Proceedings of the CADE-14 Workshop

on

Strategies in Automated Deduction

Sunday, July 13, 1997
Townsville, Australia

Bernhard Gramlich and Hélène Kirchner (eds.)

Preface

The concept of strategy allows describing and guiding computations and deductions in automated theorem provers, proof checkers, logical frameworks. Strategies are used for various purposes, e.g. for
- proof search in theorem proving,
- combining different proof techniques and computation paradigms,
- program transformation,
- developing heuristics for playing games and finding proofs.

Rules are not sufficient to capture every proof development. A mechanism is needed for instance to formalise the search for different solutions, the check of context conditions, the request for user input to instantiate variables, the processing of subgoals in a particular order. Strategies are used to guide the application of rules, but may also involve iteration, case analysis, deterministic and non-deterministic choices. One may want to program strategies, to transform them, to prove certain properties on the computations or the proofs that they describe.

The workshop aims at gathering different experiences on the use of strategies, under various terminology (tactic, tactical, method, heuristic, proof planning...), and in various application domains (first-order / higher-order / inductive theorem proving, program transformation, operational semantics...). Based on these experiments, several points are addressed:

- Strategy languages:
  Which basic constructs are needed? Which are relevant choices: expressiveness versus efficiency, meta-language versus reflexivity, higher-order versus first-order syntax?
- Strategy analysis:
  Which properties are required for strategies? How to automate as much as possible the design of strategies in some logical fragments? How to evaluate applicability and performance?
- Strategies in theorem proving:
  Which kind of architecture, modular design is suitable for such systems? How to deal with user-interaction, input-output? How to reuse past proof experience for search-guiding heuristics?

The contributions included in these proceedings deal with several aspects of these topics. Papers have been available before the workshop on the web, to encourage reactions and discussions at the workshop.

We thank the CADE-14 Program Committee and the Local Arrangements Committee for giving us the opportunity to hold this workshop. We also thank the participants for presenting and discussing their experiences and points of view on strategies in automated deduction.

Bernhard Gramlich and Hélène Kirchner

Programme

Sunday, July 13

12.00am - 12.10pm	OPENING
	SESSION 1: Strategy Languages for Computational Systems
12.10pm - 12.40pm	**Internal Strategies in a Reflective Logic** Manuel Clavel and José Meseguer
12.40pm - 1.10pm	**Strategies and Rewriting in ELAN** Peter Borovanský, Claude Kirchner and Hélène Kirchner
1.10pm - 1.40pm	**Tacticals in ILF - and Why They Are Superfluous** Bernd Ingo Dahn
1.40pm - 2.10pm	BREAK
	SESSION 2: Strategy Analysis
2.10pm - 2.40pm	**Strategies for Logic Programming Languages** Tatjana Lutovac and James Harland
2.40pm - 3.10pm	**Machine-Independent Evaluation of Theorem-Proving Strategies** Maria Paola Bonacina
3.10pm - 3.40pm	BREAK
	SESSION 3: Strategies in Theorem Proving: Applications
3.40pm - 4.10pm	**Flexible Re-enactment of Proofs** Matthias Fuchs
4.10pm - 4.40pm	**Equalizing Terms by Difference Reduction Techniques** Dieter Hutter
4.40pm - 5.30pm	DISCUSSION
5.30pm	END OF WORKSHOP

Table of Contents

Internal Strategies in a Reflective Logic*

Manuel Clavel and José Meseguer

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

Abstract. After introducing the basic notions of reflective logic and internal strategies, we discuss in detail how reflection can be systematically exploited to design a strategy language internal to a reflective logic in the concrete case of rewriting logic and Maude; and we illustrate the advantages of this new approach to strategies by showing how the rules of inference for Knuth-Bendix completion can be given strategies corresponding to completion procedures in a completely modular way, not requiring any change whatsoever to the inference rules themselves.

1 Introduction

This paper is part of a long-term effort to study and axiomatize reflective logics and to systematically exploit reflection to bring deduction strategies within the fold of logic. In this way, more powerful, modular, and flexible notions of deduction strategy can be defined and verified. In earlier work [7, 8] we have proposed general axiomatic notions of reflective logic and internal strategy language, and have begun to show how they can be realized in a concrete language based on reflective rewriting logic like Maude [6]. Here, we discuss in greater detail how reflection can be systematically exploited to design a strategy language internal to a reflective logic in the case of rewriting logic and Maude; and we illustrate the advantages of this new approach to strategies by showing how the rules of inference for Knuth-Bendix completion can be given strategies corresponding to completion procedures in a completely modular way, not requiring any change whatsoever to the inference rules themselves.

A reflective approach to strategies seems entirely natural, and, as explained later, has also been adopted by other authors in the context of higher-order logics. Since strategies naturally generalize proofs—in the sense that they yield sets of proofs as their results—in a reflective logic they can be *reified*, that is, they can be brought down to the object level the same way that other metalevel entities such as proofs and theories can. Furthermore, their semantics can be *internalized* in the logic in question, that is, they can be given a declarative semantics within the same logic that they control, in such a way that their execution becomes logical deduction. It then becomes possible to reason formally about such internal

* Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9633363, and by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA).

strategies using all the formal methods available in the logic. Internal strategies replace the traditional "Logic + Control" treatment of strategies as extralogical constructs by the inclusion "Control $\subseteq$ Logic" of strategies as logical entities.

1.1 Reflective Logics and Internal Strategies in a Nutshell

Given a logic, we say that it is *reflective* relative to a class $\mathcal{C}$ of theories—where typically the class $\mathcal{C}$ corresponds to theories that are describable using a finite piece of language, such as finitely presented or recursively enumerable theories—if we can find inside $\mathcal{C}$ a *universal theory* $\mathcal{U}$ that can simulate all other theories in the class, in the sense that—denoting by $sen(T)$ the set of meaningful sentences in the language of a theory T—there is a function, called a *representation function*, $\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(\mathcal{U})$ such that for each $T \in \mathcal{C}$ and $\varphi \in sen(T)$, $T \vdash \varphi \iff \mathcal{U} \vdash \overline{T \vdash \varphi}$. Since $\mathcal{U}$ itself is representable, representation can be iterated, so that we immediately have a 'reflective tower' $T \vdash \varphi \iff \mathcal{U} \vdash \overline{T \vdash \varphi} \iff \mathcal{U} \vdash \overline{\mathcal{U} \vdash \overline{T \vdash \varphi}} \ \ldots$ This definition of reflective logic assumes general axiomatic notions of logic and entailment such as those provided by the theory of general logics [18]. This is the approach taken in [7, 8] to make the above definition fully precise.

Given a logical theory T in a logic, a *strategy* is a computational way of looking for certain proofs of some theorems of T (see [8] for a general axiomatization of deductive strategies). This may be done by having a strategy language $S(T)$ associated with T in which strategies controlling deduction in T can be defined. For example, in theorem provers in the LCF tradition, there is a metalanguage, namely ML, in which "tactics" and "tacticals" can be defined for exactly this purpose [23]. Even if quite flexible strategies can thus be defined, such a language may remain a programming language external to the logic that it is controlling. In this situation, control becomes an extralogical feature of the system.

If strategies can be defined *inside* the logic that they control, we are in a much better situation, since formal reasoning within the system can be applied to the strategies themselves. Basically, an *internal strategy language* is a theory-transforming function S that sends each theory T to another theory $S(T)$ in the same logic whose deductions simulate controlled deductions of T. Reflective logics offer good opportunities for defining internal strategy languages of this kind, because the metalevel being controlled can be expressed within the logic. In [8] we gave a general method for defining and proving correct an internal strategy language, and showed how the correctness of a universal theory greatly simplifies the proof of correctness of a given strategy language. We have used the Maude rewriting logic language [6] as an experimental vehicle for the exploration of these techniques. Similar ideas about specifying internal strategies are also being adopted for the strategies of the ELAN rewriting logic language [4], whose strategy language was initially defined as an external language. However, in ELAN reflection and strategies have not yet been fully integrated. In the context of typed lambda calculi, the important advantages of having an internal strategy language have also been stressed by several authors [1, 9, 25].

1.2 Rewriting Logic and Maude

A rewrite theory T consists of a signature Σ of operators, a set E of equations, and a set of labeled rewrite rules. The deductions of T are rewrites modulo E using such rules. Such rewriting deductions are formalized as proofs using four simple inference rules [19].

Rewriting logic is a reflective logic[2] in the sense we have just explained [8]. Since the sentences meaningful in the language of a rewrite theory T are rewrite sequents $t \longrightarrow t'$ where t, t' are Σ-terms, the general notion of reflection takes in this case the following form. The class $\mathcal{C}$ is that of finitely presentable rewrite theories—that is, theories whose signature, and sets of equations and rules are all finite. There is then a finitely presentable theory $\mathcal{U}$ and a representation function $\overline{(\_ \vdash \_)}$ which encodes a pair consisting of a rewrite theory T in $\mathcal{C}$ and a sentence $t \longrightarrow t'$ in T as a sentence $\langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$ in $\mathcal{U}$, in such a way that $T \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$, where the function $\overline{(\_)}$ recursively defines the representation of rules, terms, etc. as terms in $\mathcal{U}$.

Maude is a logical language based on rewriting logic [19, 22, 20]. An introduction to Maude and its interpreter implementation can be found in [6]. For our present purposes the key point is that the Maude implementation supports an arbitrary number of levels of reflection and gives the user access to important reflective capabilities, including the possibility of defining and using internal strategy languages. In [8, 6] we explained how such languages can be defined, and introduced a simple but interesting internal strategy language. Here we define another such strategy language, and use it to show how reflection can give a fully modular and declarative executable specification of completion procedures.

1.3 Reflection and Internal Strategies for Completion

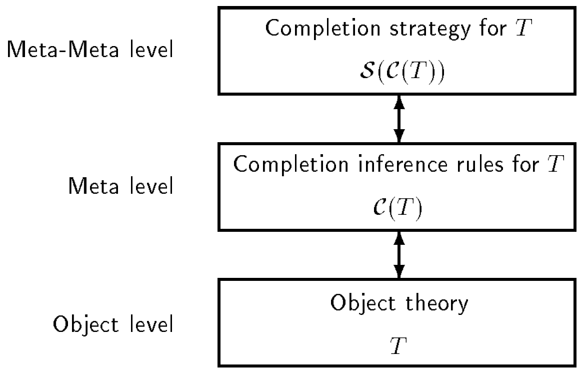
As it is well known, the Knuth-Bendix completion algorithm [14] attempts to construct a convergent rewrite system from a finite axiomatization of an equational theory. An abstract view of completion as an inference system $\mathcal{C}$ was proposed by Bachmair, Dershowitz and Hsiang [2]. In this view, a (*standard*) *completion procedure* is any program that takes a set of equations and rules, $(E_0 \cup R_0)$, and a reduction ordering $\succ$ containing R_0, and uses some strategy to apply the inference rules of $\mathcal{C}$ to generate a derivation from $(E_0 \cup R_0)$. A completion procedure is called correct if all non-failing derivations result in a convergent system.

By proving correctness of this more abstract notion of completion, Bachmair, Dershowitz and Hsiang were able to guarantee the correctness of a wide class of different completion procedures. A key element of their methodology is to cleanly separate the logical, or proof-theoretical, component of the completion method from issues pertaining to strategy and control. This is precisely what reflection and a convenient internal strategy language allow us to accomplish.

[2] The proof in [8] treats the case of unsorted and unconditional rewrite theories; the extension to sorted and conditional theories is unproblematic and will appear elsewhere.

There have been several attempts to use this abstract view of completion in actual rewrite-rule implementations of completion procedures [15, 13]. The challenge has been to control the rewriting inference process generated by the rules in $\mathcal{C}$—which in principle could go in many undesired directions—by means of adequate *strategies*. Unfortunately, these implementations have not been able to preserve the separation between the logical and the control components of the completion method. For example, control of the rewriting process is deeply based on the data structure on which the inference rules operate. As a consequence, improvements in the control of the rewriting process required changes in the original data structure and consequently in the inference system $\mathcal{C}$ as shown in [15]. In particular, proofs of correctness based on the original inference system need to be redone for the new inference rules combining inference and control.

We believe that the difficulties encountered in previous approaches are quite intrinsic and probably cannot be overcome without the use of new techniques such as reflection. Essentially, the possibility of allowing several levels of reflection is the key to a fully modular approach in which no change whatsoever is required in the original inference rules. This clean separation by levels is illustrated in the figure below. The theory T that we want to complete appears at the object level. The inference rules $\mathcal{C}$ —parameterized by T— are clearly metalevel rules defining a theory one level up. The key point is that the strategies controlling the application of such inference rules and therefore yielding different completion procedures appear as terms in the theory $\mathcal{S}(\mathcal{C}(T))$, corresponding to instantiating an internal strategy language $\mathcal{S}$ with the inference rules $\mathcal{C}$ as its object theory. Since $\mathcal{S}$ transforms any theory into a metalevel theory, and $\mathcal{C}$ is already a metalevel theory, $\mathcal{S}(\mathcal{C}(T))$ is a theory in the meta-metalevel. Keeping these three levels of reflection separate is the key to maintaining modularity. Of course, in other applications, involving for example meta-strategies that control the application of lower-level strategies, more than three levels of reflection may in fact be required.



2 A Reflective Kernel in Maude

It is possible to define many internal strategy languages in Maude. However, the implementation and the proof of correctness of such languages should rely on the notion of a basic *reflective kernel*, that is, on some basic functionality provided by the universal theory $\mathcal{U}$. The idea proposed in [8] is to first define a *strategy language kernel* as a function *Meta-Level* of rewrite theories, that sends T to a definitional extension of $\mathcal{U}$—or a suitable subtheory of $\mathcal{U}$— that defines how rewriting in T is accomplished at the metalevel. A typical semantic definition that one wants to have in *Meta-Level*(T) is that of *meta-apply*$(\bar{l}, \bar{t})$, that simulates at the metalevel one step of rewriting at the top of a term t using the rule labeled l in T. Proving the correctness of such a small strategy language kernel is then easily done, by using the correctness of $\mathcal{U}$ itself as a universal theory. The next step is to define a strategy language of choice, say *Strategy*, as a function sending each theory T to a theory that extends *Meta-Level*(T) by additional strategy expressions and corresponding semantic rules, all of which are recursive definitional extensions of those in the kernel in an appropriate sense, so that their correctness can then be reduced to that of the kernel.

The Maude implementation supports this novel style of (meta)programming of strategies through the module META-LEVEL below. Formally, META-LEVEL is a module-transforming operation which maps a module T to another module META-LEVEL[T] that is a definitional extension of $\mathcal{U}$. In particular, the following operations, that specify how rewriting in T is accomplished at the metalevel and whose correctness is based on the correctness of $\mathcal{U}$, are defined:

1. meta-reduce takes the meta-representation t of a term as its only argument and is evaluated as follows: (a) t is converted to the term it represents; (b) this term is fully reduced using the equations E in T; (c) the resulting term is converted to a meta-term which is returned.
2. meta-apply2 takes the meta-representation t of a term, a rule label l, a natural number n in Peano representation, and a substitution σ, and is evaluated as follows: (a) t is converted to the term it represents; (b) this term is fully reduced using the equations E in T; (c) the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded; (d) the first n successful matches are discarded; (e) if there is an $(n + 1)$th match, its rule is applied using that match; otherwise {error*, empty} is returned; (f) the new term is fully reduced using the equations E in T; (g) the resulting term is converted to a meta-term which is returned paired with the match used in the reduction. The operator {_,_} is used to construct this pair.
3. meta-apply takes the meta-representation t of a term, a rule label l, and a natural number n in Peano representation, and the result is the first component of the pair returned by the evaluation of meta-apply2 with t, l, n, and the empty substitution as its arguments.

Therefore, META-LEVEL can be considered in our terminology as a kernel internal strategy language for rewriting logic. Its semantic equations extend those of $\mathcal{U}$ and are omitted. The operations ext-term and ext-subs below extract the first and second component respectively from a pair constructed with {_,_}. The actual signature of the module META-LEVEL is the following:

```
mod META-LEVEL[T :: MODULE] is
sorts Op Var Term TermList Label Nat Assignment Substitution ResultPair .
subsort Op < Term . subsort Var < Term . subsort Term < TermList .
subsort Assignment < Substitution .
op 0 : → Nat . op suc_ : Nat → Nat .
op _[_] : Op TermList → Term . op _,_ : TermList TermList → TermList .
op error* : → Term .
op _←_ : Var Term → Assignment . op empty : → Substitution .
op _;_ : Substitution Substitution → Substitution [assoc comm] .
op {_,_} : Term Substitution → ResultPair .
op ext-term : ResultPair → Term . op ext-subs : ResultPair → Substitution .
op meta-reduce : Term → Term . op meta-apply : Term Label Nat → Term .
op meta-apply2 : Term Label Nat Substitution → ResultPair . endm
```

3 Internal Strategies in Maude

In many cases we need to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate strategies. Many systems, for example theorem provers and declarative languages implementations, support certain strategies of this nature. However, such strategies are often *external* to the language they control: they may constitute a separate programming language external to the logic, or may be part of the language's "extralogical features". Since rewriting logic is reflective, strategies can be made *internal* to the logic. That is, they can be defined by rewrite rules, and can be reasoned about as with rules in any other theory.

