'] & D_o^{t\,\Omega}
\end{tikzcd}
$$

Fig. 3. Commutative diagram expressing correct implementation

a program semantics and let $P \subseteq D_i$ and $Q \subseteq D_o$ be predicates, *i.e.,* subsets of regular data. f is called *(relatively) correct with respect to* (pre- and post-conditions) P and Q ($\langle\, P \,\rangle\, f \,\langle\, Q \,\rangle$ for short), iff

$$f(P) \;\subseteq\; Q \,\cup\, A_o \tag{2}$$

The following theorem says that f_t correctly implements f_s if and only if relative correctness of f_s implies relative correctness of f_t for all pre- and post-conditions P and Q. A proof can be found in [17], where we also discuss some variations of the notions defined here.

Theorem 1 (Preservation of relative correctness). *f_t correctly implements f_s ($\rho_i; f_t \sqsupseteq f_s; \rho_o$) if and only if for all $P \subseteq D_i^s$ and $Q \subseteq D_o^s$ we have*

$$\langle\, P \,\rangle\, f_s \,\langle\, Q \,\rangle \implies \langle\, \rho_i(P) \,\rangle\, f_t \,\langle\, \rho_o(Q) \,\rangle\, .$$

Relative program correctness generalizes the classical notions of partial or total program correctness. In order to see this, let $f \;\in\; (\, D_i \rightharpoonup D_o \,)$ be an original unextended program semantics and let $P \;\subseteq\; D_i$ and $Q \;\subseteq\; D_o$ be pre- and post-conditions. f is called *partially correct w.r.t.* P and Q ($\{P\}\, f\, \{Q\}$ for short), if $f(P) \;\subseteq\; Q$, *i.e.,* if the following holds for f: Whenever the precondition P holds for an input $d \in D_i$ and if $f(d) \subseteq D_o$ is defined, then any $d' \in f(d)$ fulfills the postcondition Q. And f is called *totally correct w.r.t.* P and Q ($[P]\, f\, [Q]$ for short), if f additionally is guaranteed to be defined on any $d \in P$, *i.e.,* if f is partially correct w.r.t. P and Q and the domain dom_f of f comprises P (that is if additionally $\mathrm{dom}_f \supseteq P$ holds).

Let us choose particular error sets $A = A_i = A_o =_{\text{def}} \{a\}$ and $U = U_i = U_o =_{\text{def}} \{u\}$ with acceptable error element a and unacceptable error element u for both domains D_i and D_o. Hence, $\Omega \;=\; \Omega_i \;=\; \Omega_o \;=_{\text{def}}\; A \;\dot{\cup}\; U$. Note that $\perp \;\in\; \{a, u\}$. We may now consider $\perp$ acceptable ($A_i = A_o =_{\text{def}} \{\perp\}$, $U_i = U_o =_{\text{def}} \{u\}$) or unacceptable ($A_i = A_o =_{\text{def}} \{a\}$, $U_i = U_o =_{\text{def}} \{\perp\}$). Relative correctness of the extended f^{ext} is equivalent to partial correctness of f

$$\langle\, P \,\rangle\, f^{ext} \,\langle\, Q \,\rangle \iff \{P\}\, f\, \{Q\}$$

if $\perp = a$ is considered acceptable, and it is equivalent to total correctness, if $\perp = u$ is considered unacceptable:

$$\langle\, P \,\rangle\, f^{ext} \,\langle\, Q \,\rangle \iff [\, P \,]\, f\, [\, Q \,]\, .$$

But our notion of correct (relative) implementation also generalizes well-known implementation correctness notions. If we consider $\perp$ acceptable, then

$$\rho_i^{ext} ; f_t^{ext} \sqsupseteq f_s^{ext} ; \rho_o^{ext} \quad\iff\quad \rho_i ; f_t \subseteq f_s ; \rho_o$$

exactly expresses preservation of partial program correctness, and if we consider $\perp$ unacceptable, on the other hand, then preservation of relative correctness is equivalent to preservation of total correctness (see again [17] for details).

In particular, if we come back to our earlier remarks, preservation of partial program correctness often suffices if we want to trust in target program execution and if we do not depend on sophisticated optimizations. It gives us a mathematically rigorous guarantee that target programs compute *at most* correct results. Hence, preservation of partial correctness ([12, 30, 29], or L-simulation [5]) is a practically motivated adequate special case:

Definition 2 (L-simulation). *Let π and m denote source and target programs with semantics $[\![\,\pi\,]\!]_{\text{SL}} \in (\, D_i^{\text{SL}} \rightharpoonup D_o^{\text{SL}}\,)$ and $[\![\,m\,]\!]_{\text{TL}} \in (\, D_i^{\text{TL}} \rightharpoonup D_o^{\text{TL}}\,)$, respectively. Then we say that the implementation step $\pi \mapsto m$* preserves partial correctness *(or that m* L-simulates *π), if and only if*

$$\rho_i \,;\, [\![\,m\,]\!]_{\text{TL}} \subseteq [\![\,\pi\,]\!]_{\text{SL}} \,;\, \rho_o \,. \tag{3}$$

That is to say: If the target program m is defined on the representation of a regular source program input, then the source program π can return a corresponding result as well. If π and hence m are deterministic, this means that π will return *the* corresponding result.

$$\begin{tikzcd}
{[\![\,\pi\,]\!]_{\text{SL}} \in \mathrm{Sem}_{\text{SL}} :} \arrow[d] & D_i^s \arrow[r, "{[\![\,\pi\,]\!]_{\text{SL}}}"] \arrow[d, "\rho_i"'] \arrow[dr, phantom, "\subseteq"] & D_o^s \arrow[d, "\rho_o"] \\
{[\![\,m\,]\!]_{\text{TL}} \in \mathrm{Sem}_{\text{TL}} :} & D_i^t \arrow[r, "{[\![\,m\,]\!]_{\text{TL}}}"'] & D_o^t
\end{tikzcd}$$

Fig. 4. Commutative diagram expressing preservation of partial program correctness

Corollary 1 (Preservation of partial correctness). *An implementation step $\pi \mapsto m$ preserves partial correctness if and only if m L-simulates π, i.e.,*

$$\begin{aligned} &\forall\, P, Q : (\, \{\, P \,\}\, \pi \,\{\, Q \,\} \implies \{\, \rho_i(P) \,\}\, m \,\{\, \rho_o(Q) \,\}\,) \\ &\qquad\iff\quad \rho_i \,;\, [\![\,m\,]\!]_{\text{TL}} \subseteq [\![\,\pi\,]\!]_{\text{SL}} \,;\, \rho_o \end{aligned}$$

Preservation of relative correctness gives us the necessary means to define adequate and more elaborate notions of correct implementation also for realistic correct compilation, not only of preservation of partial correctness, but also for optimizing compilers where target program inputs have to fulfill additional optimization (total correctness) conditions (do not cause particular runtime errors) for the target program to run trustworthily [17].

We may compare the previous section to a *requirements analysis* as of usual software engineering processes. We might even call it a requirements analysis in proof engineering for the correctness of compilers. The result is a mathematical framework which allows us to precisely and also formally express the theorem(s) which we have to prove in order to call a compiler correct.

5 Proving Compiler Source Level Correctness

Let us now give an example and sketch through a concrete compiler source level correctness proof which we mechanically ran through the Boyer/Moore theorem prover ACL2 [24].

In [9, 10] we prove preservation of partial correctness for a Lisp compiler, and we want to give a sketch and some brief remarks on source and target language, the compiler and its correctness proof. In [9] we define an abstract stack machine, and a compiler that generates stack machine code for a small subset of ACL2. The compiler is written in its own source language so that we can execute it within ACL2, either directly as an ACL2 function, or after compilation (into abstract machine code) by executing the machine model and running the compiled compiler (machine) program on it. The latter simulates machine execution of compiler executables within ACL2, and the main focus of [9] has been on that kind of (machine) program execution. We actually proved formally and rigorously that source level verification is not sufficient to finally guarantee correctness and hence trustworthiness of compiler executables. The compiler correctness proof is part of the most recent ACL2 distributions and described in detail in [10].

The source language **SL** is a subset of ACL2 Lisp similar to the language L3 of first order mutually recursive functions as of [26], but with s-expressions $s \in \mathsf{SExpr}$ as (dynamic) data. A program is a list `defs` of function definitions, followed by a list `vars` of input variables and a *main* program expression. The (operational) semantics $[\![\, \pi \,]\!]_{\text{SL}} \in (\,\mathsf{SExpr}^* \;\rightarrow\; \mathsf{SExpr}\,)$ is defined by an ACL2 function `evaluate` which (deterministically) maps `input` values to the result of the `main` expression after binding the inputs to the input variables. A termination argument, a natural number `n`, has been added in order to model strict partial functions within the ACL2 logic of total recursive functions (see [10] for more details).

The target language **TL** is the code of an abstract stack machine. Its configuration consists of a `code` part and a separate state `stack`, which is a data stack containing Lisp s-expressions. The machine has six machine instructions including a subroutine call and a structured conditional. Machine programs (m) are sequences of (mutually recursive) subroutine declarations (d) together with a *main* instruction sequence which is to be executed on an initial `stack` after downloading the list of declarations into `code`. The semantics $[\![\, \texttt{prog} \,]\!]_{\text{TL}} \in (\,\mathsf{SExpr}^* \;\rightarrow\; \mathsf{SExpr}^*\,)$ of machine programs is a `stack`-transformation defined by stepwise executing machine instructions. It is defined by an ACL2 function `execute` which executes a program `prog` on an initial `stack` which is supposed to contain input values on top. Again, a termination argument `n` is added in order to force the machine to stop execution after at most `n` subroutine calls.

The compiler `compile-program` is written in **SL** itself. It generates stack machine code according to the *stack principle*, *i.e.,* arguments are passed on the stack and a given expression e is compiled to a sequence of machine instructions pushing the value of e onto the stack leaving the stack below invariant. Functions or operators consume (pop) their arguments and push the result.

We prove that `compile-program` preserves partial correctness (Figure 5). The proof is mechanically checked by ACL2 and the complete ACL2 proof script is part of the recent ACL2 distribution. For those readers familiar with ACL2 let us cite the main theorem from [10]:

$$\begin{tikzcd}
(s_1,\ldots,s_n) \in \mathsf{SExpr}^* \arrow[r, "\texttt{(evaluate } \pi \cdot \texttt{n)}"] \arrow[d, "\rho_{tos}"'] & \mathsf{SExpr} \ni s \arrow[d, "\rho_{tos}"] \\
(s_n,\ldots,s_1,\ldots) \in \mathsf{Sexpr}^* \arrow[r, "\texttt{(execute } m \cdot \texttt{n)}"'] & \mathsf{Sexpr}^* \ni (s,\ldots)
\end{tikzcd}$$

Fig. 5. `compile-program` preserves partial correctness; the data representation relation ρ_{tos} relates s-expressions (or s-expression lists) to target machine `stacks` which contain the s-expression (list) on top (in reverse order)

```
(defthm compiler-correctness-for-programs
 (let ((new-stack (execute (compile-program prog)
                           (append (rev inputs) old-stack) n))
       (value (car (evaluate prog inputs n))))
  (implies (and (wellformed-program prog)
                (defined new-stack)
                (true-listp inputs) (equal (len vars) (len inputs)))
           (equal new-stack (cons value old-stack)))))
```

If the source program is well-formed, and if the target program (applied to an initial stack containing the correct number of `inputs` in reverse order on top) returns a non-erroneous result on top (is `defined`), then this result is `equal` to the semantics (`evaluate`) of the program applied to the inputs.

The crucial part of the proof, however, is to prove correctness of expression compilation. The correctness theorem for programs is, after some technical lemmas, a simple consequence of a similar theorem for correct expression compilation, applied to the main expression of the program. For correct expression compilation we prove two theorems simultaneously by induction. The first is the correctness theorem for forms (or expressions), and the second is the theorem for form lists. Both of them are very similar. Informally, for well-formed expressions and expression lists in well-formed programs we prove: If the machine, executed on a compiled expression (list), is defined on a `stack` for an `n`, then the following three conjectures hold:

1. The semantics of the expression (list) – in the given function environment and with the free variables bound to their values in the current stack-frame – is defined for the same n,
2. the machine returns a new stack with the value(s) of the expression(s) on top (in reverse order), and
3. the stack just below the result value(s) remaines unchanged.

The proof is actually a combined computational and structural induction on the termination argument n and the structural depth of the expression (in ACL2 terminology this is a well-founded induction on the ordinals). The induction is suggested by a large admissible ACL2 function which explicitly lists the entire set of induction hypotheses that we need for the proof to succeed.

We need a lot of lemmas. Most of them are technical, but some reflect crucial insights necessary to find the proof. So, for instance, it is interesting, that we actually have to prove the definedness of the source code semantics and the correctness of the result of machine execution simultaneously as well. The reason is the conditional. The definedness of the conditional inductively depends on the value of the condition. The conditional needs not be strict in both alternatives, and will not be for instance in recursive definitions. The conditional has actually been the challenging case to find the proof, not the function application as the reader might have expected. Function application is just captured by computational induction, whereas the conditional crucially influences the proof structure in the large. But we do not want to go into further detail. We refer to [9, 10] instead.

6 On the Impact of Toy Examples

In a sense, our example above is a toy example compared to the work necessary for the verification of realistic compilers for practical source programming languages and real target processors [17]. Our source language is very small, and the abstract target stack machine code is unrealistically abstract and far away from the ugly reality of real world processors. In that sense, [10] only presents an exercise. But the proof is an interesting exercise, and it contributes to realistic compiler verification in two important respects:

- It can be accomplished by the mechanical verification of subsequent compilation phases, so for instance (b) to a stack machine with a linear heap store (data refinement), (c) to linear assembly code, and (d) to real binary machine code of a concrete processor [15].
- In the context of the *Verifix* project on correct compilers [12, 15, 18], the proof has been generalized to a mechanical PVS [34] proof of preservation of partial correctness for a larger imperative source language [6].

The correctness of step (b) has already been proved manually, and the mechanical proof is nearly completed [11]. And step (c) and (d) are implemented [15, 16, 14] and designed to preserve partial correctness. Thus, in addition to being a nice exercise, our proof can be seen as part of a medium to large scale proof effort to provide an initial fully verified compiler executable that preserves partial source program correctness [15].

In the PVS proof we use structural operational semantics and inductive relations to formalize partial functions. It is interesting, that we can essentially reuse the proof idea and overall proof structure of the ACL2 proof presented here; The PVS proof also is by combined structural and computational induction. In our opinion, this is a good news with respect to proof engineering.

In both respects we benefit from the toy example. It shows that such a proof is manageable in principle, without premature introduction of realistic complexity. It actually has been the first mechanical proof of preservation of partial correctness. And such a proof is necessary in order to provide a mathematically rigorous proof also of the correctness of binary compiler executables. In fact, the latter has been the original motivation for the Verifix compiler correctness project: If we bootstrap compilers, we need a guarantee of the partial correctness of the generated compiler executable. We need to handle hard resource limitations of target machines and support for full recursion and dynamic data types of source languages.

7 Conclusions

Software verification can succeed, although it is often considered hard or even impossible – in particular for large and complex systems. Mechanical proof support can help finding proofs (and errors). It may help to redo (or reuse) proofs after slight modifications. And it might establish correctness. But in the latter case we depend on the prover. Its positive results, i.e., the successful proofs, should in principle be understandable and/or checkable, manually or by trusted proof checking programs. Otherwise we would leave a serious gap in the mechanically supported process of correct software construction and proof documentation.

Thus, we also have some wishes, in particular with respect to *proof checkability* and to *intuitive modeling*: First of all, in order to convince our customers, it is not only necessary that a mechanical proof finally succeeds. Once it does, the prover should also contribute to a consistent and complete proof documentation. If proofs are not mechanically checkable by trustworthy proof checkers, they should be readable so that we can check them manually. This is very similar to programming: It is at the end not sufficient that a program finally runs. Once it does, we also need documentation. We need to know and to communicate why.

The second wish is a bit more technical. Compiler verification deals with programming language semantics, and very often programs mean partial functions. In ACL2, we modeled them by error strict total functions, which is kind of tedious and always calls for an additional justification. In PVS, we use inductive relations, which are also a bit tedious to handle. In both cases, however, we need meta-arguments to justify the formalization. Although we know about the problems, we sometimes wish to have a logic of partial functions, and a supporting theorem prover which is as usable and elaborate.

Acknowledgments We would like to thank our colleagues in the *Verifix* project for, in particular Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich von Henke, Ulrich Hoffmann, Hans Langmaack, Vincent Vialard, Wolf Zimmermann. Special thanks to Hans Langmaack, and to Markus Müller-Olm and Andreas Wolf for their contributions to the notion of relative program correctness and its preservation. We thank Roy Bartsch for his work and Uwe Haferstroh, Sven Nordhoff and DTK for supporting the work on the Relais Master case study.

References

1. RelaisMaster Documentation and Manuals. DTK - Gesellschaft für technische Kommunikation MBH, Palmaille 82, Hamburg, Germany, 1996.
2. R. Bartsch and W. Goerigk. Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System (Abstract). In *AAAI Workshop on Model Based Validation of Intelligence MBVI'2001*, Stanford, CA, U.S.A., March 2001.
3. Roy Bartsch. Mechanisch verifizierte Programmprüfung für die Korrektheit von Prüfplänen in der Bahntechnik. Master's thesis, Institut für Informatik, CAU, Kiel, 2000.
4. M. Blum, M. Luby, and R. Rubinfeld. Program result checking against adaptive programs and in cryptographic settings. In *DIMACS Workshop on Distributed Computing and Crypthography*, 1989.
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.
6. Axel Dold and Wolfgang Goerigk. Proving Preservation of Partial Correctness in PVS. Technical report, University of Ulm, 2000. In preparation.
7. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.
8. A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation*. PhD thesis, University of Texas at Austin, 1992.
9. W. Goerigk. Compiler Verification Revisited. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.
10. W. Goerigk. Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof. In M. Kaufmann and J S. Moore, editors, *Proceeding of the ACL2'2000 Workshop*, University of Texas, Austin, Texas, U.S.A., October 2000.
11. W. Goerigk. Trusted Program Execution. Habilitation thesis. Techn. Faculty, Christian-Albrechts-Universität zu Kiel, May 2000. To be published.
12. W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.-W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.
13. W. Goerigk, Th. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.
14. W. Goerigk and U. Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU, October 1998.
15. W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 122 – 136, 1998.

16. W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998.
17. W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Technical Report 2015, Institut für Informatik, CAU, 2001.
18. G. Goos and W. Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201 – 230. Springer Verlag, 1999.
19. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
20. T. Jeron and P. Morel. Test generation derived from model-checking. *Lecture Notes in Computer Science*, 1633:108–??, 1999.
21. C. Jones, R.E. Bloomfield, P.K.D. Froome, and P.G. Bishop. Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP). Contract Research Report 337/2001, Health and Safety Executive, Adelard, London, UK, 2001.
22. C. B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.
23. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
24. M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.
25. H. Lange, R. Möller, and B. Neumann. Avoiding Combinatorial Explosion in Automatic Test Generation: Reasoning about Measurements is the Key. In *Proceedings of KI'96 Conference on Artificial Intelligence*, Dresden, 1996. Springer Verlag.
26. Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition)*. John Wiley and Sons, New York, N.Y., 1987.
27. J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.
28. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Press, Dordrecht, The Netherlands, 1996.
29. M. Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, October 1996.
30. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.
31. M. Müller-Olm and A. Wolf. On Excusable and Inexcusable Failures: Towards an Adequate Notion of Translation Correctness. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of Formal Methods FM'99*, volume 1709 of *LNCS*, pages 1107–1127, Toulouse, France, 1999. Springer.
32. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 January 1997.
33. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, 17–19 June 1998.
34. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.
35. A. Pnueli and P. Traverso, editors. *Proceedings of the FLoC'99 International Workshop on "Runtime Result Verification"*, Trento, Italy, 1999.

36. W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.
37. A. Wolf. *Weakest Relative Precondition Semantics - Balancing Approved Theory and Realistic Translation Verification*. PhD thesis, Technische Fakultät der Christian-Albrechts-Universität, Kiel, 2001.

A Logic for Rewriting Strategies*

Richard B. Kieburtz

Oregon Graduate Institute
20000 N.W. Walker Road
Beaverton, OR 97006 USA
dick@cse.ogi.edu

Abstract. Rewriting strategies can become quite complex and are not easy to comprehend or reason about when they are expressed in operational terms. This paper develops a *weakest precondition* logic for reasoning about strategies programmed in the strategy language *Stratego*. This logic embeds the modal mu-calculus, allowing it to express properties of terms of arbitrary depth. Its use is illustrated by characterizing properties of several reduction strategies for the lambda calculus with explicit substitutions.

1 Introduction

Strategies for term rewriting are widely used to implement syntactic theories in systems for automated deduction, including theorem-proving and program transformation. Strategies evaluate conditions for the application of rewrite rules, determine the order in which subterms are explored, and prescribe bindings and scope of pattern variables. Formulating strategies is a programming task that can be as complex as any other that we know. It can be made easier with appropriate programming language support and better understood through logical characterization.

This paper is a first step towards defining a programming logic for strategies. Stragegies are understood as programs over a domain of terms. Control of strategies is accomplished with recursion and nondeterministic choice. A *weakest-precondition* logic furnishes a natural formalism for reasoning about such programs. However, predicates in this logic are interpreted over a domain of term structures. As a logic for terms, we have adopted the μ-calculus [7], enriched with modalities that express path quantification in terms.

Rules have been developed in this logic for the constructions of *Stratego* [12, 13, 15, 14], a domain-specific language designed specifically for programming strategies. *Stratego* provides a compositional semantics with explicit recursion, allowing strategies to be applied at sites deep within terms. Strategies, and therefore patterns, are first-class constructs of *Stratego*. As in a logic programming language, conditional control in *Stratego* is based upon the success or finite failure of strategies, rather than if-then-else expressions that test conditions coded as boolean values. Although essentially a

* This research was done while the author was a Visiting Professor at INRIA-Rocquencourt laboratory. Partial support was provided by the National Science Foundation under Grant No. CCS-9803880.

first-order language, *Stratego* also supports a particular form of higher-order strategies, namely *term congruences*, which lift term constructors into strategy constructors.

Section 2 introduces a weakest-precondition (*wp*) logic for strategies. In Section 3, several strategies for reducing terms in the lambda-calculus are proposed and properties of reduction strategies are characterized in the *wp* logic. Section 4 discusses related work and Section 5 presents conclusions.

2 A weakest-precondition logic for strategies

Rewriting strategies are designed to produce terms that exhibit particular forms, by a directed series of rewriting steps. To reason about strategic rewriting, we'd like to know a set of input terms from which a given rewriting strategy is assured to produce an output term in a specified form. A predicate characterizing the largest set of such input terms is a *weakest precondition* for the strategy to produce a specified form of output.

In a weakest-precondition logic, each rewrite rule or strategy is characterized by a *predicate transformer*, a function from predicates to predicates. Since predicates characterize sets in a given universe, we can think of a *wp*-logic as interpreted in relations over a universe.

The universe we have in mind is a Herbrand universe of terms generated by a finite signature, Σ. Call this universe $\mathcal{T}(\Sigma)$. Predicates are interpreted as subsets of this universe, by an interpretation function, $\mathcal{I} : \mathit{Pred} \to \mathcal{T}(\Sigma)$. The distinguished predicates *True* and *False* have the interpretations $\mathcal{I}(\mathit{True}) = \mathcal{T}(\Sigma)$ and $\mathcal{I}(\mathit{False}) = \emptyset$. Term variables in the logic range over the universe.

Connectives ($\neg$), ($\vee$), ($\wedge$) and ($\Rightarrow$) are used to form compound predicate formulas. They have the interpretations

$$\begin{aligned} \mathcal{I}(\neg P) &= \{t \mid t \notin \mathcal{I}(P)\} \\ \mathcal{I}(P \vee Q) &= \mathcal{I}(P) \cup \mathcal{I}(Q) \\ \mathcal{I}(P \wedge Q) &= \mathcal{I}(P) \cap \mathcal{I}(Q) \\ \mathcal{I}(P \Rightarrow Q) &= \mathcal{I}(P) \subseteq \mathcal{I}(Q) \end{aligned}$$

In the sequel, we shall indulge in a common abuse of notation by using predicates to denote the sets that they characterize.

If s is a strategy for term rewriting, then by $\mathit{wp}_s : \mathit{Pred} \to \mathit{Pred}$ we denote the predicate transformer associated to s. The expression $\langle s \rangle\, t$ denotes the application of strategy s to term t. We denote by $\mathit{Dom}(s)$ the set $\{t \mid \langle s \rangle\, t \in \mathcal{I}(\mathit{True})\}$. That is, $\mathit{Dom}(s)$ is the set on which strategy s succeeds.

We also require a predicate characterizing terms on which a strategy, s, fails. This requires a bit more subtlety than just taking the complement of $\mathit{Dom}(s)$, because finite failure of a strategy is used for control, whereas failure by nontermination obviously cannot be. Denote by $\overline{\mathit{Dom}(s)}$ the set on which strategy s fails finitely. In case strategy s terminates uniformly, $\mathit{Dom}(s) \vee \overline{\mathit{Dom}(s)} = \mathit{True}$.

2.1 Rules of *wp* logic for *Stratego*

The weakest-precondition logic satisfies several rules, including

$$\frac{P \Rightarrow Q}{wp_s(P) \Rightarrow wp_s(Q)}$$

$$wp_s(\mathit{True}) = \mathit{Dom}(s) \qquad wp_s(\mathit{False}) = \mathit{False}$$

$$wp_{s;t}(P) = wp_s(wp_t(P))$$

$$wp_{s<+t}(P) = wp_s(P) \ \vee \ (wp_t(P) \wedge \overline{\mathit{Dom}(s)})$$

$$wp_{s+t}(P) = (wp_s(P) \wedge \overline{\mathit{Dom}(t)}) \ \vee \ (wp_t(P) \wedge \overline{\mathit{Dom}(s)})$$

$$wp_s(\neg P) \wedge wp_s(P) = \mathit{False}$$

The rules for alternatives combinators (+) and ($<+$) are those of *committed choice*. With committed choice, a potential choice strategy will not be effective on a term if there is an alternative choice that might either have succeeded or failed to terminate when applied to the current term.

A weakest precondition is intended to characterize the largest set from which a given strategy can be assured to produce a term satisfying a specified postcondition. However, this is a *constructive* interpretation. If $t \in wp_s(P)$, then the strategy application $\langle s \rangle\, t$ is accepted as constructive evidence of a term that satisfies P. Thus it should come as no surprise that the characterization it gives for nondeterministic choice (+) is rather weak. For example,

$$\begin{aligned} wp_{s+s}(P) &= (wp_s(P) \wedge \overline{\mathit{Dom}(s)}) \vee (wp_s(P) \wedge \overline{\mathit{Dom}(s)}) \\ &= wp_s(P) \wedge \overline{\mathit{Dom}(s)} \\ &= \mathit{False} \end{aligned}$$

Since the strategy $s + s$ specifies a nondeterministic choice between two strategies that have exactly the same domain, there is no way to determine which of the two strategies is applied to produce a result. Thus interpreting an application of either strategy as evidence of a result would be constructively unsound.

We call this constructive interpretation the *weakest discriminating precondition*, because it specifies a domain on which the component strategies of a choice can be discriminated. The weakest discriminating precondition $wp_{r+s}(P)$ can supply definite information only over a domain on which strategies r and s both terminate but cannot both succeed.

Although it is tempting to give a more optimistic interpretation of nondeterministic choice, such an interpretation could give rise to difficulties. In a so-called *angelic* interpretation of nondeterminism, $wp_{s+s}(P) = wp_s(P)$, although that it cannot be said which of the possible choices succeeded in producing a result. This is incompatible with a committed choice semantics.

In the sequel, we shall only consider weakest discriminating preconditions.

2.2 Variables and environments

Thus far, we have discussed strategies as if only ground terms were transformed. However, the real power of rewriting strategies is only realized when we consider terms with variables. Term variables range over the ground terms in a universe. A term with variables may be valued as a ground term by providing an environment in which its variables are bound. A strategy can have the effect of binding variables in an environment, as well as transforming the term to which it is applied into a new form.

In *Stratego*, a binding environment for terms is implicit in every strategy. To express a property of a term, we shall need to express some properties of the environment.

Formally, an environment is a list of binding pairs, $[(x_1, t_1),\ (x_2, t_2), \ldots,\ (x_n, t_n)]$, in which $x_1,\ x_2, \ldots,\ x_n$ designate variables and $t_1,\ t_2, \ldots,\ t_n$ are ground terms. We shall write $[(x, t) \mid e]$ to designate an environment in which the pair (x, t) occurs at the head of the list. Environments are represented as lists rather than sets because a variable binding may be shadowed by the addition of new bindings of the same variable.

The fundamental judgment form is term equality, relative to an environment. Axioms of the judgment form include the usual axioms of equality, plus

$$[(x, t) \mid e] \ \models\ x = t \ :\ \mathit{True}$$

$$\frac{e \ \models\ x = t \ : \mathit{True}}{[(y, t') \mid e] \ \models\ x = t \ :\ \mathit{True} \qquad (y \neq x)}$$

We say that an environment e_1 *refines* an environment e_2, writing $e_1 \prec e_2$, if e_1 is compatible with e_2 on all bindings visible in e_2, but it may also contain additional bindings.

$$e_1 \prec e_2 =_{\mathit{def}} \forall x, t.\, e_2 \ \models x = t \ :\ \mathit{True} \ \Rightarrow\ e_1 \ \models x = t \ :\ \mathit{True}$$

A weakest precondition asserts a predicate characterizing a set of (term, environment) pairs. It asserts properties of terms containing variables in a context in which the variables are bound.

Properties of elementary strategies will be characterized in terms of a small-step semantics. In the formulation below, $\Xi_{t,e}$ denotes the characteristic predicate of a set comprised of a single pair, $\{(t, e)\}$. The weakest precondition for a predicate P to hold of the result of a strategy, s is defined in terms of a predicate transformer, wp'_s, restricted to the transformation of singleton predicates. This elementary predicate transformer can be defined directly with term equality judgments for each primitive strategy.

This formulation allows the question of admissibility of a predicate to be considered separately from the formulation of *wp*-rules. We say that a predicate, P, is *admissible* if its satisfiability can be defined inductively from a basis of term-equality judgments.

$$\mathit{Admissible}(P) \iff \mathit{wp}_s(P) = \{(t, e) \mid \exists t_1.\, (\mathit{wp}'_s(\Xi_{t_1,\emptyset}) \Rightarrow \Xi_{t,e}) \ \wedge\ \emptyset \models t_1 : P\}$$

The above definition allows us to calculate the weakest precondition of a composition of two strategies:

$$\begin{aligned}
\mathit{wp}_{r;s}(P) &= \mathit{wp}_r(\mathit{wp}_s(P)) \\
&= \mathit{wp}_r(\mu\xi.\, \{(t, e) \mid \exists t_1.(\mathit{wp}'_s(\Xi_{t_1,\emptyset}) \Rightarrow\ \Xi_{t,e}) \wedge \\
&\qquad\qquad \emptyset \models t_1 : P \ \wedge\ e \models t \ :\ \xi\}) \\
&= \mu\xi.\, \{(t, e) \mid \exists t_1, t_2, e_2.\, (\mathit{wp}'_r(\Xi_{t_2,e_2}) \Rightarrow \Xi_{t,e}) \wedge \\
&\qquad\qquad (\mathit{wp}'_s(\Xi_{t_1,\emptyset}) \Rightarrow \Xi_{t_2,e_2}) \wedge \\
&\qquad\qquad \emptyset \models t_1 \ :\ P \ \wedge\ e \models t \ :\ \xi\}
\end{aligned}$$

in which μ designates the least fixed-point binding operator of the μ-calculus [7].

This *wp* composition rule shows how a composition of strategies propagates refinements of an environment.

Pattern-matching and term-building strategies Elementary strategies of *Stratego* include *pattern-matching*, which succeeds on terms that unify with the pattern given in the strategy, and *term-building*, which creates a new term, using a pattern given in the strategy and the bindings found in the current environment. Predicate transformations for the elementary strategies of pattern-matching and term building are:

$$wp_{?t}(P) = \{(t', e \mid \exists e'.\, e' \prec e \,\wedge\, e' \,\models t' = t \,:\, P\}$$

$$wp_{!t}(P) = \{(t', e) \mid e \,\models t \,:\, P\}$$

A pattern strategy succeeds if an initial term and the given pattern can be matched by instantiating variables. When it succeeds, it produces a refinement of the initial environment, binding variables that occur in the initial term or the pattern. The weakest precondition for a pattern strategy characterizes a set of initial term-environment pairs for which the term matches the pattern and satisfies the asserted postcondition.

The term-building rule does not introduce new bindings. It characterizes those environments in which the form given in the term-building strategy can satisfy the asserted postcondition. Since the result of a term-building strategy does not depend upon the initial term, the weakest precondition imposes no restriction on initial terms.

The *test*(*s*) strategy The strategy *test*(s) succeeds whenever the strategy s succeeds, but it does not commit the bindings of variables made by s and restores the original term. The weakest precondition can be expressed as

$$wp_{test(s)}(P) = \{(t,e) \mid (t,e) \in wp_s(True) \,\wedge\, e \,\models t \,:\, P\}$$

Restricting the scope of variables New variables may be introduced in the scope of a particular strategy. The notation $[(x,\perp) \mid e]$ indicates an environment in which x is a new (unbound) variable that shadows any previous binding for a variable of the same name. Then

$$wp_{\{x_1,\ldots,x_m:\, s\}}(P) = \{(t,e) \mid (t, [(x_1,\perp),\ldots,(x_m,\perp) \mid e]) \in wp_s(P)\}$$

Derived strategies Many complex strategies can be defined in terms of these basic binding and building strategies. For instance, a strategy that applies a given strategy, s to a specific term is $!t;\ s$. This strategy can be written in a function-application style with the syntax $\langle s\rangle\, t$.

Another example is a *rule* in *Stratego*, which has the (sugared) syntax $\backslash\ p \ \rightarrow t$ **where** $r\backslash$. A rule is defined in terms of the compound strategy $\{x_1,\ldots,x_n\} :?p; r; !t$, with the side condition that $FV(p) \cup FV(r) \cup FV(t) \subseteq \{x_1,\ldots,x_n\}$, i.e. the rule contains no free occurrences of term variables. The weakest-precondition transformer for such a rule is the composition of the weakest-precondition transformers of its components.