We illustrate this idea by partially specifying a basic internal strategy language in Maude as a module-transforming operation STRATEGY which maps a module T to another module STRATEGY[T] that extends the strategy kernel META-APPLY, defined in Section 2.

We introduce the different function symbols and we give informal explanations when needed. Due to lack of space the formal semantics of some of the functions symbols is omitted; it will appear elsewhere (see also [7] for a complete specification of an internal strategy language similar to the one presented here). Notice that, as an important consequence of the methodology we are following, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. Also, even if some users decide to adopt a particular strategy language because of its good features, such a language remains fully *extensible*, so that new features and new strategies can be defined on top of them.

As expected, the module STRATEGY is parameterized by the theory MODULE and imports the metalevel of its parameter, that is, META-LEVEL[T] for the theory T whose rewriting inference process we want to control. A strategy expression in STRATEGY has the form rew $\langle\ t \parallel w\ \rangle$ with S, where t stands for the metarepresentation of a term in T, S is the rewriting strategy we wish to compute, and w records information relevant for S. Since this information typically has to do with terms in T that somehow are important in the computation of S, w stands for the metarepresentation of a term constructed in T, which may involve some convenient data structures defined by the user. More generally, w can be a term of a sort *Record* containing the sort *Term* of metarepresentations of terms as a subsort. As we will see, as the computation of a given strategy expression proceeds, t gets rewritten by controlled aplication of rules in T, w may be updated, and S is rewritten into the remaining strategy to be computed. In case of termination, this is the idle strategy and we are done. The strategy expression failure is returned when a requested strategy cannot be carried out.

mod STRATEGY [T :: MODULE] **is**
protecting META-LEVEL[T] **.**
sorts *Record Condition Operation MetaVar Binding Strategy StrategyExpression* **.**
subsort *Term* $<$ *Record* **.**
op rew $\langle$_||_$\rangle$ with_ **:** *Term Record Strategy* $\longrightarrow$ *StrategyExpression* **.**
op failure **:** $\longrightarrow$ *StrategyExpression* **. op** idle **:** $\longrightarrow$ *Strategy* **.**

The most basic strategy we can express in our language is to apply a rule l to a term t. A strategy of the form apply [l] will rewrite t with l using the built-in function meta-apply with the first successful match, if any; otherwise, it will return failure. We can also apply a rule l partially instantiated with a substitution σ, using a strategy of the form apply [l] with σ. In this case, the built-in function meta-apply2 is called.

op apply [_] **:** *Label* $\longrightarrow$ *Strategy* **.**
op apply [_] with _ **:** *Label Substitution* $\longrightarrow$ *Strategy* **.**
var t **:** *Term* **. var** l **:** *Label* **. var** σ **:** *Substitution.* **var** w **:** *Record* **.**
vars S S' **:** *Strategy* **.**
eq rew $\langle$ t || w $\rangle$ with apply [l] = **if** meta-apply(t, l, 0) == error*
  then failure **else** rew $\langle$ meta-apply(t, l, 0) || w $\rangle$ with idle **fi .**
eq rew $\langle$ t || w $\rangle$ with apply [l] with σ = **if** meta-apply2(t, l, 0, σ) == {error*, empty}
  then failure **else** rew $\langle$ ext-term(meta-apply2(t, l, 0, σ)) || w $\rangle$ with idle **fi .**

A strategy of the form apply [l, n] if c will rewrite a term t with the first successful match after the n^{th} one, if any, that also satisfies a user-defined condition c over $\langle$ t || w $\rangle$; otherwise, it will return failure. We also define apply [l, n] with σ if c, with its expected semantics. The operation instc instantiates a condition with a given substitution. The semantics of eval c in $\langle$ t || w $\rangle$ will typically be given by the user as an extension of STRATEGY.

op apply [_,_] if _ **:** *Label Nat Condition* $\longrightarrow$ *Strategy* **.**
op apply [_,_] with _ if _ **:** *Label Nat Substitution Condition* $\longrightarrow$ *Strategy* **.**
op instc **:** *Condition Substitution* $\longrightarrow$ *Condition* **.**
op eval _ in $\langle$_||_$\rangle$ **:** *Condition Term Record* $\longrightarrow$ *Term* **.**
var c **:** *Condition* **. var** n **:** *Nat* **.**
eq rew $\langle$ t || w $\rangle$ with apply [l, n] if c =
  if meta-apply2(t, l, n, empty) == {error*, empty} **then** failure
  else if eval instc(c, ext-subs(meta-apply2(t, l, n, empty))) in $\langle$ t || w $\rangle$ == 'true
    then rew $\langle$ ext-term(meta-apply2(t, l, n, empty)) || w $\rangle$ with idle
    else rew $\langle$ t || w $\rangle$ with apply [l, suc(n)] if c **fi fi .**

The control strategies declared below should be fairly easy to understand by now. Notice that given a sequence of strategies, rew-aux forces a step-by-step computation of the sequence, in the sense that a strategy will not be considered until the previous one has not been fully computed. We use this fact to define the strategy exitloop, which causes the innermost enclosing loop to be exited.

```
op _;_ : Strategy Strategy → Strategy .
op or {_} else {_} : Strategy Strategy → Strategy .
op if _ then {_} else {_} : Condition Strategy Strategy → Strategy .
op iterate {_} : Strategy → Strategy . op while _ {_} : Condition Strategy → Strategy .
op loop {_} : Strategy → Strategy . op exitloop : → Strategy.
op rew-aux _ with_ : StrategyExpression Strategy → StrategyExpression .
eq rew ⟨ t || w ⟩ with S ; S' = if rew ⟨ t || w ⟩ with S == failure
   then failure else rew-aux (rew ⟨ t || w⟩ with S) with S' fi . [...]
eq rew ⟨ t || w ⟩ with loop { S } = rew-aux (rew ⟨ t || w ⟩ with S) loop { S } .
eq rew-aux (rew ⟨ t || w ⟩ with idle) with S = rew ⟨ t || w ⟩ with S .
eq rew-aux (rew ⟨ t || w ⟩ with exitloop) with S ; S' = rew ⟨ t || w ⟩ with exitloop . [...]
eq rew-aux (rew ⟨ t || w ⟩ with exitloop) with loop { S } = ⟨ t || w ⟩ .
```

A strategy of the form rew $\langle\ t \parallel w\ \rangle$ with let (b) $\{\ S\ \}$ may be used to bind metavariables to certain values inside a strategy S. A metavariable is a constant in the metalevel that can be instantiated to the metarepresentation of any term. A strategy of the form rew $\langle\ t \parallel w\ \rangle$ with apply $[l]$ andlet (b) $\{\ S\ \}$ can be used to apply a rule l to a term t and to bind metavariables inside a strategy S to values that depend on the way in which l is applied. In a similar way, we define apply $[l]$ with σ andlet (b) $\{\ S\ \}$, apply $[l,\ 0]$ if c andlet (b) $\{\ S\ \}$, and apply $[l,\ 0]$ with σ if c andlet (b) $\{\ S\ \}$, with their obvious semantics. Finally, the task of updating the relevant information for strategies to come is accomplished with strategies of the form set o, where o is an operation on $\langle\ t \parallel w\ \rangle$ that, in the same form that a condition, is defined using eval.

```
subsort MetaVar < Term . subsort Operation < Term .
op _↪_ : Term Term → Binding . op _,_ : Binding Binding → Binding [assoc] .
op let (_) {_} : Binding Strategy → Strategy . op bind : Strategy Binding → Strategy .
op set _ : Operation → Strategy . [...]
var b : Binding . var o : Operation .
eq rew ⟨ t || w ⟩ with let (b) { S } = rew ⟨ t || w ⟩ with bind(S, b) . [...]
eq rew ⟨ t || w ⟩ with set o = rew ⟨ t || eval o in ⟨ t || w ⟩ ⟩ . endm
```

4 Completion Procedures as Internal Strategies

To illustrate our ideas and the semantics of the strategies introduced above, we give a full specification of Huet's algorithm for Knuth-Bendix completion [11] as a strategy in an extension of STRATEGY. The system module COMPLETION below specifies in rewriting logic the completion inference rules KB presented in [10] which are expressed as rewrite rules. Completion of an equational theory is a method that naturally takes place at the metalevel of the theory. Consequently, COMPLETION is parameterized by MODULE and imports the metalevel of the theory we want to complete. As expected, COMPLETION is also parameterized by the theory OTERM that introduces an operation $\succ$ as a well-founded partial (irreflexive) ordering on metarepresentations of terms, that is, there exists no infinite chain $t_1 \succ t_2 \succ \cdots$. In addition to the contructors needed to build sets of equations and rules, the following operations are declared: $\triangleright$, that defines a well-founded ordering on rules for a given reduction ordering $\succ$; nm, that reduces a term to its normal form with respect to a given set of rules; and cp that computes the critical pairs between two rules.

mod COMPLETION[ET :: EQUATIONAL-TH , O :: OTERM] **is**
protecting META-LEVEL[ET] .
sorts *Eqt SetEqt Rl SetRl ERPair* . **subsort** *Eqt* < *SetEqt* . **subsort** *Rl* < *SetRl* .
op $\{\_ \simeq \_\}$: *Term Term* $\rightarrow$ *Eqt* [comm] .
op $\_ \cup \_$: *SetEqt SetEqt* $\rightarrow$ *SetEqt* [**assoc comm id:** empty] .
op $\{\_ \rightarrow \_\}$: *Term Term* $\rightarrow$ *Rl* .
op $\_ \cup \_$: *SetRl SetRl* $\rightarrow$ *SetRl* [**assoc comm idem id:** empty] .
op $\_;\_$: *SetEqt SetRl* $\rightarrow$ *ERPair* . **op** $\_\triangleright\_$: *Rl Rl* $\rightarrow$ *Bool* .
op nm : *Term SetRl* $\rightarrow$ *Term* . **op** cp : *Rl Rl* $\rightarrow$ *SetEqt* .
vars $s\ t\ l\ r$: *Term* . **var** E : *SetEqt* . **var** R : *SetRl* .
crl [delete] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E; R)$ **if** $s == t$.
crl [compose] : $(E; R \cup \{s \rightarrow t\}) \Rightarrow (E; R \cup \{s \rightarrow \mathsf{nm}(t, R)\})$ **if** $\mathsf{nm}(t, R)$ =/= t .
rl [simplify] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E \cup \{\mathsf{nm}(s, R) \simeq \mathsf{nm}(t, R)\}; R)$.
crl [orient] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E; R \cup \{s \rightarrow t\})$ **if** $s \succ t$.
crl [collapse] : $(E; R \cup \{s \rightarrow t\} \cup \{l \rightarrow r\}) \Rightarrow (E \cup \{\mathsf{nm}(s, \{l \rightarrow r\}) \simeq t\}; R \cup \{l \rightarrow r\})$
if $\mathsf{nm}(s, \{l \rightarrow r\})$ =/= s **and** $s \rightarrow t \triangleright l \rightarrow r$.
rl [deduce] : $(E; R \cup \{s \rightarrow t\} \cup \{l \rightarrow r\}) \Rightarrow$
$(E \cup \mathsf{cp}(\{s \rightarrow t\}, \{l \rightarrow r\}); R \cup \{s \rightarrow t\} \cup \{l \rightarrow r\})$.

In [11] Huet proved the correctness of a Knuth-Bendix completion algorithm in which critical pairs are generated incrementally, that is, critical pairs between rules that have been previously considered are not recomputed. In order to provide the strategy language with a convenient data structure to keep this information, we extend COMPLETION with the constructor info, to record for any rule its status—marked or unmarked—and a list of the rules with which critical pairs have been already generated; we also introduce the concatenation operation _,_, to form an ordered list of these records. Finally, we define several operations to update and retrieve this information, with the expected semantics. The operation adequate checks the ordering of two rules in the list and whether or not critical pairs between them have been already generated. Finally, non-empty-set-eqt checks if a set of equations is empty.

sorts *Status ListRl InfoRl ListInfoRl* . **subsort** *InfoRl* < *ListInfoRl* .
ops marked unmarked : $\rightarrow$ *Status* . **op** info : *Rl Status ListRl* $\rightarrow$ *InfoRl* .
op _,_ ; *ListInfoRl ListInfoRl* $\rightarrow$ *ListInfoRl* [**assoc**] . **op** empty-record : $\rightarrow$ *ListInfoRl* .
op unmarked-rl? : *ERPair ListInfoRl* $\rightarrow$ *Bool* . **op** non-empty-set-eqt? : *ERPair* $\rightarrow$ *Bool* .
op adequate : *Rl Rl ListInfoRl* $\rightarrow$ *Bool* . **op** select-unmarked-rl : *ListInfoRl* $\rightarrow$ *Rl* .
op add-unmarked-rl : *Rl ListInfoRl* $\rightarrow$ *ListInfoRl* . **op** mark-rl : *Rl ListInfoRl* $\rightarrow$ *ListInfoRl* .
op add-cp-rl : *Rl Rl ListInfoRl* $\rightarrow$ *ListInfoRl* . [...] **endm**

Now we can define STRATEGY-COMPLETION for an equational theory and an ordering as an extension of STRATEGY applied to the corresponding COMPLETION module. Notice how operations and conditions are easily defined using user-defined operations.

As in Huet's algorithm, one of three things will happen to a rewriting process started with a term rew $\langle\, \overline{(E\,;\,\emptyset)} \,||\, \overline{\text{empty-record}}\,\rangle$ with H-completion: 1) it will stop with success, rewriting to a strategy expression of the form rew $\langle\, \overline{(\emptyset\,;\,R)} \,||\, \overline{w}\,\rangle$ with idle, where R is a convergent rewrite system; 2) it will stop with failure, rewriting to rew $\langle\, \overline{(E'\,;\,R')} \,||\, \overline{w}\,\rangle$ with idle, where E' is a non-empty set of equations; or 3) it will loop forever. Finally, notice that because COMPLETION itself imports the metalevel of another theory, STRATEGY-COMPLETION in fact operates at the meta-metalevel, as explained in Sect. 1.3. Since Maude is truly reflective, rewriting at any level in the 'reflective tower' is supported.

```
mod STRATEGY-COMPLETION[ET :: EQUATIONAL-TH, O :: OTERM] is
protecting STRATEGY[COMPLETION[ET, O]] .
ops α ρ γ δ : → MetaVar . ops non-empty-set-eqt? unmarked-rl? : → Condition .
op adequate : Term Term → Condition . op select-unmarked-rl : → Operation .
ops add-unmarked-rl mark-rl : Term → Operation .
op add-cp-rl : Term Term → Operation . op H-completion : → Strategy .
vars t r r' : Term . var w : Record .
eq eval unmarked-rl? in ⟨ t || w ⟩ = meta-reduce('unmarked-rl?[t, w]) . [...]
eq H-completion =
loop{ while (non-empty-set-eqt){
            apply [simplify] andlet (λ ↪ 'nm['s, 'R], ρ ↪ 'nm['t, 'R]){
                                  or{ apply [delete] with ('s ← λ ; 't ← ρ) } else{
                                    or { apply [orient] with ('s ← λ ; 't ← ρ) } else
                                        { exitloop } ;
                                    set add-unmarked-rl('→[λ, ρ]) ;
                                    iterate { apply [collapse] with ('s ← λ ; 't ← ρ) } ;
                                    iterate { apply [compose] } } } } ;
    if (unmarked-rl) then { let ('→[λ, ρ] ↪ select-unmarked-rl) {
                                iterate { apply [deduce] with ('s ← λ ; 't ← ρ)
                                          if adequate('→[λ, ρ], '→['l, 'r])
                                          andlet (δ ↪ 'l, γ ↪ 'r) {
                                              set add-cp-rl('→[λ, ρ], '→[δ, γ]) } } ;
                                set add-marked-rl('→[λ, ρ]) }
                            } else { exitloop } } . endm
```

5 Concluding Remarks

A slightly different version of the completion procedure presented here has been executed in Maude to complete a number of example theories. The slight differences are either purely syntactic, or due to the fact that matching modulo idempotency has not yet been implemented, so that only matching modulo associativity and commutativity (AC) is used. In spite of the fact that three levels of reflection and AC matching are involved, the performance obtained, about 1,500 rewrites per second on a 120MHz Pentium laptop, seems adequate for a prototype. Besides completion, we have experimented with several other examples, such as an inductive theorem-prover for equational logic, strategies for different rewriting logic interpreters (these are strategies for the universal theory $\mathcal{U}$), and several games and AI problem-solving examples. In all cases our solutions cleanly separate the declarative part, given by the rules of the object theory, from the control part, given by internal strategies. Pita and Martí-Oliet [24] report an interesting application of Maude reflection to specify the management of broadband telecommunication networks in which an internal strategy language again permits a clean separation of the object specification from the control aspects.

Aknowledgements. We cordially thank Steven Eker for his extensions to the metalevel of Maude, on which the present experiments are based. We also thank him, Francisco Durán, Patrick Lincoln, and Narciso Martí-Oliet for their very helpful comments to a previous draft and many discussions on these topics. The suggestion that completion algorithms would be a good testing ground for our ideas on strategies was kindly made to us by Mark Stickel some time ago. We have also benefit from discussions on strategies with Maria Paola Bonacina, Claude and Hèlene Kirchner, Peter Borovanský and Pierre-Etienne Moreau. We express our sincere thanks to all of them.

References

1. W. E. Aitken, R. L. Constable, and J. L. Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical Report, Computer Sci. Dept., Cornell University, 1993; also, lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.
2. L. Bachmair, N. Dershowitz, and J. Hsiang. Ordering for equational proofs. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 346–357, 1986.
3. M. P. Bonacina and J. Hsiang. Parallelization of deduction strategies: An analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.
4. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 168–188. Elsevier, September 1996.
5. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier, September 1996.
6. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, September 1996.
7. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288. Xerox PARC, 1996.
8. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 125–147. Elsevier, September 1996.
9. R. L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *Computer and System Sciences*, pages 109–144. Springer, 1995.

10. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.
11. G. Huet. A complete proof of correctness of the Knuth and Bendix completion algorithm. *Journal of Computer and System Sciences*, 23(1):11–21, 1981.
12. G. Kiczales, editor. *Proceedings of Reflection'96*. Xerox PARC, 1996.
13. H. Kirchner and P. E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.
14. D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
15. P. Lescanne. Completion procedures as transition rules + control. In M. Diaz and F. Orejas, editors, *TAPSOFT'89*, volume 351 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1989.
16. P. Lincoln, J. Meseguer, and L. Ricciulli. The Rewrite Rule Machine Node Architecture and its Performance. In B. Buchberger and J. Volkert, editors, *Proceedings of CONPAR'94*, pages 509–520, 1994.
17. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, editor, *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
18. J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
21. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, Lecture Notes in Computer Science, pages 331–372. Springer-Verlag, 1996.
22. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 253–293. Springer-Verlag, 1992.
23. L. Paulson. Tactics and tacticals in Cambridge LCF. Technical Report 39, University of Cambridge, 1983.
24. I. Pita and N. Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. In *Proceedings of APPIA-GULP-PRODE*, 1997.
25. H. Rueß. Reflection of formal tactics in a deductive reflection framework. Manuscript, Universität Ulm, Abt. Künstliche Intelligenz, January 96.
26. B. Smith and A. Yonezawa, editors. *Proceedings of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture*. Research Institute of Software Engineering, 1992.

Strategies and rewriting in ELAN

Peter Borovanský, Claude Kirchner and Hélène Kirchner

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
email: borovan,ckirchne,hkirchne@loria.fr
http://www.loria.fr/equipe/protheo.html

Abstract. ELAN implements computational systems, a concept that combines rewriting logic with the description of strategies. ELAN can be used either as a logical framework or to describe and execute deterministic as well as non-deterministic rule based processes. We describe here the strategy language offered by ELAN and some applications.

1 Introduction

ELAN is an environment dedicated to prototype, experiment and study the combination of different deduction systems for constraint solving, theorem proving and logic programming paradigms. Its evaluation mechanism is rewriting, whose elegance and expressiveness as a computational paradigm are no more to be stressed. Less evident is the weakness that comes from the absence of controlling mechanism over rewriting. In many existing rewriting-based languages or systems, the term reduction strategy is hard-wired and is not accessible to the designer of an application. Even for medium size applications, controlling rewriting is an important issue.

The language is based on the concept of computational systems [KKV95,Vit94,BKK+96b] given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is a rewrite theory in rewriting logic [Mes92], [MOM93], together with a notion of strategy to select relevant computations. Each ELAN module defines its own signature, labelled rewrite rules and strategies. ELAN is implemented in C++.

This paper explains first in Section 2 how non-deterministic computations are handled in ELAN and combined with deterministic ones. Then Section 3 describes the strategy language of ELAN which provides some predefined primitives and the possibility for the user to define his own strategies in a very flexible way using the same paradigm of rewriting. For instance the user may want to develop his own search space library and use it for solving different problems. This is illustrated in Section 4. The conclusion addresses some further research perspectives.

2 Evaluation with rules and strategies

ELAN is a system for prototyping non-deterministic computations thanks to rules and strategies. An ELAN program is composed of a signature part describing operators with their types, a set of rules and a set of strategies. Strategies is one of the main originality of ELAN compared to other algebraic specification languages based on rewriting. A strategy is a way to describe which computations the user is interested in, and specifies where a given rule should be applied in the term to be reduced. We describe informally here the evaluation mechanism and how it deals with rewrite rules and strategies.

Rules are labelled conditional rewrite rules with local variable affectations

$$[\ell] : l \Rightarrow r \ \ \textbf{if} \ v \ \ \textbf{where} \ y := [S]u$$

ℓ is the rule label, l and r the respective left and right-hand sides, v the condition and $y := [S]u$ a local affectation, assigning to the local variable y the result of the strategy S applied to the term u.

For applying such a rule on a term t, say at top position, first l is matched against t, using if necessary, associative commutative matching (if some operators in the signature are declared as associative and commutative). Then the expressions in **where** and **if** parts, instantiated with the matching substitution, are evaluated in order. When there is no failure, this usually instantiates the local variables (such as y) in **where** expressions, and this extends the matching substitution. When every condition is satisfied, the replacement by the instantiated left-hand side is performed. Indeed the power of this mechanism comes from the fact that **where** expressions may invoke strategy evaluation.

Application of a rewrite rule in ELAN yields, in general, several results. This is first due to equational matching (for instance AC-matching) and second to the **where** assignment, since it may itself return several possible assignments for variables, due to the use of strategies. When a rewrite rule or a strategy returns an empty set of terms, we say that it *fails*.

Thus the language provides a way to handle this non-determinism. This is done using two basic strategy operators: **dc** standing for dont care choose and **dk** standing for dont know choose. For a rewrite rule $[\ell] : l \Rightarrow r$ the strategy **dc**(ℓ) returns *at most one* result which is undeterministically taken among the possible results of the application of the rule. On the contrary, if the ℓ rule is applied using the **dk**(ℓ) strategy, then *all* possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation. This is extended to the application of several rules: the **dk** strategy results in the application of all substrategies and yields the concatenation of all results; the application of the **dc** strategy returns the set of results of one (the first) non-failing rule application. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.

Strategies can be expressed using **dk** and **dc** operators and regular expressions built on the alphabet of rule labels. Since they may be called in the **where** construction of rule definitions, they can be applied everywhere in a term, and with only a few primitives for concatenation and non-deterministic choices, the expressive power of strategies in ELAN is far more than just regular expressions formed from rule labels.

3 The strategy language

This section elaborates the idea of controlling rewriting *by rewriting*, more precisely by defining a strategy language based on rewriting.

The notion of strategy as subset of proof terms of a rewrite theory was proposed in [KKV95,Vit94]. The set of *(closed) proof terms* Π is defined as the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;\})$ built on function symbols $\mathcal{F}$ of the rewrite theory, labels $\mathcal{L}$ and the symbol ';' standing for concatenation. The symbol $\_ : \_ \Rightarrow \_$ is defined on $\Pi \times \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ and $\pi : t \Rightarrow t'$ means that there is a chain of rewrite steps from the term t to the term t' encoded by the proof term π. It is easy to see that for each proof term $\pi \in \Pi$, the sets
$dom(\pi) = \{t \mid \exists\, t'\ s.t.\ \pi : t \Rightarrow t'\}$ and $cod(\pi) = \{t' \mid \exists\, t\ s.t.\ \pi : t \Rightarrow t'\}$
are either both singletons or both empty. Thus, we can also define a partial function $[\_]\_ : \Pi \times \mathcal{T}(\mathcal{F}) \mapsto \mathcal{T}(\mathcal{F})$, such that $[\pi]dom(\pi) = cod(\pi)$.

A *strategy* S is defined as a subset of Π. The definition of the symbol $\_ : \_ \Rightarrow \_$ is extended to strategies as follows:
$S : t \Rightarrow t'$ if there exists $\pi \in S$, such that $\pi : t \Rightarrow t'$, and
$[S]t = \{t' \mid \pi \in S,\ \pi : t \Rightarrow t'\}$.
The definition of domain can be extended too:
$dom(S) = \{t \mid \exists t' \in \mathcal{T}(\mathcal{F}),\ \exists \pi \in S,\ \text{s.t. } \pi : t \Rightarrow t'\}$.

An arbitrary strategy $S \subseteq \Pi$ may be very complicated or irregular from the computational point of view (for instance a non-recursive set). This is why we concentrate on describing special subclasses of strategies and define two subclasses of general strategies, called elementary and defined strategies whose operational semantics are given using rewriting logic. The main difference between these two classes is that elementary strategies are predefined in ELAN. while defined strategies are user-definable, and may be recursive, parameterised and typed as well.

3.1 Elementary strategies

The first step is the construction of elementary strategies, where *an elementary strategy* S is an element of the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;, \mathbf{id}, \mathbf{dc}, \mathbf{dk}, \mathbf{first}, \mathbf{fail}\})$. Roughly speaking, elementary strategies represent non-recursive non-deterministic computations.

The strategy operator **dk** stands for *dont-known-choose*, **dc** for *dont-care-choose* and **first** is a 'sequential version' of dont-care-choose, which takes always the first, in textual order, successful branch. The strategy constants **id** and **fail** respectively represent identity and failure. $\mathbf{l}(S_1, \ldots, S_n)$ corresponds to an application of a rewrite rule labelled by $\ell \in \mathcal{L}$, which also applies substrategies S_i on values of variables x_i after matching and before replacing the matched subterm by the instantiated right-hand side of the rewrite rule. $\mathbf{f}(S_1, \ldots, S_n)$ is an application of substrategies S_i to subterms t_i of the term $f(t_1, \ldots, t_n)$ with root $f \in \mathcal{F}$.

The description of operational semantics of elementary strategies is achieved through the definition of an interpreter described by labelled rewrite rules, given in Figure 1. These rules formally define the binary function symbol $[\_]\_$, that stands for the non-deterministic application of elementary strategy S to the term t. The evaluation of a term $[S]t$ is not deterministic, due to the two first rules defining $[\mathbf{dc}(S_1, S_2)]t$ that can apply simultaneously. So we can obtain as many results as **dc** operators contained in the strategy S.

The result res of these rules application is a finite subset of terms. For the representation of results, we use an abstract data type **List** defining lists as usual with concatenation denoted by ., the empty list denoted by $\emptyset$, and an append operation denoted by $\cup$. For convenience, more sophisticated operators are also defined over the type **List**, such as:

$$\bowtie f(w_1, \ldots, w_n) = \{\mathbf{f}(e_1, \ldots, e_n) \mid e_i \in w_i,\ i = 1 \ldots n\},\ \text{and}$$
$$\bowtie t(w_1, \ldots, w_n)\ = \{\{x_i \mapsto e_i\}t \mid x_i \in Var(t),\ e_i \in w_i,\ i = 1..n\},$$

The operator $[[\_]]\_$ is the generalisation of $[\_]\_$ on lists. All these operators are formally defined by rewrite rules too.

The notation $res = ([S]t)_\downarrow$ means that res is a set in normal form according to the rules in Figure 1. Note that this is in general just *one of all* irreducible forms of $[S]t$. Similarly, the notation $true = (t \in dom(S))_\downarrow$ means that there is a derivation of the term $t \in dom(S)$ terminating with $true$. Indeed the semantics of the symbol $\_ \in dom(\ \_\ )$ can be similarly described by a rewrite theory. The symbol $\ll^?$ stands for matching in the empty theory. A rewrite theory for matching is also added to the interpreter.

[DK]	$[\mathbf{dk}(S_1, S_2)]t$	$\Rightarrow [S_1]t \cup [S_2]t$
[DC$_1$]	$[\mathbf{dc}(S_1, S_2)]t$	$\Rightarrow res_1$ if $true = (t \in dom(S_1))_\downarrow$ and $\emptyset \neq res_1 = ([S_1]t)_\downarrow$
[DC$_2$]	$[\mathbf{dc}(S_1, S_2)]t$	$\Rightarrow res_2$ if $true = (t \in dom(S_2))_\downarrow$ and $\emptyset \neq res_2 = ([S_2]t)_\downarrow$
[DC$_3$]	$[\mathbf{dc}(S_1, S_2)]t$	$\Rightarrow \emptyset$ if $true \neq (t \in dom(S_1))_\downarrow$ and $true \neq (t \in dom(S_2))_\downarrow$
[FIRST$_1$]	$[\mathbf{first}(S_1, S_2)]t$	$\Rightarrow res_1$ if $true = (t \in dom(S_1))_\downarrow$ and $\emptyset \neq res_1 = ([S_1]t)_\downarrow$
[FIRST$_2$]	$[\mathbf{first}(S_1, S_2)]t$	$\Rightarrow [S_2]t$ if $true \neq (t \in dom(S_1))_\downarrow$
[ID]	$[\mathbf{id}]t$	$\Rightarrow \{t\}$
[FAIL]	$[\mathbf{fail}]t$	$\Rightarrow \emptyset$
[FSYM$_1$]	$[\mathbf{f}(S_1, \ldots, S_n)]f(x_1, \ldots, x_n)$	$\Rightarrow \bowtie f([S_1]x_1, \ldots, [S_n]x_n)$
[FSYM$_2$]	$[\mathbf{f}(S_1, \ldots, S_n)]g(x_1, \ldots, x_m)$	$\Rightarrow \emptyset$ if $f \neq g$
[LAB$_1$]	$[\mathbf{l}(S_1, \ldots, S_n)]u(x_1, \ldots, x_n)$	$\Rightarrow \bowtie u'(\ldots, [S_i]x_i, \ldots)$ where $[\ell]u \Rightarrow u' \in \mathcal{R}$
[LAB$_2$]	$[\mathbf{l}(S_1, \ldots, S_n)]t$	$\Rightarrow \emptyset$ if $\emptyset = (u \ll^? t)_\downarrow$ where $[\ell]u \Rightarrow u' \in \mathcal{R}$
[CONC]	$[(S_1; S_2)]t$	$\Rightarrow [[S_2]]\ [S_1]t$

Fig. 1. The interpreter of elementary strategies

Elementary strategies can be given sorts, based on the set of sorts $\mathcal{S}$ defined in the user's rewrite theory. This introduces new strategy sorts $\langle s' \mapsto s'' \rangle$ for the sort of all elementary strategies transforming terms of sort s' into terms of sort s''.

3.2 Defined strategies

The goal is now to be able to define new strategies by rewrite rules. Let us take the example of **map**, that can be defined by a strategy rewrite rule in the following way:

$$\mathbf{map}(S) \Rightarrow \mathbf{dc}(\mathbf{nil}, S.\mathbf{map}(S)) \qquad (1)$$

The right-hand side of this definition means that whenever the strategy **map** with an argument S (i.e. $\mathbf{map}(S)$) is applied to a term t, either t is nil, or the strategy S is applied on the head of t (i.e. t should be a non-empty list) and $\mathbf{map}(S)$ is further applied on the tail of t.

This strategy definition can be formulated also using the strategy application symbol $[\_]\_$:

$$\begin{array}{ll} [\mathbf{map}(S)]nil & \Rightarrow nil \qquad\qquad (2) \\ [\mathbf{map}(S)]a.as & \Rightarrow [S]a.[\mathbf{map}(S)]as \end{array}$$

The difference relies on the fact that a list, which the functional **map** is applied on, is an explicit argument in the second (or third) definition, while in the first one, it is implicit.