2.3 Strategies for control

The cut strategies—*not, try* and *repeat* Three *Stratego* strategy constructors allow a compound strategy to succeed after an argument strategy has failed. The first such strategy requires failure of its argument; the other two always succeed if the argument strategy terminates. The corresponding logical rules are:

$$
\begin{aligned}
wp_{not(s)}(P) &= P \wedge \overline{Dom(s)} \\
wp_{try(s)}(P) &= wp_s(P) \ \vee \ (P \wedge \overline{Dom(s)}) \\
wp_{repeat(s)}(P) &= \mu\xi.\ wp_s(\xi) \ \vee \ (P \wedge \overline{Dom(s)}) \\
&= \lim_{n\to\infty} \bigvee_{k=0}^{n} wp_{s^k}(P \ \wedge \ \overline{Dom(s)})
\end{aligned}
$$

in which s^k denotes a k-fold repetition of the strategy s. To establish satisfiability of $wp_{repeat(s)}(P)$, one must demonstrate that there is a finite index k, for which $wp_{s^k}(P \wedge \overline{Dom(s)})$ is satisfied. It is necessary to show that the strategy terminates in order to establish that the *wp* formula is satisfiable.

2.4 CTL—a term logic

A logic capable of characterizing strategies must be able to express properties of the substructure of terms. For such a capability we turn to *Computational Tree Logic* (CTL)[3, 6]. CTL is a modal logic conceived originally as a *branching-time* temporal logic. Nodes of a tree can be interpreted as the possible future states of a system as time progresses. The root of the tree represents the current state. Each path from the root represents a possible trajectory of the system being modeled.

However, characterizing possible future trajectories of a system is only one interpretation that can be made of a CTL formula. Its essential aspect is that it allows quantification of assertions independently along two dimensions of a tree—along a path, which may be finite or infinite, and across alternate paths, which are only finitely branching.

The along-paths, or depth quantifiers of CTL are **G**, read “globally”, which quantifies (universally) over all subterms along a path that descends from the root of a tree, and **F**, read “eventually”, which selects (existentially) a term somewhere along a path. Added to these is the specific along-path quantifier **X**, read “child”, which selects the immediate subterm of the root along a given path.

The path, or breadth quantifier **A**, read “all paths”, quantifies over all paths descending through a tree from its root, and **E**, read “some path”, existentially selects a path from the root. Used together, the depth and breadth quantifiers allow one to express specific properties of a term and its subterms.

2.5 The modal mu-calculus

CTL modalities allow us to express logical formulas interpreted over terms, with separate quantifications over paths (breadth of a term) and levels (depth of a term). However, there are cases in which we should like to express depth quantification in a more detailed way. The μ-calculus [7] is a classical logic that provides least and greatest fixed-point binding operators (denoted by symbols μ and ν, respectively) well suited to expressing depth quantification. The *modal* μ-calculus is the basic μ-calculus enriched with the modal quantifiers **A** and **E** which express path quantification in a tree and the modal operator **X** to designate a property of immediate subterms.

Uses of the CTL depth quantifiers **G** and **F** can be expressed in terms of fixed-point expressions in the modal μ-calculus. For example, the CTL formula $\mathbf{AG}\, P$ (everywhere P) is logically equivalent to the formula $\mu\xi.\, P \wedge \mathbf{AX}\,\xi$ in the modal μ-calculus, and $\mathbf{EF}\, P$ (somewhere P) is equivalent to $\mu\xi.\, P \vee \mathbf{EX}\,\xi$. In the following sections, we shall use modal μ-calculus formulas to express weakest-preconditions of recursive strategies over terms.

2.6 Path quantification by term congruence

Path quantification can also be made more explicit by referring to paths directed through specific arguments of constructed terms. For example, a *Let* constructor (see Sec. 3) takes a triple of arguments, each of which is given a different interpretation in a language that embeds *Let* expressions. One might wish to quantify with respect to the last two arguments, omitting quantification over the first argument.

A mechanism that can express such selective quantification is *term congruence*, a device already employed to define strategies in the *Stratego* language. A term congruence lifts a term constructor to a constructor in another domain. For instance, the *Let* constructor has the signature

$$\mathit{Let} : \mathit{String} * \mathit{Expr} * \mathit{Expr} \rightarrow \mathit{Expr}$$

where *String* and *Expr* are sorts of type *Term*. When lifted to a domain of predicates, its signature becomes

$$\mathit{Let} : \mathit{Pred} * \mathit{Pred} * \mathit{Pred} \rightarrow \mathit{Pred}$$

Thus we can write $\mathit{Let}(P, P, P)$ to express the proposition $\mathit{Dom}(?\mathit{Let}(\_,\_,\_)) \wedge \mathbf{AX}\, P$. However, term congruence also permits one to express a property that is more specific with respect to paths, such as $\mathit{Let}(\mathit{True}, P, P)$, which asserts the predicate P over only the second and third subterms of a *Let* construction.

2.7 Weakest preconditions of non-local strategies

When a modal logic is interpreted over terms, a weakest precondition for success of a local strategy can be extended to characterize a global strategy whose effects occur throughout a term. For example, the strategy constructor *all*( ) applies a strategy s, given as its argument, to the children of a top-level term constructor and succeeds if and only if s succeeds at every one of the children. The weakest precondition for this strategy construction is expressed by

$$wp_{all(s)}(\mathbf{AX}\, P) = \mathbf{AX}\,(wp_s(P))$$

The weakest precondition for the strategy construction *some* to succeed on at least one term is

$$wp_{some(s)}(\mathbf{EX}\, P) = \mathbf{EX}\,(wp_s(P))$$

However, we often wish to make a stronger assertion about the result of applying a strategy constructed with *some*, one that accounts for its "greedy" nature. This is captured by the weakest precondition for a strategy *some*(s) to establish a condition P uniformly for all children of a node:

$$wp_{some(s)}(\mathbf{AX}\,P) = \mathbf{EX}\,(wp_s(P)) \,\wedge\, \mathbf{AX}\,(wp_s(P) \,\vee\, (P \,\wedge\, \overline{Dom(s)}))$$

A *bottom-up* strategy construction applies a strategy, s, to all subterms of a given term in bottom-up order. Thus an intermediate form might consist of a term, each of whose children had already been transformed by an application of *bottom-up*(s). A bottom-up strategy succeeds if and only if the argument strategy succeeds at every subterm. Its definition in *Stratego* is

```
bottom-up(s) = rec r(all(r); s)
```

Suppose the expected result of a bottom-up strategy is a term that satisfies a common property, P, throughout. A bottom-up strategy is characterized by a weakest-precondition defined as a least fixed-point:

$$wp_{bottom\text{-}up(s)}(\mathbf{AG}\,P) = \mu\xi.\,\mathbf{AX}\,\xi \,\wedge\, (\mathbf{AX}\,P \Rightarrow wp_s(P \,\wedge\, (\mathbf{AX}\,P)))$$

The implication expresses the condition that the common property P at every subterm is a sufficient precondition for the strategy s to succeed and establish the property P of the resulting term.

Analogously, a *top-down* strategy construction applies its argument to the subterms of a given term in top-down order. Its definition in *Stratego* is

```
top-down(s) = rec r(s; all(r))
```

Like a bottom-up strategy, a top-down strategy may also produce a result term characterized by a common property that holds throughout. The top-down strategy is characterized by:

$$wp_{top\text{-}down(s)}(\mathbf{AG}\,P) = \mu\xi.\,wp_s(\mathbf{AX}\,\xi \,\wedge\, (\mathbf{AX}\,P \Rightarrow P))$$

The strategy constructors *somebu* and *sometd* are similar, but only require the strategy application to children of a node to succeed on at least one, rather than all of the children. The *somebu*(s) and *sometd*(s) strategies succeed if there are one or more paths from the root of a term clear through to its fringe, along which the strategy s succeeds. Logical characterizations of these two strategies are:

$$wp_{somebu(s)}(\mathbf{EG}\,P) = \mu\xi.\,\mathbf{EX}\,\xi \,\wedge\, (\mathbf{EX}\,P \Rightarrow wp_s(P \,\wedge\, (\mathbf{EX}\,P)))$$

$$wp_{sometd(s)}(\mathbf{EG}\,P) = \mu\xi.\,wp_s(\mathbf{EX}\,\xi \,\wedge\, (\mathbf{EX}\,P \Rightarrow P))$$

To express that strategies *somebu*(s) or *sometd*(s) should produce a term with a property that holds everywhere, the weakest precondition must allow the possibility that the asserted property already holds in subterms on which the parameter strategy, s, does not succeed. For *sometd*, this is:

$$wp_{sometd(s)}(\mathbf{AG}\,P) = \mu\xi.\,wp_s(\mathbf{EX}\,\xi \,\wedge\, (\mathbf{EX}\,P \Rightarrow P) \,\wedge\, \mathbf{AX}\,(\xi \,\vee\, \mathbf{AG}\,P))$$

3 Example: Characterizing reduction strategies for lambda-calculus

As an example, let's consider reduction strategies for the lambda calculus with explicit substitution. An explicit substitution calculus affords more opportunities for control in reduction than does the calculus with implicit substitution. We begin with a signature for lambda terms, written in *Stratego*:

```
module lambda
signature
  sorts Expr
  constructors
    Var : String -> Expr
    Abs : String * Expr -> Expr
    App : Expr * Expr -> Expr
    Let : String * Expr * Expr -> Expr
```

Reduction rules of the calculus are given in the module, `lambda-rules`, printed below.

The rules `Alpha` and `Beta` are conversion/reduction rules of the lambda calculus. These rules suspend substitutions in the form of `Let` constructions. The *Stratego* library strategy `new` is a term-builder that upon each invocation, generates a new identifier not previously occurring in any term.

Rules `LetVar, LetApp` and `LetAbs` implement substitution of a given term for all free occurrences of a specified variable in a host term. These rules constitute a standard formulation of lambda-calculus with explicit substitution.

```
module lambda-rules
imports lambda lib
rules
  // lambda calculus rules
  Alpha : Abs(x,e) -> Abs(y,Let(x,Var(y),e))
                      where new => y

  Beta : App(Abs(x,m),n) -> Let(x,n,m)

  // Let distribution -- the substitution rules
  LetVar : Let(x,e,Var(x)) -> e
  LetVar : Let(x,e,Var(y)) -> Var(y)
           where <not(eq)> (x,y)

  LetApp : Let(x,e,App(m,n)) -> App(Let(x,e,m),Let(x,e,n))

  LetAbs : Let(x,e,Abs(x,m)) -> Abs(x,m)

  LetAbs : Let(x,e,Abs(y,m)) -> Abs(z,Let(x,e,n))
           where <Alpha> Abs(y,m) => Abs(z,n)
```

In the module lambda-red, we formulate three different strategies for reduction in the lambda calculus, using the set of rules given in lambda-rules. All use a common substitution strategy, subst, the first strategy declared in the module.

```
module lambda-red
imports lambda-rules
strategies
  // a strategy to eliminate Let constructions by forcing substitution
  subst = rec r (Let(id,id,r)
                <+ sometd(LetVar + LetApp + LetAbs))

  // various strategies for reduction:
  left-outer = rec r (Beta + subst + App(r,id))

  all-outer  = rec r (App(id,r)
                     <+ Beta + Let(id,id,r)
                     <+ subst + App(r,id))

  reduce-all = rec r (App(id,r)  + Abs(id,r)
                     <+ Beta + Let(id,id,r)
                     <+ subst + App(r,id))
```

3.1 Properties of the substitution strategy

The strategy subst applies the Let-elimination rules top-down, pushing the Let construct deeper into terms until it can be eliminated by an instance of a LetVar or LetAbs rule. When the top-level expression is a Let construction on which none of the Let-elimination rules succeeds, the subst strategy applies itself recursively to the matrix of a Let term.

By recursively eliminating Let-terms nested within a Let construction, we avoid the need for an explicit LetLet rule to handle nested Let terms. The recursion is effective only in the matrix of a Let term. This strategy is consistent with outermost reduction of Beta redexes, but would not be consistent with innermost reduction. Thus the strategy is a bit subtle.

To give a weakest-precondition formula for the subst strategy we follow the outline of the *wp* formula for a top-down strategy. However, a general top-down strategy would apply path quantification over all paths, whereas in the subst strategy, the recursion is effective only over the particular paths specified by a term-congruence. To simplify the formulation, let's factor the substitution strategy so that the top-down application of Let-elimination rules becomes a strategy parameter.

```
let-elim = LetVar + LetApp + LetAbs

subst'(s) = rec r (Let(id,id,r) <+ s)

subst = subst'(sometd(let-elim))
```

The recursive strategy subst'(s) applies its parameter strategy, s, bottom-up in the matrix of possibly nested *Let*-terms. Thus subst'(s) can be effective on nested Let constructions.

$$Dom(subst'(s)) = \mu\xi.\, Let(True, True, \xi) \,\vee\, Dom(s)$$

In particular, when s is specialized to the strategy *sometd*(*let-elim*), the domain can be seen to cover all possible forms of *Let* constructions:

$$\begin{aligned} Dom(subst'(sometd(let\text{-}elim))) = \mu\xi.\, & Let(True, True, \xi) \vee \\ & Let(True, True, Var(True)) \vee \\ & Let(True, True, App(True, True)) \vee \\ & Let(True, True, Abs(True, True)) \end{aligned}$$

Letting $NotLet =_{def} Var(True) \,\vee\, Abs(True, True) \,\vee\, App(True, True)$, we see that

$$\overline{Dom(subst'(sometd(let\text{-}elim)))} \equiv NotLet$$

Thus, the weakest-precondition under which an application of the strategy $subst'(sometd(let\text{-}elim))$ is assured to produce a let-free term can be shown to be

$$\begin{aligned} & wp_{subst'(sometd(let\text{-}elim))}(\mathbf{AG}\, NotLet) = \\ & \quad \mu\xi.\, Let(True, True, \xi) \vee \\ & \qquad (wp_{sometd(let\text{-}elim)}(\mathbf{AG}\, NotLet) \wedge \\ & \qquad\qquad \overline{Let(True, True, Dom(subst'(sometd(let\text{-}elim))))}) \end{aligned}$$

A more detailed characterization of $wp_{sometd(let\text{-}elim)}(\mathbf{AG}\, NotLet)$ has been omitted to save space.

3.2 Properties of the reduction strategies

Each of the three reduction strategies given in the module `lambda-red` has a different domain formula.

$$\begin{aligned} Dom(left\text{-}outer) = \mu\xi.\, & App(Abs(True, True), True) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, True) \end{aligned}$$

$$\begin{aligned} Dom(all\text{-}outer) = \mu\xi.\, & App(True, \xi) \vee \\ & App(Abs(True, True), \neg\xi) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, \neg\xi) \end{aligned}$$

$$\begin{aligned} Dom(reduce\text{-}all) = \mu\xi.\, & App(True, \xi) \,\vee\, Abs(True, \xi) \vee \\ & App(Abs(True, True), \neg\xi) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, \neg\xi) \end{aligned}$$

The strategy *all-outer* reduces strictly more terms than does *left-outer* and *reduce-all* reduces more terms than does *all-outer*.

Normalization strategies The module `normalize` contains normalization strategies that iterate the reduction strategies of module `lambda-red`. The *Stratego* library strategy `stdio` accepts input in textual format from the standard input file and delivers the output of its argument strategy to the standard output file.

```
module normalize
imports lambda-red io
strategies
  whnf = stdio(repeat(left-outer))

  hnf  = stdio(repeat(all-outer))

  normalize = stdio(repeat(reduce-all))
```

The normal forms achieved by each of the above strategies can be characterized in terms of modal μ-calculus formulas. The simplest to characterize is the `normalize` strategy. We first define a predicate to say what forms are β-redexes,

$$\mathit{IsRedex} = \mathit{App}(\mathit{Abs}(\mathit{True}, \mathit{True}), \mathit{True})$$

Then a β-reduced normal form is one that contains no β-redex nor `Let` term,

$$\mathit{BetaNormal} = \mathbf{AG}\,(\neg \mathit{IsRedex} \ \wedge\ \mathit{NotLet})$$

Head-normal form is slightly more troublesome to define, as normalization is not required under an abstraction. To express this distinction, we need a two-place modality operator analogous to the “until” operator of linear temporal logic, which we here call **above**. Its definition as a modal μ-calculus formula is

$$P \ \mathbf{above}\ Q \equiv \mu\xi.\, Q \ \vee\ (P \ \wedge\ \mathbf{AX}\,\xi)$$

Then head-normal form is

$$\begin{aligned}&\mathit{HeadNormal} = \\ &\qquad (\neg \mathit{IsRedex} \ \wedge\ \mathit{NotLet})\ \mathbf{above}\ (\mathit{Abs}(\mathit{True}, \mathit{True}) \ \wedge\ \mathbf{AG}\,\mathit{NotLet})\end{aligned}$$

Finally, to express weak head-normal form, which is the expected form of terms normalized by the `whnf` strategy, we resort to term-congruence operators to specify selective path quantification;

$$\begin{aligned}&\mathit{WeakHead} = \\ &\quad (\mu\xi.\, (\mathit{Var}(\mathit{True}) \ \vee\ \mathit{App}(\xi, \mathit{True})))\ \mathbf{above}\ (\mathit{Abs}(\mathit{True}, \mathit{True}) \ \wedge\ \mathbf{AG}\,\mathit{NotLet})\end{aligned}$$

Notice that in the latter formula, the embedded fixed-point formula describes only the condition in the prefix of the **above** operator, it does not encompass the entire geography of a term that contains embedded abstractions.

4 Other transformation systems—related work

The antecedent of all strategy languages is the prototypical ML language designed by Robin Milner to support proof construction in LCF [10]. In the last decade, new languages have evolved, reflecting lessons learned from logic programming on the one hand, and on the other, from understanding and implementing efficient term rewriting as a computational paradigm. Maude [8] implements a rewriting logic based upon a theory of term equality relative to an environment. Maude does not cater explicitly for programming strategies but supports strategy programming via reflection in the language [9].

ASF+SDF [11] is a general-purpose language to support term manipulation and has been used for the construction of parsers and pretty-printers as well as transformations. It does not support strategy-controlled rewriting directly, but a notion of traversal strategies can be implemented in this programming environment.

Strategies to control rewriting were introduced in ELAN [16], a comprehensive TRS with support for commutative and associative-commutative rewriting. ELAN employs reflection in the language [2], allowing strategies themselves to be expressed in terms of rewrite rules, although several specific strategy constructions have been built-in.

In ELAN, the idea of programming strategies for term rewriting was made an explicit goal [1]. ELAN has experimented with three primitives for controlling choice among possible alternate strategies: left-biased choice **case**, nondeterministic committed choice (called **dc**, for “don’t care”) and nondeterministic choice by consequence (called **dk**, for “don’t know”) which requires either backtracking or an equivalent implementation mechanism. In designing *Stratego*, choice-by-consequence has been rejected as computationally expensive and rarely needed in practice. Instead, the *Stratego* programmer is expected to anticipate the consequences of alternatives and specify an ordering of choices when the domains of alternative strategies may overlap.

ELAN is formally defined by a denotational semantics [1] which provides a reference model for implementation. In principle, the semantics also furnishes a basis for reasoning about transformation strategies. However, reasoning directly in terms of a semantics model can be tedious, as it is encumbered by details of the model.

5 Conclusions

The contribution of this paper lies in showing that two computational logics, each developed for a somewhat different purpose, can be used in combination to yield a programming logic for term transformation strategies, a domain for which no completely satisfactory logical characterization had previously been developed.

Weakest-precondition logic was originally proposed by Edsger Dijkstra [4] to cope with problems arising from nondeterministic choice, concurrency and potential nontermination of programs. Analogous problems arise when attempting to characterize properties of transformation strategies.

Strategies incorporate control to program traversals over complex terms in a variety of ways. CTL and the modal μ-calculus were originally developed for temporal applications in which paths in terms are thought of as evolving through time. However, these formalisms are equally applicable to terms whose paths are spatial. These notations provide the ability to quantify separately over paths (the breadth of a term), or over depth in a term. We have added to the generic quantification, operations that specify path quantification by lifting the constructors of terms to the status of logical quantifiers, an implicit term congruence.

The vehicle for this investigation into transformation strategies is *Stratego*, a domain-specific language that inherits from both logic and functional programming traditions. *Stratego* provides a compositional approach to programming strategies for term transformation that was lacking in earlier systems.

Acknowledgments: The author is grateful to Eelco Visser for conversations on transformation strategies and their formulation in *Stratego*, and to Tony Hoare, for a patient explanation of the subtleties of weakest precondition logics.

References

1. Peter Borovansky. *Le controle de la reecriture: etude et implantation d'un formalisme de strategies*. PhD thesis, Universite Henri Poincare–Nancy I, October 1998.
2. Peter Borovansky, Claude Kirchner, and Helene Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 5, 1997.
3. E. M. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
5. X. Du, K McDonnell, E. Nanos, Y. S. Ramakrishan, and Scott A. Smolka. Software design, specification and verification: Lessons learned from the Rether case study. In *Proc. of Sixth International Conference on Algebraic Methods and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 185–198. Springer Verlag, 1997.
6. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronizations skeletons. *Science of Computer Programming*, 2:241–266, 1982.
7. Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.
8. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 5. Elsevier, September 1996.
9. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 5. Elsevier, September 1996.
10. Robin Milner, Mike Gordon, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.
11. M. G. J. van den Brand, S. M. Eijkelkamp, D. K. A. Geluk, H. Meijer, M. J. F. Polling, and H. R. Osburne. Program transformation using ASF+SDF. Technical Report P9504, Programming Research Group, University of Amsterdam, 1995.
12. Eelco Visser. A bootstrapped compiler for strategies. In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 78–83, July 1999.
13. Eelco Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer Verlag, July 1999.

14. Eelco Visser and Zine el Abidine Benaissa. A core language for rewriting. In Claude Kirchner and Helene Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science. Elsevier, September 1998.
15. Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proc. of 1998 ACM/SIGPLAN International Conference on Functional Programming*, pages 13–26. ACM Press, September 1998.
16. M. Vittek. A compiler for nondeterministic rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*. Springer Verlag, July 1996.

Termination of Rewriting with Local Strategies

Olivier Fissore, Isabelle Gnaedig, Hélène Kirchner

LORIA-INRIA & LORIA-CNRS
P 239 F-54506 Vandœuvre-lès-Nancy Cedex
e-mail: {fissore,gnaedig,Helene.Kirchner}@loria.fr

Abstract. In this paper, we propose a method for specifically proving termination of rewriting with particular strategies: local strategies on operators. An inductive proof procedure is proposed, based on an explicit induction on the termination property. Given a term, the proof principle relies on alternatively applying the induction hypothesis on its subterms, by abstracting the subterms with induction variables, and narrowing the obtained terms in one step, according to the strategy. The induction relation, an $\mathcal{F}$-stable ordering having the subterm property, is not given a priori, but its existence is checked along the proof, by testing satisfiability of ordering constraints.

Keywords: Rewriting, termination, local strategy on operators, rule-based languages, induction, narrowing, ordering constraints.

1 Introduction

Termination of rewriting is a crucial problem in automated deduction, for equational logic, as well as in programming, for rule-based languages. As it is undecidable in general, it is ensured in particular contexts with sufficient conditions. A lot of termination proof techniques have been proposed, most of them using noetherian orderings on terms. But they usually tackle the property for the standard rewriting relation and essentially work on free term algebras. In the context of rule-based languages such as ASF+SDF [15], OBJ3 [14], Maude [5], CafeOBJ [10], Stratego [19], or ELAN [3], where programs are sets of rules and executions consist in rewriting ground expressions, it would be useful to have more specific termination proof tools: methods allowing to prove termination under specific reduction strategies, or to prove termination on the ground term algebra, for term rewriting systems (TRSs in short) that are not terminating on the free term one. The proof method we propose here, based on an explicit induction on the termination property, enables us to tackle these problems.

In the context of programming, there are sets of rules, that lead to divergent computations when all derivations are considered, but that terminate for particular strategies. A famous example is the evaluation of a recursive function defined with an $if\_then\_else\_$ expression, and that can diverge if the first argument is not evaluated first.

Local strategies on operators are used in this context, in particular to force the evaluation of expressions to terminate. This kind of strategy is allowed by languages such that OBJ3, CafeOBJ or Maude, and studied in [7] and [18]. It is defined in the following way: to any operator f is attached an ordered list of integers, giving the positions of the subterms to be evaluated in a given term, whose top operator is f. For example, the TRS

$$2nd(cons(x, cons(y, z))) \to y$$
$$inf(x) \to cons(x, inf(s(x)))$$

does not terminate for the standard rewriting relation, but does wit the following strategy: $LS(cons) = [1]$, $LS(2nd) = [1; 0]$, $LS(inf) = [1; 0]$, $LS(s) = [1]$.

As far as we know, the specific termination problem of rewriting with strategies has only been tackled for the innermost case on free term algebras [1] and for the innermost and the outermost cases for ground term ones [12]. Here, we propose a termination proof method for the case of local strategies on operators, following the induction proof principle proposed in [12]. Note that with our approach we handle the leftmost innermost strategy, which is a particular case of local strategy. Let us also cite termination results for a restricted kind of rewriting called context-sensitive rewriting [17, 16, 20, 11]. In this context, rewriting is allowed only at some specified position in the terms, which is different from local strategies, that are more specific: in the second case, not only allowed rewriting positions are specified, but also the order to consider them. Except for particular cases of local strategies, the two kinds of strategy are distinct.

The main idea of our proof method is to use explicit induction on the termination property in order to prove that any element t of a given set of terms T terminates i.e. there is no infinite derivation chain starting from t. Our induction principle uses an ordering on ground terms having the subterm property. It is based on the simple idea that if reducing a term t according to a given strategy first requires to normalize a subterm t' of t, we can suppose, by induction hypothesis, that t' terminates for the same strategy. If we replace t' by an induction variable X representing any of their normal forms, it then remains to prove that the term u obtained by replacement of t' by X in t is terminating, to prove that t is terminating. A rewriting step is then performed on u following the different possible values of X: it is computed by narrowing. This process is iterated until obtaining a no narrowable term, or a term, the induction hypothesis applies on. Note that the induction ordering is not given a priori but constrained during the proof by setting ordering constraints. Applying the induction hypothesis then lies on testing whether these constraints are satisfiable.

On the previous example, our method consists in proving termination of the constants, and of the terms of the form $s(T), cons(T_1, T_2), inf(T), 2nd(T)$, for the previously given strategy, whatever the values of the ground terms T, T_1, T_2. Let us suppose here we have a constant 0 in the signature. Obviously, 0 is in normal form and then terminating. For $s(T)$, using an induction ordering $\succ$ such that $s(T) \succ T$, by induction hypothesis, we can suppose that T is terminating. So is $s(T)$, since s is a constructor (i.e. s is not a top symbol of left-hand side of rule).

By definition of the strategy, normalizing $cons(T_1, T_2)$ consists in normalizing T_1, into $T_1\downarrow$ if its exists, and then $cons(T_1\downarrow, T_2)$ at the top position. For the same ordering $\succ$ as previously, we have $cons(T_1, T_2) \succ T_1$. Then, by induction hypothesis, T_1 terminates. Let $T_1\downarrow$ be any of its normal forms (there can be several normal forms if the system is not confluent). The termination of $cons(T_1, T_2)$ is then reduced to the termination of $cons(T_1\downarrow, T_2)$, which does not reduce at the top position by definition of the strategy.

Normalizing $2nd(T)$ also first consists in normalizing T. In a similar way than previously, the induction hypothesis can be applied to T. Let $T\downarrow$ be any of its normal forms. Reducing $2nd(T\downarrow)$ at the top position can give two results, according to the form of $T\downarrow$. If $T\downarrow$ is of the form $cons(T_3, cons(T_4, T_5))$, we obtain T_4, which is in normal form since $cons(T_3, cons(T_4, T_5))$ is in normal form. If $T\downarrow$ is not of the form $cons(T_3, cons(T_4, T_5))$, then $2nd(T\downarrow)$ does not rewrite at the top position and then is already in normal form.

Normalizing $inf(T)$ also first consists in normalizing T. Like previously, $T\downarrow$ exists thanks to the induction hypothesis. The resulting term $inf(T\downarrow)$ rewrites to

$$cons(T\downarrow, inf(s(T\downarrow))),$$

whose first subterm has then to be normalized. As this subterm is already in normal form, $cons(T\downarrow, inf(s(T\downarrow)))$ is in normal form for the given strategy. So any ground term is terminating. Our goal here is to provide a procedure implementing such a reasoning.

In Section 2, the background is presented. Section 3 introduces the basic notions formalizing our induction principle. In Section 4, a rule-based algorithm mechanizing the proof principle is given, its correctness is established and examples are given.

2 The background

We assume that the reader is familiar with the basic definitions and notations of term rewriting given for instance in [6]. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of terms built from a given finite set $\mathcal{F}$ of function symbols having an arity $n \in \mathbb{N}$, and a set $\mathcal{X}$ of variables denoted $x, y \ldots$. $\mathcal{T}(\mathcal{F})$ is the set of ground terms (without variables). The terms composed by a symbol of arity 0 are called constants; $\mathcal{C}$ is the set of constants of $\mathcal{F}$. Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the top position. The symbol at the top position of a term t is written $top(t)$. Let p and p' be two positions. The position p is said to be prefix of p' (and p' suffix of p) if $p' = p\lambda$, where λ is a non empty sequence of integers. Given a term t, $\mathcal{O}(t)$ is the set of positions in t, inductively defined as follows: $\mathcal{O}(t) = \{\epsilon\}\ if\ t \in \mathcal{X},\ \mathcal{O}(t) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{O}(t_i)\}\ if\ t = f(t_1, \ldots, t_n)$. This set is partitioned into $\overline{\mathcal{O}}(t) = \{p \in \mathcal{O}(t) \mid t|_p \notin \mathcal{X}\}$ and $\mathcal{O}_{\mathcal{V}}(t) = \{p \in \mathcal{O}(t) \mid t|_p \in \mathcal{X}\}$ where the notation $t|_p$ stands for the subterm of t at position p. If $p \in \mathcal{O}(t)$, then $t[t']_p$ denotes the term obtained from t by replacing the subterm at position p by the term t'.

A substitution is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = (x \mapsto t) \ldots (y \mapsto u)$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We identify a substitution $\sigma = (x \mapsto t) \ldots (y \mapsto u)$ with the finite set of equations $(x = t) \wedge \ldots \wedge (y = u)$. The result of applying σ to a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is written $\sigma(t)$ or σt. The domain of σ, denoted $Dom(\sigma)$ is the finite subset of $\mathcal{X}$ such that $\sigma x \neq x$. The range of σ, denoted $Ran(\sigma)$, is defined by $Ran(\sigma) = \bigcup_{x \in Dom(\sigma)} Var(\sigma x)$. A ground substitution or instantiation is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F})$. Id denotes the identity substitution. The composition of substitutions σ_1 followed by σ_2 is denoted $\sigma_2\sigma_1$. Given two substitutions σ_1 and σ_2, we write $\sigma_1 \leq \sigma_2$ iff $\exists\theta$ such that $\sigma_2 = \theta\sigma_1$. Given a subset $\mathcal{X}_1$ of $\mathcal{X}$, we note $\sigma_{\mathcal{X}_1}$ for the restriction of σ to the variables of $\mathcal{X}_1$, i.e. the substitution such that $Dom(\sigma_{\mathcal{X}_1}) \subseteq \mathcal{X}_1$ and $\forall x \in Dom(\sigma_{\mathcal{X}_1}) : \sigma_{\mathcal{X}_1}x = \sigma x$.

Given a set R of rewrite rules or term rewriting system on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, a function symbol in $\mathcal{F}$ is called a constructor if it does not occur in R at the top position of the left-hand side of a rule, and is called a defined function symbol otherwise. The set of constructors of $\mathcal{F}$ for R is denoted by $Cons_R$, the set of defined function symbols of $\mathcal{F}$ for R is denoted by Def_R (R is omitted when there is no ambiguity). The rewriting relation induced by R is called standard rewriting relation and is noted $\rightarrow_R$ ($\rightarrow$ if there is no ambiguity on R). We note $s \rightarrow_{p,l\rightarrow r,\sigma} t$ (or $s \rightarrow^{p,l\rightarrow r,\sigma} t$ where either p or $l \rightarrow r$ or σ may be omitted) if s rewrites into t at position p with the rule $l \rightarrow r$ and the substitution σ. The transitive (resp. reflexive transitive) closure of the rewriting relation induced by R is denoted by $\rightarrow^+_R$ (resp. $\rightarrow^*_R$). If it exists, the last term of a finite derivation starting from t is said to be in normal form, and is denoted by $t{\downarrow}$.

An ordering $\succ$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is said to be noetherian (or well-founded) iff there is no infinite decreasing chain for this ordering. It is $\mathcal{F}$-stable iff for any pair of terms t, t' of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, for any context $f(\ldots\ \ldots)$, $t \succ t'$ implies $f(\ldots t \ldots) \succ f(\ldots t' \ldots)$. It has the subterm property iff for any t of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $f(\ldots t \ldots) \succ t$. Note that if $\succ$ is $\mathcal{F}$-stable and has the subterm property, then it is noetherian. If, in addition, $\succ$ is stable by substitution (for any substitution σ, any pair of terms $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t \succ t'$ implies $\sigma t \succ \sigma t'$), then it is called a simplification ordering. Let t be a term of $\mathcal{T}(\mathcal{F})$; let us recall that t terminates if and only if any rewriting derivation (or derivation chain) starting from t is finite.

3 Induction for termination with local strategies

We now tackle the termination problem for rewriting with local strategies on operators, as expressed in [14] and studied in [7]. A local strategy is defined in the following way.

Definition 3. *An* LS rewriting strategy *(or LS-strategy) on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ (resp. of $\mathcal{T}(\mathcal{F})$) is a function LS from $\mathcal{F}$ to the set of lists of integers $\mathcal{L}(\mathbb{N})$, defining a rewriting strategy as follows.*

Given a LS-strategy such that $LS(f) = [p_1, \ldots, p_k]$, $p_i \in [0..arity(f)]$ for all $i \in [1..k]$, for some symbol $f \in \mathcal{F}$, normalizing a term $t = f(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ (resp $\in \mathcal{T}(\mathcal{F})$) with respect to $LS(f) = [p_1, \ldots, p_k]$, consists in normalizing all subterms of t at positions $p_1, \ldots, p_k$ successively, according to the strategy. If there exists $i \in [1..k]$ such that $p_1, \ldots, p_{i-1} \neq 0$ and $p_i = 0$, then

- *if the current term t' obtained after normalizing $t|_{p_1}, \ldots, t|_{p_{i-1}}$ is reducible at the top position into a term $g(u_1, \ldots, u_n)$, then $g(u_1, \ldots, u_n)$ is normalized with respect to $LS(g)$ and the rest of the strategy $[p_{i+1}, \ldots, p_k]$ is ignored,*
- *if t' is not reducible at the top position, then t' is normalized with respect to $p_{i+1}, \ldots, p_k$.*

Note that for $x \in \mathcal{X}$, $LS[x] = []$ since a term is not reducible at a variable position.

At each rewriting step, the term t is said to LS-rewrite into a term t'. If t does not rewrite for the LS-strategy, it is said to be in LS-normal form (or in normal form if there is no ambiguity). If any LS-rewriting chain starting from t leads to a LS-normal form then t is said to be LS-terminating (or to LS-terminate). If the evaluation strategy of a term t' is the empty list, then t' is in LS-normal form.

3.1 Induction for local strategies

For proving that a term t of $\mathcal{T}(\mathcal{F})$ LS-terminates, we proceed by induction on $\mathcal{T}(\mathcal{F})$ with a noetherian ordering $\succ$ (more precisely, an $\mathcal{F}$-stable ordering having the subterm property), assuming that for any t' such that $t \succ t'$, t' LS-terminates. We first prove that a basic set of minimal elements for $\succ$ LS-terminates. As the subterm property for $\succ$ is required, the set of minimal elements is a subset of the set of constants of $\mathcal{F}$.

We then consider the case of any term t of $\mathcal{T}(\mathcal{F})$. For that, we observe the rewriting derivation tree for the LS-strategy starting from a term $t_{ref} = g(x_1, \ldots, x_m)$, for any $g \in \mathcal{F}$, where $x_1, \ldots, x_m$ are induction variables that can be instantiated by any ground term. The LS rewriting relation on ground terms is simulated by the two mechanisms below to follow the derivation tree starting from t_{ref}, and whose current term is t. Let $LS(top(t)) = [p_1, \ldots, p_n]$, and p_k the first element of $[p_1, \ldots, p_n]$ such that $p_k = 0$.

- First, the subterms $t|_{p_1}, \ldots, t|_{p_{k-1}}$ of t have to be LS-normalized, by definition of the above LS-strategy. If $t_{ref} \succ t|_{p_1}, \ldots, t|_{p_{k-1}}$ we can suppose, by induction hypothesis, that these subterms are LS-terminating. We then replace them in t by *abstraction variables* X_i representing respectively any of their normal forms $t_i{\downarrow}$: these variables will only be instantiated by terms in normal form. Reasoning by induction allows us to only suppose the existence of the $t_i{\downarrow}$ *without explicitly computing them*; this step will be called abstraction step or abstraction of the subterms of t. We also say that t is abstracted into a term v.
- Second, rewriting the resulting term v at position ϵ, following all possible ground instantiations of v. This is computed by a narrowing step on v. Two cases may happen:
  - if v is not narrowable at the top position, the subterms $v|_{p_{k+1}}, \ldots, v|_{p_n}$ of v then have to be LS-normalized, and we try to abstract them like above;
  - if v is narrowable at the top position, the narrowing step is computed with all possible rules and all possible substitutions $\sigma_1, \ldots, \sigma_l$ to give terms $w_1, \ldots, w_l$, that have to be considered respectively with the strategies $LS(top(w_1)), \ldots,$ $LS(top(w_l))$. So the two mechanisms above are again applied on the terms $w_1, \ldots, w_l$. In addition, instances of v that are not considered by the narrowing have to be reduced at the positions $p_{k+1}, \ldots, p_n$. So the two mechanisms described above are also applied on v at positions $p_{k+1}, \ldots, p_n$ for the instances of v that are not instances of $\sigma_i v, i \in [1..l]$.
- The process stops on the current terms t having an empty LS-strategy or on current terms the induction hypothesis can be applied on (i.e. such that $t_{ref} \succ t$; in this case, t is supposed to be LS-terminating).

Note that if there does not exist p_k in $\{p_1, \ldots, p_n\}$ such that $p_k = 0$, then only the first point is processed, abstracting every subterm $t|_{p_i}$ of $t, i \in [1..n]$.

3.2 Abstraction

We now give some new definitions to formalize the above mechanisms. Abstraction needs the use of special variables representing LS-normal forms.

Definition 4. *Let $\mathcal{N}$ be a set of new variables disjoint from $\mathcal{X}$. Symbols of $\mathcal{N}$ are called* NF-variables*. Substitutions and instantiations are extended to $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ in the following way. Let $X \in \mathcal{N}$; for any substitution σ (resp. instantiation θ) such that $X \in Dom(\sigma)$, σX (resp. θX) is in normal form.*

LS-strategies can be extended to terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, by stating $LS(X) = []$ for any $X \in \mathcal{N}$.

Note that for abstracting the current term $f(u_1, \ldots, u_m)$, it is not useful to introduce an abstraction variable for the u_j that are ground terms already in normal form, nor for the u_j that are already NF-variables.

Definition 5. *The term $f(u_1, \ldots, u_m)$ is* abstracted *into $f(U_1, \ldots, U_m)$ at positions $\{i_1, \ldots, i_p\} \subseteq [1..m]$ if :*

- *$\{i_1, \ldots, i_p\}$ are the positions of $[1..m]$ such that $u_{i_1}, \ldots, u_{i_p}$ are neither ground terms in normal form, nor NF-variables,*
- *$U_j = X_j$ where X_j is a fresh NF-variable, if $j \in \{i_1, \ldots, i_p\}$, $U_i = u_i$ otherwise.*

We will prove LS-termination on $\mathcal{T}(\mathcal{F})$, reasoning on terms with abstraction variables, i.e. on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$.

3.3 Constraints

Let us now define the different constraints needed by our proof process. Unlike in classical approaches using induction, the induction ordering is not given a priori. Constraints are set along the proof, following the requirements appearing when induction hypotheses have to be applied. Such ordering constraints are cumulated in a set C and the satisfiability of C is tested any time the induction hypotheses have to be applied.

We now formally define the satisfiability of ordering constraints.

Definition 6. An ordering constraint $(t > t')$ *on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ is* satisfiable *if there exists an ordering $\succ$ and at least one instantiation θ such that $\theta t \succ \theta t'$. We say that $\succ$ and θ satisfy $(t > t')$.*

A conjunction C of ordering constraints is satisfiable if there exists an ordering and an instantiation satisfying all conjuncts. The empty conjunction, always satisfied, is denoted by $\top$.

Along our induction process, when abstracting subterms t_i by X_i, we state constraints on NF-variables to express that their instances can only be the normal forms of the corresponding instances of the t_i. They are of the form $t\downarrow = X$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and $X \in \mathcal{N}$, or more generally of the form $t\downarrow = t'$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. Let us call such a constraint an abstraction constraint.

Definition 7. *An* abstraction constraint $(t{\downarrow} = t')$ *where* $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ *is* satisfiable *if there exists at least one instantiation* θ *such that* $\theta t{\downarrow} = \theta t'$. *We say that* θ *satisfies* $(t{\downarrow} = t')$.

A constraint formula A *is a formula of the form* $\bigwedge_i (t_i{\downarrow} = t'_i) \bigwedge_j (\bigvee_{k_j} (x_{k_j} \neq u_{k_j}))$, $x_{k_j} \in \mathcal{X} \cup \mathcal{N}, u_{k_j} \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ *and the* $(t_i{\downarrow} = t'_i)$ *are abstraction constraints. The empty formula is denoted* $\top$. *A formula* A *is* satisfiable *if there exists at least one instantiation* θ *such that* $\bigwedge_i (\theta t_i{\downarrow} = \theta t'_i) \bigwedge_j (\bigvee_{k_j} (\theta x_{k_j} \neq \theta u_{k_j}))$. *We say that* θ *satisfies* A.

In this paper, we consider constraint problems composed of 2-tuples (A, C) where A is constraint formula and C is a conjunction of ordering constraints.

Definition 8. *Let* A *be a constraint formula and* C *a conjunction of ordering constraints. The constraint problem* (A, C) *is satisfied by an ordering* $\succ$ *if* A *is satisfiable, and for all instantiations* θ *satisfying* A, $\succ$ *and* θ *satisfy* C. (A, C) *is satisfiable if* A *is satisfiable and there exists an ordering* $\succ$ *as above.*

Deciding the satisfiability of (A, C) would require to express all instantiations satisfying A. As we will see later, an interesting point of our method is that we do not need to characterize all those instantiations. It is enough to exhibit one of them to prove the satisfiability of A. In such a case, a sufficient condition for an ordering $\succ$ to satisfy (A, C) is that $\succ$ is stable by substitution (the induction ordering is then a simplification ordering) and $t \succ t'$ for any unequality $t > t'$ of C.

3.4 Narrowing

After the abstraction of the term $f(u_1, \ldots, u_m)$ into $f(U_1, \ldots, U_m)$ at positions $\{i_1, \ldots, i_p\}$, where the u_{i_j} are supposed to have a normal form $u_{i_j}{\downarrow}$, and are replaced by abstraction variables X_{i_j}, we test whether the ground instances of $f(U_1, \ldots, U_m)$ are reducible with a case study on the syntactic form of the possible instantiations of the X_{i_j}. This test consists in narrowing $f(U_1, \ldots, U_m)$ at position ϵ with all possible substitutions instantiating the X_i only with irreducible terms, and all possible rewrite rules.

Let us now recall the definition of narrowing.

Definition 9. *Let* $\mathcal{R}$ *be a TRS on* $\mathcal{T}(\mathcal{F}, \mathcal{X})$. *A term* t *is* narrowed *into* t', *at the non variable position* p, *using the rewrite rule* $l \to r$ *of* R *and the substitution* σ, *when* σ *is a most general unifier of* $t|_p$ *and* l, $t' = \sigma(t[r]_p)$. *This is denoted* $t \rightsquigarrow_R^{p, l \to r, \sigma} t'$ *where either* p, *or* $l \to r$ *or* σ *may be omitted. It is always assumed that there is no variable in common between the rule and the term, i.e. that* $Var(l) \cap Var(t) = \emptyset$.

The requirement of disjoint variables is easily fulfilled by an appropriate renaming of variables in the rules when narrowing is performed. Note that for the most general unifier σ used in the above definition, $Dom(\sigma) \subseteq Var(l) \cup Var(t)$ and we can choose $Ran(\sigma) \cap (Var(l) \cup Var(t)) = \emptyset$, thus introducing in the range of σ only fresh variables. Thus $Var(t) \cap Var(t') = \emptyset$ if in addition, variables of $\mathcal{V}ar(t) - Dom(\sigma)$ are renamed through a substitution denoted σ_{ren}.

Moreover, if narrowing is performed after an abstraction step, the range $Ran(\sigma)$ of the narrowing substitution σ only contains NF-variables. Indeed, abstraction gives a term $f(U_1, \ldots, U_m)$ in which some subterms U_i are abstraction variables $X_{i_1}, \ldots, X_{i_p}$. Thus, on $\{X_{i_1}, \ldots, X_{i_p}\}, \sigma$ is of the form $(X_{i_1} = v_{i_1}, \ldots, X_{i_p} = v_{i_p})$. As $X_{i_1}, \ldots, X_{i_p}$ are NF-variables, all their ground instances must be in normal form, and the same holds for ground instances of the v_i's. So the variables of $v_{i_1}, \ldots, v_{i_p}$ have to be NF-variables.

As we will see below, in our proof process, we will also have to consider the negation of a substitution.

Definition 10. *Let σ be a substitution on $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ defined by $\bigwedge_i (x_i = t_i)$ $x_i \in \mathcal{X} \cup \mathcal{N}$, $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. The* negation *of σ, denoted $\overline{\sigma}$ is the formula $\bigvee_i (x_i \neq t_i)$.*

4 A rule-based algorithm

4.1 The inference rules

Inference rules describing our termination proof mechanism for local strategies work on sets of 4-tuples $T = (\{u\}, [p_1, \ldots, p_m], A, C)$, where:

- $\{u\}$ is a set of terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, containing the current term u whose ground instances have to be proved LS-terminating. This is either a singleton or the empty set.
- $[p_1, \ldots, p_m]$ is the list of positions with respect to whom the current term u has to be evaluated. This is a sublist of $LS(top(u))$.
- A is a constraint formula. The sub-formulas of the form

$$u\!\downarrow = X, u \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N}), X \in \mathcal{N}$$

  are stated each time a subterm u of the current term is abstracted by a new NF-variable X.
- C is a conjunction of ordering constraints completed by the abstraction steps.

Let us now present the inference rules.

- The rule $\mathbf{Abstract}$ processes the abstracting step. It applies to

$$(\{f(u_1, \ldots, u_m)\}, [p_1, \ldots, p_n], A, C),$$

  when there exists $k \in [2..n], p_k = 0$ and $p_1, \ldots, p_{k-1} \neq 0$. The term $u = f(u_1, \ldots, u_m)$ is abstracted at positions $i_1, \ldots, i_p \in \{p_1, \ldots, p_{k-1}\}$ if there exists an $\mathcal{F}$-stable ordering having the subterm property and such that $(A, C \wedge t_{ref} > u_{i_1}, \ldots, u_{i_p})$ is satisfiable. Indeed, by induction hypothesis, all ground instances of $u_{i_1}, \ldots, u_{i_p}$ LS-terminate. So $\{f(u_1, \ldots, u_m)\}$ is replaced by $f(U_1, \ldots, U_m)$. The list of positions then becomes $[0, p_{k+1}, \ldots, p_n]$.
- The rule $\mathbf{Abstract{-}Stop}$ processes the abstracting step as above, when there is no position 0 in the strategy of the current term. Any ground instance of the term obtained after abstraction is irreducible, by definition of the LS-strategy, which ends the proof on the current derivation chain. The set containing the current term is then replaced by the empty set.

Table 1. Inference rules for t_{ref} LS-termination

Abstract:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [p_1,\ldots,p_n],\ A,\ C)\}}{T \cup \{(\{f(U_1,\ldots,U_m)\},\ [0,p_{k+1},\ldots,p_n],\ A \bigwedge_{i\in\{i_1,\ldots,i_p\}} (u_i{\downarrow} = X_i),\ C \bigwedge_{i\in\{i_1,\ldots,i_p\}} t_{ref} > u_i)\}}$$

where $f(u_1,\ldots,u_m)$ is abstracted by $f(U_1,\ldots,U_m)$ at the positions $i_1,\ldots,i_p \in \{p_1,\ldots,p_{k-1}\}$

if $\exists k \in [2..n] : p_1,\ldots,p_{k-1} \neq 0, p_k = 0$ and $(A, C \bigwedge_{i\in\{i_1,\ldots,i_p\}} t_{ref} > u_i)$ is satisfiable

Abstract-Stop:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [p_1,\ldots,p_n],\ A,\ C)\}}{T \cup \{(\emptyset,\ [],\ A,\ C \bigwedge_{i\in[i_1..i_p]} t_{ref} > u_i)\}}$$

where $f(u_1,\ldots,u_m)$ can be abstracted by $f(U_1,\ldots,U_m)$ at the positions $i_1,\ldots,i_p \in \{p_1,\ldots,p_n\}$
if $p_1,\ldots,p_n \neq 0$ and $(A, C \bigwedge_{i\in[i_1..i_p]} t_{ref} > u_i)$ is satisfiable

Narrow-Y:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [0,p_1,\ldots,p_n],\ A,\ C)\}}{T \cup_{i\in[1..l]} \{(\{w_i\},\ LS(top(w_i)),\ \sigma_i A,\ C)\} \cup COMPL}$$

if $\exists\sigma$ such that $f(u_1,\ldots,u_m) \rightsquigarrow_{\epsilon,\sigma} w$ and σA is satisfiable
where $w_i, i \in [1..l]$, are all terms such that $f(u_1,\ldots,u_m) \rightsquigarrow_{\epsilon,\sigma_i} w_i$ and $\sigma_i A$ is satisfiable,

$$COMPL = \begin{cases} \{(\{f(u_1,\ldots,u_m)\},\ [p_1..p_n],\ A \bigwedge_{i=1}^{l} \overline{\sigma_i \mathcal{V}_{ar(f(u_1,\ldots,u_n))}},\ C)\} & \text{if } (A \bigwedge_{i=1}^{l} \overline{\sigma_i \mathcal{V}_{ar(f(u_1,\ldots,u_n))}}) \text{ satisfiable} \\ \emptyset & \text{otherwise.} \end{cases}$$

Narrow-N:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [0,p_1,\ldots,p_n],\ A,\ C)\}}{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [p_1,\ldots,p_n],\ A,\ C)\}}$$

if $f(u_1,\ldots,u_m)$ is not narrowable at the top position
or $\forall\sigma$ narrowing substitution of $f(u_1,\ldots,u_m)$ at the top position , σA is not satisfiable.

Stop-Ind:

$$\frac{T \cup \{(\{u\},\ [p_1,\ldots,p_n],\ A,\ C)\}}{T \cup \{(\emptyset,\ [],\ A,\ C \wedge t_{ref} > u)}$$

if $(A, C \wedge t_{ref} > u)$ is satisfiable

Stop-A:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [p_1,\ldots,p_n],\ A,\ C)\}}{T \cup \{(\{\sharp\},\ [],\ A,\ C)\}}$$

if $p_1 \neq 0$ and neither Abstract nor Abstract$-$Stop applies.

Stop:

$$\frac{T \cup \{(\{f(u_1,\ldots,u_m)\},\ [],\ A,\ C)\}}{T \cup \{(\emptyset,\ [],\ A,\ C)\}}$$

- The rule $\mathbf{Narrow-Y}$ processes the narrowing step at position 0 (i.e. the top position) of the current term u. If u is narrowable with a substitution satisfying the current constraint formula A, then u is narrowed in all possible ways in one step, with all possible rewrite rules of the rewrite system R, and all possible substitutions σ_i, into $w_i, i \in [1..l]$. Then $(\{u\}, [0, p_1, \ldots, p_n], A, C)$ is replaced by $\{(\{w_i\}, LS(top(w_i)), \sigma_i A, C), i \in [1..l]\}$, where σ_i is the most general substitution allowing narrowing of u into terms w_i. This narrowing step means that $\sigma_1 u, \ldots, \sigma_l u$ are all the instances of u that are reducible at the top position. It involves that if $\Phi = \overline{\sigma_1} \wedge \ldots \wedge \overline{\sigma_l}$ is satisfiable, for each substitution μ satisfying Φ, μu is not reducible at the top position. Then, as these μu have to be reduced at positions $[p_1, \ldots, p_n]$, if Φ is satisfiable, to the previous set we must add the set : $(\{u\}, [p_1, \ldots, p_n], A \bigwedge_{i=1}^{l} \overline{\sigma_i}, C)\}$. Note that if $\exists i$ such that σ_i is just a renaming of variables, then $\Phi = \emptyset$. Moreover, since the set of variables occuring in A is disjoint from the set of variables occuring in the rewrite rules, we had rather consider $\overline{\sigma_{i\mathcal{V}ar(u)}}$ instead of $\overline{\sigma_i}$.
- The rule $\mathbf{Narrow-N}$ handles the case where u is not narrowable at position 0 or is narrowable with a substitution that does not satisfy the current constraint formula A. Then no narrowing is processed and the current term is evaluated at positions following the top position in the strategy. The list of positions then becomes $[p_1, \ldots, p_n]$.
- We also can test for the current term whether there exists an ordering having the subterm property such that $(A, C \wedge t_{ref} > u)$ is satisfiable. Then, by induction hypothesis, any ground instance of u terminates for the LS-strategy, which ends the proof on the current derivation chain. The $\mathbf{Stop-Ind}$ rule then replaces the set containing the current term by the empty set.
- The rule $\mathbf{Stop-A}$ allows to stop the inference process when neither $\mathbf{Abstract}$ nor $\mathbf{Abstract-Stop}$ applies, replacing u by the particular symbol $\sharp$.
- The rule $\mathbf{Stop}$ allows to stop the inference process when the list of positions is empty.

The set of inference rules is given in Table 1.

Once $\mathbf{Abstract}$ is applied, the evaluation list's first element is 0, so the only rule that applies then is one of $\{\mathbf{Narrow-Y}, \mathbf{Narrow-N}\}$. When $\mathbf{Narrow-Y}$ does not apply, $\mathbf{Narrow-N}$ applies. When $\mathbf{Abstract}$ does not apply, and the evaluation list's first element is not 0, either $\mathbf{Abstract-Stop}$ or $\mathbf{Stop-A}$ applies, and then no rule applies anymore. The strategy for applying these rules is:

repeat*($\mathbf{Stop}$; $\mathbf{Stop-Ind}$; $\mathbf{Abstract}$; $\mathbf{Abstract-Stop}$; $\mathbf{Stop-A}$;
($\mathbf{Narrow-Y}$/$\mathbf{Narrow-N}$))

where ";" expresses the sequential application of the rules, r_1/. . ./r_n expresses that the rules $r_1, \ldots, r_n$ are mutually exclusive and that one of them will be applied, and repeat* (r_1; . . .; r_n) stops if none of the r_i applies anymore.

We write $SUCCESS(g, \succ)$ if application of the inference rules on $(\{g(x_1, \ldots, x_m)\}, LS(g), \top, \top)$, whose conditions are satisfied by $\succ$, gives a state of the form $(\emptyset, [], A, C)$ on every branch of the derivation tree.

Theorem 1. *Let R be a TRS on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, and $LS : \mathcal{F} \mapsto L(\mathbb{N})$ a LS-strategy such that the constants of $\mathcal{F}$ LS-terminate. If there exists an $\mathcal{F}$-stable ordering $\succ$ having the subterm property, such that for every non constant defined symbol g, $SUCCESS(g, \succ)$, then every term of $\mathcal{T}(\mathcal{F})$ LS-terminates.*

Note that if there exists a simplification ordering $\succ$ such that $l \succ r$ for any rewrite rule $l \to r$ of a TRS $\mathcal{R}$, then we have $SUCCESS(g, \succ')$, for any defined symbol $g \in \mathcal{D}ef_R$ and any $\mathcal{F}$-stable ordering $\succ'$ having the subterm property. Indeed, for any $t_{ref} = g(x_1, \ldots, x_m)$, we have $\mathcal{U}(t_{ref}) = \mathcal{R}$, with every rule oriented by $\succ$. Then, thanks to Proposition 1, we have $TERMIN(t_{ref})$, and then $\mathbf{Stop{-}Ind}$ applies.

4.2 Extending the induction principle

When the induction hypothesis cannot be applied on a term u, the inductive reasoning can be completed as follows. It can sometimes be possible to prove termination of any ground instance of u_i (resp. u) by another way. Let $TERMIN(u)$ be a predicate that is true iff any ground instance of u LS-terminates. In the first, second and fifth previous inference rules we can then replace the condition $t > u_i$ for some i (resp. $t > u$) by the alternative predicate $TERMIN(u_i)$ (resp. $TERMIN(u)$). Obviously, in this case, the ordering constraint $t > u_i$ (resp. $t > u$) is not added to C.

As in [13], for establishing that $TERMIN(u)$ is true, in some cases, the notion of usable rules can be used. Given a TRS $\mathcal{R}$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ and a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, we determine the only rewrite rules that are likely to apply to any of its ground instances, for the standard rewriting relation, until its ground normal form is reached, if it exists. Then we try to find a simplification ordering $\succ_N$ so that these rules are oriented. Thus any ground instance αt is bound to terminate for the standard rewriting relation: indeed, if $\alpha t \to t_1 \to t_2 \to \ldots$, then, thanks to the previous hypotheses, $\alpha t \succ_N t_1 \succ_N t_2 \succ_N \ldots$ and, since the ordering $\succ_N$ is noetherian, the rewriting chain cannot be infinite. More formally, given a TRS $\mathcal{R}$, we call *usable rules* of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, as in [2], a calculable superset of the set of rules of R used in all possible standard derivations starting from t, and defined as follows. When t is a variable of $\mathcal{X}$, then the usable rules of t are $\mathcal{R}$ itself. Moreover, the set of usable rules associated to a NF-variable is empty, since the only possible instances of such a variable are ground terms in normal form.

Definition 11. *Let $\mathcal{R}$ be a TRS on a set $\mathcal{F}$ of symbols. Let $Rls(f) = \{l \to r \in \mathcal{R} \mid top(l) = f\}$. For any $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, the set of usable rules of t, denoted $\mathcal{U}(t)$, is defined by:*

- $\mathcal{U}(t) = \mathcal{R}$ *if* $t \in \mathcal{X}$,
- $\mathcal{U}(t) = \emptyset$ *if* $t \in \mathcal{N}$,
- $\mathcal{U}(f(u_1, \ldots, u_n)) = Rls(f) \bigcup_{i=1}^{n} \mathcal{U}(u_i) \bigcup_{l \to r \in Rls(f)} \mathcal{U}(r)$.

Lemma 1. *[13] Let $\mathcal{R}$ be a TRS on a set $\mathcal{F}$ of symbols and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. Whatever αt ground instance of t and $\alpha t \to_{p_1, l_1 \to r_1} t_1 \to_{p_2, l_2 \to r_2} t_2 \to \ldots \to_{p_n, l_n \to r_n} t_n$ rewrite chain starting from αt, then $l_i \to r_i \in \mathcal{U}(t),\ \forall i \in [1..n]$.*

We then can give a sufficient criterion for ensuring termination for the standard rewriting relation (and then LS-termination) of any ground instance of a term t.

Proposition 1. *[13] Let $\mathcal{R}$ be a TRS on a set $\mathcal{F}$ of symbols, and t a term of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. If there exists a simplification ordering $\succ$ such that $\forall l \to r \in \mathcal{U}(t) : l \succ r$, then any ground instance of t is terminating.*

Let us now illustrate our method on the example given in the Introduction.

Example 1. Let R be the following TRS. We prove that R is terminating on $\mathcal{T}(\mathcal{F})$ ($\mathcal{F} = \{cons : 2, 2nd : 1, inf : 1, s : 1, 0 : 0\}$) with the following evaluation strategy : $LS(cons) = [1]$, $LS(2nd) = [1; 0]$, $LS(inf) = [1; 0]$, $LS(s) = [1]$.

$$\begin{array}{ll} 2nd(cons(x, cons(y, z))) & \to y \\ inf(x) & \to cons(x, inf(s(x))) \end{array}$$

The defined symbols of $\mathcal{F}$ are $2nd$ and inf. The term 0 is obviously terminating. Applying the rules on $2nd(x_1)$, we get :

$$\begin{array}{ll}
2nd(x_1) & [1;0] \\
 & A = \top \\
 & C = \top \\
\textbf{Abstract} & \\
2nd(X_1) & [0] \\
 & A = (x_1{\downarrow} = X_1) \\
 & C = (2nd(x_1) > x_1) \\
\textbf{Narrow}-\textbf{Y} & \\
X_3 & [] \\
 & \sigma = (X_1 = cons(X_2, cons(X_3, X_4)) \\
 & \quad \wedge x = X_2 \wedge y = X_3 \wedge z = X_4) \\
 & A = (x_1{\downarrow} = cons(X_2, cons(X_3, X_4))) \\
 & C = (2nd(x_1) > x_1) \\
2nd(X_1) & [] \\
 & A = (x_1{\downarrow} = X_1) \\
 & \quad \wedge (X_1 \neq cons(X_2, cons(X_3, X_4))) \\
 & C = (2nd(x_1) > x_1)
\end{array}$$

The first constraint formula A can be satisfied by $\theta = (x_1 = cons(0, cons(0, 0)) \wedge X_2 = 0 \wedge X_3 = 0 \wedge X_4 = 0)$, the second one by $\theta = (x_1 = 0 \wedge X_1 = 0)$.

$$\begin{array}{ll}
\textbf{Stop} \text{ (twice)} & \\
\emptyset & [] \\
 & A = (x_1{\downarrow} = cons(X_2, cons(X_3, X_4))) \\
 & C = (2nd(x_1) > x_1) \\
\emptyset & [] \\
 & A = (x_1{\downarrow} = X_1 \wedge X_1 \neq cons(X_2, cons(X_3, X_4))) \\
 & C = (2nd(x_1) > x_1)
\end{array}$$

Applying the rules on $inf(x_1)$, we get :

$$
\begin{array}{ll}
inf(x_1) & [1;0] \\
 & A = \top \\
 & C = \top \\
\textbf{Abstract} & \\
inf(X_1) & [0] \\
 & A = (x_1{\downarrow} = X_1) \\
 & C = (inf(x_1) > x_1) \\
\textbf{Narrow}-\textbf{Y} & \\
cons(X_2, inf(s(X_2))) & [1] \\
 & \sigma = (X_1 = X_2 \wedge x = X_2) \\
 & A = (x_1{\downarrow} = X_2) \\
 & C = (inf(x_1) > x_1)
\end{array}
$$

The constraint formula A is easily satisfied with the substitution $\theta = (x_1 = 0 \wedge X_2 = 0)$.

$$
\begin{array}{ll}
\textbf{Abstract} & \\
cons(X_2, inf(s(X_2))) & [] \\
 & A = (x_1{\downarrow} = X_2) \\
 & C = (inf(x_1) > x_1) \\
\textbf{Stop} & \\
\emptyset & [] \\
 & A = (x_1{\downarrow} = X_2) \\
 & C = (inf(x_1) > x_1)
\end{array}
$$

Note that the substitution of narrowing used in the application of $\textbf{Narrow}-\textbf{Y}$ merely consists of a renaming of variable. Therefore all the instances of $inf(X_1)$ are covered, and there is no need to define an extra derivation.

Another example is the famous case of the conditional expression.

Example 2. Let us consider the following system $\mathcal{R}$ built on $\mathcal{F} = \{f : 1, zero : 1, if\_then\_else\_ : 3, h : 1, s : 1, i : 1, g : 1, 0 : 0\}$, and denote $if\_then\_else\_$ by ite for short :

$$
\begin{array}{ll}
f(i(x)) & \rightarrow ite(zero(x), g(x), f(h(x))) \\
zero(0) & \rightarrow true \\
zero(s(x)) & \rightarrow false \\
ite(true, x, y) & \rightarrow x \\
ite(false, x, y) & \rightarrow y \\
h(0) & \rightarrow i(0) \\
h(x) & \rightarrow s(i(x))
\end{array}
$$

The LS-strategy is the following : $LS(ite) = [1;0], LS(f) = LS(zero) = LS(h) = [1;0]$ and $LS(g) = LS(i) = [1]$.

Before showing LS-termination of $\mathcal{R}$, let us note a few points :

- because of the first rule and the last one, $\mathcal{R}$ cannot be oriented;
- $\mathcal{R}$ does not terminate in general; indeed, it suffices to consider the following derivation :
  $f(i(0)) \rightarrow_{\epsilon}$
  $ite(zero(0), g(0), f(h(0))) \rightarrow_{3.1}$
  $ite(zero(0), g(0), f(i(0))) \rightarrow \ldots$

Obviously, the constant 0 LS-terminates. Applying the inference rules on $f(x_1)$, we get :

$$\begin{array}{ll}
f(x_1) & [1;0] \\
 & A = \top \\
 & C = \top \\
\textbf{Abstract} & \\
f(X_1) & [0] \\
 & A = (x_1{\downarrow} = X_1) \\
 & C = (f(x_1) > x_1)
\end{array}$$

Abstract applies, since C is satisfiable by any ordering having the subterm property. A is satisfiable with any instantiation θ such that $\theta x_1 = \theta X_1 = 0$.

$$\begin{array}{ll}
\textbf{Narrow-Y} & \\
ite(zero(X_2), g(X_2), f(h(X_2))) & \sigma = (X_1 = i(X_2) \\
 & \quad \wedge x' = X_2) \\
 & [1;0] \\
 & A = (x_1{\downarrow} = i(X_2)) \\
 & C = (f(x_1) > x_1) \\
f(X_1) & [] \\
 & A = (x_1{\downarrow} = X_1) \\
 & \quad \wedge (X_1 \neq i(X_2)) \\
 & C = (f(x_1) > x_1)
\end{array}$$

Note that x' comes from the renaming of x in the first rule. Here, the first constraint formula A is satisfiable by any instantiation θ such that $\theta X_2 = 0$ and $\theta x_1 = i(0)$. The second constraint formula is satisfied by any instantiation θ such that $\theta x_1 = \theta X_1 = \theta X_2 = 0$.

Narrow−Y expresses the fact that $\sigma f(X_1)$ is reducible if σ is such that $\sigma X_1 = i(X_2)$, and that the other instances ($\sigma' f(X_1)$ with $\sigma' X_1 \neq i(X_2)$) cannot be reduced.

$$\begin{array}{ll}
\textbf{Stop} & \\
ite(zero(X_2), g(X_2), f(h(X_2))) & \sigma = (X_1 = i(X_2) \\
 & \quad \wedge x' = X_2) \\
 & [1;0] \\
 & A = (x_1{\downarrow} = i(X_2)) \\
 & C = (f(x_1) > x_1) \\
\emptyset & [] \\
 & A = (x_1{\downarrow} = X_1) \\
 & \quad \wedge (X_1 \neq i(X_2)) \\
 & C = (f(x_1) > x_1)
\end{array}$$

Stop applies, ending the second branch.

Abstract	
$ite(X_3, g(X_2), f(h(X_2)))$	$[0]$ $A = (x_1{\downarrow} = i(X_2)$ $\wedge\, zero(X_2){\downarrow} = X_3)$ $C = (f(x_1) > x_1)$
$\emptyset$	$[]$ $A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$ $C = (f(x_1) > x_1)$

The constraint formula A is satisfiable with any instantiation θ such that $\theta X_2 = 0$, $\theta X_3 = true$ and $\theta x_1 = i(0)$.

Abstract applies here, since $zero(X_2)$ can be abstracted, thanks to Proposition 1. Indeed,

$$\mathcal{U}(zero(X_2)) = \{zero(0) \to true, zero(s(x)) \to false\},$$

and both rules can be oriented by a LPO $\succ$ with the precedence $zero \succ_{\mathcal{F}} true$ and $zero \succ_{\mathcal{F}} false$. Then we have $TERMIN(zero(X_2))$.