Rule (1) applied without special strategy may lead to infinite computations. This justifies the concept of meta-strategies, i.e. strategies which control the execution by rewriting of defined strategies. Such a strategy can be either constructed by enriching the previous interpreter, or explicitly defined by the user from labelled rules on strategies.

A rule like $\mathbf{map}(S) \Rightarrow \mathbf{dc}(\mathbf{nil}, S.\mathbf{map}(S))$ is a definition rewrite rule, in that it defines recursively the strategy **map**. Indeed other rules involving **map** can be written, to express in particular some equivalence on defined strategies, for instance the distributivity of **map** on the concatenation ;

$$\begin{array}{ll} \mathbf{map}(S_1)\ ;\ \mathbf{map}(S_2) & \Rightarrow \mathbf{map}(S_1\ ;\ S_2) \\ \mathbf{map}(\mathbf{id}) & \Rightarrow \mathbf{id} \end{array}$$

To formalize these ideas, a *strategy definition* is given by

- A set of strategy symbols $\mathcal{F}_\mathcal{D}$ with signatures: $\mathbf{d} : (s_1, \ldots, s_n)\langle s' \mapsto s''\rangle$ where $s_1, \ldots, s_n$ are strategy sorts of the arguments, and $\langle s' \mapsto s''\rangle$ is the sort of the result.
- A finite set of rewrite rules
  - either in an implicit form $\mathbf{d}(S_1, \ldots, S_n) \Rightarrow S$
  - or in an explicit form $[\mathbf{d}(S_1, \ldots, S_n)]x \Rightarrow [S]x$

  where $\mathbf{d} \in \mathcal{F}_\mathcal{D}$, S_i for $i = 1, \ldots, n$ are elementary strategies, $x : s'$ is a new variable in $\mathcal{X}$, S is a term built from defined strategy symbols, elementary strategy symbols, rule labels and variables.

The interpreter described for elementary strategies is expanded in both cases of the previous definition with new rules, labelled with DSTR:

$$[DSTR] \quad [\mathbf{d}(S_1, \ldots, S_n)]x \Rightarrow [S]x$$

Different evaluation modes for strategy application is offered via different possibilities for labels. A rewrite rule on strategy labelled by [] will be applied just as unlabelled rules on terms in an innermost-outermost way. A rewrite rule on strategy labelled by the special label $[\cdot]$ will be applied in a lazy way using the built-in strategy of the interpreter of strategies. A rewrite rule on strategy labelled by $[\ell]$ will be applied just as labelled rules of terms with a user-designed strategy built from these labels.

Example 1. The definition of the strategy **map** is for instance given by the definition of the strategy symbol **map** : $(\langle s \mapsto s\rangle)\langle list[s] \mapsto list[s]\rangle$ and the rule
$\mathbf{map}(S) \Rightarrow \mathbf{dc}(\mathbf{nil}, S \cdot \mathbf{map}(S))$, where $S : \langle s \mapsto s\rangle$ and $s \in \mathcal{S}$. The following rule with the label $\mathsf{DSTR}_{\mathbf{List}[list[s]]}$ is added $[\mathbf{map}(S)]x \Rightarrow [\mathbf{dc}(\mathbf{nil}, S \cdot \mathbf{map}(S))]x$, where $x : list[s]$.

Several simple strategy definitions are given below.

$$\mathit{variables} \qquad x : \langle s \mapsto s\rangle \quad xs : \langle list[s] \mapsto list[s]\rangle \quad ys : \langle u \mapsto v\rangle$$

iterate : $(\langle s \mapsto s\rangle)\langle s \mapsto s\rangle$	$\mathbf{iterate}(x)$	$\Rightarrow \mathbf{dk}(x; \mathbf{iterate}(x),\ \mathbf{id})$
repeat : $(\langle s \mapsto s\rangle)\langle s \mapsto s\rangle$	$\mathbf{repeat}(x)$	$\Rightarrow \mathbf{first}(x; \mathbf{repeat}(x),\ \mathbf{id})$
map1 : $(\langle s \mapsto s\rangle)\langle list[s] \mapsto list[s]\rangle$	$\mathbf{map1}(x)$	$\Rightarrow \mathbf{dc}(\mathbf{nil}, x \cdot \mathbf{map1}(x))$
map2 : $(\langle list[s] \mapsto list[s]\rangle)\langle list[s] \mapsto list[s]\rangle$	$\mathbf{map2}(nil)$	$\Rightarrow \mathbf{nil}$
	$\mathbf{map2}(x.xs)$	$\Rightarrow x \cdot \mathbf{map2}(xs)$

The **iterate** strategy differs from **repeat** in a such way that it returns all intermediate forms during the normalisation of a term by the application of a strategy x, while **repeat** returns only the last one (i.e. normal form). The strategy **map**1 applies a fixed strategy x on all elements of a list and produces a new list of transformed elements. The strategy **map2** is driven by a list of strategies which are respectively applied to elements of a list of same length.
For instance, with the rewrite rule: $[\ell] \quad x + 0 \Rightarrow x$ and strategies defined above:
$[\mathbf{iterate}(\mathbf{l}(\mathbf{id}))](a + 0) + 0$ yields $\{(a + 0) + 0, a + 0, a\}$
$[\mathbf{repeat}(\mathbf{l}(\mathbf{id}))](a + 0) + 0$ yields $\{a\}$

4 A search space library

We have shown in [KM95] how saturation-based theorem provers like completion can be easily and efficiently implemented using rewrite rules and strategies. We have also implemented a mechanical theorem prover for first-order predicate calculus [BKK96a]. Now we give an example of a small strategy library, i.e. several defined strategies, providing primitives for depth-first search in a computation tree. Two natural applications of this strategy library are examples of a winning strategy for a tic-tac-toe-like game, and a guided strategy helping to find out a path (or, all paths) towards an exit of a labyrinth.

The strategy library supports four primitives for depth-first search. The current situation of the partially discovered search tree is represented by a path from the initial state (the root of the tree) to the current state (the left-most leaf). It also remembers all possible alternatives (or, choice points) met along this path. Each step of the path is represented by a list of states, where its head is the currently chosen state, and the rest represents all non-explored alternative states. The limitation of this approach is that the number of successive (alternative) states of any state should be finite, i.e. the search tree has a finite degree. Each step of the path is a list of states, thus, the whole path could be naturally represented as $list[list[state]]$. This sort also describes the current situation during the depth-first search in this tree.
We define four primitive strategies over the sort $list[list[state]]$, which develop and maintain this search tree. Their intended semantics are the following:

- $call(S)$ applies the strategy S on the current state, where S is a strategy transforming a state into a state, i.e. of the sort $\langle state \mapsto state\rangle$. All possible results of this application, if there are any, are formed into a new step (level) attached to the current path.
- *next* throws away the current state and continues searching with the next possibility of the previous step (the latest choice-point).
- *exit* leaves the current state ignoring all alternatives and it returns the control to the previous step of the current path.
- *cut* eliminates all alternative states of the latest step.

```
stratop global
 call(@)          : (<state->state>) <list[list[state]]->list[list[state]]>;
 next             : <list[list[state]]->list[list[state]]>;
 exit             : <list[list[state]]->list[list[state]]>;
 cut              : <list[list[state]]->list[list[state]]>;
end
strategies for list[list[state]]
 s                : <state->state>;
 state            : state;
 level,
 new_level        : list[state];
 space            : list[list[state]];
explicit
 [.]      [call(s)]state.level.space => new_level.state.level.space
                           where new_level:=()setof(s,state)
                           if new_level != nil                 end
 [.]      [next]state.level.space => level.space                end
 [.]      [exit]state.level.space => space                      end
 [.]      [cut]state.level.space => state.nil.space             end
end
```

Fig. 2. Search-space library

For better understanding of their formal definitions in Figure 2, let us recall that on the left-hand sides, the variable *state* represents the current state, the variable *level* represents all alternatives of the current state, i.e. $state.level : list[state]$ is one step of the path, where the others steps are recorded in the variable *space*. The only non-trivial strategy among the four primitives is the strategy $call(S)$, which uses the function symbol *set_of* to collect all results of an application of a strategy S.

Tic-tac-toe game The use of the depth-first search library is illustrated by the definition of a game winning strategy for a simple game with a finite search space. The solution described below is split into game-dependent and independent parts. The game-independent part does not rely on rules of a particular game. These particularities are hidden in the game-dependent kernel.

The particular game chosen for this example is the tic-tac-toe game with the play-board of the size 3×3. Two players with different signs alternatively put one sign on the board. The winner is the player who succeeds in having 3 signs in a same column, row, or diagonal.

The game-dependent part defines the state (Figure 3) as a pair encoding

```
operators global
 X        : Sign;           { sign of the 1-st player              }
 O        : Sign;           { sign of the 2-nd player              }
 -        : Sign;           { empty sign                           }
 * @,                       { next player's sign                   }
 |@,@,@|,                   { playboard : 1-st row                 }
 |@,@,@|,                   { playboard : 2-nd row                 }
 |@,@,@| *                  { playboard : 3-rd row                 }
          : ( Sign Sign Sign Sign Sign Sign Sign Sign Sign Sign ) state;
end
```

Fig. 3. A state of tic-tac-toe game

in its first component the next player and in its second component the play-board. There are three basic strategies *win*, $put(row, column)$, *moves*, all of the sort $\langle state \mapsto state \rangle$. *nxt* is a function that switches players. Their definitions are given in a straightforward exhaustive way, thus, we show only a small part of them in Figure 4.

The strategy $win : \langle state \mapsto state \rangle$ tries to put the last sign and to win the game. It fails, if there is no correct winning move. The deterministic strategy $put(row, column)$ puts the sign on the field with coordinates *row* and *column*, if it is possible. The non-deterministic strategy *moves* generates all possible moves (according to the game rules) from the current situation.

The game-independent part specifies the winning strategy *movetowin* using an auxiliary strategy *findloosing* in Figure 5. The strategy *movetowin*, first of all, tries to apply the basic strategy *win* searching a winning move. This application is realised by calling the library primitive $call(win)$. It succeeds if there is a winning move. Otherwise, the strategy *movetowin* produces the set of all possible moves by $call(moves)$. These successive possibilities are filtered by

```
strategies for state
a, b, c, d, e, f, g, h, i, Y              : Sign;
explicit
 [.]      [win] *Y,|-,Y,Y|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
 [.]      [win] *Y,|Y,-,Y|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
 [.]      [win] *Y,|Y,Y,-|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
 . . . . . . . . . . . . . . . . . . . . . . . . .
 [.]      [put(1,1)] *Y,|-,b,c|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,b,c|,|d,e,f|,|g,h,i|* end
 [.]      [put(1,2)] *Y,|a,-,c|,|d,e,f|,|g,h,i|* => *nxt(Y),|a,Y,c|,|d,e,f|,|g,h,i|* end
 [.]      [put(1,3)] *Y,|a,b,-|,|d,e,f|,|g,h,i|* => *nxt(Y),|a,b,Y|,|d,e,f|,|g,h,i|* end
 . . . . . . . . . . . . . . . . . . . . . . . . .
implicit
 [.]      moves => dk(       put(1,1), put(1,2), put(1,3),
                             put(2,1), put(2,2), put(2,3),
                             put(3,1), put(3,2), put(3,3)
                    )
end
```

Fig. 4. Basic moves of tic-tac-toe game

```
stratop global
 movetowin          : <list[list[state]]->list[list[state]]>;
 findloosing        : <list[list[state]]->list[list[state]]>;
 guess              : <list[list[state]]->list[list[state]]>;
end
strategies for list[list[state]]
implicit
 [.]      movetowin => first(call(win), call(moves);findloosing;cut) end
 [.]      findloosing =>     if movetowin then exit; next; findloosing
                             orelse id
                             fi                                          end
 [.]      guess => first(movetowin, call(moves);heuristic)               end
end
```

Fig. 5. A winning strategy

the strategy *findloosing* removing all winning configurations of the adversary. If there is at least one possible move such that the adversary's configuration is not winning, the strategy *findloosing* succeeds, and all alternative moves are ignored by the primitive *cut*. Otherwise, the strategy *findloosing* fails, and *movetowin* fails also. These two mutually recursive strategies represent the kernel of the game-independent part. The winning strategy *movetowin* may be encapsulated by a strategy *guess*, which either proposes to the player a winning move, or, if there is no winning move, proposes one of the non-winning configuration chosen with an heuristic strategy. The strategy *heuristic* saying what to do if we are not in a winning configuration, depends on the particular game and some defensive tactic. When no specific heuristic is proposed, it can be replaced by the strategy **id**. Figure 6 illustrates the use of these strategies on a small game session. With a similar encoding of the game state and rules, we can also specify other games, such as the nim-game in [CELM96].

```
elan's first move from
 [guess] *X,|-,-,-|,|-,-,-|,|-,-,-|* to *O,|X,-,-|,|-,-,-|,|-,-,-|*
player's first move from
 [put(2,2)] *O,|X,-,-|,|-,-,-|,|-,-,-|* to *X,|X,-,-|,|-,O,-|,|-,-,-|*
elan's attack from
 [guess] *X,|X,-,-|,|-,O,-|,|-,-,-|* to *O,|X,X,-|,|-,O,-|,|-,-,-|*
player's defense from
 [put(1,3)] *O,|X,X,-|,|-,O,-|,|-,-,-|* to *X,|X,X,O|,|-,O,-|,|-,-,-|*
elan's defense from
 [guess] *X,|X,X,O|,|-,O,-|,|-,-,-|* to *O,|X,X,O|,|-,O,-|,|X,-,-|*
player's defense from
 [put(2,1)] *O,|X,X,O|,|-,O,-|,|X,-,-|* to *X,|X,X,O|,|O,O,-|,|X,-,-|*
elan's defense from
 [guess] *X,|X,X,O|,|O,O,-|,|X,-,-|* to *O,|X,X,O|,|O,O,X|,|X,-,-|*
no more chance for the player from
 [put(3,2)] *O,|X,X,O|,|O,O,X|,|X,-,-|* to *X,|X,X,O|,|O,O,X|,|X,O,-|*
no more chance for elan from
 [guess] *X,|X,X,O|,|O,O,X|,|X,O,-|* to *O,|X,X,O|,|O,O,X|,|X,O,X|*

game is over, draw game
```

Fig. 6. Tic-tac-toe game

Labyrinths Another application of our strategy library is a definition of a strategy helping to find out an exit from a labyrinth. An alternative re-formulation of this problem could be to find a path (or, all paths) between two nodes of a graph. As in the previous section, we split the solution into a part dependent on the labyrinth and an independent one.

The dependent part contains a specification of a state, which is a pair of coordinates, and a definition of four basic moves in the labyrinth. They are realized as state transforming strategies $left$, *right*, *up*, *down* of sort $\langle state \mapsto state\rangle$ such that these strategies fail, if some movement in a given state is not possible (because of a wall). All exits of the labyrinth are specified by a strategy *exitable*, which fails, if the current state has 'no exit doors'. Otherwise, it is equivalent to the identity strategy.

The independent part of the strategy (Figure 7) consists of a definition of the strategy *search* of sort $\langle list[list[state]] \mapsto list[list[state]]\rangle$ and an auxiliary strategy *moves* of sort $\langle state \mapsto state\rangle$. The strategy *search* on an exit-able state (a room with exit doors) succeeds. Otherwise, if the current path contains a loop, it backtracks and it takes the next possible state for continuation. If it does not contain a loop, it either moves to the neighbour state and continues to search, or, if no exit has been found, it ignores the current choice, and backtracks to the next alternative.

Intuitively, if we need find out all possible paths, we ought to replace deterministic constructions (i.e. **first**) in the definition of the strategy *search* by a non-deterministic one (i.e. **dk**). The obtained solution can be consequently optimised thanks to two strategy equivalences $\mathbf{dk}(S_1; S, S_2; S) = \mathbf{dk}(S_1, S_2); S$ and **if** *cond* **then** $S_1; S$ **else** $S_2; S$ **fi** $=$ **if** *cond* **then** S_1 **else** S_2 **fi** ; S to the form in Figure 8.

```
strategies for state
implicit
 [.]      moves => dk(right, down, left, up)                end
end
strategies for list[list[state]]
 state    : state;
 level    : list[state];
 space    : list[list[state]];
explicit
 [.]      [search] state.level.space =>
                   [first(   call(exitable),
                             if occurs state in space then
                              next;search
                             else
                              first(call(moves);search, next;search)
                             fi
                   )] state.level.space
 end
```

Fig. 7. One path in the labyrinth

```
 [.]      [allsearch] state.level.space =>
                   [dk(       call(exitable),
                             if occurs state in space then
                              next
                             else
                              dk(call(moves), next)
                             fi;
                             allsearch
                   )] state.level.space
 end
```

Fig. 8. All paths in the labyrinth

5 Conclusion

The definition of the strategy language described in this paper is related to a view of strategies in reflective logics (in particular, rewriting logic) developed in [CM96]. From this point of view, the strategy language described here can be classified as an internal strategy language, whose semantics and implementation are described in the same logic, while the strategy language available in the first distribution of ELAN and described in [Vit94,Vit96] is an external one.