Narrow−Y	
$g(X_4)$	$\sigma = (X_3 = true \wedge X_2 = X_4$ $\wedge\, x'' = g(X_4) \wedge y'' = f(h(X_4)))$ $[1]$ $A = (x_1{\downarrow} = i(X_4)$ $\wedge\, zero(X_4){\downarrow} = true)$ $C = (f(x_1) > x_1)$
$f(h(X_4))$	$\sigma = (X_3 = false \wedge X_2 = X_4$ $\wedge\, x'' = g(X_4) \wedge y'' = f(h(X_4)))$ $[1; 0]$ $A = (x_1{\downarrow} = i(X_4)$ $\wedge\, zero(X_4){\downarrow} = false)$ $C = (f(x_1) > x_1)$
$ite(X_3, g(X_2), f(h(X_2))$	$[]$ $A = (x_1{\downarrow} = i(X_2) \wedge zero(X_2){\downarrow} =$ $X_3) \wedge (X_3 \neq true \wedge X_3 \neq false)$ $C = (f(x_1) > x_1)$
$\emptyset$	$[]$ $A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$ $C = (f(x_1) > x_1)$

The first constraint formula A is satisfiable by any instantiation θ such that $\theta X_4 = 0$ and $\theta x_1 = i(0)$. The second one is satisfiable by any instantiation θ such that $\theta X_4 = s(0)$ and $\theta x_1 = i(s(0))$. The third one is satisfiable by any instantiation θ such that $\theta X_3 = zero(i(0))$, $\theta X_2 = i(0)$ and $\theta x_1 = i(i(0))$.

The following step ends the third branch, whose strategy evaluation list is empty.

Stop

$g(X_4)$ $[1]$
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = true)$
$C = (f(x_1) > x_1)$

$f(h(X_4))$ $[1;0]$
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = false)$
$C = (f(x_1) > x_1)$

$\emptyset$ $[]$
$A = (x_1{\downarrow} = i(X_2) \wedge zero(X_2){\downarrow} = X_3)$
$\wedge (X_3 \neq true \wedge X_3 \neq false)$
$C = (f(x_1) > x_1)$

$\emptyset$ $[]$
$A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$
$C = (f(x_1) > x_1)$

Abstract $(twice)$

$g(X_4)$ $[]$
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = true)$
$C = (f(x_1) > x_1)$

$f(X_5)$ $[0]$
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = false$
$\wedge h(X_4){\downarrow} = X_5)$
$C = (f(x_1) > x_1)$

$\emptyset$ $[]$
$A = (x_1{\downarrow} = i(X_2) \wedge zero(X_2){\downarrow} = X_3)$
$\wedge (X_3 \neq true \wedge X_3 \neq false)$
$C = (f(x_1) > x_1)$

$\emptyset$ $[]$
$A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$
$C = (f(x_1) > x_1)$

Abstract trivially applies on $g(X_4)$: since X_4 is an NF-variable, there is no need to abstract it.

The second **Abstract** applies to $f(h(X_4))$, thanks to Proposition 1. Indeed,

$$\mathcal{U}(h(X_4)) = \{h(0) \to i(0), h(x) \to s(i(x))\}$$

and both rules can be oriented by the same LPO as previously with the additional precedence $h \succ_{\mathcal{F}} i$ and $h \succ_{\mathcal{F}} s$. Then we have $TERMIN(h(X_4))$.

The first constraint formula has not changed, while the second one is now satisfiable by any instantiation θ such that $\theta X_5 = s(i(s(0)))$, $\theta X_4 = s(0)$ and $\theta x_1 = i(s(0))$.

Narrow−N (*on the second branch*)

$g(X_4)$ []
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = true)$
$C = (f(x_1) > x_1)$

$f(X_5)$ []
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = false \wedge h(X_4){\downarrow} = X_5)$
$C = (f(x_1) > x_1)$

$\emptyset$ []
$A = (x_1{\downarrow} = i(X_2) \wedge zero(X_2){\downarrow} = X_3) \wedge (X_3 \neq true \wedge X_3 \neq false)$
$C = (f(x_1) > x_1)$

$\emptyset$ []
$A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$
$C = (f(x_1) > x_1)$

One could have tried to narrow $f(X_5)$, by using the first rule and the narrowing substitution $\sigma = (X_5 = i(X_6) \wedge x''' = X_6)$. But then $\sigma A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = false \wedge h(X_4){\downarrow} = i(X_6))$. For any θ satisfying A, θ must be such that $\theta h(X_4){\downarrow} = h(\theta X_4{\downarrow}){\downarrow} = i(\theta X_6)$. If $\theta X_4{\downarrow} \neq 0$, then, according to $\mathcal{R}$, $h(\theta X_4{\downarrow}) \rightarrow s(i(\theta X_4{\downarrow}))$, where s is a constructor. Then we cannot have $h(\theta X_4{\downarrow}){\downarrow} = i(\theta X_6)$, and therefore θ must be such that $\theta X_4{\downarrow} = 0$. But then $\theta zero(X_4){\downarrow} = true$, which makes A unsatisfied. Therefore there is no narrowing.

The process is ended by a double application of **Stop**.

Stop (*twice*)

$\emptyset$ []
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = true)$
$C = (f(x_1) > x_1)$

$\emptyset$ []
$A = (x_1{\downarrow} = i(X_4) \wedge zero(X_4){\downarrow} = false \wedge h(X_4){\downarrow} = X_5)$
$C = (f(x_1) > x_1)$

$\emptyset$ []
$A = (x_1{\downarrow} = i(X_2) \wedge zero(X_2){\downarrow} = X_3) \wedge (X_3 \neq true \wedge X_3 \neq false)$
$C = (f(x_1) > x_1)$

$\emptyset$ []
$A = (x_1{\downarrow} = X_1) \wedge (X_1 \neq i(X_2))$
$C = (f(x_1) > x_1)$

As for the defined symbols $ite, zero, h$, the inference rules apply successfully through one **Abstract**, **Narrow−Y**, **Abstract** without abstraction, **Narrow−N** and **Stop** application. Therefore $\mathcal{R}$ is LS-terminating.

5 Conclusion

In this paper, we have proposed a method to prove termination of term rewriting with local strategies on operators by explicit induction on the termination property. Our method works on the ground term algebra using as induction relation an $\mathcal{F}$-stable ordering having the subterm property. The general proof principle relies on the simple idea that for establishing termination of a ground term t, it is enough to suppose that terms smaller than t for this ordering are terminating, and that rewriting the context leads to terminating chains. Iterating this process until obtaining a context which is not reducible anymore establishes the termination of t.

More precisely, the method is applied on terms of the form $g(x_1, \ldots, x_m)$, where g is a defined symbol, and consists in iterating the application of two steps: an abstraction step, replacing immediate subterms by NF-variables, representing any of their normalized instances, and a narrowing step, reducing the resulting term according to the different possible instances of the NF-variables. These two steps are iterated until getting a term for which one can easily say that all ground instances are terminating. The important point to automatize our proof principle is the satisfaction of the ordering constraints for $\mathbf{Abstract}$, $\mathbf{Abstract{-}Stop}$ and $\mathbf{Stop{-}Ind}$.

On many examples as those shown in the paper, this is immediate since they are ensured by the subterm property. In many other cases, a LPO is sufficient to satisfy these constraints. Note that such an LPO does not suffice when it is used in the classical way (any left-hand side of rule is greater than any corresponding right-hand side) since we can handle systems that are not terminating for standard rewriting. Testing satisfiability of a constraint formula A remains simple to handle in practice.

We now have a semi-automatic implementation of the inference rules and the strategy for the leftmost innermost case, that can be expressed by a local strategy on operators. It has been implemented in ELAN [4], which is a logical environment for specifying and prototyping deduction systems in a rule based language with strategies. In this case, as any list of reduction positions ends with the top position, the formula A never contains disequations. Sufficient conditions are implemented to detect unsatisfiability of A, by identifying the reducible right-hand sides. The subterm property of the induction ordering to be found is also implemented, allowing the first application of the rule $\mathbf{Abstract}$ to be completely automatic. Given a TRS, the program interacts with the user and builds the derivation tree resulting from the application of the inference rules according to the strategy we have defined in this paper [8]. Execution examples are available[1]. We also have proposed a variant of the previous implementation, reducing the interaction with the user, by ignoring the satisfiability problem of A. In this case, the obtained proof derivation tree contains the tree we would obtain in using A: states for which A is not satisfiable just correspond to empty sets of ground terms. We thus have in general more computations, but with considerably less user interactions. Moreover, for many examples, the subterm property is the only required property on the induction ordering, so there are no user interactions to test the satisfiability of C and the algorithm is completely automatic. We are now working on automatizable sufficient conditions for proving satisfiability of A.

[1] `http://www.loria.fr/~fissore/Demo/description.html`

Proofs and additional examples can be found in the full version of the paper [9].

References

1. T. Arts and J. Giesl. Proving innermost normalisation automatically. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer-Verlag, 1997.
2. T. Arts and Giesl. J. Proving innermost normalization automatically. Technical Report 96/39, Technische Hochschule Darmstadt, Germany, 1996.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and Ch. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. Second Intl. Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson (France), September 1998. Elsevier.
4. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. Report LORIA 98-R-316.
5. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications*, volume 5 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, USA, September 1996. North Holland.
6. Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.
7. S. Eker. Term rewriting with operator evaluation strategies. In C Kirchner and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, Pont-à-Mousson, France, September 1998.
8. O. Fissore. Terminaison par induction. Mémoire de DEA, Université Henri Poincaré – Nancy 1, June 2000.
9. O. Fissore, I. Gnaedig, and H. Kirchner. Termination of rewriting with local strategies. Technical report, LORIA, Nancy, France, 2001. http://www.loria.fr/˜gnaedig/INDO/STRAT-LO/strat-lo.ps.gz.
10. K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.
11. J. Giesl and Middeldorp A. Transforming Context-Sensitive Rewrite Systems. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pages 271–285, Trento, Italy, 1999. Springer-Verlag.
12. I. Gnaedig, H. Kirchner, and O. Fissore. Induction for Termination. Technical Report A00-R-357, LORIA, Nancy (France), 2000.
13. I. Gnaedig, H. Kirchner, and O. Fissore. Induction for innermost and outermost ground termination. Internal report, submitted, 2001. http://www.loria.fr/˜fissore/Papers/in-out.ps.gz.
14. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ3. Technical report, Computer Science Laboratory, SRI International, march 1992.
15. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

16. S. Lucas. Termination of context-sensitive rewriting by rewriting. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 122–133. Springer-Verlag, 1996.
17. S. Lucas. Context-sensitive rewriting strategies. Technical Report DSIC-II/7/00, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Spain, 2000.
18. Nakamura M. and Ogata K. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proceedings of the Third International Workshop on Rewriting Logic and its Applications, WRLA'2000*, pages 211–227, Kanazawa City Cultural Halt, Kanazawa, Japan, September 2000. Electronic Notes in Theoretical Computer Science.
19. E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description for Stratego 0.5. In Aart Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*. Springer-Verlag, 2001. To appear.
20. H. Zantema. Termination of context-sensitive rewriting. In *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 1997.

An Algorithm for Guiding Temporal Resolution*

M. Carmen Fernández Gago, Michael Fisher, and Clare Dixon

Department of Computer Science, University of Liverpool, L69 7ZF, United Kingdom
{M.C.Gago, M.Fisher, C.Dixon}@csc.liv.ac.uk

Abstract. The clausal resolution method developed for discrete temporal logics involves translation to a normal form, classical resolution on formulae within states (termed step resolution) and temporal resolution between states. Step resolution may generate an unnecessarily large set of clauses. In addition, the most expensive part of the method is the application of the temporal resolution operation. In this paper we develop an algorithm to guide the search for the set of clauses needed for the application of temporal resolution. The algorithm is based on the outputs of a refined temporal resolution rule which allows us to generate temporal resolvents earlier within the process. In particular, this can also help us to avoid unnecessary step resolution and focus search for the most relevant clauses.

1 Introduction

The effective mechanisation of temporal logic is vital to the application of temporal reasoning in many fields, for example the verification of reactive systems [11], the implemention of temporal query languages [4], and temporal logic programming citeAbadiManna89-templog. Consequently, a range of proof methods have been developed, implemented and applied. The development of proof methods for temporal logic has followed three main approaches: tableau [17], automata [15] and resolution [2, 3, 9, 16]. Resolution based methods have the advantage that, as in the classical case [13], a range of strategies can potentially be used. Also the complexity of satisfiability of PTL, the logic used in this paper, is PSPACE-complete [14] which motivates the need for strategies to guide proof search.

In classical resolution a popular strategy has been the set of support strategy [18], which restricts the application of the resolution rule, pruning the search space. Our general aim here is to develop a set of support strategy to temporal resolution. We have extended this strategy for PTL without eventualities ('$\Diamond$', meaning sometime in the future) as this follows from the classical case. The extension of the strategy to full PTL is non trivial and we intend to achieve it using the algorithm which we propose in this paper together with the revised temporal resolution rule presented.

Throughout, we use a proof method for temporal logics based upon the use of clausal resolution [9]. The resolution procedure is characterised by the translation to a normal form, the application of a classical style resolution between formulae that occur at the same moment in time (*step resolution*), together with a novel *temporal*

* This work was partially supported by EPSRC grant GR/M44859.

resolution rule, which derives contradictions over temporal sequences. Although the clausal temporal resolution method has been defined, proved correct and implemented, it sometimes generates an unnecessarily large set of formulas that may be irrelevant to the refutation. Not only that, but temporal resolution operations occur only after all step resolution inferences have been carried out. Thus no search for temporal resolution can be made earlier in the process. This means that in cases where a large amount of step resolution can occur the method may be very expensive.

The temporal resolution operation requires us to search for a set of clauses which satisfy a specified condition. As the search for candidates for this operation is the most expensive part but, is certain to be required for the temporal resolution process, our intention is to try to avoid all unnecessary step resolution operations and apply temporal resolution earlier in the process. In this sense, we propose an algorithm (and show soundness, completeness and termination of it) based on the outputs of a revised temporal resolution rule [8]. This temporal resolution rule allows us to generate temporal resolvents earlier, with the resolvents generated then being used in order to guide further search.

The structure of the paper is as follows. In section 2 we define the temporal logic considered, namely Propositional Temporal logic (PTL) [10]. In section 3 we review the basic resolution method while in section 4 we introduce the refinement of the temporal resolution rule which will be used in the rest of the paper. In the subsequent sections we propose the algorithm for guiding the temporal resolution search and give an example of its use.

2 Syntax and Semantics of PTL

In this section we present the syntax and semantics of a discrete, linear temporal logic with finite past and infinite future (PTL). The future-time connectives that we use include '$\Diamond$' (*sometime in the future*), '$\bigcirc$' (*in the next moment in time*), '$\Box$' (*always*) '$\mathcal{U}$' (*until*), and '$\mathcal{W}$' (*unless, or weak until*). A choice for domain in which to interpret such temporal connectives is $\mathbb{N}$, i.e., the Natural Numbers ordered by the usual 'less than' relation.

Formulae are constructed using the following connectives and proposition symbols.
- A set, $\mathcal{P}$, of propositional symbols.
- Nullary connectives: **true** and **false**.
- Propositional connectives: $\neg$, $\vee$, $\wedge$ and $\Rightarrow$
- Temporal connectives: $\bigcirc$, $\Diamond$, $\Box$, $\mathcal{U}$, $\mathcal{W}$ and **start**.

The set of well-formed formulae of PTL[1], denoted by WFF_p, is defined as follows:
- Any element of $\mathcal{P}$ is in WFF_p.
- **true**, **false** and **start** are in WFF_p.
- If ϕ and ψ are in WFF_p then so are

$\neg\phi$, $\phi \vee \psi$, , $\phi \wedge \psi$, $\phi \Rightarrow \psi$, $\Diamond\phi$, $\Box\phi$, $\phi\mathcal{U}\psi$, $\phi\mathcal{W}\psi$, $\bigcirc\phi$

We define a model, M, for PTL as a structure $\langle \mathbb{N}, <, \pi_p \rangle$ where $\pi_p : \mathbb{N} \times \mathcal{P} \rightarrow T, F$ is a function assigning T or F to each atomic proposition at each moment in time.

[1] As usual, parentheses are also allowed to avoid ambiguity

As usual we define the semantics of the language via the satisfaction relation '$\models$'. For PTL, this relation holds between pairs of the form $\langle M, u\rangle$ (M is a model and u$\in$ $\mathbb{N}$) and well-formed formulae. The rules defining the satisfaction relation are as follows

$\langle M, u\rangle \models p$ iff $\pi_p(u, p) = T$ (where $p \in \mathcal{P}$)
$\langle M, u\rangle \models \mathbf{true}$
$\langle M, u\rangle \models$ **start** iff $u = 0$
$\langle M, u\rangle \models \phi \wedge \psi$ iff $\langle M, u\rangle \models \phi$ and $\langle M, u\rangle \models \psi$
$\langle M, u\rangle \models \neg\phi$ iff $\langle M, u\rangle \not\models \phi$
$\langle M, u\rangle \models \bigcirc\phi$ iff $\langle M, u+1\rangle \models \phi$
$\langle M, u\rangle \models \square\phi$ iff for all $j \in \mathbb{N}$, if $j \geq u$ then $\langle M, u\rangle \models \phi$
$\langle M, u\rangle \models \phi\,\mathcal{U}\,\psi$ iff there exists a $k \in \mathbb{N}$, s.t. $k \geq u$ and $\langle M, k\rangle \models \psi$ and for all $j \in \mathbb{N}$, if $i \leq j < k$ then $\langle M, j\rangle \models \phi$

We define '$\vee$', '$\Rightarrow$', '$\Leftrightarrow$', '$\diamondsuit$' and '$\mathcal{W}$' as follows:

$$\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi) \quad \phi \Leftrightarrow \psi \equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi) \quad \phi\mathcal{W}\psi \equiv (\phi\,\mathcal{U}\,\psi) \vee \square\phi$$
$$\phi \Rightarrow \psi \equiv \neg\phi \vee \psi \quad \diamondsuit\phi \equiv \neg\square\neg\phi \quad \textbf{false} \equiv \neg\,\textbf{true}$$

3 Clausal Resolution Method for PTL

Discrete temporal logics are diffcult to reason about principally because we must make sure that formulae being resolved holds at the same moment in time. Further problems occur due to the inductive interaction between the $\square$-operator and $\bigcirc$-operator. The resolution method described in [9] addresses these problem, by utilising a normal form, called Separated Normal Form (SNF), which separates out complex formulae from their contexts through the use of *renaming* [12], and a new temporal resolution operation introduced specifically for formulae in the normal form.

3.1 Separated Normal Form

The resolution method depends on formulae being transformed into a normal form (SNF), inspired by Gabbay's separation result [10]. The normal form, which is presented in [7], comprises formulae that are implications with present-time formulae on the left-hand side and (present or) future-time formulae on the right-hand-side. The transformation of formulae into SNF depends on three main operations: the renaming of complex subformulae; the removal of temporal operators; and classical style rewrite operations. In this section we review SNF but do not consider the transformation procedure (we note that the transformation to SNF preserves satisfiability [7]).
Formulae in SNF are of the general form $\square \bigwedge_i(\phi_i \Rightarrow \psi_i)$, where each $\phi_i \Rightarrow \psi_i$ is known as a *clause* and is one of the following forms

$\textbf{start} \Rightarrow \bigvee_c l_c$ (an *initial* clause)

$\bigwedge_a k_a \Rightarrow \bigcirc \bigvee_d l_d$ (a *step* clause)

$\bigwedge_b k_b \Rightarrow \diamondsuit l$ (a *sometime* clause)

where each k_a, k_b, l_c, l_d and l represent literals.

3.2 Resolution Operations

Step resolution consists of the application of the standard classical resolution rule in two different contexts. Pairs of initial or step clauses may be resolved as follows:

$$\frac{\begin{array}{l}\textbf{start} \Rightarrow \psi_1 \lor l\\ \textbf{start} \Rightarrow \psi_2 \lor \neg l\end{array}}{\textbf{start} \Rightarrow \psi_1 \lor \psi_2} \qquad \frac{\begin{array}{r}\phi_1 \Rightarrow \bigcirc(\psi_1 \lor l)\\ \phi_2 \Rightarrow \bigcirc(\psi_2 \lor \neg l)\end{array}}{(\phi_1 \land \phi_2) \Rightarrow \bigcirc(\psi_1 \lor \psi_2)}$$

The *simplification operations* are similar to those used in the classical case, consisting of both simplification and subsumption. An additional operation is required when a contradiction in a state is produced:

$$\frac{\phi \Rightarrow \bigcirc\textbf{false}}{\begin{array}{l}\textbf{start} \Rightarrow \neg\phi\\ \textbf{true} \Rightarrow \bigcirc\neg\phi\end{array}}$$

This means that, if a formula ϕ leads to a contradiction in the next moment, then ϕ must never be satisfied.

Temporal resolution operations resolve one sometime clause with a set of step clauses [9] as follows:

$$\frac{\begin{array}{rcl}\phi_1 & \Rightarrow & \bigcirc\psi_1\\ \vdots & \vdots & \vdots\\ \phi_n & \Rightarrow & \bigcirc\psi_n\\ \chi & \Rightarrow & \Diamond\neg l\end{array}}{\chi \Rightarrow (\neg \bigvee_{i=1}^{n} \phi_i)\mathcal{W}\neg l}$$

with the side condition that for all i, $1 \leq i \leq n$, then $\models \psi_i \Rightarrow l$ and $\models \psi_i \Rightarrow \bigvee_{i=1}^{n} \phi_i$, from which we can derive $\bigwedge_{i=1}^{n}(\phi_i \Rightarrow \bigcirc(l \land \bigvee_{j=1}^{n} \phi_j))$. This side condition ensures that the set of $\phi_i \Rightarrow \bigcirc\psi_i$ clauses together imply $\bigvee_{i=1}^{n} \phi_i \Rightarrow \bigcirc\Box l$. Such a set of clauses is known as a *loop* in l. The resolvent produced includes a $\mathcal{W}$ operator that must be translated into SNF before any further resolution steps can be applied.

Termination. If **start**$\Rightarrow$ **false** is produced, the original formula is unsatisfiable and the resolution process terminates.

Correctness. The soundness and (refutation) completeness of the original temporal resolution method have been established in [9].

4 A Refined Rule

In this paper we use a version of the temporal resolution operation described in [8]. The basic idea behind this operation is that, rather than insisting we already have a set of clauses that characterise a loop, we derive a more complex resolvent that allows for the possibility that such a loop does not exist. If the set of clauses chosen do comprise a loop, then the new resolvent turns out to be the resolvent from the original temporal resolution rule. More importantly, using this revised temporal resolution rule, if the clauses chosen do not comprise a loop, we will show, in this paper, how the resolvents produced can be used to guide further resolution in such a way that allows us to detect a loop if possible.

This new temporal resolution rule is

$$\frac{\begin{array}{rcl} \phi_1 & \Rightarrow & \bigcirc\psi_1 \\ \vdots & \vdots & \vdots \\ \phi_n & \Rightarrow & \bigcirc\psi_n \\ \chi & \Rightarrow & \Diamond\neg l \end{array}}{\chi \Rightarrow \left[\Box(\bigwedge_{i=1}^{n}(\phi_i \Rightarrow \bigcirc(l \wedge \bigvee_{j=1}^{n}\phi_j))) \Rightarrow (\neg\bigvee_{i=1}^{n}\phi_i)\mathcal{W}\neg l\right]}$$

Notice that there is no side-condition. This has been incorporated into the resolvent, which can be further transformed using classical manipulation to

$$\chi \Rightarrow \left[(\Diamond\neg\bigwedge_{i=1}^{n}(\phi_i \Rightarrow \bigcirc(l \wedge \bigvee_{j=1}^{n}\phi_j))) \vee (\neg\bigvee_{i=1}^{n}\phi_i)\mathcal{W}\neg l\right]$$

This new resolvent can be seen to introduce a new eventuality, $\Diamond\neg$**looping**, (where **looping** is the side condition for the original resolution rule), that effectively provides a check to establish that an appropriate loop was chosen. If an appropriate loop is not present, then this check will fail.

Once we rename $\Diamond\neg$**looping** as $\Diamond s$ and $s \Rightarrow \neg$**looping** then step resolution will give us **true**$\Rightarrow \bigcirc\neg s$ if we have chosen the correct loop (as **looping** is valid. This will be proved below) and the *temporal check* (essentially a minimal form of temporal resolution) will take place between the clauses $\Diamond s$ and **true**$\Rightarrow \bigcirc\neg s$.

As the choice of ϕ_i in temporal resolution is crucial, our main aim is to guide the search for a loop using this new temporal resolution rule. More specifically, we will use the output of a failed temporal check process to guide the subsequent search for loops. We proceed as follows. We guess a formula for the ϕ_i such that if our guess is correct, that is $\bigvee_{i=1}^{n}\phi_i \Rightarrow \bigcirc\Box l$, then the loop search phase terminates. Otherwise we use output from subsequent resolution steps to make a new guess based on the previous one. We show that this process terminates with the detection of a loop if one exists.

We assume that correct loops are in their simplest form, that is, factoring rule (similar to the classical case) and sbsumption have been applied wherever possible.

5 Algorithm

In this section we propose an algorithm to guide the search for a loop based on the temporal resolution rule introduced in section 4. For each eventuality $\Diamond\neg l$ occuring on the right hand side of a sometime clause, the algorithm constructs a sequence of DNF formulae, G_i, using the previous one and F_j, where F_j are disjunctions of literals which appear in the outputs of the revised temporal resolution rule. The algorithm is the following.

1. Choose $G_{-1} =$ **true**.
2. Given G_i apply the revised temporal resolution rule to it.
3. For all clauses **true** $\Rightarrow \bigcirc(\neg s \vee F_j)$, $1 \leq j \leq m$ obtained during the last proof attempt, then G_{i+1} is obtained as $G_{i+1} = G_i \wedge (\bigvee_{j=1}^{m} \neg F_j)$.
4. Go to 2 until either
   (a) **true** $\Rightarrow \bigcirc\neg s$ is obtained and, hence, a loop has been found or
   (b) $G_{i+1} =$ **false**, in which case we terminate without having found a loop.

6 Completeness

In the following we will prove completeness for this algorithm by relating it to the completeness of the Breadth-First Search Algorithm proposed in [5]. First, we introduce the Breadth-First Search Algorithm.

6.1 Breadth-First Search Algorithm

The Breadth-First Search Algorithm constructs a sequence of formulae, H_i for $i \geq 0$, that are formulae in Disjunctive Normal Form and contain no temporal operators. They are constructed from the conjunctions of literals on the left hand sides of step clauses or combinations of step clauses in the SNF-clause-set that satisfy certain properties (see below). Assuming we are resolving with $\Diamond\neg l$ each formula H_i satisfies $H_i \Rightarrow \bigcirc l$ and given H_i each new formula H_{i+1} satisfies $H_{i+1} \Rightarrow \bigcirc H_i$. When termination occurs we have $H_{i+1} \Leftrightarrow H_i$ so that $H_i \Rightarrow \bigcirc\Box l$ for resolution with $\Diamond\neg l$. We assume that all necessary step resolution has been carried out.

Breadth-First Search Algorithm For each eventuality $\Diamond\neg l$ occuring on the right hand side of a sometime clause do the following.

1. Search for all the step clauses of the form $C_k \Rightarrow \bigcirc l$, for $k = 0$ to b (called *start PTL-clauses*), disjoin the left hand sides and generate the *top formula* H_0 equivalent to this, i.e.
$$H_0 \Leftrightarrow \bigvee_{k=0}^{b} C_k.$$
   Simplify H_0. If $\models H_0$ we terminate having found a loop-formula (**true**).

2. Given formula H_i, build formula H_{i+1} for $i = 0, 1, \ldots$ by looking for step clauses or combinations of clauses of the form $A_j \Rightarrow \bigcirc B_j$, for $j = 0mboxtoc$ where $\models B_j \Rightarrow H_i$ and $\models A_j \Rightarrow H_0$. Disjoin the left hand sides so that

$$H_{i+1} \Rightarrow \bigvee_{j=0}^{c} A_j$$

and simplify as previously.

3. Repeat (2) until
   (a) $\models H_i$. We terminate having found a loop-formula and return **true**.
   (b) $\models H_i \Leftrightarrow H_{i+1}$. We terminate having found a loop-formula and return the DNF formula H_i.
   (c) The new formula is empty. We terminate without having found a loop-formula.

Soudness, Completeness and Termination for the BFS-algorithm Given a set of SNF clauses R, that contains a loop $A \Rightarrow \bigcirc \Box l$, applying BFS algorithm will output a DNF formula A' such that $A' \Rightarrow \bigcirc \Box l$ and $A \Rightarrow A'$. Termination is also established.[5]

6.2 Completeness of the New Algorithm

To show the completeness of the algorithm we will prove that for all $i \geq 0, G_i \Leftrightarrow H_i$ by induction. Let R be a set of SNF-clauses and $\Diamond \neg l$ be the right hand side of a sometime clause, we assume that R contains a loop in l.

Lemma 2. $G_0 \Leftrightarrow H_0$

Proof (Outline). In order to obtain G_0, according to the algorithm, the revised temporal resolution rule must be applied to G_{-1}. Some of the clauses derived during the proof are **true** $\Rightarrow \bigcirc(\neg s \vee$ **true**) and $s \Rightarrow \bigcirc(\neg l \vee \neg$**true**), which are the clauses **true** $\Rightarrow \bigcirc$**true** and $s \Rightarrow \bigcirc \neg l$ (1).

As R contains a loop, in the initial set of clauses there must be some clauses such that they may be resolved together to obtain $A_i \Rightarrow \bigcirc l$, $1 \leq i \leq k$. By resolution with clause (1) the resolvents are $s \wedge A_i \Rightarrow \bigcirc$**false**, $1 \leq i \leq k$, and by applying simplification to the above clauses, **true** $\Rightarrow \bigcirc(\neg s \vee \neg A_i)$, $1 \leq i \leq k$, is obtained. These latter clauses are used to obtain G_0 as follows: $G_0 =$ **true** $\wedge (A_1 \vee ... \vee A_k) = A_1 \vee ... \vee A_k$. In building H_0 by Breadth-First Search Algorithm, the left hand sides of the clauses $A_i \Rightarrow \bigcirc l$ are disjoined, giving, $H_0 \Leftrightarrow A_1 \vee ... \vee A_k$. Hence, $G_0 \Leftrightarrow H_0$

Theorem 1. *For all* $i \in \mathbb{N}$ $H_{i+1} \Leftrightarrow G_{i+1}$.

Proof (Outline).
<u>Base case</u>: By Lemma 2 $H_0 \Leftrightarrow G_0$.
<u>Induction case</u>: We assume $H_k \Leftrightarrow G_k$ for all $k \leq i, k \in \mathbb{N}$ and we prove the hypothesis for $i + 1$. We know the following valid statements about H_{i+1} from the definition of the Breadth First Search algorithm:

$(a).\ H_{i+1} \Rightarrow \bigcirc H_i$ $(c).\ \ G_i \ \Leftrightarrow H_i$ (Induction Hypothesis)
$(b).\ H_{i+1} \Rightarrow \ \bigcirc l$ $(d).\ H_{i+1} \Rightarrow H_i$

Because of the translation into the normal form of the resolvent obtained from applying the new temporal resolution rule to G_i, we obtain the clauses
1. $s \Rightarrow \bigcirc(\neg l \vee \neg G_i)$ and 2. **true** $\Rightarrow \bigcirc(\neg s \vee G_i)$.
Using property (c) these two clauses are transformed to
3. $s \Rightarrow \bigcirc(\neg l \vee \neg H_i)$ and 4. **true** $\Rightarrow \bigcirc(\neg s \vee H_i)$
Then, applying step resolution, we obtain:
5. $s \wedge H_{i+1} \Rightarrow \bigcirc \neg H_i$ [Step Resolution b,3]
6. $s \wedge H_{i+1} \Rightarrow \bigcirc$**false** [Step Resolution 5,a]
7. **true** $\Rightarrow \bigcirc(\neg s \vee \neg H_{i+1})$ [Simp.6]
In order to obtain G_{i+1} the algorithm is applied, where the clauses **true** $\Rightarrow \bigcirc(\neg s \vee F_j)$ considered in this case are 4 and 7.
$G_{i+1} = G_i \wedge [\neg H_i \vee H_{i+1}] = G_i \wedge [\neg G_i \vee H_{i+1}] = (G_i \wedge \neg G_i) \vee (G_i \wedge H_{i+1}) =$ **false** $\vee\ (H_i \wedge H_{i+1}) = H_i \wedge H_{i+1}$
and by property (d) $H_i \wedge H_{i+1} \Leftrightarrow H_{i+1}$, which means $G_{i+1} \Leftrightarrow H_{i+1}$

Theorem 2. *Let L be a DNF formula. Then by applying the new temporal resolution rule, the clause* **true** $\Rightarrow \bigcirc\neg s$ *is obtained if L is a loop.*

Proof (Outline). Let $L = D_1 \vee D_2 \vee ... \vee D_n$. We are assuming L is a loop, i.e., $\square[L \Rightarrow \bigcirc(L \wedge l)]$. After renaming, the clauses $t \Rightarrow \Diamond s$ and $s \Rightarrow \neg[L \Rightarrow \bigcirc(L \wedge l)]$ are obtained.

Because of the translation of the second clause into the normal form, the clauses produced are $s \Rightarrow L$ and $s \Rightarrow \bigcirc(\neg l \vee \neg L)$. The first clause produces **true** $\Rightarrow \bigcirc(\neg s \vee D_1 \vee ... \vee D_n)$ (1) and the second one, $s \Rightarrow \bigcirc(\neg l \vee \neg D_i)$ (B_i) $1 \leq i \leq n$.

As L is a loop, there must be a set of clauses in the initial set of clauses such that together represent $L \Rightarrow \bigcirc l$ and $L \Rightarrow \bigcirc L$. Clauses $B_1, ..., B_n$ together with the previous ones produce the clauses $s \wedge D_i \Rightarrow \bigcirc$**false**, $1 \leq i \leq n$. Applying simplification to these clauses we obtain **true** $\Rightarrow \bigcirc(\neg s \vee \neg D_1)$, $1 \leq i \leq n$. The previous clauses can be resolved with clause (1) producing **true**$\Rightarrow \bigcirc\neg s$.

Theorem 3. *If G_i is a loop, then $G_i = G_{i+1}$.*

Proof. By Theorem 2, if G_i is a loop, then **true** $\Rightarrow \bigcirc\neg s$ is generated, so according to the algorithm $G_{i+1} = G_i \wedge [\neg$**false**$] = G_i \wedge$ **true** $= G_i$

Theorem 4. *The algorithm terminates.*

Proof. 1. If there exists a loop in the initial set of clauses, then we know that Breadth Search Algorithm terminates finding a loop H_n. By Theorem 1 $H_n \Leftrightarrow G_n$, so G_n is a loop and then by Theorem 2 **true** $\Rightarrow \bigcirc\neg s$ is obtained.
2. If there does not exist a set of clauses which comprise a loop in the initial set of clauses, then at some point it will not be possible to produce the clause $s \wedge C_i \Rightarrow \bigcirc$**false** and therefore no clauses **true** $\Rightarrow \bigcirc(\neg s \vee F_i)$ will be derived during the proof apart from the clause **true** $\Rightarrow \bigcirc(\neg s \vee G_n)$, then $G_{n+1} = G_n \wedge [\neg G_n] =$ **false**

6.3 Example

Let the loop be $a \wedge b \wedge c \wedge d \Rightarrow \bigcirc \square l$ and the set of SNF clauses be as follows:

1. $a \Rightarrow \bigcirc l$	4. $d \wedge a \Rightarrow \bigcirc b$
2. $b \wedge c \Rightarrow \bigcirc d$	5. $a \wedge b \Rightarrow \bigcirc c$
3. $c \wedge d \Rightarrow \bigcirc a$	6. $\chi \Rightarrow \diamondsuit \neg l$

According to the algorithm the first choice is $G_{-1} = \textbf{true}$. So, some of the resolvents are

7. $\textbf{true} \Rightarrow \bigcirc(\neg s \vee \textbf{true})$
8. $s_{-1} \Rightarrow \bigcirc \neg l$
9. $s_{-1} \wedge a \Rightarrow \bigcirc \textbf{false}$ [SRES 1,8]
10. $\textbf{true} \Rightarrow \bigcirc(\neg s_{-1} \vee \neg a)$ [SIMP 9]

So, the next G_i will be, $G_0 = \textbf{true} \wedge a = a$.

11. $\textbf{true} \Rightarrow \bigcirc(\neg s_0 \vee a)$
12. $s_0 \Rightarrow \bigcirc(\neg l \vee \neg a)$
13. $s_0 \wedge a \wedge c \wedge d \Rightarrow \bigcirc \textbf{false}$ [1,3,12]
14. $\textbf{true} \Rightarrow \bigcirc(\neg s_0 \vee \neg a \vee \neg c \vee \neg d)$ [SIMP 13]
15. $\textbf{true} \Rightarrow \bigcirc(\neg s_0 \vee \neg c \vee \neg d)$ [SRES 11,14]

Clause 15 subsumes clause 14, so $G_1 = a \wedge [\neg a \vee (c \wedge d)] = a \wedge c \wedge d$.

16. $\textbf{true} \Rightarrow \bigcirc(\neg s_1 \vee a)$
17. $\textbf{true} \Rightarrow \bigcirc(\neg s_1 \vee c)$
18. $\textbf{true} \Rightarrow \bigcirc(\neg s_1 \vee d)$
19. $s_1 \Rightarrow \bigcirc(\neg l \vee \neg a \vee \neg c \vee \neg d)$
20. $s_1 \wedge a \wedge b \wedge c \wedge d \Rightarrow \bigcirc \textbf{false}$ [SRES 1,2,3,5,19]
21. $\textbf{true} \Rightarrow \bigcirc(\neg s_1 \vee \neg a \vee \neg b \vee \neg c \vee \neg d)$ [SIMP 20]
22. $\textbf{true} \Rightarrow \bigcirc(\neg s_1 \vee \neg b)$ [SRES 16,17,18,21]

Clause 22 subsumes clause 21. According to the algorithm
$G_2 = (a \wedge c \wedge d) \wedge (\neg a \vee \neg c \vee \neg d \vee b) = \textbf{false} \vee \textbf{false} \vee \textbf{false} \vee (a \wedge c \wedge d \wedge b) = (a \wedge c \wedge d \wedge b)$, which is the correct loop, so if the algorithm is applied again $\textbf{true} \Rightarrow \bigcirc \neg s$ will be obtained, which means termination.

7 Conclusions and Future Work

In this paper we have presented an algorithm based upon a refined clausal temporal resolution rule. This new rule has not only allowed us to develop such an algorithm, but allows more flexibility in the search for a refutation.

The algorithm uses outputs from the previous attempt to select a loop to guide the choice for the next guess and its correctenss is shown with respect to an existing loop search algorithm.

In the future we intend to apply these results to the development of strategies for temporal resolution that allow us to reduce the search space. In particular, we are interested in incorporating the set of support strategy [18, 6], which has been pariculary successful in the classical case. The algorithm presented in this paper provides a basis for such a strategy and allows the flexible interleaving of step resolution and loop search operations.

As our main aim is to develop a set of support strategy for the temporal case, an important task to carry out will be to combine and to compare such a strategy with alternative proof strategies and to examine the practical efficiency of such combined strategies. In this future work the results presented in this paper will be essential.

References

1. M. Abadi and Z. Manna. Temporal Logic Programming. *Journal of Symbolic Computation*, 8: 277–295, 1989.
2. M. Abadi and Z. Manna. Nonclausal Deduction in First-Order Temporal Logic. *ACM Journal*, 37(2):279–317, April 1990.
3. A. Cavalli and L. Farinas del Cerro. A Decision Method for Linear Temporal Logic. In R.E.Shostak, editor, *Proceedings of the 7th International Conference on Automated Deduction*, volume 170 of *LNCS*, pages 113–127. Springer-Verlag, 1984.
4. J. Chomicki and D. Niwinski. On the Feasibility of Checking Temporal Integrity Constraints. *Journal of Computer and System Sciences*, 51(3):523–535, 1995.
5. C. Dixon. Temporal Resolution using a Breadth-First Search Algorithm. *Annals of Mathematics and Artificial Intelligence*, 22:87–115, 1998.
6. C. Dixon and M. Fisher. The Set of Support Strategy in Temporal Resolution. In *Proceedings of TIME-98 the Fifth Workshop on Temporal Representation and Reasoning*, Sanibel Island, Florida, May 1998. IEEE Computer Society Press.
7. M. Fisher. A Normal Form for Temporal Logic and its Application in Theorem-Proving and Execution. *Journal of Logic and Computation*, 7(4):429–456, 1997.
8. M. Fisher and C. Dixon. Guiding Clausal Temporal Resolution. In *Advances in Temporal Logic*, volume 16 of *Applied Logic Series*, pages 167–184. Kluwer, 2000.
9. M. Fisher, C. Dixon, and M. Peim. Clausal Temporal Resolution. In *Transactions on Computational Logic 2(1)*. January 2001.
10. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. The Temporal Analysis of Fairness. In *Proceedings of the 7th ACM Symposium on the Principles of Programming Languages*, pages 163–173, Las Vegas, Nevada, January 1980.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
12. D. A. Plaisted and S. A. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
13. J. A. Robinson. A Machine–Oriented Logic Based on the Resolution Principle. *ACM Journal*, 12(1):23–41, January 1965.
14. A. P. Sistla and E. M. Clarke. Complexity of propositional linear temporal logics. *ACM Journal*, 32(3):733–749, July 1985.
15. A. P. Sistla, M. Vardi, and P. Wolper. The Complementation Problem for Buchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.
16. G. Venkatesh. A Decision Method for Temporal Logic based on Resolution. *Lecture Notes in Computer Science*, 206:272–289, 1986.
17. Pierre Wolper. The Tableau Method for Temporal Logic: An overview. *Logique et Analyse*, 110–111:119–136, June-Sept 1985.
18. L. Wos, G. Robinson, and D. Carson. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *ACM Journal*, 12:536–541, October 1965.

Goal-Driven Inference Search in Classical Propositional Logic

Alexander Lyaletski and Andrey Paskevich

Cyberetics Department, Taras Shevchenko Kyiv National University,
2, Glushkov avenue, building 6, 03022 Kyiv, Ukraine,
tel.: +38 (044) 266 3108, fax: +38 (044) 266 1059,
e-mail: lav@tc.unicyb.kiev.ua, andrey@raptor.kiev.ua

Abstract. Two goal-driven strategies for inference search in propositional logic are presented in the form of special sequent calculi. Some results on their soundness and completeness are given, and comparison of these calculi is made. It is shown that no one of them is preferable to another in the sense of constructing minimal inferences.

1 Introduction

A programme of automated theorem proving in mathematics called Evidence Algorithm (EA) was advanced by Academician V.Glushkov in his paper [1]. The programme EA paid much attention to (interactive) control of the strategies for inference search. The first heuristic routine proposed for EA [2] provided already this possibility. Afterwards, that routine was extended to a complete Kanger-type [3] calculus [4], thus becoming a machine-oriented version of the usual sequent calculus [5]. Because of absence of the skolemization, this calculus has significantly lower efficiency than resolution-type calculi [7]. In order to overcome this obstacle (without introducing the skolemization), an original notion of an admissible substitution [8] have been developed and then have been used in a so-called a-sequent calculus (AS calculus) [9]. This calculus became a core of a prover implemented on the ES-series computers as a part of a software system called SAD (System of Automated Deduction) [10].

In the paper [9] the description of the AS calculus had partially an algorithmic character. This deviation from the traditional Gentzen notation was corrected, when a number of the modifications of AS was constructed in the frame of the INTAS-project 96-0760 (see, for example, [11, 12]). These modifications possesses the following features: goal-oriented reduction of an assertion to be proven to a number of auxiliary assertions; special quantifier handling technique (without the skolemization), which permits to achieve a search efficiency comparable with the efficiency of methods requiring the preliminary skolemization; separation of a deduction from equality processing (equation solving); use of human-like proof methods such as applications of definitions and auxiliary propositions; flexible interactive search mode.

All the above features are discussed in [11, 12]. However, these papers do not contain reasons on positive or negative peculiarities of different strategies for the AS calculus and its modifications. The present paper is intended to fill partially this gap and it concerns two "goal-driven" strategies. Note that the strategies proved to be essentially different even in the case of propositional logic. In this connection these strategies are convenient to define as two different a-sequent calculi. All considerations are restricted to the propositional level.

2 Main Definitions

We consider classical propositional logic with the following connectives: implication ($\supset$), disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$). Below propositional variables, or atoms, are denoted by A, B, C, or D, propositional literals (i.e. atoms and their negations) are denoted by L or M, propositions (formulas) are denoted by F, G, P, or Q. All the letters can be subscripted.

The expression $F^{\neg}$ denotes the result of one-step carrying of the negation into a proposition F: $(P \supset Q)^{\neg}$ is $P \wedge \neg Q$, $(P \vee Q)^{\neg}$ is $\neg P \wedge \neg Q$, $(P \wedge Q)^{\neg}$ is $\neg P \vee \neg Q$, $(\neg P)^{\neg}$ is P, and $A^{\neg}$ is $\neg A$.

We define the *positive* ($P[F^+]$) *and negative* ($P[F^-]$) *occurrences* of a proposition F to a proposition P by the following:

I. $F[F^+]$ and $F[\bar{F}^-]$ always hold, where $\bar{F}$ denotes a proposition G if F is $\neg G$, and $\bar{F}$ denotes $\neg F$ otherwise. ($\bar{F}$ is called a *complementary* of F.)

II. If a proposition F occurs in a proposition P, then:

$$
\begin{array}{llll}
P[F^+] \text{ implies } (\neg P)[F^-] & & P[F^-] \text{ implies } (\neg P)[F^+] \\
P[F^+] \text{ implies } (P \wedge Q)[F^+] & & P[F^-] \text{ implies } (P \wedge Q)[F^-] \\
P[F^+] \text{ implies } (Q \wedge P)[F^+] & & P[F^-] \text{ implies } (Q \wedge P)[F^-] \\
P[F^+] \text{ implies } (P \vee Q)[F^+] & & P[F^-] \text{ implies } (P \vee Q)[F^-] \\
P[F^+] \text{ implies } (Q \vee P)[F^+] & & P[F^-] \text{ implies } (Q \vee P)[F^-] \\
P[F^+] \text{ implies } (P \supset Q)[F^-] & & P[F^-] \text{ implies } (P \supset Q)[F^+] \\
P[F^+] \text{ implies } (Q \supset P)[F^+] & & P[F^-] \text{ implies } (Q \supset P)[F^-]
\end{array}
$$

The main object of the calculi is an *a-sequent*, which is an expression of the form $[\Lambda]\ \Gamma \rightarrow \Pi$, where Λ, Γ, Π are sequences (possibly, empty) of propositions. The propositions from Γ are called *premises*, the propositions from Π are called *goals*, and the propositions from Λ are called *framed propositions*. Here we consider a-sequents containing *one goal* and *framed literals* only.

Below we formulate two a-sequent calculi and give some results on them. Both calculi are based on the following principle: when a goal to be proven "interacts" with some premise in an a-sequent, new goals are generated. In this connection we call these calculi goal-driven. In the first calculus (denoted here by GD-1), the "interacting" premise is removed from the a-sequent, and new premises are generated if needed. In the second calculus (denoted by GD-2), the "interacting" premise is preserved without generating any new premise.

When a proof of an initial a-sequent is searched in GD-1 (GD-2), an *inference tree* is constructed. At the beginning of a search process the tree consists of the initial a-sequent. The subsequent nodes of the inference tree are generated in accordance with the rules of GD-1 (GD-2). Inference trees grow "from top to bottom" in accordance with the order of inference rule applications. An inference tree is considered to be a *proof tree* if every leaf in it is an axiom. An initial a-sequent of a proof tree in GD-1 (GD-2) is called *deducible* in GD-1 (GD-2).

The following a-sequent is an axiom of the calculi GD-1 and GD-2:

$$[\ ] \to \#$$

Note that the main results concern the "reducibility" of both GD-1 and GD-2 to the sequent calculus **G** [6] (restricted to classical propositional logic and denoted here by **Gal**), and vice versa. The soundness and completeness of GD-1 and GD-2 are the consequences of this "reducibility", since the calculus **Gal** is sound and complete.

3 Calculus GD-1

Goal-Splitting Rules (GS):

$$(\to\supset)_1 : \quad \frac{[\Lambda]\ \ \Gamma \to F \supset G}{[\Lambda]\ \ \Gamma, F \to G} \qquad (\to\supset)_2 : \quad \frac{[\Lambda]\ \ \Gamma \to F \supset G}{[\Lambda]\ \ \Gamma, \neg G \to \neg F}$$

$$(\to \vee)_1 : \quad \frac{[\Lambda]\ \ \Gamma \to F \vee G}{[\Lambda]\ \ \Gamma, \neg F \to G} \qquad (\to \vee)_2 : \quad \frac{[\Lambda]\ \ \Gamma \to F \vee G}{[\Lambda]\ \ \Gamma, \neg G \to F}$$

$$(\to \wedge) : \quad \frac{[\Lambda]\ \ \Gamma \to F \wedge G}{[\Lambda]\ \ \Gamma \to F \qquad [\Lambda]\ \ \Gamma \to G} \qquad (\to \neg) : \quad \frac{[\Lambda]\ \ \Gamma \to \neg F}{[\Lambda]\ \ \Gamma \to F^{\neg}}$$

Auxiliary-Goal Rules (AG):

$$(\supset\to)_1 : \quad \frac{[\Lambda]\ \ \Gamma, F[L^-] \supset G, \Delta \to L}{[\Lambda]\ \ \Gamma, (\neg F)[L^+], \Delta \to L \qquad [\Lambda, L^{\neg}]\ \ \Gamma, \Delta \to \neg G}$$

$$(\supset\to)_2 : \quad \frac{[\Lambda]\ \ \Gamma, F \supset G[L^+], \Delta \to L}{[\Lambda]\ \ \Gamma, G[L^+], \Delta \to L \qquad [\Lambda, L^{\neg}]\ \ \Gamma, \Delta \to F}$$

$$(\vee \to)_1 : \quad \frac{[\Lambda]\ \ \Gamma, F[L^+] \vee G, \Delta \to L}{[\Lambda]\ \ \Gamma, F[L^+], \Delta \to L \qquad [\Lambda, L^{\neg}]\ \ \Gamma, \Delta \to \neg G}$$

$$(\vee \to)_2 : \quad \frac{[\Lambda]\ \ \Gamma, F \vee G[L^+], \Delta \to L}{[\Lambda]\ \ \Gamma, G[L^+], \Delta \to L \qquad [\Lambda, L^{\neg}]\ \ \Gamma, \Delta \to \neg F}$$

$$(\wedge \to)_1 : \quad \frac{[\Lambda]\ \ \Gamma, F[L^+] \wedge G, \Delta \to L}{[\Lambda]\ \ \Gamma, F[L^+], G, \Delta \to L}$$

$$(\wedge \to)_2 : \quad \frac{[\Lambda]\ \ \Gamma, F \wedge G[L^+], \Delta \to L}{[\Lambda]\ \ \Gamma, F, G[L^+], \Delta \to L}$$

$$(\neg \to) : \quad \frac{[\Lambda]\ \ \Gamma, \neg(F[L^-]), \Delta \to L}{[\Lambda]\ \ \Gamma, F^{\neg}[L^+], \Delta \to L}$$

Termination-by-Premise Rule (TP):
$$\frac{[\Lambda]\ \ \Gamma, L, \Delta \to L}{[\ \ ] \to \#}$$

Termination-by-Framed-Literal Rule (TF):
$$\frac{[\Lambda_1, L, \Lambda_2]\ \ \Gamma \to L}{[\ \ ] \to \#}$$

Remark. Whenever an AG-rule is applied, the corresponding positive occurrence of a goal literal must be fixed, so that the next AG-rule application is considered to refer to a premise containing the fixed occurrence. This means that any GD-1-inference cannot contain, for example, the "chain":

$$\cfrac{\cfrac{[\Lambda]\ (L \vee M) \wedge F, L \wedge G \to L}{[\Lambda]\ L \vee M, F, L \wedge G \to L}}{[\Lambda]\ L \vee M, F, L, G \to L}$$

because the last AG-rule application is made w.r.t. the formula $L \wedge G$, which does not contain the occurrence of L fixed in the previous AG-rule application (this fixed occurrence of L contains the formula $L \vee M$). The proof of Prop. 2 makes use of this remark.

Proposition 2. *If an a-sequent $[L_1, \ldots, L_k]\ P_1, \ldots, P_n \to G$ is deducible in GD-1, then the sequent $L_1, \ldots, L_k, P_1, \ldots, P_n \to G$ is deducible in **Gal**.*

Proof (scheme). First we prove that all the inference rules of **Gal** are "reversible", i.e. applying any rule to a deducible sequent, we generate deducible sequent(s). Then we show that carrying the negation in some formula of a sequent also does not affect its deducibility in **Gal**. Afterwards, it is easy to prove that every rule of GD-1 corresponds to a certain "superposition" of steps that preserve the deducibility in **Gal**.

Proposition 3. *Let $P_1, \ldots, P_n$ be a consistent set of propositions and G be a proposition. If the sequent $P_1, \ldots, P_n \to G$ is deducible in **Gal**, then the a-sequent*

$$[\ \ ]\ P_1, \ldots, P_n \to G$$

is deducible in GD-1.

Proof (scheme). We prove the theorem by induction in the number of binary connectives in an initial sequence. There are two main cases.

I. The formula G is not a literal. In this case we generate deducible a-sequent(s) after applying an appropriate GS-rule (since the rules of **Gal** can be applied in any order without losing the deducibility).

II. The formula G is a literal. In this case we prove that there always exists some positive occurrence of G in premises of an a-sequent under consideration such that a deducible a-sequent(s) is generated after applying an appropriate AG-rule.

Corollary 2 (soundness and completeness of GD-1). *A proposition F is valid in classical propositional logic if and only if the a-sequent $[\ \ ] \to F$ is deducible in GD-1.*

Proof. This assertion follows from Prop. 1 and 2 and from soundness and completeness of **Gal**.

4 Calculus GD-2

Goal-Splitting Rules (GS):

$(\to\supset)_1: \quad \dfrac{[\Lambda]\ \ \Gamma \to F \supset G}{[\Lambda]\ \ \Gamma \to G}$ $\qquad (\to\supset)_2: \quad \dfrac{[\Lambda]\ \ \Gamma \to F \supset G}{[\Lambda]\ \ \Gamma \to \neg F}$

$(\to \vee)_1: \quad \dfrac{[\Lambda]\ \ \Gamma \to F \vee G}{[\Lambda]\ \ \Gamma \to G}$ $\qquad (\to \vee)_2: \quad \dfrac{[\Lambda]\ \ \Gamma \to F \vee G}{[\Lambda]\ \ \Gamma \to F}$

$(\to \wedge): \quad \dfrac{[\Lambda]\ \ \Gamma \to F \wedge G}{[\Lambda]\ \ \Gamma \to F \qquad [\Lambda]\ \ \Gamma \to G}$ $\qquad (\to \neg): \quad \dfrac{[\Lambda]\ \ \Gamma \to \neg F}{[\Lambda]\ \ \Gamma \to F^{\neg}}$

Auxiliary-Goal Rule (AG): $\quad \dfrac{[\Lambda]\ \ \Gamma, F[L^{+}], \Delta \to L}{[\Lambda, L^{\neg}]\ \ \Gamma, F, \Delta \to \neg F}$

Termination-by-Premise Rule (TP): $\quad \dfrac{[\Lambda]\ \ \Gamma, L, \Delta \to L}{[\ ]\ \to \#}$

Termination-by-Framed-Literal Rule (TF): $\quad \dfrac{[\Lambda_1, L, \Lambda_2]\ \ \Gamma \to L}{[\ ]\ \to \#}$

Remark. Whenever an AG-rule is applied, the corresponding positive occurrence of a goal literal must be fixed, so that the next GS-rule applications must never remove this fixed occurrence from a proof. This means that any GD-2-inference cannot contain, for example, the "chain":

$$\cfrac{\cfrac{[\Lambda]\, L \wedge M \to L}{[\Lambda, \neg L]\, L \wedge M \to \neg L \vee \neg M}}{[\Lambda, \neg L]\, L \wedge M \to \neg M}$$

because the last GS-rule removes $\neg L$ and preserves $\neg M$, though it must preserve $\neg L$. The proof of Prop. 4 makes use of this remark.

Proposition 4. *If an a-sequent $[L_1, \ldots, L_k]\ P_1, \ldots, P_n \to G$ is deducible in GD-2, then the sequent $L_1, \ldots, L_k, P_1, \ldots, P_n \to G$ is deducible in **Gal**.*

Proof (scheme). Using the proof of the Prop.1, we can show that adding instances of the same formula in a sequent does not affect to the deducibility of the sequent in the calculus **Gal**. Then we show that every rule of GD-2 corresponds to a certain "superposition" of steps that preserve the deducibility in **Gal**.

Proposition 5. *Let $P_1, \ldots, P_n$ be a consistent set of propositions and G be a proposition. If the sequent $P_1, \ldots, P_n \to G$ is deducible in **Gal** then the a-sequent*

$$[\ ]\ P_1, \ldots, P_n, \neg G \to G$$

is deducible in GD-2.

Proof (scheme). We show that every inference in GD-1 can be transformed to an inference in GD-2 by the following way. Whenever we apply an AG-rule of GD-1 to a premise P, which is not an initial premise (i.e. it is derived), we choose one of the initial premises and split it up to P in GD-2. Since **Gal** is reducible to GD-1 (by Prop. 2), it is reducible to GD-2.

Corollary 3 (soundness and completeness of GD-2). *A proposition F is valid in classical propositional logic if and only if the a-sequent $[\ ]\ \neg F \to F$ is deducible in GD-2.*

Proof. This assertion follows from Prop. 3 and 4 and from the soundness and completeness of **Gal**.

Remark. Note that some GS-rule applications in GD-2 remove subformulas from "parental" a-sequents. Therefore, we must insert $\neg F$ as the premise of the initial a-sequent in order to preserve the completeness (cp. Exam. 2).

5 Examples of Inferences

Here we give the examples of the proof trees $\mathbf{T}$ and $\mathbf{T}'$ in both calculi. We write an applied inference rule to the right of each step.

Example 3. Let us prove the a-sequent $[\ ]\ A \vee B, A \supset B \to (B \supset C) \supset C$ in GD-1:

$\mathbf{T}$ is

$$\dfrac{\dfrac{[\ ]\ A\vee B, A\supset B \to (B\supset C)\supset C}{[\ ]\ A\vee B, A\supset B, B\supset C \to C}\,(\to\supset)_1}{\dfrac{[\ ]\ A\vee B, A\supset B, C\to C}{[\ ]\to \#}\,(\mathrm{TP}) \qquad \dfrac{[\neg C]\ A\vee B, A\supset B\to B}{\dfrac{[\neg C]\ A\vee B, B\to B}{[\ ]\to\#}\,(\mathrm{TP}) \quad \mathbf{T}_1}\,(\supset\to)_2}\,(\supset\to)_2$$

$\mathbf{T}_1$ is

$$\dfrac{[\neg C, \neg B]\ A\vee B\to A}{\dfrac{[\neg C,\neg B]\ A\to A}{[\ ]\to\#}\,(\mathrm{TF}) \qquad \dfrac{[\neg C,\neg B,\neg A]\ \to\neg B}{[\ ]\to\#}\,(\mathrm{TF})}\,(\vee\to)_1$$

Example 4. Let us prove the a-sequent $[\ ]\ \Gamma \to A \vee D$ in GD-2, where the premises of Γ are $A \vee C, C \supset D, \neg(A \vee D)$:

$\mathbf{T}'$ is

$$\dfrac{\dfrac{[\ ]\ \Gamma\to A\vee D}{\dfrac{[\ ]\ \Gamma\to D}{\dfrac{[\neg D]\ \Gamma\to\neg(C\supset D)}{[\neg D]\ \Gamma\to C\wedge\neg D}\,(\to\neg)}\,(\mathrm{AG})}\,(\to\vee)_2}{\dfrac{\dfrac{[\neg D]\ \Gamma\to C}{[\neg D,\neg C]\ \Gamma\to\neg(A\vee C)}\,(\mathrm{AG})}{\mathbf{T}'_1}\,(\to\neg) \qquad \dfrac{[\neg D]\ \Gamma\to\neg D}{[\ ]\to\#}\,(\mathrm{TF})}\,(\to\wedge)$$

$$\mathbf{T'}_1 \text{ is } \quad \dfrac{[\neg D, \neg C]\ \Gamma \to \neg A \wedge \neg C}{\dfrac{[\neg D, \neg C]\ \Gamma \to \neg A}{\dfrac{[\neg D, \neg C, A]\ \Gamma \to \neg\neg(A \vee D)}{\dfrac{[\neg D, \neg C, A]\ \Gamma \to A \vee D}{\dfrac{[\neg D, \neg C, A]\ \Gamma \to A}{[\ ]\ \to \#}\ (\mathrm{TF})}\ (\to \vee)_1}\ (\to \neg)}\ (\mathrm{AG}) \qquad \dfrac{[\neg D, \neg C]\ \Gamma \to \neg C}{[\ ]\ \to \#}\ (\mathrm{TF})}\ (\to \wedge)$$

Note that we use both subformulas of the initial goal $A \vee D$. Thus, it was necessary to have its negation in Γ.

6 Comparison of the Calculi

When an inference search is made in GD-1 or GD-2, we apply subsequently the GS-rules until the goal of an a-sequent becomes a literal proposition.

When the goal becomes a literal (let us denote it by L), first of all we try to find it among the framed literals of the a-sequent in order to apply the termination rule (TF). If the TF-rule cannot be applied, we choose some premise P containing a positive occurrence of L. If P is L, we apply the termination rule (TP) and derive the axiom. In the opposite case we act in accordance with the calculus under consideration (GD-1 or GD-2).

In the case of GD-1 we subsequently apply the AG-rules until P is split up to L. When we do this, new branches can appear in an inference tree (with a new proposition in the goal).

In the case of GD-2 we apply the AG-rule, which does not split premise P, but writes its negation as a goal (preserving P as a premise). The complementary of L becomes a framed literal.

The main difference between the calculi is that GD-2 allows us to use initial premises many times in the same branch of an inference tree, while GD-1 keeps subformulas, which were not used in the deduction and can be used later.

The following examples demonstrate this difference. We define the *weight of a proof tree* as the number of a-sequents in it. A proof tree of some a-sequent is considered to be *minimal* if and only if it has the minimal weight among other proof trees for this a-sequent.

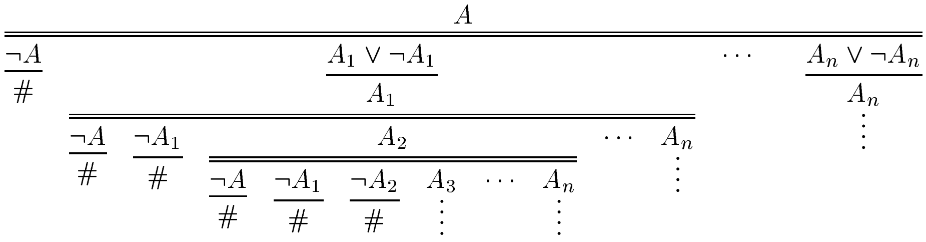
Example 5. Let us consider a-sequents of the form:

$$[\ ]\ (A_1 \wedge \neg A_1) \vee \cdots \vee (A_n \wedge \neg A_n) \vee A \to A$$

Below we give the scheme of their inference in GD-1 (some steps are omitted).

$$\begin{array}{c} [\ ]\ (A_1 \wedge \neg A_1) \vee \cdots \vee (A_n \wedge \neg A_n) \vee A \to A \\ \hline\hline \dfrac{[\ ]\ A \to A}{[\ ]\ \to \#} \qquad \dfrac{[\neg A]\ \to A_1 \vee \neg A_1}{\dfrac{[\neg A]\ A_1 \to A_1}{[\ ]\ \to \#}} \qquad \cdots \qquad \dfrac{[\neg A]\ \to A_n \vee \neg A_n}{\dfrac{[\neg A]\ A_n \to A_n}{[\ ]\ \to \#}} \end{array}$$

Let us construct the proof tree scheme for these a-sequents in GD-2. Since we do not change the set of premises during inference search, and since framed literals of any a-sequent in an inference tree are just complementaries of literal-goals of parent a-sequents, it will be sufficient to give the goals of a-sequents only:

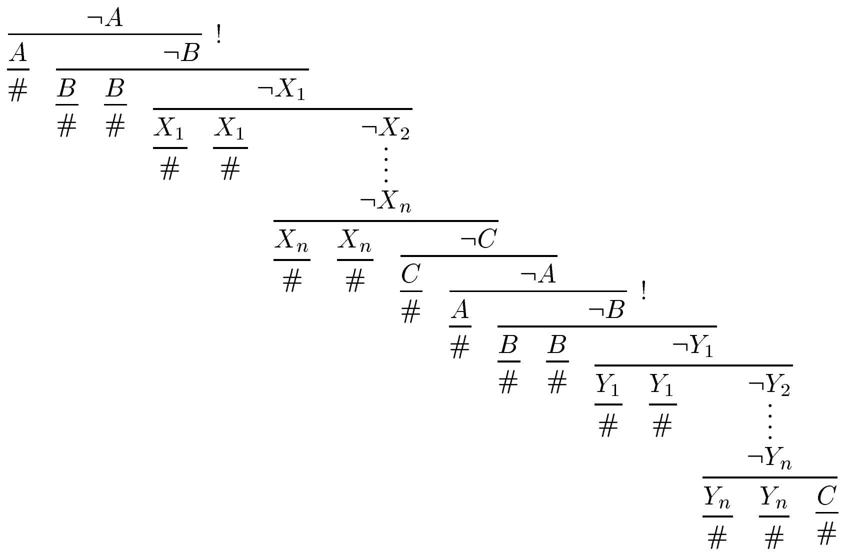


We see that weights of the corresponding minimal proof trees are linear by n in GD-1 and are exponential by n in GD-2 for the given class of a-sequents.

Example 6. Let us consider a-sequents of the form:

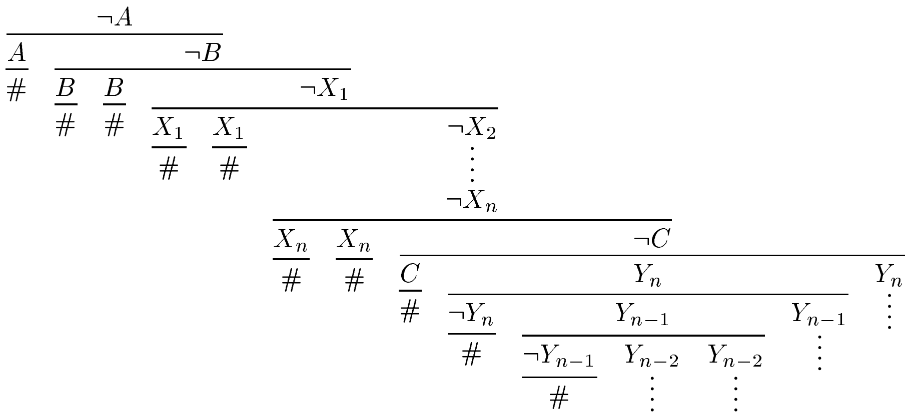
$$[\ ]\ B \supset (B \supset X_1),\ X_1 \supset (X_1 \supset X_2),\ \ldots,\ X_n \supset (X_n \supset C),\ C \supset A,$$
$$B \supset (B \supset Y_1),\ Y_1 \supset (Y_1 \supset Y_2),\ \ldots,\ Y_n \supset (Y_n \supset \neg C),\ A \supset B\ \rightarrow\ \neg A$$

Below we give the scheme of their inference in GD-2.



Note that the premise $A \supset B$ is used twice in the branch (the corresponding steps are marked by '!').

When deducing this a-sequent in GD-1, we pass the first half of the proof in the same manner. However, we cannot use again the premise $A \supset B$. Therefore, we must "inverse" the refutation of the literal C:



This a-sequent can be deduced in GD-1 in several ways, but no one of them gives a smaller inference. Thus, corresponding minimal weights are linear by n in GD-2 and are exponential by n in GD-1 for the given class of a-sequents.

7 Conclusion

The end of the 1950s and the beginning of the 1960s can be characterized as the period of coming into existence computers with a sufficiently high performance rate and information capacity. In response to this computing progress, a series of papers appeared, which contained the issues of implementation for Gentzen [5] calculi and for inference search methods relying on results of Skolem [13] and Herbrand [14]. For more detail, see, for example, [15].

Those first papers gave an answer to the principal question about a possibility to use computers for mathematical reasoning, but they did not contain special theorem-proving methods for constructing proofs of a rather high efficiency.

Investigations on improving the efficiency of theorem proving technique resulted in coming into existence Kanger's calculus [3] (Gentzen approach) and Robinson's resolution method [7] (Skolem-Herbrand approach). For those days, the latter approach yielded the most efficient machine-oriented inference search technique for 1st-order classical logic. As a result, studies on automated theorem proving were later concentrated upon improving the potentialities of the Skolem-Herbrand approach mainly. Thus, subsequent attempts for the advancement of the Skolem-Herbrand approach resulted in arising such methods as tableaux methods, the connection graph method, goal-oriented search methods, etc.