Another question which comes with the reflective approach is how to control computation at the meta-level. Roughly speaking, there are two levels of rewriting: the object (or first-order term) level, and the meta-level that controls the object-level. The first solution is that the meta-level computation (i.e. the evaluation of strategies) is controlled by a built-in strategy. But computation at the meta-level can also be controlled by meta-strategies. Using a reflective logic allows using the same formalism at all levels, which might be viewed as an advantage of a reflective approach.

The capability offered by this approach of being able to rewrite strategies leads to consider the possibility of partial evaluation and further on, of compilation of strategies. This is a promising direction of our current research.

Acknowledgements: We sincerely thank the members of the Protheo group in Nancy and José Meseguer, Steven Eker, Manual Clavel at SRI for many interesting discussions on strategies.

References

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[CM96] M. G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

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

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

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

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

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, July 1996.

Tacticals in ILF - and why they are superfluous

Ingo Dahn

Humboldt University Berlin
Mathematical Institute
Ziegelstr. 13a
D-10099 Berlin
dahn@mathematik.hu-berlin.de

In its history, the ILF system has been used as a shell for interactive theorem provers for several calculi - e.g. for model elimination, tableaux calculus and the ProofPad which works with block structured proofs.These interactive provers have been implemented either in the same Prolog system in which the major part of ILF was developed or in C using the tool kit ILFA.

The rules of inference are the basic proof tactics. Proof tactics in ILF are *Prolog programs* which access the proof data structures only through other proof tactics. Hence their application guarantees the correctness of the resulting proofs.

The tactic language of ILF *is* Prolog. It is as generic as a tactic language can be. Since proof tactics are - by definition - only programs which transform proofs, any other programming language would do as well.The only necessary condition is the ability, that it's programms (tactics) can call other programms (basic tactics). For the sake of safety it would be desirable to protect proof data structures against access by non-primitive proof tactics.

Prolog was chosen because it supported proof search by backtracking and by automatic completion of partially instantiated tactics through unification. Moreover it was available at no cost as the main implementation language of ILF. Prolog does not permit secure protection of data structures like object oriented languages. However the module system of modern Prolog dialects can prevent unintended modification of proof data structures by keeping them as private predicates in separate modules which are different from the data modules used by proof tactics.

The programming of proof tactics is supported in ILF by tacticals. The most basic tactical executes two tactics - one after each other. There is no implementation of this tactical in ILF since it is already a basic construct of Prolog - even with the additional feature that the first tactic can react to a failure of the second tactics by backtracking.

ILF also has other tacticals similar to those of the NuPrl/Oyster system implemented in a generic way. An interactive theorem prover has to provide a few simple procedures in order to make these tacticals available for its proof editing process. Apparently, these procedures are also necessary for the basic proof editing process. Moreover, the calculation of the list of subgoals is also useful for visualizing proof structures by ILF tools.

The advantage of these generic tacticals is, that they can be used instantly, whenever a new interactive prover is integrated into ILF. But since their implementation is relatively easy, they are in actual practice soon replaced by more efficient implementations which take specific properties of the given proof structure into account. Here by proof structure we mean the *logical* structure of the proof, not the *data* structure which represents it and which cannot be directly accessed by proof tactics.

Another reason for the rare use of generic tacticals is, that the vast majority of problems in the implementation of proof tactics is concerned with the analysis of specific data. These are ordinary programming tasks. E.g. a frequent problem is - during the run of a tactic - to remember the positions at which formulas of a particular form are inserted. Later, the tactic may use these positions e.g. as parameters for calls of rules of inference.

A third reason is, that the programmers of proof tactics usually have (and need) programming skills. Hence they tend to implement tactics in a programming language they know.

When automated theorem provers are used as a support for an interactive theorem prover - like in the ProofPad of ILF - the whole task of proof tactics changes dramatically. It is no longer necessary to produce subgoals with a given status like "proved" or "complete" since this can be left to the automated provers. Instead, proof tactics have to decompose proof obligations so that they can be solved by automated provers. This is much more than a transformation guided by the structure of the formula. In such a system, many tactics have also to analyse the logical structure of the proof and the structure of the available axiom system in order to isolate the assumptions that can hopefully be used by an automated prover to solve the generated subgoals. This requires tools which are not only specific for an interactive prover and the cooperating automated provers, but also specific for a given field of knowledge.

Strategies for Logic Programming Languages

Tatjana Lutovac and James Harland

Department of Computer Science
Royal Melbourne Institute of Technology
GPO Box 2476V, Melbourne, 3001, Australia
{tanja, jah} @cs.rmit.edu.au

Abstract. Logic programs consist of formulas of mathematical logic and various proof-theoretic techniques can be used to design and analyse execution models for such programs. We briefly review the main problems, which are questions that are still elusive in the design of logic programming languages, from a proof-theoretic point of view. Existing strategies which lead to the various languages are all rather sophisticated and involve complex manipulations of proofs. All are designed for analysis on paper by a human and many of them are ripe for automation. We aim to perform the automation of some aspects of strategies for logic programming language, in order to assist in the design of these languages. In this paper we describe the first steps towards the design of such an automatic analysis tool. We investigate the usage of particular proof manipulations for the analysis of logic programming strategies. We propose a more precise specification of sequent calculi inference rules that we use as a framework for the generalisation and automation-oriented specification of the permutation process.

1 Introduction

It is well known that logic programming (LP) may be thought of as the application of the techniques of mathematical logic to programming tasks. Logic programs may be considered as collections of formulas and their computation may be identified as searching for proofs: given a program $\mathcal{P}$ and a goal G we attempt to satisfy G by searching for a proof of $\mathcal{P} \rightarrow G$ using the inference rules of a given logic. In addition, logic programs may be analysed declaratively, *i.e.* as statements of logic, without reference to a sequence of operations, and so are more amenable to program transformation and optimisation than many imperative languages. Also, programs written in LP languages tend to be somewhat more concise, and hence more comprehensible to a human, than the same programs written in an imperative language (such as C or Pascal).

There have been various LP systems implemented (Prolog [15], Lolli[6], LinLog[1], λProlog[12], Forum[11], Lygon[17], etc). But still there is no precise concept of what constitutes a LP language as well as what is the essential difference between a LP system and a theorem prover. One proof-theoretic characterisation and a criterion for identifying LP language is formalised in [10] by the identification of *goal-directed proofs* which are usually defined in terms of a *sequent calculus* system of proofs. Essentially, a goal-directed proof is a (cut-free) proof in which a goal is decomposed before the program, and hence the program is not used until the goal is reduced to just an atom. The properties of such proofs have been thoroughly investigated for various fragments of first and higher-order intuitionistic logic (classes of formulas called Horn clauses [5] and hereditary Harrop formulas [10]). Whilst this analysis is quite thorough, there remains the question of how to "lift" the concept of goal-directed proofs to other logics (for example multiple-conclusioned logic, linear logic, light linear logic, affine logic, relevant linear logic, etc.). Given the evolution of programming languages towards higher and higher level languages, with a corresponding increase in the computational power of the execution models of these languages, a natural demand is a wide range of expressive logics and more powerful inference facilities in which to write programs. The relationship between the logical fragment and the proof strategy is fundamental. The proof search strategy should be *complete* (to always find a proof, if one exists) for a given fragment of logic. For example, the strategy of *uniform proofs* [10] can prove all results for a Horn clause fragment or hereditary Harrop fragment of intuitionistic logic. At the same time, in general, we need complete and feasible classes of proofs with a view to automating and optimising proof search. The choice of logical fragment is a compromise between the expressiveness of the logic and the difficulty of proof search automation. This compromise affects the ability of the language to express simply and naturally the solution to a given problem. In some cases, we are able to select a logical fragment on the basis of a given problem to be solved. Still there is no clear technical statement of why, and what kind of restrictions on some classes of formulas are necessary, in order to use classes of formulas in logic programs.

There have been various proof-theoretic techniques, approaches and analyses [1, 2, 3, 5, 6, 9, 13] used to design and analyse LP languages. Many of the existing approaches and analyses which lead to some of the answers are all rather sophisticated and involve complex manipulations of proofs. Many are restricted to particular logic or classes of formulae. Almost all are designed for analysis on paper by a human and many of them are ripe for automation, being formally defined in precise detail, and yet somewhat overwhelming for humans. A general answer and formal definition requires a thorough understanding of all the existing LP systems, as well as of the relationship between theorem proving and LP and, in particular, how to use proof-theoretic techniques to design and analyse execution models for LP languages.

We present initial work on the problem of automating the strategies for the design of LP languages. We propose a more precise specification of sequent calculi inference rules that we use for the generalisation and automation-oriented specification of the permutation process. The full version may be found in [8].

1.1 Aims and expected outcomes

The design of new LP languages demands detailed and systematic analyses of existing LP proof systems and precise answers to the questions:

What class of formulae can be used in programs? What strategy can be applied?

What is the influence and the actual impact of selected choices [18] on the characteristics of the corresponding LP language (for example on expressibility, efficiency, supporting modular programming, abstract data types, higher-order functions, simple operational behaviour, providing proofs as *answers* as well as all possible answer substitutions, etc)?

We plan to perform a systematic study of existing approaches, solutions, strategies and to investigate ways of modifying them to make them more suitable for automated analysis and proof-search. The plan is to design a system, an *automatic analyser, i.e.* a *prototype language compiler generator* which will be able to:

- *recognise the appropriate logical fragment which may be used as a basis for a LP language;*
- *identify what combinations of inference rules are necessary for the execution of a given fragment;*
- *recognise (and perhaps define) proofs which conform to a given strategy;*
- *identify fragments for which a given strategy is complete.*

Many implementations of LP languages have many features in common, such as the use of unification and some form of the resolution rule. The analyser would be a first step towards providing a library of tools such as unification methods and algorithms, top-down and bottom-up proof search procedures and other implementation techniques. But it would be more than just an automatic support tool for the proof-theoretic analyses. It should be an *adaptive system* capable of systematically analysing a given set of inference rules in order to extract and to determine the most appropriate and optimised way to assemble the various pieces into a LP system. It is expected that such systems in the future will be able to determine the appropriately optimised execution model.

2 Proof Manipulations as a tool for Strategy Analysis

Analysis and specification of LP strategies is not just a matter of determining whether a given goal is provable on the base of a given program, but of providing more information about the proof search process. Unlike a theorem prover, an LP system is used to find answers, rather than proofs per se *i.e.* other properties of the proof are taken into account other than just proof existence. Thus, for example, for a non-ground query $\exists x P(x)$, the LP system is expected to return not only *yes* or *no*, but also an *answer i.e.* substitution for the (existentially quantified) variables of the goal: a value of x for which $P(x)$ is true. It is well known that a proof contains much more information than the simple fact that its conclusion is true. The so-called *meta-level* properties of a proof, include, among others, identification of the strategy applied to a given proof and subgoals where more than one strategy is applicable, recognising families of proofs which contain a common structure, finding all possible proofs as well as proofs which lead to different answers, determining the number of answers for an existentially quantified (sub)goal, etc.

There have been various *proof manipulations* which enable extracting meta-level information about a proof, and which are, therefore, a necessary tool in the analysis of existing LP systems and strategies. Proof manipulations include various procedures for modification and adaptation of a proof to different search strategies and to different requirements such as elimination of redundant steps and loops in a proof, decreasing number of rules or the amount of branching in a proof, cut elimination and elimination of other rules with the high level of non-determinism, etc. In the rest of this section we point out two proof manipulations of particular interest: *permutations* and *hypothesis elimination*.

2.1 Permutations

The *permutation* of two adjacent inference rules of a given proof is reversing their order in the proof but without disturbing the rest of the proof (the part above and below the inferences modulo a duplication of some proof branches and a renaming of certain free variables) as a result of which we get the *equivalent* proof to the given one:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash a, c, d, \Gamma\end{array} \quad \begin{array}{c}\vdots\\ \vdash b, \Delta\end{array}}{\vdash c, d, a \otimes b, \Gamma, \Delta}\otimes}{\begin{array}{c}\vdash c \wp d, a \otimes b, \Gamma, \Delta\\ \vdots\end{array}}\wp \quad \Longleftrightarrow \quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash c, d, a, \Gamma\end{array}}{\vdash c \wp d, a, \Gamma}\wp \quad \begin{array}{c}\vdots\\ \vdash b, \Delta\end{array}}{\begin{array}{c}\vdash c \wp d, a \otimes b, \Gamma, \Delta\\ \vdots\end{array}}\otimes$$

Permutations are essential proof manipulations. The dynamics of the proof that results from the permutation analysis clearly has a strong influence on the operational semantics of the resulting LP language. For example, Lygon[17] is based on the search strategy that *some* permutations of right-hand side rules will lead to a proof, whereas Forum[11] is based on the search strategy that *any* permutation of right-hand rules will lead to a proof. Here we give a brief overview of some applications of permutations for the analysis of particular LP strategies.

Example 1 (Defining a proof search strategy).
Permutation properties of the inference rules of a given logical fragment, have direct impact on defining efficient proof search strategies which deal with the order of inference rules. We explain this impact through the so-called *"Normal proof"*[3] strategy which is applicable in LP. It imposes constraints for the choice of formula to be decomposed in the next step of the proof construction. These constraints are based on the permutability of inference rules and contribute to reduction of some sources of non-determinism during the proof construction. In a *bottom-up proof construction* (in which search starts at the root of the tree) this approach always first applies rule for which there is a "guarantee" that in the case of a provable sequent there is a proof with that rule in that particular place in the proof tree. Also, it is important to detect groups of rules that are easy to apply from the conclusion (when are applying in a bottom up direction). It is desirable to apply such rules *as soon as possible* in a proof construction and to postpone the others (which introduce don't know non-determinism when applied in up direction) *as late as possible*. The degree of the realization of this aim *to apply a rule I as soon as possible* or *to apply a rule I as late as possible* is directly dependent on the permutability relationships between I and other inference rules in a given logical fragment *i.e.* in a given proof.

For instance, consider the sequent $\vdash A \otimes B, C \wp D, \Gamma$. During proof construction we must have a complete strategy for selecting the next formula to be decomposed. This selection is crucial for bottom-up proof construction. On the basis of the permutation results given in [3] we have the following table, where the case (t_1, t_2) in the table contains: p iff inference t_1 *permutes down* over inference t_2, and, np iff there exists a proof in which inferences t_1 and t_2 are not permutable.

Table 1.

	t_1					
t_2	$\wp$	$\forall$	$\otimes$	$\oplus$	$\exists$	$w?$
$\wp$	p	p	np	p	p	p
$\forall$	p	p	p	p	np	p
$\otimes$	p	p	p	p	p	p
$\oplus$	p	p	p	p	p	p
$\exists$	p	p	p	p	p	p
$w?$	p	p	p	p	p	p

As a consequence of np for the case $(\otimes, \wp)$ and p for the case $(\wp, \otimes)$, for any provable sequent $\vdash A \otimes B, C \wp D, \Gamma$ there is a proof where $\wp$ is applied before (closer to the root than) the $\otimes$ rule, but not necessary the proof with $\otimes$ applied before $\wp$. For instance, the proof construction of the sequent $\vdash a \otimes b, a^\perp \wp b^\perp$ terminates successfully only when $\wp$ precedes $\otimes$:

$$\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}Ax \qquad \dfrac{}{\vdash b, b^\perp}Ax}{\vdash a \otimes b, a^\perp, b^\perp}\otimes}{\vdash a \otimes b, a^\perp \wp b^\perp}\wp \qquad \dfrac{\dfrac{\dfrac{?}{\vdash a, a^\perp, b^\perp}}{\vdash a, a^\perp \wp b^\perp}\wp \qquad \dfrac{?}{\vdash b}\wp}{\vdash a \otimes b, a^\perp \wp b^\perp}\otimes \qquad \dfrac{\dfrac{?}{\vdash a}\wp \qquad \dfrac{\dfrac{?}{\vdash b, a^\perp, b^\perp}}{\vdash b, a^\perp \wp b^\perp}\wp}{\vdash a \otimes b, a^\perp \wp b^\perp}\otimes$$

while in a proof construction of the sequent $\vdash a \otimes b, a^\perp \wp ?b^\perp, b^\perp$, the order of applied rules does not matter:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}Ax}{\vdash a, a^\perp, ?b^\perp}w? \qquad \dfrac{}{\vdash b, b^\perp}Ax}{\vdash a \otimes b, a^\perp, ?b^\perp, b^\perp}\otimes}{\vdash a \otimes b, a^\perp \wp ?b^\perp, b^\perp}\wp \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}Ax}{\vdash a, a^\perp, ?b^\perp}w?}{\vdash a, a^\perp \wp ?b^\perp}\wp \qquad \dfrac{}{\vdash b, b^\perp}Ax}{\vdash a \otimes b, a^\perp \wp ?b^\perp, b^\perp}\otimes$$

Thus, in the proof construction of a sequent $\vdash A \otimes B, C \wp D, \Gamma$ the choice to decompose $C \wp D$ first has a greater chance to be successful *i.e.* to be the choice that lead to a proof.