The lower efficiency of the Gentzen approach (as compared with the Skolem-Herbrand approach) can be explained by the fact that even Kanger-type calculi result in arising the superfluous enumeration caused by unordered logical rule applications and by the absence of a preliminary skolemization. At the same time, machine-oriented Kanger-type methods reflect proof technique that is more "natural" for a human. Also, it enables to control automated reasoning strategies and to construct rather flexible interactive search modes. Exactly these reasons caused the appearance of different modifications of AS.

All the modifications of AS use goal-driven approach, but only this paper contains some analysis how the choice of propositional rules has an influence on the efficiency of an inference search in sequent calculi. The above comparison of the calculi GD-1 and GD-2 shows that no one of them can be preferred to another, though GD-1 could seem to give always smaller minimal inferences than GD-2. This witnesses to the fact that even syntax of the problem under consideration influences essentially the choice of an appropriate strategy.

The authors thank referees for useful advice.

References

1. Glushkov,V.M.: Some problems of automata theory and artificial intelligence (in Russian). Kibernetika **2** (1970) 3–13
2. Anufriyev, F.V., Fediurko, V.V., Letichevski, A.A., Asel'derov, Z.M, Didukh, I.I.: On one algorithm of theorem proof search in Group Theory (in Russian). Kibernetika **1** (1966) 23–29
3. Kanger, S.: Simplified proof method for elementary logic. In: Comp. Program. and Form. Sys.: Studies in Logic, North-Holl. Publ. Co. (1963)
4. Anufriyev, F.V.: An algorithm of theorem proof search in logical calculi (in Russian). In: Teoriya avtomatov **5**, Institute of Cybernetics, Kiev (1969) 3–26
5. Gentzen, G.: Untersuchungen uber das Logische Schliessen. Math. Zeit. **39** (1934) 176–210
6. Gallier J.: Logic for Computer Science: Foundations of Automatic Theorem Proving. Harper and Row, Inc., New York (1986) 513 p.
7. Robinson, J.: A machine-oriented logic based on resolution principle. J. of the ACM (1965) 23–41
8. Lyaletski, A.V.: Gentzen calculi and admissible substitutions. In: Actes prelim. du Symp. Franco-Sovetigue "Informatika-91", Grenoble, France (1991)
9. Degtyarev, A.I., Lyaletski, A.V.: Logical inference in SAD (in Russian). In: Matematicheskiye osnovy sistem iskusstvennogo intellekta, Institute of Cybernetics, Kiev (1981)
10. Glushkov, V.M.: The System for Proving Automation (in Russian). In: Avtomatizatsiya obrabotki matematicheskikh tekstov, Institute of Cybernetics, Kiev (1980)
11. Degtyarev, A.I., Lyaletski, A.V., Morokhovets, M.K.: Evidence Algorithm and Sequent Logical Inference Search, In: Lect. Notes in Art. Intell. **1705** (1999) 44–61
12. Degtyarev, A.I., Lyaletski, A.V., Morokhovets, M.K.: On the EA-style integrated processing of self-contained mathematical texts. In: Proc. of the Intern. Workshop CALCULEMUS'2000, Great Britain (2000)
13. Skolem, T.: Logisch-kombinatorische Untersuchungen uber die Erfullbarkeit oder Beweisbarkeit mathematischer Satze. Skriftner utgit ar Videnskapsselskaper i Kristiania, **4** (1920) 4–36.
14. Herbrand, J.: Recherches sur la Theorie de la Demonstration. Travaux de la Societe des Sciences et de Lettres de Varsovie, Class III, Sciences Mathematiques et Physiques **33** (1930).
15. Chang, C., Lee, R.: Symbolic logic and mechanical theorem proving. Academic Press, New York (1973).

A Pragmatic Approach to Reuse in Tactical Theorem Proving

Axel Schairer[1], Serge Autexier[2], and Dieter Hutter[1]

[1] German Research Center for Artificial Intelligence (DFKI GmbH), Stuhlsatzenhausweg 3, 66111 Saarbrücken, Germany
schairer@dfki.de, hutter@dfki.de

[2] FR 6.2 Informatik, Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany,
autexier@ags.uni-sb.de

Abstract. In interactive theorem proving, tactics and tacticals have been introduced to automate proof search. In this scenario, user interaction traditionally is restricted to the mode in which the user decides which tactic to apply on the top-level, without being able to interact with the tactic once it has begun running. We propose a technique to allow the implementation of derivational analogy in tactical theorem proving. Instead of replaying tactics including backtracked dead ends our framework makes choice points in tactics explicit and thus avoids dead ends when reusing tactics. Additionally users can override choices a tactic has made or add additional steps to a derivation without terminating the tactic. The technique depends on an efficient replay of tactic executions without repeating search that the original computation may have involved.

1 Introduction

In interactive theorem proving, tactics and tacticals have been introduced to automate proof search. In its simplest case, a tactic is a sequence of rules to be applied in order. Virtually all tactic languages provide additional control structures like iteration, conditional branching or indeterministic choice over tactics. These control structures are called tacticals. Tacticals provide a means to write more powerful tactics. Invoking a tactic on a goal produces a representation of the (partial) proof that the tactic constructs while simplifying the original goal.

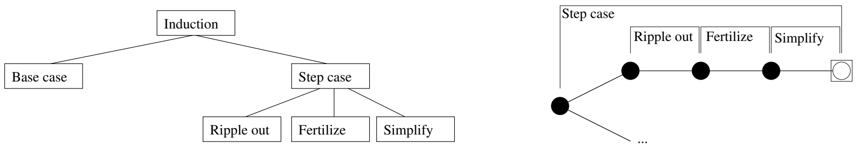
Tactics decompose a goal into subgoals and call other tactics or themselves recursively to solve these subgoals. In this sense tactics are a way to structure strategic knowledge about proof search. This describes the static call graph of a set of related tactics. E.g. an induction tactic sets up the induction, applies a difference reduction tactic to the induction conclusion and hypothesis, applies the induction hypothesis, and finally simplifies the result.

Executing such a tactic gives rise to another dynamic structure, namely the tree of tactic calls that were actually carried out while solving a problem. This structure is closely related to the call graph but depends on the goal the tactic is run on. It is a summary of the call stack over the time of the tactic execution and describes the computation that was carried out to apply the tactic to a concrete goal. Because tactics involve search (usually implemented by indeterministic choice and explicit failure with backtracking) whole subtrees of this structure may correspond to branches of the search space that turned out to be dead ends. I.e. they describe computations that do not manifest themselves in the final proof representation. As an example, depending on the goal, the difference reduction tactic might call itself twice on subproblems and a simplification tactic inside one of the recursive calls. Also it might have to undo the second call to itself because the call did not produce the desired result and try again in a different way after some additional rewriting.

If we have a closer look at this structure we see two ways in which a user can interact with a tactical theorem prover. First, the user can decide which tactics to apply at the top-level and let the tactics fill in the details. I.e. the user decides on the overall structure of the proof while the machine decides on the details. This techique is applied in many theorem provers like Isabelle, PVS, Coq, KIV, Inka.

Another way the user should be able to interact with the theorem prover is to let a tactic decide on the overall proof. If the prover gets stuck the user decides on the details by overriding decisions of the tactic, inserting intermediate steps or suggesting a different way to solve a particular subproblem without otherwise terminating the tactic. I.e. after the intervention by the user the tactic carries on with what was left for it to do when the user intervened. In current theorem provers this way of interaction is not possible without changing the code of tactics and rerunning them. This has two severe disadvantages: the code of the tactics needs to be changed by the user and the tactics have to be rerun, repeating all the search that the tactic carried out before the intervention. In current theorem provers, when a tactic gives up the prover either does backtracking and leaves the user with the original proof state the tactic started out with, or it leaves a proof state corresponding to the partial success the tactic had. In any case the tactic call stack is gone and the user is back at the top-level. There is no way to intervene and then resume the original tactic at the place it gave up. That is, once a tactic has started it either produces a result fully automatically, or it does not produce a result at all. An integration of automation and interaction is not possible once a tactic has been started.

In this paper we describe a technique that enables the second mode of interaction. The basic idea is the following: while executing a tactic on a goal the prover collects a trace of the computation. Each element of this trace corresponds to a point of time in the execution of the tactic. If the user wants to suggest an intermediate step to the tactic, the prover can reexecute the initial part of the computation along the trace that it had collected earlier. When it reaches the point where the user wants to intervene it can execute a tactic that was given by the user. We call such a tactic *callback*. Since the prover has reexecuted the initial part of the computation, the callback can be executed in exactly the same situation in which the theorem prover was when the original, supposedly unsuccessful subtactic was executed. In particular all information that was collected by the execution of the tactic up to this point in time in the original execution is again present (including information entered by the user in the original execution). Our technique takes care that the reexecution of the trace is carried out in synchronization with the tactic that generated the trace in the first place. This means that not only can the callback be executed in the situation in which the original tactic would have



(a) Structure of example induction tactic (b) Proof representation for failed proof

Fig. 1.

been carried on, but also can the original tactic be resumed at exactly the same place in its execution in which it was intercepted.

The user can also override a decision by the tactic. Assume that the tactic has decided to apply a particular lemma at some point in the proof search, but the user has decided that it would be better to use another lemma. This is achieved by reexecuting the initial part of the computation up to the point at which the wrong lemma was chosen by the tactic. This decision is overriden by the lemma determined by the user and the tactic resumes with the new lemma instead of the old one.

In order to allow this mode of interaction, we assume a standard tactic language that offers facilities most tactic languages in use provide. In particular the language supports explicit choice from a list of alternatives and explicit failure to initiate backtracking. The language also supports calls to functions defined in the underlying programming language which does not support the kind of backtracking needed for implementing choice and failure. These operations interface the tactics to the proof representation and objects of the logic, and they also serve to use the full power of an efficient programming language to compute heuristic information.

Since the technique depends on the reexecution of tactics involving search it is crucial that dead branches of the search space can be cut off when repeating a computation.

The rest of this paper is organised as follows. In Sect. 2 we give a motivating example of a situation in which we can use our technique. Sect. 3 informally describes the technique. Sect. 4 revisits the earlier example and shows how the technique works in this case. Sect. 5 discusses how the technique can be used to reuse proofs analogically. Finally, we describe related work in Sect. 6 and conclude.

2 An Example

Suppose we want to prove the theorem $\forall x.\ \exists y.\ \text{half}(y) = x$ by induction on x, given natural numbers generated by 0 and $s(\_)$ and the rewrite rules $\text{half}(0) = 0$ and $\text{half}(s(s(X))) = s(\text{half}(X))$. We will use an induction strategy which is implemented as a tactic with the structure given in Fig. 1.(a). The figure shows the call graph of the tactic: 'step case', e.g., calls 'ripple out', 'fertilize', and 'simplify' in this order.

'Induction' sets up the base and step case. For brevity, we will ignore the base case in the rest of the example. The formula to prove in the step case is

$$\forall x.\ [\exists y.\ \text{half}(y) = x] \Longrightarrow [\exists y.\ \text{half}(y) = s(x)]$$

from which the quantifiers are removed by skolemization. ‘Ripple out’ uses rippling to make the induction hypothesis applicable to the induction conclusion. Rippling [3, 9] is a reasoning technique exploiting the semantic knowledge that the induction hypothesis must be applied in the step case, and additionally assumes that the induction conclusion contains syntactically the induction hypothesis. The differences between induction conclusion and induction hypothesis are called *wave-fronts* and are denoted by boxes, while the common parts compose the *skeleton* and are denoted by underlining them. E.g., in our example the differences between induction conclusion and induction hypothesis are represented by

$$\text{half}(y) = x \Longrightarrow \text{half}(Y) = \boxed{s(\underline{x})}$$

The Rippling technique tries to apply so-called wave rules, like the definition of half

$$\text{half}(\boxed{s(s(\underline{U}))}) = \boxed{s(\underline{\text{half}(U)})},$$

that preserve the skeleton while moving the differences to the outside, in order to eventually end up in a situation where the induction hypothesis is a proper subterm of the induction conclusion. In the running example there is no wave rule to move the only wave front occurring in

$$\text{half}(y) = x \Longrightarrow \text{half}(Y) = \boxed{s(\underline{x})}$$

and the induction hypothesis is directly applicable anyway for $Y \leftarrow y$. So ‘ripple out’ returns and ‘fertilize’ applies the induction hypothesis:

$$\text{half}(y) = x \Longrightarrow x = \boxed{s(\underline{x})}$$

The subsequent simplification then fails because s is a free constructor. So, the system comes back to the user saying that there is an open goal it could not solve. The resulting proof state (proof representation) that has been built and which is displayed to the user (cf. Fig. 1.(b)) includes the open goal, and links – representing reasoning steps – which are annotated by the part of the tactic that has produced them. Browsing the sequence of reasoning steps that lead to the open goal, the user will notice Y would have to be instantiated to a term different from y. And indeed, we can repair the proof by introducing an additional wave front by instantiating Y with $s(s(\_))$ before we ripple out (we will come back to this example in Sect. 4).

In current systems, what we have to do in this case is either to change the tactic code and try again from scratch, or to remove the parts of the proof representation that was built after ‘normalize’ and carry out the introduction of wave rules and the rest of the induction tactic manually. Since our tactics are procedural (rather than declarative), we cannot use the induction tactic in the latter case because we would have to restart it right in the middle. In Sect. 3 we describe a technique that enables us to interact with the theorem prover in such a way that we can intercept the induction tactic before ‘ripple out’ is called, introduce the wave front manually, and then resume the tactic as if the wave front had been introduced by the tactic itself.

3 Tactic Language

In this section we explain our technique in an informal way. We describe the semantics of tactics in continuation passing style which is a standard technique (cf. [1, 8, 17, 15, 6, 5]). The idea is to describe the evaluation of an expression e with respect to (wrt.) a continuation S which represents the future of the computation after e has been evaluated. A continuation S is a function of one argument. In order to evaluate e wrt. S, the continuation S is applied to the value that e evaluates to. Evaluation of a literal l wrt. S is, therefore, described by $S(l)$. Evaluation of complex expressions is decomposed in such a way that a part of the expression is evaluated wrt. a new continuation which combines the rest of the evaluation of the complex expression and also considers the original continuation. As an example, if e is a list of expressions $e_1, e_2, \ldots, e_n$, its evaluation relative to S can be described by saying that e_1 should be evaluated (assume the result is h) wrt. the continuation that evaluates $e_2, \ldots, e_n$ (assume the result is t) and then applies the original continuation S to the list composed of the head h and the tail t. Evaluation of complex expressions can be decomposed until the expression to evaluate is a literal or variable and the rest of the evaluation is encoded in the success continuation. This defines a non-standard interpreter for the tactic programming language in continuation passing style. So we can also view the definition operationally as the way in which a tactic expression is executed. The general idea is that evaluation is flattened (or sequentialised) into a sequence of basic evalutation steps. This is in contrast to the standard evaluation strategy, which evaluates an expression by evaluating its subexpressions along its syntactic structure, its abstract syntax tree, and does not explicitly sequentialise the evaluation.

Tactics can be defined and named, e.g. as in “tactic $f(x) = e$”, where the formal parameter x of the named tactic f may be free in the body e. Tactics may call other named tactics to solve subgoals of the current goal. Depending on the proof representation this necessitates setting up the proof state to focus on the subproblem before evaluating the body of the called tactic. It may also necessitate cleaning up after evaluating the body. This is dependent on the concrete proof representation, and the semantics has to take care that setting and cleaning up are done appropriately, i.e. before and after a tactic is called. Note that the interpreter knows which named tactic is being called and the actual values it is called with when it sets up the proof state. So it is possible to make the proof state dependent on the tactic that is called.

3.1 Backtracking

We assume the tactic language includes a special operator `choose` which implements search. It evaluates its argument, which should be a list of alternatives, and then chooses the first alternative for which the remaining part of the evaluation is successful, i.e. does not fail. If there is no such alternative, `choose` fails itself. As usual this is realised by backtracking and can be done straightforwardly in the setting of continuations by using a second continuation F that encodes the future of the evaluation if the evaluation of an expression fails. Again this is a standard technique covered in the literature cf. [1]. Tactics can explicitly call `fail` which is an abbreviation for `choose([])` and fails straightaway. In order for this to work properly, side effects of tactics need to be restricted such that they can be backtracked over.

In particular, because there is an interaction between the evaluation of tactic expressions and the proof representation, the semantics has to ensure that the proof representation is adjusted when backtracking occurs. This is important in particular when updating the proof representation is done by destructively updating proof data structures. In this case, since evaluation of a tactic expression can modify the proof state between the time a choice point is set up and the time the subsequent computation fails, the proof state has to be remembered before a choice is tried and has to be rewound to the remembered state before a new alternative is tried. This has been built into the semantics of `choose` in such a way that only one computation for each choice point, corresponding to trying the first successful choice from the list of alternatives, has left modifications in the proof tree.

The net effect is that the proof representation is always well-formed and only contains parts that correspond to successful evaluation paths. This is, of course, relative to the concrete definition of the proof representation and its interface to the tactic evaluation. The technical details are out of the scope of this paper.

3.2 Replay

Remember that our aim is to be able to intervene a tactic in the middle of its execution and later resume it in the same context. The way we realise this is by reexecuting the tactic up to the point in its execution at which we want to intervene. When we reexecute a tactic, we do not want to repeat the search that was caried out when the tactic was originally executed. The idea is to remember the successful alternatives when executing a tactic and use them as an oracle for choosing the successful alternative without search on reexecution.

Therefore, we defined evaluation of a tactic expression such that it collects a trace of the sequentialised computation. A trace is a sequence the elements of which are either tactic call markers, written $\texttt{call\_to}(f)$, or choices, written $\texttt{choice}(v)$. When we start evaluating an expression e from the top-level we start with an empty trace $\langle\rangle$. We pass around the trace which we have collected so far from one step of the evaluation to the next. Whenever we reach a choice point and choose a value from the list of alternatives, say h, we add the choice we have made to the trace T we have collected so far, written $\langle T.\texttt{choice}(h)\rangle$, and then carry on collecting the trace of the rest of the computation starting with $\langle T.\texttt{choice}(h)\rangle$. Whenever we have to backtrack to this choicepoint later and choose another alternative we start with T again and add the new choice and the trace of the remaining computation to the end of T. In a similar manner we collect function call markers. Whenever we have evaluated the arguments of a named tactic f and make the next step, i.e. we evaluate the body, we extend the current trace by the tactic call marker $\texttt{call\_to}(f)$. When we backtrack over a call to a named tactic, the tactic call marker is removed from the collected trace as explained for choicepoints above. We define the trace for a tactic expression that succeeds, written *trace*(e), to be the trace collected for the evaluation of e.

There is also a possibility to extract the trace of a computation that is not successful: we have defined an operation `abort` that tactics can call to give up completely, i.e. abort to the top-level. However, the trace collected so far is passed back. Tactics will call `abort` on timeout or on manual interruption. We will ignore this detail for the rest of the paper, however.

The computation trace that we have collected can be used to replay a computation without repeating the search that is involved in finding the appropriate alternative in each choice point. We use the successful choices represented in the computation trace as an oracle to choose the successful alternative without search on reexecution.

Simple Replay. Evaluation of a tactic expression is now defined wrt. continuations and a computation trace t that we have collected as described above. We simply pass through t except for evaluation of `choose` and calls to named tactics. For `choose`, instead of setting up a choicepoint and searching for a successful alternative, we simply take the first event in the trace t, say $\texttt{choice}(v)$, and use v as the value of the `choose` form without setting up a choicepoint. For calls to a named tactic f we simply ignore the first event in the computation trace, which is $\texttt{call\_to}(f)$.

This has the consequences that if $\mathit{trace}(e) = t$ then replaying the trace, i.e. evaluating e wrt. t, yields the same return value and has the same effect as the original evaluation, except that the former is more efficient: it does not carry out the search to determine successful choices because it can use the trace as an oracle to determine the correct guess.

Note that there are several possibilities to guess the successful choice. We can either, as was described above, remember and reuse the value that was chosen in the computation. Or we can remember the position of the successful choice in the list of alternatives and reuse the element at that position. Since for this paper we assume that nothing has changed with the axiomatisation or the theorem these two possibilities are equivalent. If we allowed changes to the theorem or the axiomatisation we have to transfer the decisions made for each individual choicepoint to its analogous counterpart. We will discuss this point in more detail in Sect. 5.

would have to use an entirely different notion of what "reusing the old choice" means. This is not in the scope of this paper, however.

As described in Sect. 2 the main point of the replay in our scenario is to be able to replay an initial part of a computation to recover the context for the rest of the computation. Obviously, the initial part of a computation is associated with a prefix of a computation trace. Therefore, we define evaluation of a tactic expression also wrt. a trace t which is a prefix of the trace s collected when the expression was evaluated earlier. Evaluation is now defined as for simple replay if t is non-empty, and is defined as for evaluation from scratch when t is empty. In other words, we replay the computation corresponding to the prefix t and switch back to evaluation from scratch when we run out of events in the trace. Again, if $\mathit{trace}(e) = s$ and t is a prefix of s, then replaying the trace t yields the same result as the original computation. The replay is more efficient, however, because it does not repeat the search for the initial part of the computation described by the trace t.

Replay with Intervention. With the evaluation strategy given so far we can replay an intial part of the computation corresponding to the evaluation of a tactic expression, and then revert to evalution from scratch in the proper context. In order to be able to intervene the tactic at this point in the evaluation instead of continuing, we allow the last event of a trace prefix to be changed before the prefix is replayed. If the last event is $\texttt{choice}(v)$, we allow this event to be replaced by another event $\texttt{choice}(v')$. Instead of choosing the alternative v as in the original computation, on reexecution v' is chosen and the tactic is resumed with v' instead of v. This allows overriding choices that the tactic has made without interrupting the tactic. As an example, a lemma that a tactic may have chosen to apply at at specific point may simply be the wrong one. By changing the event the tactic can be forced to use another lemma and carry on as before.

Another way to interact with tactics is to replace a call to a named tactic by the evaluation of another tactic expression, called a callback. Consider the example in Sect. 2 where the step case tactic applied ripple out without introducing the additional wave front. In this case, instead of calling the tactic ripple out, step case should let the user carry out a reasoning step interactively and then call ripple out and carry on with the tactic. To achieve this, if the last event of an initial trace is $\texttt{call\_to}(f)$, we allow this event to be replaced by $\texttt{callback}(e)$. On reexecution, now instead of calling the tactic f we evaluate the expression e in the context in which f was called when t was collected. Since the context in which e is evaluated includes the lexical bindings of program variables, e may be an expression in the programming language that includes free occurrences of the program variables that were visible at the place in the original tactic definition at which f was called.

In any case, the replay collects a trace that includes the choice made to the trace before reexecution was started, and the trace for the subsequent computation. This means that the resulting trace can be replayed, in effect replaying the computation with the overridden decision without any further user intervention in the replay. This is an important property of our technique. The trace of any computation can be replayed, yielding the same collected trace. I.e. if $s = \mathit{trace}(e)$ when replaying t, then $\mathit{trace}(e) = s$ when replaying s. Also, for such an s, replaying s is equivalent to the original computation that produced s, i.e. it builds up the same proof representation.

4 The Example Revisited

We now go back to the example of Sect. 2 and show how the technique described in the previous section is applied to repair the failed proof attempt. We noted in passing in Sect. 2, that the user spotted a possible problem with the instantiation of Y to y in

$$\text{half}(y) = x \Longrightarrow \text{half}(Y) = s(x) \tag{1}$$

where Y should probably have the form $s(s(\_))$. The user will, therefore, want to retry the proof attempt with Y instantiated with $s(s(Y'))$, where Y' is a new metavariable. So, before ‘ripple out’ is called, the formula is changed manually to

$$\text{half}(y) = x \Longrightarrow \text{half}(s(s(Y'))) = s(x)$$

and then the induction tactic is resumed. ‘Ripple out’ marks the additional wave front,

$$\text{half}(y) = x \Longrightarrow \text{half}(\boxed{s(s(\underline{Y'}))}) = \boxed{s(\underline{x})}$$

applies the definition (2) of 'half' to push the left wave front outside the 'half', removes the s on both sides of the equation to make the induction hypothesis applicable.

$$\text{half}(y) = x \Longrightarrow \boxed{s(\underline{\text{half}(Y')})} = \boxed{s(\underline{x})}$$
$$\text{half}(y) = x \Longrightarrow \text{half}(Y') = x$$

At this point, 'fertilize' succeeds and without further simplification the proof is complete.

In the process of repairing this proof the user is never aware of the computation traces and their replay. Pointing to the node in the proof representation where the manual intervention should take place presents the user with a choice of possible actions, one of them being "Intervene and retry". In the example, the relevant node is the one which contains the formula (1) on which the 'ripple out' has been applied.

Internally, each node of the proof representation is annotated with the computation trace that was collected when the node was created in the failed proof attempt and the tactic that was called on the relevant proof node. Therefore, the system can execute the rippling tactic from scratch by replaying the initial trace t up to (but excluding) the call to 'ripple out' (the computation trace stored with the node). Before replaying the tactic, the computation trace t that is to be replayed is extended by the callback event which calls a tactic implementing user interaction and, when the interaction tactic returns, calls 'ripple out'. The evaluation rules of the tactic language ensure that after 'ripple out' returns, the original tactic is resumed as if the user interaction had not happened and the wave front had been inserted by the tactic itself. This only necessitates the user to make one interaction instead of simulating the rest of the induction tactic manually.

The callback and the user interaction are both stored in the trace that is collected when the successful proof is constructed. This means that when the computation needs to be replayed again later the user interaction is replayed by the system as well and no user interaction is necessary.

5 From Replay to Reuse by Analogy

As tactics decompose a goal into subgoals by calling other tactics, an implicit proof plan is encoded into the call graph of the used tactics. In our running example we decided to prove the theorem by induction. In more detail we tackled the step case by rippling out, fertilization and simplification tactics. In even more detail, we had to instantiate Y to enable the use of the definition of half inside the rippling tactic. Thus, replaying the same tactics on a different problem corresponds to reuse by analogy. In contrast to a simple replay, we have to translate the choicepoints of the orginal problem to corresponding choicepoints of the target problem and use the choice that is analogous to the choice made in the source proof. In [10] we illustrated how annotations can be used to maintain the proof history of individual symbol occurrences during the proof. Relating symbol occurrences of source and target proof if they share the same annotations (i.e. proof history), we can implement a sophisticated approach for transformational analogy. Given this mapping of symbol occurrences of source and target problem (which is automatically extended to derived formulas with the help of the annotated calculus

[10]), we are able to map the choices made inside the source problem to corresponding choices to be made inside the target problem.

6 Related Work

Theorem proving by analogy has been investigated since the early days of automated theorem proving. For a long time, computational accounts of this analogy have been dominated by the idea of mapping symbols and single proof steps of a source proof to a target theorem (see e.g. [16]). This turned out to be rather insufficient and is far to limited to capture the intuitive notion of a proof by analogy. Inspired by explanation based learning techniques [14], Kolbe and Walther [12, 11] developed a method for reusing proofs which is based on generalization of proofs and conjectures by replacing occurrences of function symbols by second order functions. Felty's and Howe's [7] approach on proof reuse is also based on proof generalization introducing meta-variables in proofs to find a minimal proof using given rule schemata.

Melis [13] applied the paradigm of derivational analogy to theorem proving at the more abstract planning level. The notion of derivational analogy was invented by Carbonell [4, 19] for planning and general problem solving where the "lines of reasoning"(i.e. the sequence of decisions and their justifications) are replayed to solve a target problem. Melis' approach is based on the paradigm of proof planning and thus on an explicit representation of control knowledge in terms of methods and proof plans. Our approach can be easily integrated into standard tactic languages thus allowing to implement derivational analogy in a procedural (instead of declarative) control language.

Richardson and Smaill [18] use a technique similar to ours to be able to intervene with and then resume declarative methods. They store an explicit form of the continuation with the proof representation and do not collect or replay computation traces.

7 Conclusion

In this paper we have described a technique that allows user interaction in tactics without terminating the tactics. Users can override choices of the tactic and insert other proof steps to be carried out before the tactic resumes. The technique works by replaying initial parts of tactic executions and then carrying on with the computation in the same situation in which the tactic execution was intercepted. The tactic can then be resumed.

Our approach is simple and can be implemented easily in existing systems. We have implemented the semantics straightforwardly in Common Lisp using continuations. While this implementation works and is useful for experimenting with the technique, it is not efficient enough for a production strength theorem prover.

Therefore, we have implemented a much more efficient version by compiling the tactic language into common lisp code. This has several advantages: tactic expressions are analysed at compile-time, not run-time; careful compilation avoids the need to construct most of the lexical closures at run-time; program variables are translated into common lisp variables; and finally, the resulting code can be fed through the common lisp compiler, so all its optimisations apply to the tactics code for free, in particular most calls to named tactics and success continuations are tail calls and so do not consume stack space. The tactic compiler is integrated with the underlying programming language by a common lisp macro. We plan to integrate the compiled version into the INKA system [2].

The advantage of the technique is that both the user and the machine can determine the overall structure of a proof, and both can fill in details of a proof that the other is working on without interrupting it. This is not possible in existing theorem provers.

The technique is a particular simple starting point to implement derivational analogy in tactical theorem provers. More sophisticated forms of analogy can be implemented by refining the way in which choice points are transferred from the source to the target proof attempt. Also, ideas from transformational analogy can be used to patch traces before they are replayed. We will investigate these extensions in the future.

References

1. H. Abelson and G. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.
2. S. Autexier, D. Hutter, H. Mantel, and A. Schairer. System description: INKA 5.0 – a logical voyager. In *Proc. 16th International Conference on Automated Deduction (CADE-16)*, 1999.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, August 1993.
4. J.G. Carbonell. Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M.Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*. M. Kaufmann, 1986.
5. M. Carlsson. On implementing prolog in functional programming. *New Generation Computing*, 2(4), 1984.
6. C. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*. MIT Press, 1991.
7. A. Felty and D. Howe. Generalization and reuse of tactic proofs. In *5th International Conference on Logic Programming and Automated Reasoning (LPAR-94)*, 1994.
8. P. Graham. *On Lisp – Advanced Techniques for Common Lisp*. Prentice Hall, 1994.
9. D. Hutter. Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997. Kluwer-Publishers.
10. D. Hutter. Annotated Reasoning. *Annals of Mathematics and Artificial Intelligence*, 29:183–222, 2000. Kluwer-Publishers.
11. T. Kolbe. *Optimizing Proof Search by Machine Learning Techniques*. PhD thesis, Technische Hochschule Darmstadt, 1997.
12. T. Kolbe and C. Walther. Reusing proofs. In *Proc. 11th European Conference on Artificial Intelligence (ECAI-94)*, 1994.
13. E. Melis. A model of analogy-driven proof-plan construction. In *Proc. 14th International Joint Conferences on Artificial Intelligence (IJCAI-95)*, 1995.
14. T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation based generalization: a unifying view. *Machine Learning*, 1:47–80, 1986.
15. P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. M. Kaufmann, 1992.
16. S. Owen. *Analogy for Automated Reasoning*. Academic Press, 1990.
17. J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 1993.

18. J. Richardson and A. Smaill. Continuations of proof strategies. In *Proc. 4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, 2001.
19. M. Veloso and J.G. Carbonell. Towards scaling up machine learning: A case study with derivational analogy in prodigy. In S. Minton, editor, *Machine Learning Methods for Planning*. M. Kaufmann, 1993.

A Proof-Planning Framework with Explicit Abstractions Based on Indexed Formulas

Serge Autexier

FR 6.2 Informatik, Saarland University, P.O. Box 15 11 50, 66041 Saarbrücken, Germany,
e-mail:autexier@ags.uni-sb.de

Abstract. A major motivation of proof-planning is to bridge the gap between high-level, cognitively adequate reasoning for specific domains, and calculus-level reasoning to ensure soundness. For high reasoning levels the cognitive adequacy of representation and reasoning techniques is a major issue, while for lower reasoning levels the adequacy wrt. the modelled domain is important. Furthermore, proof construction is an engineering task and there is a need to support the design and application of proof-search engineering methods. To this end we present a framework to explicitly support different reasoning levels. To structure reasoning levels the framework allows for an explicit representation of abstractions and proof-search refinement techniques. In order to ensure soundness within a reasoning level, we use techniques developed in the context of matrix characterisation relying on the notion of indexed formulas. Furthermore, we introduce a uniform concept for contextual reasoning, and sketch basic tacticals for the definition of tactics to organise the overall proof-search inside and across different reasoning levels.

Keywords: Proof-planning, Methodology, Abstraction, Matrix Characterisations

1 Introduction

A major motivation for proof-planning is to allow on the one hand for a high-level reasoning which is (cognitively) adequate for specific problems and domains. On the other hand the final product of the reasoning process must be a calculus level proof wrt. some logic, in order to have a verifiably sound proof. Besides the cognitive motivation, another reason for the different reasoning levels is that the construction of a proof for some theorem is an *engineering task*. Analogously to the development of software, where one starts with a high-level specification of a program that is subsequently refined to an executable program in some target programming language, we view proof-planning as a methodology to design (and subsequently apply) different reasoning levels as well as methods to stepwise refine *proof specifications* (i.e. proof plans). Hence, proof-planning has been introduced as a methodology to refine high-level proof-plans of a theorem to a calculus level.

The objectives of the different reasoning levels are different: for high reasoning levels, the adequacy of the representation and the reasoning techniques is a key issue, in order to allow for an intuitive reasoning, which is as close as possible to say the style of a human. Although the reasoning inside one reasoning level should be sound, there is another notion of *relative soundness* due to the presence of different reasoning levels. A high reasoning level is *relatively sound* wrt. some lower reasoning level, if any high level proof can be refined to some low level proof. Usually, this relative soundness does not hold, which is a well-known effect in the presence of abstractions [5]. We do not want to enforce the *relative soundness* of high reasoning levels, as this usually may hamper the adequacy of representations, whereas the primary purpose of those levels is to allow for an efficient and adequate search for a proof plan. The relative soundness of a *proof* on a high reasoning level is established by stepwise refinement of this proof to a calculus level proof. On the lower reasoning levels we are less concerned with the cognitive adequacy of representations, but rather with the adequacy wrt. the modelled domain, as for example some program behaviour or a mathematical domain.

In this paper we present a proof-planning framework that supports the design and application of proof search engineering methods. First, we define in Sect. 2 the notion of a *reasoning level*, which consists of *reasoning objects* (e.g formulas, sequents, etc.) and *reasoning procedures* (e.g. tactics, planning algorithms, general theorem proving algorithms, etc.). For example, in a proof-planner like [10, 2], the reasoning objects are sequents as well as the pre- and postconditions of some methods, which are used as inference rules. The reasoning procedure of a proof-planner is the depth-first search algorithm applying the methods. To ensure the soundness wrt. some logic within a reasoning level, a variety of logics are predefined in the framework, and a uniform mechanism to encode those logics is introduced. For sake of representational adequacy, the logics are not embedded in some meta-logic. Instead we present in Sect. 4 techniques developed in the context of matrix characterisation [12, 9] that rely on the notion of *indexed formulas*. Those allow for a uniform treatment of a large class of classical, modal and intuitionistic logics, without having to change the syntax of the formulas.