Example 2 (Completeness of some strategies).
Permutation properties can be used to either prove that, for a given set of inference rules, a given search strategy will find all possible proofs, or construct an example of a provable sequent which can not be proved with that strategy. This is specific for strategies which deal with the order of inference rules. Strategies which pass this test(s) may be amenable to further analysis, such as minimising the amount of branching in the proof, or delaying certain choices until the optimum amount of information is available.

We explain this on a simplified version of *"normal proof"* strategy whose full version is given in [3]. Let assume that, in the logical fragment we consider, the sequents are one-sided: $\vdash \Gamma$. Γ is a multiset of goal formulas, denoted G and defined by the following grammar:

$$G := A|G\otimes G|G\wp G|G\oplus G|\forall xG|\exists xG|?G$$

Let us assume that the allowed set of inference rules for proving the specified sequents is $\{\otimes, \wp, \oplus_1, \oplus_2, \forall, \exists, w?\}$. The strategy is defined by the following algorithm:

> **If** $\vdash \Gamma$ contains a formula of the form $?F$
> **then** $\vdash \Gamma$ is the conclusion of $w?$ rule
> **else**
> **if** $\vdash \Gamma$ contains a formula whose top-most connective is $\wp$ or $\forall$
> **then** $\vdash \Gamma$ is the conclusion of rule which introduces that particular formula
> **else** $\vdash \Gamma$ contains a formula whose top-most connective is $\otimes$ or $\oplus$ or $\exists$ and
> $\vdash \Gamma$ is the conclusion of rule which introduces that particular formula

The completeness properties of the outlined strategy follow directly from the permutability properties of the concerned linear logic fragment, which are given in Table 1. Any proof, from this logical fragment, can be transformed, by permutations of its inferences, to a proof that obeys the given strategy:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash p(t), p^\perp(t)}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}\,Ax}{\vdash a, a^\perp, ?b}\,w?}{\vdash a, a^\perp \wp ?b}\,\wp}{\vdash p(t)\otimes a, p^\perp(t), ?b\wp a^\perp}\,\otimes}{\vdash p(t)\otimes a, \exists y p^\perp(y), ?b\wp a^\perp}\,\exists}{\vdash \forall x(p(x)\otimes a), \exists y p^\perp(y), ?b\wp a^\perp}\,\forall \quad \longmapsto \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^\perp}\,Ax \qquad \dfrac{\dfrac{}{\vdash p(t), p^\perp(t)}\,Ax}{\vdash p(t), \exists y p^\perp(y)}\,\exists}{\vdash p(t)\otimes a, \exists y p^\perp(y), a^\perp}\,\otimes}{\vdash p(t)\otimes a, \exists y p^\perp(y), ?b, a^\perp}\,w?}{\vdash p(t)\otimes a, \exists y p^\perp(y), ?b\wp a^\perp}\,\wp}{\vdash \forall x(p(x)\otimes a), \exists y p^\perp(y), ?b\wp a^\perp}\,\forall$$

Example 3 (Generating equivalent proofs which obey different strategies).
A logic program for which there are many proofs of the same goal (and hence there are goals which return the same answer substitution many times) is generally considered to be somewhat deficient. Different proofs are only interesting if they lead to different answers.

As example, consider the sequent $p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)$. All possible uniform proofs[1] can be classified in two groups in regard to the answer substitution for x:

Group 1:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q(b)\vdash q(b)}\,Ax}{q(b), p(a)\vdash q(b)}\,w-L \qquad \dfrac{}{p(b)\vdash p(b)}\,Ax}{p(a), q(b)\Rightarrow p(b), q(b)\vdash p(b)}\,\Rightarrow -L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash p(b)}\,\forall -L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash \exists x p(x)}\,\exists -R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q(b)\vdash q(b)}\,Ax \qquad \dfrac{}{p(b)\vdash p(b)}\,Ax}{q(b)\Rightarrow p(b), q(b)\vdash p(b)}\,\Rightarrow -L}{\forall y q(y)\Rightarrow p(y), q(b)\vdash p(b)}\,\forall -L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash p(b)}\,w-L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash \exists x p(x)}\,\exists -R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q(b)\vdash q(b)}\,Ax \qquad \dfrac{}{p(b)\vdash p(b)}\,Ax}{q(b)\Rightarrow p(b), q(b)\vdash p(b)}\,\Rightarrow -L}{p(a), q(b)\Rightarrow p(b), q(b)\vdash p(b)}\,w-L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash p(b)}\,\forall -L}{p(a), \forall y q(y)\Rightarrow p(y), q(b)\vdash \exists x p(x)}\,\exists -R$$

[1] Uniform proof [10] of a sequent $\mathcal{P}\vdash G$ means a goal-directed proof, in a sense that the goal G is decomposed uniformly, based only on its structure and without reference to the program $\mathcal{P}$, until atomic goals are reached. The program is only consulted when atomic goals are to be proved.

Group 2:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p(a) \vdash p(a)}\,Ax}{p(a), q(b) \vdash p(a)}\,w-L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash p(a)}\,w-L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)}\,\exists-R \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{p(a) \vdash p(a)}\,Ax}{p(a), \forall y q(y) \Rightarrow p(y) \vdash p(a)}\,w-L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash p(a)}\,w-L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)}\,\exists-R$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{q(b) \vdash q(b)}\,Ax}{p(b), q(b) \vdash q(b)}\,w-L \qquad \dfrac{}{p(a) \vdash p(a)}\,Ax}{p(a), q(b) \Rightarrow p(b), q(b) \vdash p(a)}\,\Rightarrow -L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash p(a)}\,\forall-L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)}\,\exists-R$$

Clearly, the question of finding all proofs which lead to different answers have some overlap with the question of finding all equivalent proofs modulo inference permutations. Since proofs from the same equivalence class lead to the same answer, it is sufficient to generate just one of them, which than will be representative for the whole class. This is reduced to the problem of finding all permutations of the proof *i.e.* of finding the appropriate equivalence class, which than will be "excluded from further consideration".

Uniformity of proofs is a key strategy for the logic programming proofs. In the above example, in the uniform proofs which provide $x \leftarrow b$ substitution, the reasons for such substitution, which is performed (because of the uniformity constraint) in the first step of proof construction, are vague:

$$\dfrac{\begin{array}{c}\vdots\\ p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash p(b)\end{array}}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)}\,\exists-R$$

Such substitution is result of analysing $\forall y q(y) \Rightarrow p(y)$, $q(b)$ part of the program. Confronted with the question of substitution for the quantified variables, an implementation might relax the uniformity constraint, recognising the advantage of the application of some other strategy: in the outlined example the application of certain left rules before the right ones. One of possible (non-uniform) proofs that can result is:

$$\dfrac{\dfrac{\dfrac{}{q(b) \vdash q(b)}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{p(b) \vdash p(b)}\,Ax}{p(a), p(b) \vdash p(b)}\,w-L}{p(a), p(b) \vdash \exists x p(x)}\,\exists-R}{p(a), q(b) \Rightarrow p(b), q(b) \vdash \exists x p(x)}\,\Rightarrow -L}{p(a), \forall y q(y) \Rightarrow p(y), q(b) \vdash \exists x p(x)}\,\forall-L \qquad (*)$$

The generated proof is equivalent, modulo permutations, to the starting one. After the step $(*)$ the origin of the idea for the substitution $x \leftarrow b$ is more clear.

Permutations - some problems and solutions There are techniques to perform an appropriate permutation analysis and transformations of a proof but they are all designed for analysis on paper by a human rather than by machine. Also, all existing techniques have been adapted to particular sets of inference rules. For permutations there is no general framework capable of handling permutabilities of inferences for different logical fragments and different proof systems. There are many situations where the permutabilities can not be explained by the existing definitions and conditions. One such situation is illustrated by the example below, where $\wp'$ presents a mix of $\wp$ and w rules from linear logic, and where the possible permutation of $c?$ and $\wp'$ in the outlined subproof can not be detected using the existing definitions ([7] or Definitions 2.1-2.4 of [3]):

$$\underset{\text{schema-rule}}{\dfrac{\vdash \phi, \psi, \Gamma}{\vdash \phi\wp'\psi, \Gamma, ?\delta}\,\wp'} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash ?A, ?A, B, \Gamma\end{array}}{\vdash ?A, B, \Gamma}\,c?}{\vdash ?A\wp' B, \Gamma, ?A}\,\wp' \quad \Longrightarrow \quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash ?A, ?A, B, \Gamma\end{array}}{\vdash ?A, ?A\wp' B, \Gamma, ?A}\,\wp'}{\vdash ?A\wp' B, \Gamma, ?A}\,c?$$

It is our contention that a more precise specification of the structure of sequent calculus rules is needed, as it is proposed by the following definition:

Definition 1. The structure of sequent calculus inference rules:

- An *active formula* of rule I is a formula in a premise, that does not exist in the conclusion. The *active part* of the i-th premise of an inference I (denoted by $\mathcal{A}_I^i$) is the (possibly empty) multiset of its active formulas.
- A *quasi-active formula* of an inference I is a formula occurrence whose presence in a premise is necessary (but not sufficient[2]) for the rule application, and which is copied unchanged into the conclusion of the rule. The *quasi-active part* of the i-th premise of an inference I (denoted by $\mathcal{QA}_I^i$) is the (possibly empty) multiset of its quasi-active formulas.
- A *principal formula* of rule I is a formula occurrence of a conclusion that does not exist in premise(s) and that is a result of the disappearance of some active formulas of I. The *principal part* of an inference I (denoted by $\mathcal{P}_I$) is the (possibly empty) multiset of its principal formulas.
- A formula occurrence of rule I that does not exist in a premise and that is not a principal formula, is called an *extra formula*. The *extra part* of an inference I (denoted by $\mathcal{E}_I$) is the (possibly empty) multiset of its extra formulas.
- The *context* of the i-th premise of an inference I (denoted by C_I^i) is the (possibly empty) multiset - complement of its active and quasi-active part.

The above definition is illustrated through examples in the Table 2 below:

Table 2.

	$\mathcal{A}^1$	$\mathcal{A}^2$	$\mathcal{QA}^1$	$\mathcal{QA}^2$	$\mathcal{P}$	$\mathcal{E}$	C^1	C^2
$\dfrac{\vdash \Gamma \quad \vdash \Delta}{\vdash \Gamma, \Delta}\,mix$							Γ	Δ
$\dfrac{\vdash A, \Gamma \quad \vdash B, \Delta}{\vdash A \otimes B, \Gamma, \Delta}\,\otimes$	A	B			$A \otimes B$		Γ	Δ
$\dfrac{\vdash A, \Gamma \quad \vdash B, \Gamma}{\vdash A \& B, \Gamma}\,\&$	A	B			$A \& B$		Γ	Γ
$\dfrac{\vdash A, \Gamma \quad \vdash B, \Delta}{\vdash \Gamma, \Delta}\,cut$	A	$A^\perp$					Γ	Δ
$\dfrac{\vdash \Gamma}{\vdash \Gamma, ?F}\,w?$						$?F$	Γ	
$\dfrac{\vdash ?F, ?F, \Gamma}{\vdash ?F, \Gamma}\,c?$	$?F$		$?F$				Γ	

[2] except for the exchange rule

According to the context of premises and conclusion, binary rules can be distinguished as: *multiplicative* rules where the context of each premise is copied unchanged into the conclusion (rules $\otimes, mix, cut$ in above table); *strong additive* rules where the context of each premise and of the conclusion are identical (rule &). Note that there are also *weak additive* rules where some part of the context of the premise and of the conclusion are identical.

We use the proposed characterisation of the structure of sequent calculi rules for the automated-oriented specification of the permutation process. Here we outline some results which have been described in details and proved formally in [8]:

- Our framework characterizes arbitrary sets of inference rules;
- The active, quasi-active, principal and extra part of an inference rule are not change by its permutation up or down, in a given proof. The context part of an inference can change during inference permutation. The appropriate changes in the context have been specified in detail (see [8]);
- "Pre-conditions" for a permutation have been focused and restricted to particular parts of inference rules and to minimally sufficient tests.
- Some dependencies between the ways of permuting rules and their generic properties and the characteristic of a given proof have been specified and proved. For example, the *upward permutation* of a strong-additive rule requires a particular proof environment. Also, the form of the permutation result depends of the generic properties of the rules in question and therefore, can be determined independently of any particular permutation.

2.2 Hypothesis elimination

This is a transformation of a *near proof* $\mathcal{D}$(derivation where not all leaves are axioms) to the actual proof which retains (at least relatively) the order of inference rules of $\mathcal{D}$ and whose conclusion is identical or contains a conclusion of $\mathcal{D}$. This is achieved by the replacement of each non-axiom leaf $\vdash S$ with a proof of $\vdash S, \Omega$ (where Ω is possibly empty multiset of formulas dependent on S and on the structure of inference rules of a given proof system) and then by, eventually, *adaptation* of subproof below $\vdash S$.

In the resulting proof of the below example the adaptation implies inserting some intermediate steps (inserted inferences are boxed).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^{\perp}}\,Ax \qquad \vdash a, b^{\perp}}{\vdash a, a^{\perp}\& b^{\perp}}\,\&}{\vdash a \oplus b, a^{\perp}\& b^{\perp}}\,\oplus}{\vdash (a \oplus b)\wp(a^{\perp}\& b^{\perp})}\,\wp \quad \longmapsto \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash a, a^{\perp}}\,Ax}{\vdash a, a^{\perp}, ?(a^{\perp}\otimes b)}\,\boxed{\mathrm{w}} \qquad \dfrac{\dfrac{\dfrac{}{\vdash a, a^{\perp}}\,Ax \quad \dfrac{}{\vdash b, b^{\perp}}\,Ax}{\vdash a, b^{\perp}, a^{\perp}\otimes b}\,\boxed{\otimes}}{\vdash a, b^{\perp}, ?(a^{\perp}\otimes b)}\,\boxed{?}}{\vdash a, a^{\perp}\& b^{\perp}, ?(a^{\perp}\otimes b)}\,\&}{\vdash a \oplus b, a^{\perp}\& b^{\perp}, ?(a^{\perp}\otimes b)}\,\oplus}{\vdash (a \oplus b)\wp(a^{\perp}\& b^{\perp}), ?(a^{\perp}\otimes b)}\,\wp$$

If we allow interpretation of a non-axiom leaf $\vdash S$ as a hypothesis of, say, a proof $\mathcal{D}$ then the above transformation is known as *hypotheses elimination* [2].

Axioms are absolute truth or static knowledge. *Hypotheses* are auxiliary assumptions or lemmas of a proof which present an extra-logical knowledge in the language of a particular logical system. They are a temporary knowledge that is assumed to be true in some stages of the deduction, or belong to a catalogue of already proven results.

There are several potential benefits of allowing hypotheses in proofs. First, for the analysis of meta-properties of proofs it could be useful to reduce the branching factor at some search node or to replace some branches or sub-proofs (whose investigation is not of interest) with sequent(s) that is(are) proven in those branches/sub-proofs. It could be also useful to adapt proof systems to retain the results of a previous proof search and to apply and combine them to a new situation. Information whether a given proof can be relieved of hypothesis and what the resulting proof looks like is important. They have a direct impact, for example, on a Cut elimination because mostly, in the systems with a Cut elimination property, the Cut can be eliminated from proofs without hypotheses and just reduced to a certain normal form for proofs with hypotheses.

Elimination of a hypothesis $\vdash \mathcal{H}$ from a proof Π is the problem of searching for a proof (without any hypotheses) of hypothesis $\vdash \mathcal{H}$, which will replace it in Π. For decidable proof systems it is possible to find, at least, one such proof iff $\vdash \mathcal{H}$ is provable. It seems reasonable to expect that in a decidable proof system every provable hypothesis $\vdash \mathcal{H}$ can be eliminated and replaced by its proof. The situation is more complicated when a hypothesis $\vdash \mathcal{H}$ (or $\mathcal{H}_2 \vdash \mathcal{H}_1$) is an unprovable sequent, or a given proof system is not decidable. It is interesting, in such situations, to consider the finding of the simplest proof (without hypotheses) whose conclusion incorporates $\mathcal{H}$ $(\mathcal{H}_1, \mathcal{H}_2)$. It is obvious, that this process depends on the hypothesis itself, as well as on the structure of inference rules of a given proof system. Andreoli has shown this on the example of one fragment of propositional classical logic [2].

Developing general methods for hypothesis elimination is research still in progress. It is possible to generalise some dependences and to extract certain, general structures of rules that are *suitable* and *useful* for hypothesis elimination. For example, the structure of weakening rule (in LK), the structure of mix rule, structure of some multiplicative rules, etc.

Also, given the considerations above we can address another interesting question: what is the minimal extension (that is sufficient, a guarantee) to provide provability of a given unprovable sequent $\mathcal{S}$ of a given proof system? The example below points out some possible extensions and additions of the unprovable sequent $\vdash p, q$ in a different logic:

classical logic: two-sided linear logic:

$$\dfrac{\dfrac{}{p \vdash p}\,Ax}{p \vdash p, q}\,w \qquad \dfrac{\dfrac{}{p \vdash p}\,Ax \quad \dfrac{}{q \vdash q}\,Ax}{p \wp q \vdash p, q}\,\wp \qquad \dfrac{}{\vdash p, q, \top}\,\top \qquad \dfrac{}{0 \vdash p, q}\,0$$

3 Conclusions and Future Work

We have outlined the first steps in the investigation of the automation of the design of LP languages. The aim is to develop a logic-independent system, which will contain various implementing and proof-theoretics techniques together with the general strategic planner, capable of determining the most appropriate way to assemble the various pieces into a LP system, but in a formalism and framework which is designed and intended for humans.

The first steps towards such an automated analyser are dedicated to machine-oriented specifications for the encapsulation, manipulation and automation of existing LP strategies. Various properties of search strategies have direct influence on the design of LP languages. We exploit some proof manipulations and their usage in the analysis of the particular search strategies. We have proposed a more detailed specification of inference rules, for a better understanding of their structure and their generic properties. Making use of the generic properties of inference rules rather than particular instances of rules will allow analyses on various logics and various classes of formulae within each logic. On the basis of introduced specifications we have generalised some definitions and meta-rules for the permutation analyses. Also, it seems that it is possible to exploit some similarities in the structure of sequent calculi inference rules from different proof systems for the extenuation and generalisation of some strategies or techniques originally designed and dedicated to particular sets of rules. For example, proof-nets, Girard's "parallel syntax for proof theory" [4], are among the most fascinating constructions for a concise graph-theoretic representation of deductions by "decontextualization of inferences". They have arisen and are based on some fragments of linear logic, principally first order $\otimes, \wp, cut$ and $\otimes, \wp, cut, mix$ fragments. The question that arises is: are the same principles for the construction and analysis of proof-nets in, say, $\wp, cut, mix$ fragment of linear logic applicable in $\vee, cut, mix$ fragment of LK, in regard to the same structure (same number of active, quasi-active, principal and extra formulas, the same type of restrictions on the structure of involved formulas and multisets, etc) of rules:

$$\text{CLL:}\quad \frac{\vdash A, B\ ;\ ;\ \Gamma\ ;}{\vdash A \wp B\ ;\ ;\ \Gamma\ ;}\ \wp \qquad\qquad \text{LK}\quad \frac{\vdash A, B\ ;\ ;\ \Gamma\ ;}{\vdash A \vee B\ ;\ ;\ \Gamma\ ;}\ \vee$$

References

1. Andreoli J.-M.: Logic Programming with focusing proofs in linear logic, Journal of Logic and Computation, 2(3) (1992) 297-347.
2. J.M. Andreoli, R. Pareschi: Logic programming with sequent systems: A linear Logic Approach. In P.Schröder-Heister ed., Proceedings of Workshop to Extensions of Logic Programming, Tübingen, LNAI 475 (1989) 1-30.
3. Galmiche D. and Perrier G.: On proof normalisation in Linear Logic, Theoretical Computer Science 135 (1994) 67-110.
4. Girard J.-Y.: Proof-nets: the Parallel syntax for Proof-theory, Logic and Algebra, Marcel Dekker, New York (1995).
5. Harland J.: A proof-theoretic Analysis of Goal-directed Provability, Journal of Logic and Computation, Vol.4 No 1 (1994) 69-88.
6. Hodas J.S. and Miller D. *Logic programming in a Fragment of Intuitionistic Linear Logic*, Journal of Information and Computation 110:2 (1994) 327-365.
7. Kleene S.C.: Permutability of Inferences in Gentzen's Calculi LK and LJ, Memoirs of the American Mathematical Society 10 (1952) 1-26.
8. Lutovac T., Harland J.: Towards the Automation of the Design of Logic Programming Languages, Technical Report TR-97-30, Department of Computer Science, RMIT University (1997).
9. Miller D.: A logical analysis of modules in logic programming, Journal of Logic Programming 6 (1989) 79-108.
10. Miller D., Nadathur G., Pfenning F., Ščedrov A.: Uniform proofs as a Foundation for Logic Programming, Annals of Pure and Applied Logic 51 (1991) 125-157.
11. Miller D.: Forum: A multiple-conclusion specification-logic, Theoretical Computer Science 165, 201-232.
12. Nadathur G. and Miller D.: An overview of λProlog, In Kenneth A. Bowen and Robert A. Kowalski, editors, Fifth International Logic Programming Conference, Seattle, Washington, August 1988, MIT Press, 810-827.
13. Pym J.D., Harland A.J.: A Uniform Proof-theoretic Investigation of Linear Logic Programming, Journal of Logic and Computation, Vol. 4 No 2, (1994) 175-207.
14. Pfenning F.: Analytic and Non-analytic Proofs, in M.E. Stickel ed., 10th International Conference on Automated Deduction, Kaiserslautern, FRG, LNAI 449 (1990) 394-413.
15. Sterling L., Shapiro E.: The Art of Prolog, MIT Press, 1986.
16. Tammet T.: Proof Search Strategies in Linear Logic, Journal of Automated Reasoning no. 12, (1994) 273-304.
17. Winikoff M., Harland J.: Implementing the Linear Logic Programming Language Lygon, Proceedings of the International Logic Programming Symposium, Portland, Oregon, December, 1995, 66-80.
18. Winikoff M., Harland J.: Deriving Logic Programming Languages, Technical Report 95/26, Department of Computer Science, The University of Melbourne, (1996).

Machine-independent evaluation of theorem-proving strategies

(Position paper)

Maria Paola Bonacina *

Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu

The evaluation of theorem-proving strategies has been done traditionally in an empirical manner: a strategy is implemented in a theorem prover, the prover is applied to a number of theorems, and the running times are reported and compared with those of other systems. In recent years, a growing effort has been devoted to make the evaluation of theorem provers more systematic. The need for a standard collection of theorem-proving problems (e.g., the TPTP library [9]) and a standard set of empirical measures has been recognized (e.g., [8]).

While benchmarking of theorem provers is necessary, and the progress in the methodology of empirical evaluation is important for the field, the problem of *strategy evaluation* remains open. A theorem prover is made of many components in addition to the strategy, including data structures, indexing techniques and service algorithms such as those for unification or term replacement. The performance of a theorem prover depends on all these components and the overall engineering of the system. It is very difficult to establish quantitatively how different features contribute to the observed performance. Therefore, empirical evaluation is evaluation of theorem-proving systems, not theorem-proving strategies. The goal of evaluating strategies independent of implementation requires the development of a theory of *"strategy analysis,"* comparable to algorithm analysis, and with potentially similar beneficial consequences, not only for theorem proving, but also for logic programming and all applications of deduction.

The idea of "strategy analysis" is new. Most of the work on search in artificial intelligence concentrates on the design of heuristics (e.g., [5]). Most of the research in complexity related to theorem proving studies the complexity of propositional proofs as part of the quest for $NP \neq co\text{–}NP$ (e.g., see [10] for a survey), or works with complexity measures based on the Herbrand theorem to determine lower bounds for sets of clauses, not upper bounds for strategies (e.g., [2, 4, 7]). In resolution theorem proving, the classical source for the modelling of search is [3], which was not concerned with evaluating the complexity of the strategies.

The primary objective of strategy analysis is to study the complexity of *searching for a proof.* An approach to this problem was proposed in [6]. It applies classical techniques from algorithm analysis to derive worst-case upper bounds

* Supported in part by the National Science Foundation with grant CCR-94-08667.

on the total size of the search space of various theorem-proving strategies in propositional Horn logic. This approach assumes that the search space is *finite*, so that presently it is not known how it could be extended to first-order logic.

An approach that aims at *infinite search spaces* was presented in [1]. The first component of a methodology for strategy analysis is a *model of the search space* and *search process* (i.e., the application of the search plan to the search space). The second step is to define *measures of complexity* in such a model, so that strategies can be evaluated and compared in terms of such measures. The third step is to carry out the analysis. In [1], the *model of the search space* is a *marked search graph*. The search graph represents all possible inferences. Unlike expansion inferences, however, contraction inferences, such as simplification or subsumption, cannot be represented by a static graph. Expansion inferences *visit* the graph. Contraction inferences *visit and modify* the graph by deletions, which cannot be represented prior to the search. Therefore, the search graph is enriched with a *marking*. The marking is used to represent dynamically the *search process*, including the selections by the search plan, the expansion steps and the contraction steps.

This model of the search space supports measures of complexity of search for infinite search spaces. The classical notion of complexity is complexity of a computation which is guaranteed to halt, and therefore deals with finite objects. For derivations that may not halt, the approach of [1] is to observe that while a strategy operates with a finite amount of data, this data represents a portion of the infinite space that the strategy is searching. In order to capture the complexity of the search, the analysis needs to involve both the *present* and the *future* of the derivation. In terms of the search graph, the present is the part explored so far, and the future is the unexplored part. The analysis needs to study how the inferences selected by the strategy affect both.

Complexity measures for this purpose are obtained by defining notions of *ancestor-graph* and *dynamic distance*, which replace the conventional notions of path and path-length. Unlike the latter, the distance is dynamic, because it depends on the actions of the strategy. For instance, if present clauses are deleted by contraction, clauses in the future may become *unreachable*. Given this notion of distance, it is possible to define the *bounded search space* with bound j as the space of clauses reachable within distance j. The infinite search space is viewed as an infinite succession (for all j) of bounded search spaces. Since the bounded search spaces are finite, they can be compared. Furthermore, they are characterized as *multisets of clauses*, so that it is possible to compare them with the multiset extension of a *well-founded ordering on clauses*. Thus, the notion of bounded search space allows us to handle infinite search spaces finitely.

In [1] this framework is applied to compare two generic strategies with the same expansion rules, the same search plan, but different contraction rules. It is shown that contraction reduces the bounded search spaces, and therefore the search complexity. Then it is proved that the strategy with more contraction induces a higher reduction of search complexity. This is the first analysis that translates the empirical observations of the efficacy of contraction in theorem proving into a formal result of reduction of complexity.

At present, this type of analysis is not asymptotic. Possible directions for future research include exploring approaches to make the analysis asymptotic, comparing strategies with different search plans, extending the framework from sequential to parallel search, and from forward-reasoning to backward-reasoning.

References

1. M. P. Bonacina and J. Hsiang. On the representation of dynamic search spaces in theorem proving. In C.-S. Yang, editor, *Proc. of the Int. Computer Symposium*, pages 85–94, 1996. Abstract presented at the *Logic Colloquium 1996*, and full version available as Tech. Rep. 95-05, Dept. of Comp. Sci., Univ. of Iowa.
2. J. Goubault. The complexity of resource-bounded first-order classical logic. In *Proc. of the 11th STACS*, volume 775 of *LNCS*, pages 59–70. Springer Verlag, 1994.
3. R. Kowalski. Search strategies for theorem proving. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 181–201. Edinburgh University Press, 1969.
4. V.-P. Orevkov. Lower bounds for increasing complexity of derivations after cut elimination. *J. of Soviet Mathematics*, pages 2337–2350, 1982.
5. J. Pearl. *Heuristics – Intelligent search strategies for computer problem solving.* Addison Wesley, Reading, MA, 1984.
6. D. A. Plaisted. The search efficiency of theorem proving strategies. In A. Bundy, editor, *Proc. of CADE-12*, volume 814 of *LNAI*, pages 57–71. Springer Verlag, 1994. Full version available as Tech. Rep. MPI I-94-233.
7. R. Statman. Lower bounds on Herbrand theorems. In *Proc. AMS 75*, pages 104–107, 1979.
8. G. Sutcliffe and C. Suttner. The design of the CADE-13 ATP system competition. Technical Report 95/15 and AR-95-05, Dept. of Comp. Sci., JCU, Australia and Inst. für Informatik, TU München, Germany, 1996.
9. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In A. Bundy, editor, *Proc. of CADE-12*, volume 814 of *LNAI*, pages 252–266. Springer Verlag, 1994.
10. A. Urquhart. The complexity of propositional proofs. *Bull. of Symbolic Logic*, 1:425–467, 1995.

Flexible Re-enactment of Proofs*

Matthias Fuchs

Center for Learning Systems and Applications (LSA)
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern, Germany
E-mail: fuchs@informatik.uni-kl.de

Abstract. We present a method for making use of past proof experience called flexible re-enactment (FR). FR is actually a search-guiding heuristic that uses past proof experience to create a search bias. Given a proof $\mathcal{P}$ of a problem solved previously that is assumed to be similar to the current problem $\mathcal{A}$, FR *searches* for $\mathcal{P}$ and in the "neighborhood" of $\mathcal{P}$ in order to find a proof of $\mathcal{A}$.
This heuristic use of past experience has certain advantages that make FR quite profitable and give it a wide range of applicability. Experimental studies substantiate and illustrate this claim.

1 Introduction

Automated deduction is essentially a search problem that gives rise to potentially infinite search spaces because of general undecidability. Despite these unfavorable conditions state-of-the-art theorem provers have gained a remarkable level of performance mainly due to (problem-specific) search-guiding heuristics and advanced implementation techniques. Nevertheless theorem provers can hardly rival a mathematician when it comes to proving "challenging" theorems. The main reason for this fact is also a major shortcoming of theorem provers: Unlike humans, theorem provers lack the ability to learn. Learning, however, is a key ability in any form of human problem solving, in particular in theorem proving.

The necessity to equip theorem provers with learning capabilities has been recognized quite early. But learning for theorem proving causes much more difficulties than learning in other areas of artificial intelligence because the premise that "*small changes of the problem cause small changes of its solution*" is not fulfilled at all. In theorem proving tiny variations of a problem specification can result in significant changes of the solution (proof). This circumstance complicates matters substantially.

Learning methods based on analogous proof transformation (e.g., [3, 2, 14, 18]) basically attempt to transform a *source proof* of a *source problem* solved previously into a *target proof* of a given *target problem* to be solved. Mostly, the mainly *deterministic* transformation procedure centers on some kind of analogy mapping obtained by comparing source and target problem. The transformation

* This work was supported by the *Deutsche Forschungsgemeinschaft (DFG)*.

may include abstraction and planning steps. Occasionally, a little search may be involved in order to patch failures of the transformation procedure (e.g., [2]).

The prevailing determinism has the advantage that analogous proof transformation can be quite fast. The downside, however, is that source and target problem have to be very similar so that the source proof can be transformed into a target proof mainly with deterministic actions. For this reason, methods based on analogous proof transformation are preferably applied to inductive theorem proving. There, the inherent proof structures provide a suitable platform for such methods. However, for theorem proving in domains without any (recognizable) structure, these methods appear to be inappropriate.