Secondly, we define in Sect. 3 vertical structuring mechanisms to relate different reasoning levels. This includes on the one hand the explicit definition of abstractions, which define how reasoning objects of a higher reasoning level are related to those of a lower level. On the other hand, we introduce the notion of refinements, which determine how higher-level reasoning procedure can be implemented by some lower-level reasoning procedures. This subsumes e.g. the notion of refinement encoded in a method of a proof-planner like [10, 2], where the tactic wrapped inside a method determines the refinement of the abstract rule described by the pre- and postcondition of the method. Based on the general notion of refinement, the refinement of an abstract proof can be defined in a uniform manner.

Thirdly, reasoning procedures should be definable in an intuitive manner, again because of the overall motivation of cognitive adequacy. To this end we present in Sect. 5 window inference reasoning, which is a formalisation of an intuitive reasoning style, that allows to focus the reasoning on sub-formulas and sub-terms, and aims at contextual reasoning. We integrate window inference reasoning with the indexed formulas from Sect. 4 and present as a by-product how this allows for contextual reasoning in a canonical way.

Finally, we present in Sect. 6 the basic programming language to define reasoning and refinement procedures and how to combine those in order to organise the overall proof search. These build upon the intuitive reasoning rules provided by window inference reasoning and have an explicit failure and success semantics.

2 Reasoning Levels

In this section we introduce the notion of *reasoning levels*. Those are the *horizontal structures* of the framework in opposition to the abstraction and refinement techniques (cf. Sect. 3) that are the *vertical structures*.

A reasoning level consists on the one hand of the *objects* we want to reason about as well as the *basic reasoning rules*, and on the other hand of the reasoning procedures organising the proof search within one reasoning level. The former are defined by a signature and a set of formulas wrt. some logic. Thus, the reasoning objects and rules of a reasoning level are given by a pair (Σ, Φ), where Σ is the signature and Φ is a set of "axioms", and we denote such a pair as a *representation*. This is similar to the notion of a specification in formal specification languages [8], which are e.g. used to define abstract datatypes and functions about abstract datatypes. For a reasoning level, the signature Σ determines the syntax of the *reasoning objects* and the available *basic reasoning rules* are determined by the "axioms" from Φ.

Example 7. As an example we present a representation called `TLA` for TLA formulas [7] and a representation `TLA-States&Transitions` for the abstraction of TLA formulas. The signature for `TLA` contains the usual logical connectives, as well as $\Box$, $\Diamond$, and also *WF* to express fairness requirements. As an example we use a simple program, which has a counter x of initial value 15 and that decreases this counter in every step if it is greater than 0. This is encoded in TLA by the following formula

$$x = 15 \land \Box(x \geq 0 \land x' = x - 1) \land \mathit{WF}(x \geq 0 \land x' = x - 1)_x$$

In order to prove that the counter eventually has the value 0 (encoded by $\Diamond x = 0$), we use as abstraction a representation consisting of states and state transitions. This representation is called `TLA-States&Transitions` and consists of states $\langle n \rangle$ representing the state, where x has the value n, and state transitions represented by $\forall n : \langle n \rangle \Rightarrow \langle n - 1 \rangle$. This is an abstraction of the transition made by the program, i.e. $x \geq 0 \land x' = x - 1$. Finding a proof-plan for $\Diamond x = 0$, consists in finding a proof for $\langle 0 \rangle$ on the (abstract) representation `TLA-States&Transitions`. $\Box$

The second component of a reasoning level are the *reasoning procedures*, which organise the proof search within the reasoning level. We adopt a general notion for a reasoning procedure, by defining it to be an algorithm in some programming language, which interacts with the proof by applying basic inference rules. Additionally a reasoning procedure comes with a notion of `success` and `failure` to reflect about its own behaviour. E.g. in case of `failure` a reasoning procedure might invoke backtracking to some previous proof state. This notion of a reasoning procedure is general enough to subsume various kinds of specific reasoning procedures developed in automated theorem proving, like, e.g., a tactic in a tactical theorem prover, the various algorithms in an automated theorem prover, or a proof-planning algorithm in some proof-planner.

In summary, we have defined a reasoning level to consist of two parts, namely *representations* defining the syntax of the reasoning objects and *reasoning procedures* to

$$
\begin{tikzcd}[column sep=small]
\text{Reasoning Level 3} = & \text{Representation-3} & + & \text{Reasoning Procedures 3} \arrow[d, "\text{Refinement}"] \\
\text{Reasoning Level 2} = & \text{Representation-2} \arrow[u, "\text{Abstraction}"] & + & \text{Reasoning Procedures 2} \arrow[d, "\text{Refinement}"] \\
\text{Reasoning Level 1} = & \text{Representation-1} \arrow[u, "\text{Abstraction}"] & + & \text{Reasoning Procedures 1}
\end{tikzcd}
$$

Fig. 1. Structuring reasoning levels

organise the proof-search inside the reasoning level. This can be sketched by the following equation:

$$\text{Reasoning level} = \text{Representation} + \text{Reasoning procedures}$$

The next step consists in relating reasoning levels. To this end *vertical structures* are introduced in the next section, that describe the mechanisms to switch between reasoning levels.

3 Structuring Reasoning Levels

In this section we define the mechanisms to structure the different reasoning levels of the framework (cf. Fig. 1). There are two dual notions of *vertical structures*: first, *abstractions* can be defined which describe how objects of a higher reasoning level can be constructed from objects of a lower reasoning levels. Formally, an abstraction is a mapping of a set of objects of a source representation R_S to a set of objects of some target representation R_T. For the purpose of the framework, we define an abstraction to consist of a *name*, the source and target representations R_S and R_T, and an *abstraction procedure* (i.e. an algorithm) to compute the mapping.

Example 8. Consider as an example the source representation `NAT` of natural numbers and labelled fragments (`LF-NAT`) as respective target representation. *Labelled fragments* are known abstractions in the context of inductive theorem proving, where they are used to abstract over the specific differences (so-called *wave-fronts*) between two terms sharing a common *skeleton*. The representation `NAT` consists of a signature for natural numbers and – among others – of two axioms defining the addition of natural numbers

$$\forall y\,.\,0 + y = y \tag{1}$$
$$\forall x, y\,.\,s(x) + y = s(x + y) \tag{2}$$

The representation `LF-NAT` consists on the one hand of a signature extending the signature of `NAT` by a function $\bullet : \mathit{Nat} \to \mathit{Nat}$ that is used to represent the wave-fronts, and on the other hand of the axioms

$$\forall y\,.\,\bullet(y) = y \tag{3}$$

$$\forall x, y\,.\,\bullet(x) + y = \bullet(x + y) \tag{4}$$

$$\forall x\,.\,\bullet(s(x)) = s(\bullet(x)) \tag{5}$$

The *labelled-fragment abstraction* is an abstraction from `NAT` to `LF-NAT` and its associated *abstraction procedure* maps the singleton $\{(1)\}$ to the singleton $\{(3)\}$, and the singleton $\{(2)\}$ to the set $\{(4), (4)\}$. □

Thus, an *abstraction* consists of the source and target *representations* and a function mapping objects from the source representation to objects of the target representation. This can be expressed in an intuitive manner by the following equation:

Abstraction = Source & Target Representations + Abstraction procedure

The dual *vertical structuring* mechanism is the *refinement* of higher-level reasoning procedures to lower-level reasoning procedures. Take as an example of a high-level reasoning procedure some planning algorithm, which computes a proof-plan from methods. This proof-plan is a "proof" on the reasoning level of the proof-planning procedure, and consists of the application of basic rewriting rules defined by the pre- and postconditions of the methods. We can associate to this planning procedure a *refinement procedure*, which takes generated proof-plan and refines it by calling the tactics wrapped inside the different methods. In this scenario the refinement information is associated to each basic reasoning rule of the higher reasoning level via the methods. A method describes that an abstract proof step satisfying a certain pattern (described by the pre- and postcondition of the method) could be refined by the tactic wrapped inside the method. The refinement procedure exploits this knowledge in order to refine a proof plan. In general it is not possible to describe this refinement information by patterns. In order to deal with the general case, we allow to associate a refinement procedure to some reasoning procedure. Such a refinement procedure takes the proof generated by the reasoning procedure as a proof-plan to compute a proof on the lower reasoning level (using the lower-level reasoning procedures). Similarly to reasoning procedures, refinement procedures come with an explicit `success` and `failure` semantics to reflect about their own behaviour. This allows for example to backtrack the refinement process and to try alternative refinements.

From the above we can summarise the notion of refinement by stating that a refinement is composed of a reasoning procedure and an associated refinement procedure. This can be sketched by the following equation:

Refinement = Reasoning procedure + Refinement procedure

Note that a refinement procedure for some reasoning procedure $\mathcal{P}$ can dually be interpreted as some kind of abstraction, where all the (lower-level) reasoning procedures used by the refinement procedure are abstracted to the reasoning procedure $\mathcal{P}$. This notion of abstraction is different from the notion presented above, since it is an abstraction of reasoning procedures while the former is an abstraction of objects and axioms. Furthermore do refinement procedures exploit knowledge about object abstractions during

α	α_0	α_1
$(\varphi \vee \psi)^+$	φ^+	ψ^+
$(\varphi \Rightarrow \psi)^+$	φ^-	ψ^+
$(\varphi \wedge \psi)^-$	φ^-	ψ^-
$(\neg\varphi)^+$	φ^-	$-$
$(\neg\varphi)^-$	φ^+	$-$

β	β_0	β_1
$(\varphi \wedge \psi)^+$	φ^+	ψ^+
$(\varphi \vee \psi)^-$	φ^-	ψ^-
$(\varphi \Rightarrow \psi)^-$	φ^+	ψ^-

γ	$\gamma_0(c)$
$(\forall x \centerdot \varphi)^-$	$(\varphi[x/c]^-$
$(\exists x \centerdot \varphi)^+$	$(\varphi[x/c]^+$

δ	$\delta_0(c)$
$(\forall x \centerdot \varphi)^+$	$(\varphi[x/c]^+$
$(\exists x \centerdot \varphi)^-$	$(\varphi[x/c]^-$

ν	ν_0
$(\Box\varphi)^-$	φ^-
$(\Diamond\varphi)^+$	φ^+

π	π_0
$(\Box\varphi)^+$	φ^+
$(\Diamond\varphi)^-$	φ^-

Fig. 2. Uniform notation for signed formulas

the refinement. Hence, an object abstraction also gives rise to some notion of refinement. The reason why we introduced abstractions mainly for objects and refinement mainly for reasoning procedures is because this is the usual way those are used.

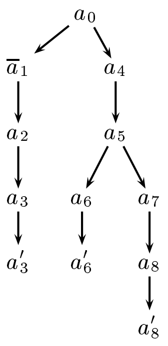
4 Handling Soundness

In this section we define a uniform mechanism to handle soundness for a variety of logics. For sake of the adequacy of representation, we want to support directly the soundness wrt. some logic without altering the syntax of formulas. Subsequently, we do not want to embed a logic in some meta-logic, like e.g. intuitionistic higher-order logic. In comparison to the *representations* introduced in Sect. 2, a *logic* also defines the syntax of reasoning objects, but comes with its own formal proof-theory (whereas a representation is relative to some logic, and thus "inherits" the proof-theory from this logic). Thereby its proof-theory determines, for example, whether in a sequent calculus for this logic an elimination rule for some object-level quantifier must satisfy the *Eigenvariable* condition, or how other parts of a sequent are affected by the elimination rule for some $\Box$ or $\Diamond$ in some modal logic.

A background mechanism to handle the proof-theory of different logics must allow for a uniform handling of soundness and completeness results for those logics. Such a basis for propositional and first-order, classical, modal and intuitionistic logics is provided by *indexed formula trees* [12] and relies on polarities of (sub-)formulas and uniform notation [12, 4, 3].

The starting point for uniform notation is the concept of *signed formulas*, which are formulas φ annotated by some positive (φ^+) or negative (φ^-) polarity. Intuitively, a signed formula φ^+ occurs in the succedent of a sequent, i.e. $\Gamma \vdash \varphi, \Delta$, and φ^- occurs in the antecedent of a sequent, i.e. $\Gamma, \varphi \vdash \Delta$. The signed formulas can be categorised into different classes, according to their behaviour during the proof search: consider as an example the sequent $\vdash \varphi \vee \psi$. The application of the respective sequent decomposition rule leads to the single sequent $\vdash \varphi, \psi$. All those kinds of signed formulas are said to be of type α. The complete list of signed formulas of this type are given in the table for α in Fig. 2.

Furthermore, those signed formulas whose decomposition leads to a split on the sequent proof tree are said to be of type β (cf. Fig. 2). Signed formulas of type γ correspond to instantiation rules, without restrictions to the substituted terms, while for δ-type signed formulas the instantiation must respect the *Eigenvariable* condition (e.g.



u	*Pol*(u)	*Lab*(u)	*Ptype*(u)	*Stype*(u)
a_0	$+$	$((\Box(\forall x \,.\, P(x))) \Rightarrow (\exists y \,.\, (P(y) \wedge \Box P(y)))$	α	$-$
$\overline{a}_1$	$-$	$\Box \forall x \,.\, P(x)$	ν	α_1
a_2	$-$	$\forall x \,.\, P(x)$	γ	ν_0
a_3	$-$	$P(x)$	$-$	γ_0
a_3'	$\circ$	x	$-$	$-$
a_4	$+$	$\exists y \,.\, (P(y) \wedge \Box P(y))$	γ	α_2
a_5	$+$	$P(y) \wedge \Box P(y)$	β	γ_0
a_6	$+$	$P(y)$	$-$	β_1
a_6'	$+$	y	$-$	$-$
a_7	$+$	$\Box P(y)$	π	β_2
a_8	$+$	$P(y)$	$-$	π_0
a_8'	$\circ$	y	$-$	$-$

Fig. 3. Indexed Formula tree

$(\forall x \,.\, \varphi)^+$, which corresponds to $\vdash \forall x \,.\, \varphi$). For modal logic formulas, the decomposition of ν-type signed formulas does not affect the other formulas in a sequent (e.g. $\Gamma, \Box\varphi \vdash \Delta$), while the decomposition of π-type signed formulas changes the other parts of the sequent (e.g. $\Gamma \vdash \Box\varphi, \Delta$).

In Fig. 2 the rules also indicate how the polarity is inherited to the sub-formulas of a signed formula. Using these rules, polarities can be assigned to all sub-formulas of a formula and the polarity of a subexpression indicates whether this sub-formula will occur in the antecedent or the succedent of a sequent, if all structural sequent decomposition rules are applied until this formula. E.g. the signed formula $(\varphi \Rightarrow \psi)^-$ corresponds to $\Gamma, \varphi \Rightarrow \psi \vdash \Delta$. The β-rule reduces this signed formula into the signed formulas φ^+ and ψ^-. This corresponds to what happens if the $\Rightarrow$-*Left*-decomposition rule is applied on $\Gamma, \varphi \Rightarrow \psi \vdash \Delta$, since it results in a split of the proof (indicated by β), and φ occurs in the succedent of the first premise $\Gamma \vdash \varphi, \Delta$, and ψ occurs in the antecedent of the second premise $\Gamma, \psi \vdash \Delta$.

Indexed formula trees as introduced in [12] exploit the tree structure of formulas and annotate each sub-formula according to the reduction rules with their respective polarity and uniform type. Consider the indexed formula tree in Fig. 3 of the signed formula $((\Box\forall x \,.\, P(x)) \Rightarrow (\exists y \,.\, (P(y) \wedge \Box P(y))))^+$. For each sub-formula there is a position a_i that is either a position constant or position variable (which are over-lined, like $\overline{a}_1$). A set of properties associated to each position is listed in the adjacent table: those are the actual sub-formula (called the *label* of the position), its polarity, its uniform type (called *primary type*), and its *secondary type*, which is its type inside the parent formula reduction rule. According to those reduction rules, only sub-formulas that are related by an α-type position may occur together in a final sequent (after application of the respective sequent calculus decomposition rules). This can be illustrated by the sub-formulas $\forall x \,.\, P(x)$ and $P(y)$ that are related by the position a_0, or, similarly, the sub-formulas $\forall x \,.\, P(x)$ and $\Box P(y)$). Hence, from the indexed formula tree we obtain a simple notion of a *context* for some subexpression, i.e. those parts of the signed formula tree, that can be "used" for this subexpression. This is exploited in Sect. 5 to handle contextual reasoning.

The tree structure of the indexed formula tree induces an ordering $\prec$ among the positions. A first-order substitution σ instantiates a variable bound on some position a of primary type γ and induces an additional ordering relationship between the position a and γ- or δ-type positions binding variables in $\sigma(x)$. Combined with the original ordering $\prec$, we obtain a relation $\lhd$. Such a first-order substitution is only sound, if $\lhd$ is irreflexive.

Furthermore, we can associate to any position a a string of positions, consisting of some of the positions governing a (i.e. smaller wrt. $\prec$) and that are of primary type ν or π. Such a string is called the modal prefix of the position a. Which positions are actually used in the prefix depends on the modal logic under consideration (K, K4, S4, etc., cf. [12] for details). The variable positions are exactly all positions of primary type ν. *Modal substitutions* are used to instantiate the position variables by modal position strings. Depending on the modal logic under consideration, there is a set of restrictions how the position variables can be instantiated (cf. [12]). Furthermore, similar to first-order substitutions a modal substitution induces an ordering among the ν- and π-type positions and again the induced overall ordering $\lhd$ must be irreflexive. All the requirements about the instantiation of variables, position variables, the irreflexivity of induced overall orderings $\lhd$ among positions are subsumed by the uniform notion of $\mathcal{L}$-admissible substitutions, where $\mathcal{L}$ is either classical first-order logic, or propositional or first-order modal logic K, K4, D, D4, T, S4, or propositional or first-order intuitionistic logic. The uniform notion of $\mathcal{L}$-admissible substitutions is the key concept allowing for a uniform treatment of a large class of logics.

Based on these techniques, we introduce the uniform notion of *valid proof states* (i.e., a signed formula tree with an irreflexive ordering $\lhd$). From [12] we obtain, that the application of $\mathcal{L}$-admissible substitutions preserves the validity of proof states. The problem to compute $\mathcal{L}$-admissible substitutions can be solved by using a standard unification procedure with occur-check for object variables, and the generic prefix unification procedure from [9], which computes only $\mathcal{L}$-admissible substitutions for position variables, where $\mathcal{L}$ is any of the above logics. The notion of valid proof states can subsequently be used to formally prove the $\mathcal{L}$-soundness of any other transformation we might define on the signed formula tree. Hence, using signed formula trees gives us a powerful and uniform mechanism to support a variety of logics inside the framework.

5 Intuitive and Contextual Reasoning

For the design of reasoning and refinement procedures we support an intuitive reasoning style. To this end we aim to focus the reasoning process on arbitrary sub-expressions of some actual expression and to have a strong support of contextual reasoning inside a focus. Window inference [11, 6] has been introduced as a formalisation of an intuitive and hierarchical reasoning style. The basic objects are windows

$$\langle [] \vdash P(a) \Rightarrow \neg(a = b)\rangle^{+}$$

which represents that the focus is on the positive $(+)$ formula $P(a) \Rightarrow \neg(a = b)$ (denoted as the *content* of the window) and the context of this focus is empty ($[]$). The basic window inference rules are

1. to focus on some subexpression of the content of a window

$$\langle [] \vdash P(a) \Rightarrow \neg(a = b)\rangle^{+} \quad \underset{\text{w}}{\longmapsto} \quad \langle [a = b] \vdash P(a)\rangle^{-} \quad \underset{\text{w}}{\longmapsto} \quad \langle [a = b] \vdash a\rangle^{\circ},$$

   where $\underset{\text{w}}{\longmapsto}$ denotes a focus derivation step. Note that this rule must be defined for any logical connective (positive $\Rightarrow$ in the example) and argument position (first argument of $\Rightarrow$ in the example), in order to adjust the context in a consistent manner. While this is not difficult for classical logics, where the context of a sub-focus is either equal or an extension of the previous context, this gets difficult as soon as we deal with non-classical logics.
2. to transform the content of a window using information from the context

$$\langle [a = b] \vdash a\rangle^{\circ} \quad \underset{\text{w}}{\longmapsto} \quad \langle [a = b] \vdash b\rangle^{\circ},$$

3. and to leave a focus and return to the parent focus

$$\langle [a = b] \vdash b\rangle^{\circ} \quad \underset{\text{w}}{\longmapsto} \quad \langle [a = b] \vdash P(b)\rangle^{-} \quad \underset{\text{w}}{\longmapsto} \quad \langle [] \vdash P(b) \Rightarrow \neg(a = b)\rangle^{+}.$$

Furthermore, window inference reasoning also allows one to focus on the condition of a window. To this set of basic rules we added a general rule to allow one to introduce a case analysis:

$$\langle [H] \vdash t\rangle^{p} \quad \underset{\text{w}}{\longmapsto} \quad \langle [A \mid H] \vdash t\rangle^{p} \text{ and } \langle [B \mid H] \vdash t\rangle^{p},$$

In order to ensure the soundness of the case analysis[1] the condition $\langle [H] \vdash A \vee B\rangle^{+}$ is generated as an additional goal. In order to allow for an intuitive design of reasoning procedures, we integrated window inference reasoning with indexed formula trees (cf. Sec. 4) by annotating positions in the indexed formula trees with windows. As focusing on subexpressions is along the tree structure, opening a sub-window corresponds to adding a window on the respective position. Similarly, leaving a focus means removing the window from a position and returning to the previous position.

The integration of indexed formula trees with windows results in the following structure of the framework: inside the framework, proof states are represented by indexed formula trees with windows. They serve as a soundness backbone managing any proof transformation, like e.g. the instantiation of variables or the application of rules from the context. The indexed formula trees are invisible to the reasoning procedures, which communicate with them via the windows annotated to specific positions and applying window inference rules, which result in a change of the proof state. An example for this is the opening of a sub-window, the instantiation of variables, etc. The difficulty remains that contextual information must be provided to the reasoning procedures, like what is the polarity of a window, which are the instantiable variables, and especially which are possible rules from the context of a window and whether they are applicable. These difficulties are elegantly resolved by using indexed formula trees:

Polarity of windows. The polarity of a window is simply the polarity of the position it is attached to in the indexed formula tree.

[1] Even for intuitionistic logic.

Instantiable Variables. The *instantiable* variables of some window are simply all variables that are bounded on some γ-type position in the formula tree. In order to check the admissibility of the instantiation of a variable, we check the $\mathcal{L}$-admissibility of the substitution (i.e. the irreflexivity of the new induced ordering $\lhd$, cf. Sec. 4).

Sampling contextual rules. The context of a window is uniformly determined by the position in the indexed formula: all parts of the indexed formula, that are related to the actual window by an α-type position, are in the *logical* context. Indeed, according to the reduction rules in Fig. 2, only α-related formulas may occur together in some final sequent after applying respective calculus decomposition rules (cf. Sec. 4). Hence, usable rules to change the content of a window are for example all *negative* equations and implications in the context. The application of a negative equation is basically a paramodulation rule, whereas the application of a negative implication is a resolution step. The *conditions* for such a rule are all the formulas that are β-related to this rule. From this we can define a uniform concept of a *replacement rule* $[\varphi]\ u \to v$: intuitively, this rule means that we can replace a content u of a window by v, if we prove the conditions φ. This notion can be defined formally as well as if it is admissible wrt. some focus (i.e. position).

Definition 12 (Admissible replacement rules). *Let a be a position of label s and polarity p in some indexed formula tree. Then $[\Phi]\ u \to v$ is an* admissible replacement rule *for a, if*

1. *u and v are either the left- and right-hand side of a negative equation, that is α-related to a, or u and v are β-related formulas with the same modal or intuitionistic prefixes, of polarities $-p$ and p respectively, and α-related to a,*
2. *and Φ contains all other formulas that are β-related to u and v.* □

Application of replacement rules. To apply a replacement rule on some focus, we must syntactically unify the left-hand side of the rule with the context of the focus as well as their modal prefixes. Furthermore, if the replacement rule is not an equation, the focus and the left-hand side of the rule must have opposite polarities. For unification we use standard unification procedures for object level variables and rely on the generic prefix unification procedure from [9], in order to compute only substitutions that are admissible for the actual logic. This allows to prove for all logics considered in [12] the soundness of the replacement rule application.

In summary, the integration of window inference reasoning with indexed formulas allows on the one hand for the design of intuitive reasoning procedures. On the other hand it provides a simple but powerful mechanism to support contextual reasoning.

6 Organising Proof Search

The overall proof search inside the framework is organised by the reasoning and refinement procedures, and some additional language constructs to combine them. The reasoning procedures are based on windows and interact with the proof in two manners: first, the content of a window can be modified by applying a replacement rule

```
procedure abstract-prove-reachable (window:TLA-
States&Transitions)
   ...
endprocedure

refinement abstract-prove-reachable-refine
   for abstract-prove-reachable
   ...
endrefinement

procedure prove-reachable (window:TLA)
 call-on-abstraction Reachable-Abstraction window abstract-prove-
reachable
endprocedure
```

Fig. 4. TLA reasoning and refinement procedures

from the context of the window. Second, they can introduce a case analysis over some arbitrary formulas $A_1 \vee \ldots \vee A_n$ at any stage of the reasoning process. Following the intuitive reasoning paradigm, reasoning procedures can focus the proof search on some sub-window by calling another reasoning procedure on the sub-window. Finally, they can invoke the reasoning on some higher reasoning level by calling a reasoning procedure on some abstraction of the actual window. To this end they indicate the abstraction to be used and the reasoning procedure of the higher reasoning level to call. In case the abstract reasoning procedure succeeds, the refinement of the (abstract) proof is achieved by calling the refinement procedure of the abstract reasoning procedure. Only if the refinement procedure succeeds, the proof planning attempt is successful.

Example 9. Take as an example a reasoning procedure `prove-reachable` that is specialised to prove temporal properties of the form $\Diamond\varphi$ (cf. Fig. 4). Hence, the argument of the reasoning procedure is declared to be from the representation `TLA`. Furthermore let `abstract-prove-reachable` be the reasoning procedure on the representation `TLA-States&Transitions` introduced in Sect. 2, and let `Reachable-Abstraction` be the abstraction from the representation `TLA` to the representation `TLA-States&Transitions`. In Fig. 4 we sketch the body of `prove-reachable`: this procedures calls the abstract reasoning procedure via `call-on-abstraction`, which abstracts the argument window to the representation `TLA-States&Transitions` and calls the abstract reasoning procedure `abstract-prove-reachable`. If this procedure succeeds, then the corresponding refinement procedure `abstract-prove-reachable-refine` is called, which refines the abstract proof on `TLA-States&Transitions` to a proof on `TLA` formulas. □

7 Conclusion

In this paper we took up the motivation that the task to construct a proof is an engineering task. This necessitates a methodology to design and apply proof engineering techniques and we emphasised that the techniques developed in proof-planning aim at these goals. Based on that we developed a proof-planning framework, that comes with an explicit notion of different reasoning levels. The reasoning levels can be structured via abstraction and refinement relations. Furthermore, we emphasised that cognitive adequate reasoning is a key issue for the design of higher reasoning levels. This entails on the one hand, that the representation of reasoning objects of the different reasoning levels should be intuitive. Subsequently, a uniform mechanism to support directly soundness wrt. a large variety of logics has been introduced, without having to alter the representation of logical formulas. Furthermore, we integrated this mechanism with an extension of the window inference reasoning style to allow for the design of intuitive reasoning procedures. A major benefit of this integration is a straightforward and elegant support of contextual reasoning, a key concept for intuitive reasoning. Finally, we sketched how proof-search engineering methods like reasoning procedures, abstractions and refinements can be defined in the framework, and how these techniques are linked together and subsequently can be applied for proof-search.

The framework serves as the basis for the ongoing implementation of the core reasoning mechanisms in the INKA 5.0 system [1]. Future work consists of finishing the implementation of the framework, and to validate the adequacy of the framework by implementing reasoning levels and structuring mechanisms for specific domains and problems. Theoretical future work is concerned with the integration of further logics into the framework.

References

1. Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. System description: INKA 5.0 – a Logic Voyager. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE)*, LNAI 1632, Trento, Italy, 1999. Springer.
2. Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiarong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Erica Melis, Andreas Meier, Wolf Schaarschmidt, Jörg Siekmann, and Volker Sorge. Ωmega: Towards a mathematical assistant. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE)*, LNAI 1249, Townsville, North Queensland, Australia, 1997. Springer.
3. M. Fitting. *First-Order Logic and Automated Theorem Proving / 2nd Edition*. Springer-Verlag New York Inc., 1996. ISBN 0-387-94593-8.
4. Melvin Fitting. Tableau methods of proof for modal logics. *Notre Dame Journal of Formal Logic*, XIII:237–247, 1972.
5. Fausto Giunchiglia and Toby Walsh. A Theory of Abstraction. *Journal of Artificial Intelligence*, 56(2-3):323–390, 1992. Also as technical report IRST-Technical Report 9001-14.
6. Jim Grundy. Window inference in the HOL system. In *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, 1991.
7. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
8. J. Loeckx, H.-D. Ehrig, and M Wolf. *Specification of Abstract Data Types*. Teubner, Chichester;New York;Brisbane, 1996. ISBN 3-519-02115-3.
9. Jens Otten and Christoph Kreitz. T-string unification: Unifying prefixes in non-classical proof methods. In P Miglioli, U Moscato, and et. al., editors, *Proceedings of 5th Workshop on theorem Proving with analytic tableaux and related methods*, LNAI 1071, pages 244–260. Springer Verlag, 1996.

10. Julian D.C. Richardson, Alan Smaill, and Ian M. Green. System description: proof planning in higher-order logic with λ-clam. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-98)*, LNAI 1421. Springer, 1998.
11. Peter D. Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. In *Journal of Logic and Computation*, volume 3, pages 47–61, 1993.
12. Lincoln Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics*. MIT Press series in artificial intelligence, 1990.

Continuations of Proof Strategies

Julian Richardson[1] and Alan Smaill[2]

[1] Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton Campus
Edinburgh EH14 4AS
julianr@cee.hw.ac.uk

[2] Division of Informatics
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
smaill@dai.ed.ac.uk

1 Introduction

Proof planning [1] is a technique for automatically constructing proofs from abstractions of tactics called *methods*. Proof planning has been successfully applied in many domains, for example: mathematical induction [1], higher-order program synthesis [7] and hardware verification [3]. In the first proof planner, *Clam* [2], application of methods during the search for a proof plan was controlled by a simple waterfall (§2.1). In this paper, we describe *methodical* expressions, which we use to define more complex proof search strategies. Methodicals compose methods in the same way that tacticals compose tactics. The work described in this paper has been implemented as part of the λ*Clam* system [11, 10].

We present a meta-interpreter which gives a semantics to methodical expressions. In conjunction with a mechanism for executing methods and a search strategy, this meta-interpreter can be used as a proof planner. This planner does not have any notion of proof state, and is unsuitable for interactive proof.

We present a set of transformation rules which we use to define a new meta-interpreter which does maintain a notion of proof state, is suitable for interactive proof, and we prove that it is equivalent to the first in the sense that it produces the same plans. This meta-interpreter has been used for some time as the basis of the interactive planner in λ*Clam*, although it does not yet treat the whole of the methodical language.

The transformation rules can be interpreted as defining a notion of *continuation* [9] for proof strategies. Previous work on a continuation semantics for Prolog [4] indicates how the meta-interpreter should be extended to cover the more problematical elements of the methodical language.

The definition of methodicals, use of formal proof to verify equivalence of proof planners, and use of continuations in proof planning are novel.

2 Tactics, Methods, Tacticals and Methodicals

A *tactic* [5] is a procedure which constructs a piece of proof in some logic. The tactic is defined as a composition of primitive rules of inference. For example, a tactic may strip all universal quantifiers from the front of a goal and then apply a lemma. When a tactic is applied to a goal in the proof, one branch is created for each subgoal which has not been proved by the tactic, and the goal is labeled with the tactic, not with the individual proof rules the tactic applied. LCF defines *tacticals* for composing tactics.

In proof planning, we distinguish between the *object-level*, where proofs are constructed using tactics, and the *meta-level* (where the control, search and proof explanation properties are better than at the object-level), where schematic proofs are constructed from abstractions of the object-level tactics called *methods*. Meta-level goals G are generated by the following grammar:

$$G ::= emptyGoal \mid A \mid allGoal\, \lambda v.G[v] \mid G **G$$

where meta-level sequents A are abstractions of object-level sequents formed by operations (which we do not here formalise) such as omitting hypotheses, annotating terms, or replacing terms by meta-variables. *allGoal* represents meta-level universal quantification, and $**$ represents pairing of meta-level goals.

A methodical expression consists of methods linked by methodicals. Methodical expressions which contain no methodicals are called *atomic methods*, otherwise they are *compound methods*. The following methodicals, and their types, are available (for reasons of clarity we simplify a little and do not distinguish between $then\_meth$ and $then\_meths$):

Methodical	Type	Description
id_meth	meth	Does nothing.
orelse_meth	meth → meth → meth.	First, which failing second
cond_meth	(goal → o) → meth → meth → meth.	If condition holds, then meth1, otherwise meth2
pair_meth	meth → meth → meth.	Pair of methods for pair of goals
try_meth	meth → meth.	Either given method, or id_meth
repeat_meth	meth → meth.	Exhaustive recursion on subgoals – oblige to succeed at least once
then_meth	meth → meth → meth.	Chain second method to all subgoals of first
map_meth	meth → meth.	Map method over compound goal structure
complete_meth	meth → meth.	Only succeed if subgoals all trivial
cut_meth	meth → meth.	Only give first solution

2.1 Hierarchical Proof Planning

Clam employed a two-deep method hierarchy, consisting of methods and *submethods*. *Clam*'s *waterfall* is a list of methods whose applicability is tested whenever a new subgoal is generated. Submethods must be invoked explicitly by methods (or submethods), and cannot appear in the waterfall. The need to explicitly invoke submethods made expressing complex proof strategies declaratively difficult. For example, the control structure of *Clam*'s *step_case* method, depicted in Figure 1, was realised by making explicit calls to the planning apparatus, thereby spoiling any chance of a declarative reading of the method.

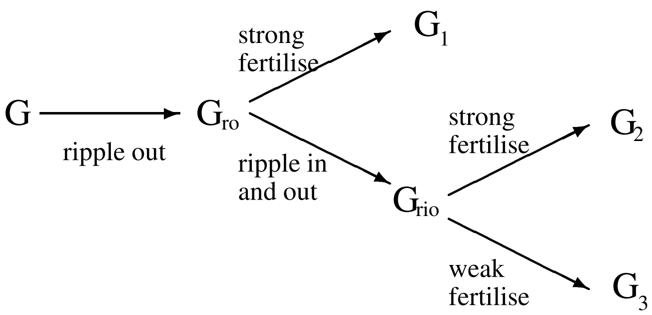


Fig. 1. The structure of *Clam*'s *step_case* method. The result of method application is the generated subgoal with the lowest numerical index.

In λ*Clam*, we use methodicals to declaratively define a compound method which imposes the structure of the step_case strategy:

```
method_def step_case
  (cut_meth (then_meth
              (repeat_meth ripple_out_once)
              (orelse_meth strong_fertilise
                           (then_meth
                              (repeat_meth ripple_inout_once)
                              (orelse_meth strong_fertilise
                                           weak_fertilise))))).
```

Such structures make explicit the control component of the planning strategy, which eases comprehension and may permit some formal reasoning about the planning process.

In λ*Clam*, the simple waterfall of *Clam* is replaced by a compound method *top_meth*. Planning normally starts by trying to expand *top_meth*. This is the default definition, but others are used to create special-purpose proof strategies with more complex forms of top-level control.

```
method_def top_meth
  (complete_meth (repeat_meth
                   (orelse_meth all2i
                   (orelse_meth taut
                   (orelse_meth sym_eval
                   (orelse_meth existential
                   (orelse_meth (repeat_meth generalise)
                                ind_strat))))))).
```

3 A Meta-Interpreter for Methodicals

We now present rules which define a meta-interpreter for methodicals. We use the notation $M_1 : G \mapsto G'$ to mean that method M_1 applies to goal G resulting in subgoal G' (which could consist of one or more subgoals paired with $**$). The rules below can be interpreted as an inductive definition of the (non-deterministic) $\mapsto$ relation. Methodical expressions are expanded by applying the rules in a goal-directed bottom-to-top way.

$$\frac{}{id\_meth : G \mapsto G} \tag{1}$$

$$\frac{M : G \mapsto G'}{repeat\_meth\ M : G \mapsto G'} \tag{2}$$

$$\frac{M : G \mapsto G'' \qquad repeat\_meth\ M : G'' \mapsto G'}{repeat\_meth\ M : G \mapsto G'} \tag{3}$$

$$\frac{M_1 : G \mapsto G'' \qquad M_2 : G'' \mapsto G'}{then\_meth\ M_1\ M_2 : G \mapsto G'} \tag{4}$$

$$\frac{M_1 : G \mapsto G'}{orelse\_meth\ M_1\ M_2 : G \mapsto G'} \tag{5}$$

$$\frac{M_2 : G \mapsto G'}{orelse\_meth\ M_1\ M_2 : G \mapsto G'} \tag{6}$$

$$\frac{}{try\_meth\ M : G \mapsto G} \tag{7}$$

$$\frac{M : G \mapsto G'}{try\_meth\ M : G \mapsto G'} \tag{8}$$

$$\frac{M_1 : G_1 \mapsto G_1' \qquad M_2 : G_2 \mapsto G_2'}{pair\_meth\ M_1\ M_2 : G_1\ ** G_2 \mapsto G_1'\ ** G_2'} \tag{9}$$

$$\frac{M : G \mapsto G'}{map\_meth\ M : G \mapsto G'} \text{if } G \text{ is not a pair} \tag{10}$$

$$\frac{map\_meth\ M : G_1 \mapsto G_1' \qquad map\_meth\ M : G_2 \mapsto G_2'}{map\_meth\ M : G_1\ ** G_2 \mapsto G_1'\ ** G_2'} \tag{11}$$

$$\frac{M_1 : G \mapsto G'}{cond\_meth\ C\ M_1\ M_2 : G \mapsto G'} \text{if } C \text{ holds} \tag{12}$$

$$\frac{M_2 : G \mapsto G'}{cond\_meth\ C\ M_1\ M_2 : G \mapsto G'} \text{if } \neg C \text{ holds} \tag{13}$$

$$\frac{M : G \mapsto G'}{complete\_meth\ M : G \mapsto G'} \text{if } G' \text{ trivial} \tag{14}$$

$$\frac{M : G[x/v] \mapsto H[x/v]}{M : allGoal\ \lambda v.G \mapsto allGoal\ \lambda v.H} x \text{ does not appear in } G \text{ or } H \tag{15}$$

We do not here define *cut_meth*, since (as for the Prolog cut, !) its meaning is procedural. We show how to handle *cut_meth* in Section 7.

These rules are non-deterministic, but when applied in conjunction with a search strategy (for example depth-first search), and a mechanism for interpreting atomic methods, they define a proof planner. Proof planning is applied to a goal G, yielding subgoals G', by expanding the expression $top\_meth : G \mapsto G'$.

4 Representation of Proof State

It is natural to represent the state of a partial proof as a tree, in which nodes are labeled with subgoals, and branching represents conjunction. As a proof proceeds, nodes are added to or deleted from the tree. Presenting such a proof tree to a user provides an excellent form of feedback, allowing the user to comprehend and perhaps influence the course of the proof attempt, as was demonstrated in the *XBarnacle* [8] interactive proof planner (§8).

In *λClam*, however, the hierarchical nature of the proof strategy as represented by methodical expressions means that we must also label each node in the partial proof tree with the state of meta-interpretation of the proof strategy. For example, consider the proof constructed by applying a proof strategy *repeat_meth (then_meth* M_1 M_2*)* to a goal G. In (4) below, we see the state of meta-interpretation (proceeding from bottom to top, as in Section 3) up to the point where method M_1 has been successfully applied for the first time. Goals which are yet to be determined are denoted by "??".

$$\dfrac{\dfrac{M_1 : G \mapsto G' \qquad M_2 : G' \mapsto ??}{\mathit{then\_meth}\ M_1\ M_2 : G \mapsto ??} \qquad \mathit{repeat\_meth}\ (\mathit{then\_meth}\ M_1\ M_2) : ?? \mapsto ??}{\mathit{repeat\_meth}\ (\mathit{then\_meth}\ M_1\ M_2) : G \mapsto ??}$$

Suppose we wish the pause the proof at this point. What information do we need to record in order that we can return later and correctly continue where we left off? We must record the generated subgoal, G', but we must also record the state of interpretation of *repeat_meth (then_meth* M_1 M_2*)*, specifically that we should next apply M_2 to G', and then either apply *repeat_meth (then_meth* M_1 M_2*)* to the subgoal which arises, or stop. In the planner of Section 3 there is no way to derive or represent this information about how to continue.

In the next section, we give transformation rules which allow us to systematically derive this "continuation" information and represent it as a methodical expression, in this case *then_meth* M_2 *(orelse_meth id_meth (repeat_meth (then_meth* M_1 M_2*)))*.

5 Methodical Transformations and Proof State

We now present rewrite rules which can be used to transform any methodical expression M into an equivalent form $F \oplus L$ such that F is a singleton method (not a compound methodical expression), and application of *then_meth* F L to a goal G produces the same subgoals as applying M to G. This transformation can be neatly defined without reference to the structure of goals, and so the transformation rules are only applied to expressions which are not of the form *pair_meth* M_1 M_2 or *map_meth* M — these are handled by rules (9,10,11). We leave treatment of *cut_meth* (see Section 7) and *cond_meth* as further work.

We transform each methodical expression into an equivalent form $F \oplus L$ using a terminating, non-deterministic rewrite relation $\Rightarrow$ defined as follows:

$$M \Rightarrow M \oplus \mathit{id\_meth} \qquad \text{if } M \text{ is a singleton} \tag{16}$$

$$\mathit{then\_meth}\ M_1\ M_2 \Rightarrow M_1 \oplus M_2 \qquad \text{if } M_1 \text{ is a singleton} \tag{17}$$

$$
\begin{aligned}
then\_meth\ M_1\ M_2 &\Rightarrow F \oplus (then\_meth\ L\ M_2) && (18)\\
&\qquad \text{where } M_1 \Rightarrow F \oplus L &&\\
repeat\_meth\ M &\Rightarrow F \oplus (then\_meth\ L\ (orelse\_meth && (19)\\
&\qquad (repeat\_meth\ M)\ id\_meth)) \text{ where } M \Rightarrow F \oplus L &&\\
orelse\_meth\ M_1\ M_2 &\Rightarrow F \oplus L \qquad \text{where } M_1 \Rightarrow F \oplus L && (20)\\
orelse\_meth\ M_1\ M_2 &\Rightarrow F \oplus L \qquad \text{where } M_2 \Rightarrow F \oplus L && (21)\\
try\_meth\ M &\Rightarrow F \oplus L \qquad \text{where } M \Rightarrow F \oplus L && (22)\\
try\_meth\ M &\Rightarrow id\_meth \oplus id\_meth && (23)
\end{aligned}
$$

The state of a partial proof is represented in a proof tree whose nodes are labelled by triples $\langle G, M, X\rangle$ of a subgoal G, the atomic method M applied at the node to generate its children (if it is not a leaf node), and the methodical expression X to apply to G in order to generate its subgoals. Planning proceeds at a node $\langle G, M, X\rangle$ by transforming X into the form $F \oplus L$, applying the atomic method F and extending the proof tree with the new node(s), labelling the new node(s) with the continuation L. We can then pause for interaction with a user or with proof critics. Proof planning can then be continued at some arbitrary (user-selected) node by applying the continuation associated with that node.

6 Equivalence of the Two Planners

We now prove that a planner which applies the transformation rules to transform each methodical into the form $F \oplus L$ before applying the meta-interpreter of section 3 is equivalent to a planner which uses only the meta-interpreter.

First some preliminaries. Two methodical expressions are equivalent if they have the same effect on goals when the standard meta-interpreter is applied:

$$M_1 \equiv M_2 \leftrightarrow (\forall G, G' \,.\, M_1 : G \mapsto G' \leftrightarrow M_2 : G \mapsto G')$$

We need two important properties of $then\_meth$:

$$
\begin{aligned}
then\_meth\ (then\_meth\ M_1\ M_2)\ M_3 &\equiv then\_meth\ M_1\ (then\_meth\ M_2\ M_3) && (24)\\
F \oplus L &\equiv then\_meth\ F\ L \quad \text{if } F \text{ is atomic} && (25)
\end{aligned}
$$

The first of these follows easily from the definition of $then\_meth$ in section 3 as long as the derivations of M_1, M_2 and M_3 are independent, which is the case in the absence of side-effects. The second is definitional.

Given a methodical expression M define the *size* of M, $|M|$ to be the number of (occurrences of) methodicals contained in M. If M is atomic then $|M| = 0$.

We first prove that every methodical expression can be transformed into the required form $F \oplus L$, and then that transformation does not change the interpretation of the methodical expression according to the rule of Section 3.

Theorem 1. *Every methodical expression can be transformed*

$$\forall M . \exists F, L . M \Rightarrow F \oplus L$$

Proof. The proof is by induction on the structure of M. We give the proof for the case where *M = repeat_meth M'*. The other cases are similar.

Base case: M atomic. Then by (16), $M \Rightarrow M \oplus id\_meth$.

Step case: by the induction hypothesis, assume $\exists F', L' . M' \Rightarrow F' \oplus L'$.

Then *repeat_meth M'*$\Rightarrow$ $F' \oplus (then\_meth\ L'\ (orelse\_meth\ (repeat\_meth\ M)\ id\_meth))$ by (19), i.e. we have produced witnesses for F and L as required.

Theorem 2. *Transformation does not alter the interpretation of methodical expressions.*

$$\forall M, F, L \forall G, G' . M \Rightarrow F \oplus L \rightarrow [M : G \mapsto G' \leftrightarrow F \oplus L : G \mapsto G']$$

Proof. Again, the proof is by induction on the size of M.

Base case: If M is atomic then there is nothing to prove, since clearly $M \Rightarrow M \oplus id\_meth$ and $M : G \mapsto G' \leftrightarrow M \oplus id\_meth : G \mapsto G'$.

Step case: Assume the result for all methodical expressions of size $k < n$. We now prove the result for methodical expressions of size n.

We here prove the result for *M=repeat_meth M'* and *M=orelse_meth* M_1 M_2. The proof is similar for the other cases.

- ***M=repeat_meth M'***:
  Assume *repeat_meth* $M' \Rightarrow\ F \oplus L$ (by Theorem 1, such a decomposition exists). From rule (19) of the definition of $\Rightarrow$, $M' \Rightarrow F \oplus L'$ for some L' and $L = (then\_meth\ L'\ (orelse\_meth\ (repeat\_meth\ M')\ id\_meth))$. Since $|M'| < |repeat\_meth\ M'|$, we can apply the induction hypothesis and assume (IH) that $M : G \mapsto G' \leftrightarrow F \oplus L' : G \mapsto G'$. Then

$$\begin{array}{c}
F \oplus then_meth\ L'\ (orelse_meth\ (repeat_meth\ M')\ id_meth) : G \mapsto G' \\
\leftrightarrow \text{ By (25,24)} \\
then_meth\ (then_meth\ F\ L')\ (orelse_meth\ (repeat_meth\ M')\ id_meth) : G \mapsto G' \\
\leftrightarrow \text{ By (4)} \\
then_meth\ F\ L' : G \mapsto H\ \bigwedge\ orelse_meth\ (repeat_meth\ M')\ id_meth) : H \mapsto G' \\
\leftrightarrow \text{ By IH} \\
M' : G \mapsto H\ \bigwedge\ orelse_meth\ (repeat_meth\ M')\ id_meth) : H \mapsto G' \\
\leftrightarrow \text{ By (5,6)} \\
(M' : G \mapsto H\ \bigwedge\ repeat_meth\ M' : H \mapsto G')\ \bigvee\ M' : G \mapsto G' \\
\leftrightarrow \text{ By (3,2)} \\
repeat_meth\ M' : G \mapsto G'
\end{array}$$

- ***M=orelse_meth*** M_1 M_2:
  Suppose that $M_1 \Rightarrow F_1 \oplus L_1$ and $M_2 \Rightarrow F_2 \oplus L_2$ (again, Theorem 1 guarantees that such a decomposition exists). Then by the induction hypothesis (IH): $M_i : G \mapsto G' \leftrightarrow F_i \oplus L_i : G \mapsto G'$.
  By (20, 21), $orelse\_meth\ M_1\ M_2 \Rightarrow F \oplus L \leftrightarrow F = F_1 \wedge L = L_1 \bigvee F = F_2 \wedge L = L_2$. Consider the case where $F = F_i$ and $L = L_i$:

$$\begin{array}{cl}
F_i \oplus L_i : G \mapsto G' & \\
\leftrightarrow & \text{By IH} \\
M_i : G \mapsto G' & \\
\leftrightarrow & \text{By (5 if } i = 1 \text{ or 6 if } i = 2) \\
orelse_meth\ M_1\ M_2 : G \mapsto G' &
\end{array}$$

7 Developing the Notion of Continuation

Continuations have been used in computer science for, amongst other things, providing a semantics for programming languages. We now outline how previous work described in [4] on a continuation semantics for Prolog can be exploited to both describe and extend the implementation of our planner.

[4] defines a language $\mathcal{B}$ which describes the control element of Prolog, treating conjunction, disjunction, predicate definition and cut. There are no terms or unification on terms; instead, predicates can perform primitive actions. Execution of a program induces a certain sequence of primitive actions. Certain features of $\mathcal{B}$ and their treatment with continuations can be usefully adapted to extend our meta-interpreter:

– Disjunction is handled in [4] by the concept of a *frame*. Execution of a disjunction introduces two frames, each of which models one of the possible subsequent execution paths. If execution fails in one of these frames then it is deleted and execution switches to the other frame.
  Similarly, in our meta-interpreter interpretation of *orelse_meth* results in the creation of two frames, each modelling one of the two possible subsequent interpretation paths. More precisely, let $\mu(G, M; E, T)$ denote that applying the methodical expression M and then applying the continuation methodical expression E. Let frames be separated by a colon, so $\mu(S : T)$ denotes the result of interpreting two alternatives, S and T. Then $\mu((\textit{orelse\_meth}\ M_1\ M_2; E) : S) = \mu((M_1; E) : (M_2; E) : S)$.
  We anticipate that explicitly modelling disjunction in this way will be useful for proof critics (§8). Clearly it also introduces space inefficiency, since it results in copying the continuation E. This can be addressed by distinguishing two different forms of *orelse_meth*: one which is interpreted explicitly using frames, and a second which is interpreted implicitly using λPrologbacktracking.
– In [4], when a cut is encountered in the body of a procedure call, the frames which represent the alternatives which will be employed if the current frame fails are replaced by the frames which represented the alternatives when the procedure call was initiated.
  Analogously, whenever the body M of a methodical expression *cut_meth M* has been successfully concluded, the frames immediately subsequent to the cut should be replaced by those immediately preceding it. More precisely, $\mu((\textit{cut\_meth}\ M; E) : S) = \mu((M; !S; E) : S)$ where $\mu((!S; E) : T) = \mu(E) : S$.

8 Applications

The definition of proof strategies using methodical expressions, and the continuation - based proof planner described in this paper have been implemented in λ*Clam*. The planner provides a global view of the state of a partial proof which supports a prototype implementation of proof critics [6]. Describing the critics mechanism is outside the scope of this paper.

A user interface has been developed for λ*Clam*, based on the XBarnacle [8] user interface for *Clam*. The combined system is called xλ*Clam*. The user interface itself is written in Tcl/Tk. It runs as a Unix process, and it communicates with λ*Clam*, which is running in a separate process, via a socket interface. The Tcl/Tk process keeps a copy of the partial proof state, while the proof plan data structure and planner described in this paper enable λ*Clam* to also keep a copy of the proof plan tree. Whenever λ*Clam* makes a change to the proof plan tree, this is communicated to the Tcl/Tk process, and conversely, instructions from the user to modify the proof plan tree are passed from the Tcl/Tk process to the λ*Clam* process. The use of continuations in the proof planner makes the planning process interruptible and places it under control of the user — the user may pause the proof, run the proof planner in non-stop mode, force the proof planner to backtrack at a node, change the current planning node, or load and edit theory files.

9 Further Work

We mention here a few of the many directions for future research:

- We have demonstrated how to define a proof planner for a persistent proof plan data structure in an entirely declarative way. How to optimise this proof planner must be the subject of future work.
- The continuation semantics outlined in Section 7 must be fleshed out. In [4], Prolog programs generate *sequences* of primitive actions, whereas we would prefer to generate *trees* of primitive actions. We should also incorporate failure continuations into our formalism.
- A continuation-based treatment of the complete methodical language, and implementation of *cut_meth* and *orelse_meth* as described in Section 7 will also be the subject of further work.
- The use of continuations suggests extension of the methodical language with constructs (such as exceptions) found in programming languages.

10 Conclusions

In this paper we have outlined how proof planning strategies can be defined using methodicals. We defined a meta-interpreter for methodical expressions which can be naturally interpreted as a proof planner. There is, however, no notion of proof state, so interpretation of methodical expressions cannot be paused.

We presented a set of transformation rules which we used to define a new meta-interpreter which is equivalent to the first (in the sense that it produces the same plans) and which does maintain a notion of proof state, and is suitable for interactive proof. This meta-interpreter has been used for some time as the basis of the interactive planner in λ*Clam*.

Finally, we showed how previous work on a continuation semantics for Prolog [4] can provide us with appropriate representations and algorithms for handling *cut_meth* and *orelse_meth*.

11 Acknowledgements

We would like to thank the following for their invaluable comments: Ian Green (who first suggested the use of the term "continuation" in this context), the anonymous referees, members of the *Dream* group, Greg Michaelson, and the participants of the CIAO 2001 workshop. This work was partially funded by EPSRC grant GR/L11724.

References

1. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.
2. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
3. Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1996.
4. Arie de Bruin and Erik P. de Vink. Continuation semantics for PROLOG with cut. In *TAPSOFT, Vol.1*, pages 178–192, 1989.
5. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
6. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.
7. D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In *Proceedings of the First International Conference on Computational Logic (CL2000)*. 2000.
8. H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag, 1997.
9. J. C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation: An International Journal*, 6:233–247, 1993.
10. J. D. C. Richardson, L. A. Dennis, J. Gow, and M. Jackson. User/programmer manual for the λ*Clam* proof planner, v.2.0.0. Technical Paper TBA, Division of Informatics, 2000.
11. J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with λClam. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.

Strategies for Interactive Proof and Program Development in Martin-Löf Type Theory

(Extended abstract)

Marcin Benke

Department of Computing Science
Chalmers University of Technology
413 96 Göteborg, Sweden

Abstract. We propose some strategies for automatic tools facilitating interactive proof and program development in the proof editor Alfa based on Martin-Löf Type Theory.

1 Introduction

1.1 Context: proof editors and type theory

Alfa [HR00] is a graphical, syntax-directed editor for the proof system Agda [Coq98], is an implementation of *s*tructured type theory (STT) [CC99], which is based on Martin-Löf's type theory [ML84]. Like its predecessors in the ALF family of proof editors [Mag94], Alfa allows one to, interactively and incrementally, define theories (axioms and inference rules), formulate theorems and construct proofs of the theorems. All steps in the proof construction are immediately checked by the system and no erroneous proofs can be constructed.

Alfa is a term-based proof editor. This means that the proof is presented and recorded as a term, rather than as a tactic expression as in tactic-based proof editors such as Coq [BBC$^+$97,pro99]. Of course this does not preclude usage of tactics.

Alternatively, you can view Alfa as a syntax-directed editor for a small purely functional programming language with a type system that provides dependent types, thus allowing specification and verification of program properties within its type system. In fact, the language is very similar to the functional language Cayenne [Aug98] by Lennart Augustsson (which in turn is based on Haskell).

As such, Alfa supports algebraic datatypes[1] , pattern matching and general recursive definitions. To preserve logical consistency, all proofs are subject to termination check as well as typechecking.

Since checking termination is undecidable, the checker approximates, keeping on the safe side, hence some definitions may be unduly rejected. It is however more liberal than for example restricting to primitive or well-founded recursion [Wah00].

[1] By an algebraic type we mean a type being a sum of products, defined by listing its constructors, eg

```
data Tree = Empty | Leaf Int | Node Tree Tree
```

1.2 Assumptions

- We consider interactive proof developments in a system based on Monomorphic Martin-Löf Type Theory.
- Proof development proceeds interactively in small steps (definitionslemmas)[2].
- User states a lemma and (optionally) hints as to which previous lemmas he thinks might be useful in proving the current one.
- System tries to prove the lemma using hints.

The task of constructing an automated tool within this framework faces us with several challenges:

Given the interactive nature of the system, response time is an important consideration. The user should expect an answer within reasonable time.

Since Type Theory is an intuitionistic, higher-order logic, applicability of "classical"[3] theorem proving methods is rather limited, even though possible for some fragments of the theory [TS98].

Every term may have many equivalent types; some of them give better clues as to possible proofs than others. Moreover the reduction relation is more complicated than in pure lambda calculus since one has to expand definitions and **case** expressions, too.

2 Strategies

Given the assumptions above, a typical situation would look like

$$\mathit{env} \vdash f\ (\boldsymbol{x} \in \boldsymbol{T}) = ? \in G$$

where *env* contains existing definitions/lemmas (which will be further referred to as *globals*), f is the current definition and $\boldsymbol{x}$ represents a lists of *parameters* along with their types. Our aim is to find a proof term of type G (goal) to replace "?" (hole). Moreover the term must be such that the definition would pass the termination check.

2.1 Directives

The main directives of our strategy are:[4]

(DG) Use of globals: since most proof developments make use of numerous library modules, this leads to a very large search space; therefore usage of globals is limited to

(DG1) globals specified in hints,
(DG2) constructors ,
(DG3) cheap globals (i.e. ones that do not increase the search space substantially)
(DG4) "local" globals (i.e. ones defined in the current module)

[2] In Martin-Löf Type Theory there is no clear distinction between definitions and theorems

[3] By "classical", we mean methods that use *tertium non datur*, for example to prove a theorem by refuting its negation.

[4] The parenthesised directive abbreviations are used as a shorthand in the example in Section 3

(DR) Recursive calls: discouraged, especially where they might lead to nontermination (we shall call this subdirective DR1) (with a notable exception of the "recursive vs recursive" strategy described in (DA) below)

(DP) Use of parameters: encouraged; solutions using **all** parameters are preferred. Notice that within the scope of **case**, parameter lists are updated accordingly.

(DA) Algebraic types: if the initial proof search fails to produce satisfactory result and one or more parameters is of an algebraic type, use pattern-matching; if a type is recursive, try "recursive vs recursive" strategy: using recursive calls on the recursive constructor arguments (this subdirective is closely related to (DR) hence we call it DR2).

(DO) Ordering candidates: during the proof search, candidates for resolving particular subgoals are ordered according to anticipated "cost" of their use. However, we propose to use type ordering rather than an integer-based ranking.

It is worth noting that the directive (DP) may have negative impact on the efficiency of our strategy, so disabling it is desirable in many cases. However, it does have some merits, one of which is illustrated in the example in the following section. It should also be noted that it has global, rather than local character (but may be also implemented in a compositional way).

3 Motivating example: *map*

As an illustration of how the directives described above work, it will be shown how they allow deducing the right-hand side of a function definition from its type and left hand side. Even though the example seems to belong more to functional programming rather than proof development, but it is short and illustrates almost all directives. Besides, one should remember that in type theory all proofs are programs and are subject to similar phenomena. The choice of example is also motivated by our interest in program verification and deriving programs from specifications.

Assume we have

$$[A] = \textbf{data}\ [\,]\ |\ : (x \in A, xs \in [A])$$

and we want the system to fill the hole in

$$\mathit{map}\ (f \in A \to B)(xs \in [A]) \in [B] = ?_0$$

pursuant to the directives stated above.

Initial proof search yields two type-correct results:

- $[\,]$ — unsatisfactory due to (DP);
- *map f xs* — rejected due to (DR1): fails termination check.[5]

[5] In fact no candidates using *map* would be even considered; recursive candidates are considered only in specific situations, such as specified in (DR2)

Since the proof search failed to produce satisfactory results and the parameter *xs* is of an algebraic type, the system proceeds to case analysis (DA), replacing $?_0$ by

$$\begin{array}{l}\textbf{case}\ xs\ \textbf{of}\\ \quad [\,]\ \rightarrow ?_1\\ \quad (x : xs')\ \rightarrow ?_2\end{array}$$

Proof search for $?_1$ yields two candidates:

- [ ]
- *map f* [ ] — not considered; we are in a "non-recursive" branch.

Proof search for $?_1$ yields five candidates (we are in a "recursive" branch, so use of recursion is allowed):

- [ ]
- *map f xs'*
- *map f* [ ]
- $(f\,x) : [\,]$
- $(f\,x) : (map\ f\ xs')$

All considered candidates pass both type- and termination-check. However the only one satisfying (DP) is

$$\begin{array}{l}\textbf{case}\ xs\ \textbf{of}\\ \quad [\,]\ \rightarrow\ [\,]\\ \quad (x : xs')\ \rightarrow\ (f\ x) : (map\ f\ xs')\end{array}$$

which is the "usual" definition of *map*!

Similar results are obtained for other functions on lists, e.g. list concatenation. Interestingly enough, when provided with a hint to use *foldr* (provided an usual definition of foldr is visible), our strategy yields an alternative definition of list *concat* using *foldr* rather than pattern matching:

$$\begin{array}{l}concat\ ::\ [[a]]\ \rightarrow\ [a]\\ concat\ xss\ =\ foldr\ (+\!\!+)\ [\,]\ xss\end{array}$$

As a side note, let us remark that our tool also succeeded to generate a similar definition of ordinary list concatenation:

$$\begin{array}{l}(+\!\!+)\ ::\ [a]\ \rightarrow\ [a]\ \rightarrow\ [a]\\ xs\ +\!\!+\ ys\ =\ foldr\ (:)\ ys\ xs\end{array}$$

But we have to admit that This is however due to a bit of luck, since (without additional restrictions) there are two solutions satisfying our criteria. To distinguish between them, one should provide a more precise specification of list concatenation.

4 Type orderings

For the purposes of ordering candidates we propose using quasi-orders on types. One interesting ordering we are currently experimenting with is based on the following system:

$$0 \leq A$$

$$A \leq 1$$

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

$$A_i \leq A_1 + A_2 \text{ for } i = 1, 2$$

$$A_1 \times A_2 \leq A_i \text{ for } i = 1, 2$$

where 0 represents empty types, 1 represents one-element types and A, B (possibly with indices) represent arbitrary types.

The ordering is intended to represent how easy (or difficult) it is to find an element of given type, with "greater" meaning "greater ranking" i.e. "easier". The intuitions behind the rules are thus:

- It is difficult to provide an element of an empty type.[6]
- It is trivial to provide an element of a unit type.
- The stronger the assumptions and the weaker the thesis, the easier it is to prove the implication. On the extreme we have *ex falso quod libet*.
- Constructing an element of a sum type amounts to constructing an element of any of its components.
- Constructing an element of a product type amounts to constructing elements of all its components.

5 Implementation

A tool based on the strategies described in this paper has been implemented as a plugin for the Alfa proof editor. The plugin (as well as Alfa itself) is written in Haskell. Initial experiences with using it seem quite promising, however a *corpus* of test cases (see below) is needed for a performance evaluation.

6 Future work

- Basing on user feedback build a *corpus* of test cases and test the strategies more extensively.
- Compare relative merits of various type orderings against each other and against integer scores.
- Improve efficiency of the implementation.

[6] In most cases it is just impossible, but we must relativize with regard to context.

– Extend the range of tactics used; better tactics for equational reasoning seem to be a priority.

Equality proofs are important for verification of functional programs. Unfortunately, current type theory style equational proofs are tedious and not easily readable. We plan to develop methods and tools that would improve this situation, basing partially on preliminary results on deriving equality for algebraic types [Ben00].

References

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Mu noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.

[Ben00] Marcin Benke. Automatic deriving of properties of algebraic datatypes in Martin-Löf type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Durham. Submitted for publication, December 2000.

[CC99] C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, Sep 1999.

[Coq98] Catharina Coquand. The AGDA Proof System Homepage. `http://www.cs.chalmers.se/~catarina/agda/`, 1998.

[HR00] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer, 2000.

[Mag94] L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1994.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[pro99] The LogiCal project. Coq Homepage. `http://coq.inria.fr/`, 1999.

[TS98] Tannel Tammet and Jan Smith. Optimized encodings of fragments of type theory in first-order logic. *JLC: Journal of Logic and Computation*, 8, 1998.

[Wah00] David Wahlstedt. Detecting termination using size-change in parameter values. Masters Thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 2000.

Proof Planning for Strategy Development

(Position paper)

Andrew Ireland

Department of Computing & Electrical Engineering
Heriot-Watt University
Edinburgh, Scotland
a.ireland@hw.ac.uk

Proof planning extends the tactic-based theorem proving paradigm through the explicit representation of proof strategies. We see three key benefits to the proof planning approach to the development of proof strategies: flexibility, re-usability and synergy. In particular we have demonstrated these benefits through our work on reasoning about imperative programs where we reuse strategies developed previously for proof by mathematical induction [IS01]. We are now about to embark upon an industrial collaboration in order to extend and further evaluate our strategies. The goal is to investigate[1] techniques for automating the formal verification of software for critical systems. In particular we will focus upon SPARK [Bar97], a subset of the Ada programming language developed by Praxis Critical Systems[2]. SPARK has a proven track record in the production of software for critical systems within the finance, aerospace, rail and telecommunications markets. The time is now ripe to test the strategic advantage of proof planning on "industrial strength" problems. SPARK and our collaboration with Praxis Critical Systems provides an ideal opportunity for achieving this goal.

References

[Bar97] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison-Wesley, 1997.

[IS01] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001.

[1] The research proposal outlined here has been funded under the EPSRC Critical Systems Programme – EPSRC grant GR/R24081.

[2] http://www.praxis-cs.co.uk

Strategies in Automated Theorem Proving

(Position Paper)

David Crocker

Escher Technologies Ltd
E-mail: dcrocker@eschertech.com

1 Background

Escher Technologies Ltd. (ETL) is a company specialising in tools for developing validated software. Established in 1995, the company has been working on a development system that combines rigorous deduction-based validation techniques with a modern object-oriented approach to building software, while minimising the need for mathematical prowess on the part of the developer. The product was released for educational use and commercial evaluation in March 2001 under the name Perfect Developer. Full commercial release is planned for September 2001. More information can be found at `www.eschertech.com`.

2 Theorem proving in the Perfect Developer tool

When performing validation, Perfect Developer first generates proof obligations from the specifications and code, then attempts to discharge them using a term rewriting system and a predominantly first-order prover based on a Rasiowa-Sikorski deduction system. When using automated deduction to prove properties of software systems, there are a number of issues that we believe assume greater importance than they do for provers not specifically targeted at software validation:

- Lots of potentially redundant information is available, corresponding to knowledge of the program state that may not be relevant to the current proof obligation.
- Occasionally, the assumptions in a proof obligation may be incompatible (e.g. when code is unreachable). In this case, rather than just flagging the obligation as discharged, we wish to warn the user. This means we have to track the assumptions of a theorem separately from the consequences we are trying to prove (or else be able to determine after finding a proof whether the consequences were relevant).
- Function symbols are of two kinds: those that must remain uninterpreted (corresponding to dynamic binding in the program) and those that can be expanded. Expanding a function symbol is often necessary to find a proof, on the other hand unnecessary expansion of function symbols may substantially increase the complexity of a proof.
- The underlying logic is a logic of partial functions so that expressions have the same meaning whether in the context of a specification or a program statement. For efficiency, the prover works with classical two-valued logic most of the time, but

the possibility of introducing terms that are ill-formed in two-valued logic must be avoided. The Perfect Developer tool (including the prover) is implemented in its own technology and gives rise to about 125,000 proof obligations.

3 Our interest in proof strategies

The current prover uses a simple heuristic based on generating new terms and favouring those of lowest complexity. A partial lookahead is implemented so that the system has some idea of the likely effect of taking each possible prover step on the complexity of the resulting terms; however this lookahead cannot take account of the potential for a subsequent unification to drastically reduce complexity. We are seeking to make improvements in the following areas:

- Better heuristics to decide when to expand function symbols, or to direct function expansion from a strategy.
- The identification and implementation of strategies that are effective in our context.
- Inductive strategies.
- Automatically using experience either to choose what proof step to take at each point or at a strategic level.