An alternative approach to learning for theorem proving is to incorporate problem-solving experience into the search-guiding heuristic (e.g., [20, 23, 8, 11, 6]). That is, when solving a target problem, the source problem is in some way exploited by the search-guiding heuristic. The theorem prover still conducts a search, but the heuristic is biased towards a certain area of the search space depending on the source proof and on the way it is utilized by the heuristic. This approach has the advantage that the demands on similarity between source and target problem do not have to be as high as for methods based on proof transformation. Naturally we incur the usual overhead of search caused by exploring areas of the search space that do not contribute to the proof eventually found.

In this paper we investigate a method for incorporating past proof experience into a search-guiding heuristic called *flexible re-enactment* (`FR`). `FR` basically attempts to re-enact the source proof *via search*, *if* possible. Flexibility is achieved by also searching in the "neighborhood" of the source proof or by using some "standard" (non-learning) heuristic if the trace of the source proof is (temporarily) lost. Both re-enactment and flexibility are combined in a monolithic structure that allows for shifting smoothly between flexibility and re-enactment depending on the search space encountered.

`FR` was first introduced in [11]. In this paper we present a systematic experimental evaluation of `FR` that illustrates its performance and range of applicability. Furthermore, we counter a frequent and unjustified criticism of `FR`, namely to be a costly disguise for adding the source proof (in the form of lemmas) to the axiomatization of the target problem.

2 Theorem Proving

Theorem provers can attempt to accomplish a task in various ways. We focus here on so-called *saturation-based theorem provers*. This type of theorem prover is very common and is employed by provers based on the resolution method (e.g., [4]) or the Knuth-Bendix completion procedure (e.g., [1]). The principle working method of such a prover is to infer facts by applying given rules of inference, starting with a given set Ax of axioms, until the *goal* λ_G (the theorem to be proved) can be shown to be a logical consequence of Ax. A proof problem $\mathcal{A}$ is hence specified by $\mathcal{A} = (Ax, \lambda_G)$.

The prover maintains two sets of facts, the set F^A of *active facts* and the set F^P of *passive* or *potential facts*. In the beginning, $F^A = \emptyset$ and $F^P = Ax$. In the *selection* or *activation step* a fact $\lambda \in F^P$ is selected, removed from F^P, and put into F^A unless there is a fact $\lambda' \in F^A$ that subsumes λ (in symbols $\lambda' \lhd \lambda$) in which case λ is simply discarded. Note that $\lambda \equiv \lambda'$ (syntactic identity modulo renaming variables) implies $\lambda' \lhd \lambda$ (and of course $\lambda \lhd \lambda'$). If λ is indeed activated (put into F^A), all (finitely many) inferences involving λ are applied exhaustively, and inferred facts are added to F^P. Facts in F^P are so to speak known to be inferable (from F^A), but are not yet considered to be actually inferred.

The activation step is the only inherently indeterministic step in the proof procedure just sketched. Commonly, a heuristic $\mathcal{H}$ is employed to resolve the indeterminism at the (inevitable) expense of introducing search. $\mathcal{H}$ associates a natural number $\mathcal{H}(\lambda) \in \mathbb{N}$ (a *weight*) with each $\lambda \in F^P$ which is called "weighting λ with $\mathcal{H}(\lambda)$". The fact with the smallest weight $\mathcal{H}(\lambda)$ is next in line for activation. Ties are usually broken in compliance with the FIFO strategy.

The search-guiding heuristic $\mathcal{H}$ is pivotal for efficiency. The quality of $\mathcal{H}$ can be measured in terms of redundant search effort. The sequence $\mathcal{S} \equiv \lambda_1; \ldots; \lambda_n$ of facts activated by $\mathcal{H}$ describes the search behavior of $\mathcal{H}$. Assuming that such a *search protocol* $\mathcal{S}$ represents a successful search, the last fact λ_n of $\mathcal{S}$ concluded the proof. By tracing back the application of inference rules starting with λ_n we can identify all those facts that actually contributed to concluding the proof. These facts are called *positive facts* and are collected in the set P of positive facts. The remaining facts constitute the set N of negative facts and represent redundant search effort. The set P obtained when solving a source problem represents past proof experience. It is utilized by `FR` as described in the following section in order to reduce redundant search effort.

3 Flexible Re-enactment

Similarity between two proof problems $\mathcal{A}$ and $\mathcal{B}$ can occur in many variations. For instance, a considerable number of the facts that contribute to a proof of $\mathcal{A}$ are also useful for proving $\mathcal{B}$ (or vice versa). This means in our terminology that the associated sets of positive facts share many facts. `FR` attempts to exploit such a similarity.

Assuming that a target problem $\mathcal{A}_T$ and a source problem $\mathcal{A}_S$ are similar in the way just described, it is reasonable to concentrate on deducing facts when attempting to prove $\mathcal{A}_T$ that also played a role in finding the source proof of $\mathcal{A}_S$, namely the set P_S of positive facts. When proving $\mathcal{A}_T$, `FR` prefers a $\lambda \in F^P$ (by giving it a relatively small weight $\mathtt{FR}(\lambda)$) if λ "bears significant resemblance" with a fact in P_S. Such a λ is called a *focus fact*. Depending on how strong the preference of focus facts is, `FR` will activate focus facts as soon as they appear in F^P and give little attention to other facts while there are focus facts. (If all source axioms occur in the target axiomatization, this essentially means that `FR` will re-enact the source proof.)

Although we assume the proofs of $\mathcal{A}_T$ and $\mathcal{A}_S$ to share many positive facts, a few of the facts useful for proving $\mathcal{A}_S$ may lead to focus facts that do not contribute to the proof of $\mathcal{A}_T$ eventually found. Besides the redundant search effort caused by these *irrelevant* focus facts, the crucial difficulty is to find the (non-focus) facts not needed for proving $\mathcal{A}_S$, but necessary for proving $\mathcal{A}_T$. These "missing" facts have to supplement the *relevant* focus facts, i.e., the focus facts that do contribute to the target proof. It is very likely that the missing facts are descendants[2] of relevant focus facts considering that the relevant focus facts already established the core of a proof and that there are only a few steps needed to conclude the proof. It is reasonable to assume that "taking the reasoning process a few steps further in the direction given by the focus facts", i.e., that also preferring their descendants will be profitable. Immediate descendants should be preferred the most. In the sequel we give a formal definition of FR.

3.1 Details of FR

For the definition of FR the notions 'difference' and 'distance' are pivotal. First, we define the *difference* *diff* between two facts λ and λ' to formalize the notion 'focus fact'.

$$\mathit{diff}(\lambda,\lambda') = \begin{cases} 0, & \lambda \lhd \lambda' \\ 100, & \text{otherwise.} \end{cases}$$

For the time being, we content ourselves with this simple definition of difference that centers on subsumption. Note that the values 0 and 100 are somewhat arbitrary but intuitive hints of percentages, denoting "perfect similarity" and "no similarity at all", respectively. As we shall see, the restriction of *diff* to $\mathbb{N}_{100} = \{0, 1, \ldots, 100\}$ entails that all further computations will produce values from $\mathbb{N}_{100}$ which makes computations more transparent.

diff is used to find out whether a given fact λ is a focus fact. We define

$$\mathcal{D}(\lambda) = \min\Big(\{\mathit{diff}(\lambda,\lambda') \mid \lambda' \in P_S\}\Big)\,.$$

Hence, $\mathcal{D}(\lambda)$ returns the minimal difference between a given fact λ (target) and the positive facts (source). If $\mathcal{D}(\lambda) = 0$ then λ is considered to be a focus fact, which complies with the ideas from above.

Now recall that also descendants of focus facts are to be favored. The preference given to them should, however, decrease with their distance from focus facts. The *distance* $d(\lambda)$ of a given fact λ measures distance, roughly said, in terms of the number of inference steps separating λ from ancestors which are focus facts. It depends on the distance of the immediate ancestors of λ from focus facts (if λ is not an axiom) and $\mathcal{D}(\lambda)$:

$$d(\lambda) = \begin{cases} \psi\Big(q, \mathcal{D}(\lambda)\Big), & \lambda \text{ is an axiom} \\ \psi\Big(d(\lambda_1), \mathcal{D}(\lambda)\Big), & \lambda_1 \text{ is the (only) immediate ancestor of } \lambda \\ \psi\Big(\gamma\big(d(\lambda_1), d(\lambda_2)\big), \mathcal{D}(\lambda)\Big), & \lambda_1 \text{ and } \lambda_2 \text{ are immediate ancestors of } \lambda. \end{cases}$$

[2] When applying an inference rule to facts $\lambda_1, \ldots, \lambda_m$, thus producing a fact λ, the facts $\lambda_1, \ldots, \lambda_m$ are the (*immediate*) *ancestors* of the (*immediate*) *descendant* λ.

The first argument of ψ represents the distance of the immediate ancestors. It is simply given as the distance of the immediate ancestor if there is just one immediate ancestor. If λ is an axiom (i.e., there are no ancestors), this value is specified by a parameter $q \in \mathbb{N}_{100}$. If λ has two immediate ancestors, then γ computes this value. (Note that $d(\lambda)$ is defined if λ has zero, one, or two immediate ancestors which is sufficient for most deduction systems. If the number of immediate ancestors should be in excess of two, then a weighted average of the ancestors' distances can be employed, for instance.) We chose a parameterized γ employing a parameter $q_1 \in [0;1]$. Depending on q_1, the result of γ ranges between the minimum and the maximum of the distances of the immediate ancestors:

$$\gamma(x,y) = \min(x,y) + \lfloor q_1 \cdot (\max(x,y) - \min(x,y)) \rfloor \ .$$

Using $q_1 = 0$ or $q_1 = 1$, γ computes the minimum or maximum, respectively. With $q_1 = 0.5$, γ computes the (integer part of the)[3] average.

The distance of immediate ancestors (or q) and $\mathcal{D}$ are combined by ψ yielding $d(\lambda)$. ψ should—for obvious reasons—satisfy the following criteria: On the one hand, $d(\lambda)$ should be minimal (i.e., 0) if $\mathcal{D}(\lambda) = 0$, in which case λ itself is a focus fact. On the other hand, the value produced by ψ should increase (reasonably) with the values obtained from γ and $\mathcal{D}$ in order to reflect the (growing) remoteness of λ from focus facts (and, in a way, from the source proof). As a matter of fact, γ already satisfies the latter criterion. Therefore, ψ is in parts identical to γ. It also uses a parameter $q_2 \in [0;1]$.

$$\psi(x,y) = \begin{cases} 0, & y = 0 \\ \min(x,y) + \lfloor q_2 \cdot (\max(x,y) - \min(x,y)) \rfloor, & \text{otherwise}. \end{cases}$$

The remaining task consists in designing FR so that it offers a reasonable degree of specialization in (i.e., focus on) the source proof that is paired with an acceptable degree of flexibility, i.e., the ability to cope with a target problem with a proof that requires minor to moderate deviations from the source proof. The use of d already provides sufficient specialization by directing the search towards the source proof. Its rudimentary flexibility can be enhanced by combining it with some "standard" ("general purpose") heuristic $\mathcal{H}$. Among several sensible alternatives we picked the following:

$$\mathtt{FR}(\lambda) = (d(\lambda) + p) \cdot \mathcal{H}(\lambda) \ , \quad p \in \mathbb{N}.$$

The parameter p controls the effect of $d(\lambda)$ on the final weight $\mathtt{FR}(\lambda)$. $d(\lambda)$ will be dominant if $p = 0$. In this case, if $d(\lambda) = 0$, $\mathtt{FR}(\lambda)$ will also be 0 regardless of $\mathcal{H}(\lambda)$. As p grows, $\mathcal{H}$ increasingly influences the final weight, thus mitigating the inflexibility of the underlying method, namely using $d(\lambda)$ *alone* as a measure for the suitability of a fact λ. For very large p, the influence of $d(\lambda)$ becomes negligible, and FR basically degenerates into $\mathcal{H}$.

[3] We restrict our computations to $\mathbb{N}$, because there is no gain in "high precision arithmetic" when dealing with weighting functions, but there would be a loss in efficiency w.r.t. computation time.

3.2 Range of Applicability

FR is most suitable in situations where each source axiom is "covered" by a target axiom, i.e., for each source axiom λ' there is a target axiom λ so that $\lambda \lhd \lambda'$. Then FR can conduct a search guided by the source proof in the sense that FR can re-enact the source proof (or a more general proof) *without search* since, for each positive fact, there will be a focus fact subsuming it. The re-enacted source proof or, more precisely, the focus facts constituting the source proof, can serve as a basis for finding the target proof by searching in the "neighborhood" of the source proof, i.e., in particular by focusing on immediate descendants of focus facts (cp. subsection 4.2, case 1). (If additionally the source goal λ_S subsumes the target goal λ_T then plain re-enactment will succeed.) Under these conditions it is sound to add the positive facts as lemmas to the target axiomatization. Therefore, FR has often been mistaken for a costly disguise for adding lemmas. Subsection 4.2 will show that this criticism is not justified. Moreover, subsection 4.2 will demonstrate that FR is also useful in case source and target axiomatizations do not agree "obviously", i.e., agreement (logical equivalence) cannot be checked with simple (syntactic) subsumption criteria. Under these conditions it is not sound to add the positive facts of the source proof to the target axiomatization because it is not known whether they are logical consequences of the target axiomatization.

Hence, FR is very versatile and covers a wide range of applicability. Nevertheless we want to point out that FR does of course not prove useful if source and target problem are not "similar enough", i.e., the sets of positive facts do not share enough facts. In other words, there are too few relevant focus facts that do not suffice to supply the core of a target proof. In addition, too many irrelevant focus facts hamper the search. (Naturally, relevant and irrelevant focus facts can only be identified at the end of a successful search.) In this case focusing on the source proof will be counterproductive. Hence it is important to find a suitable source problem. This difficulty is not addressed in this paper (see [12] or [5] instead). Here, we want to show that—when given a suitable source problem—FR allows for solving target problems that pose serious difficulties or cannot be handled at all without FR.

4 Experiments

We conducted our experimental studies with a theorem prover for problems of *condensed detachment* (CD). Subsection 4.1 explains the basics of CD and motivates this choice. We want to point out, however, that we have also successfully applied FR to equational reasoning (cp. [5]). The results of our experiments with CD are summarized in section 4.2.

4.1 Condensed Detachment

CD allows for studying logic calculi with automated deduction systems. (See [24] and [15] for motivation and a detailed theoretical background.) There is only one inference rule (also denoted by CD) that manipulates first-order terms which we shall also call facts. The set of terms (facts) $Term(\mathcal{F}, \mathcal{V})$ is defined as usual, involving a finite set $\mathcal{F}$ of function symbols and an enumerable set $\mathcal{V}$ of variables.

CD (in its basic form) is defined for a distinguished binary function symbol $f \in \mathcal{F}$. CD allows us to deduce $\sigma(t)$ from two given facts $f(s,t)$ and s' if σ is the most general unifier of s and s'. A proof problem $\mathcal{A} = (Ax, \lambda_G)$ hence consists in deducing λ_G from Ax with continuous applications of CD. Subsumption is merely a matching problem, i.e., $\lambda \lhd \lambda'$ if (and only if) there is a match σ so that $\sigma(\lambda) \equiv \lambda'$.

CD fits the theorem proving framework given in section 2. Despite the simplicity of CD the arising proof problems can be very hard, sometimes even exceeding the limits of state-of-the-art provers. Therefore, CD is widely acknowledged as a testing ground for new ideas. It has been (and still is) used for this purpose quite frequently (e.g., [19, 26, 16, 21, 27, 11]). Furthermore, CD offers problems of a varying degree of difficulty, almost continuously ranging from (nearly) trivial to (very) challenging. This constellation is important if we want to employ learning techniques (like FR).

For the experiments we used the theorem prover 'CoDe' ([7]) that realizes CD based on the concepts introduced in section 2. CoDe has a standard ("general purpose") heuristic $\mathcal{W}$ at its disposal that computes the weight $\mathcal{W}(\lambda)$ of a fact $\lambda \in F^P$ as $\mathcal{W}(\lambda) = c_\delta \cdot \delta(\lambda) + c_w \cdot w(\lambda)$, $c_\delta, c_w \in \mathbb{N}$. $w(\lambda)$ is equal to twice the number of function symbols occurring in λ plus the number of variables in λ. $\delta(\lambda)$ is the *level* or *depth* of λ: $\delta(\lambda) = 0$ if λ is an axiom; otherwise $\delta(\lambda)$ is the maximum of the levels of the immediate ancestors of λ plus 1. Furthermore, we set $\mathcal{W}(\lambda) = 0$ if $\lambda \lhd \lambda_G$. Consequently, facts subsuming the goal and hence concluding the proof are activated immediately (cp. [9] or [10]).

$\mathcal{W}$ is quite a successful heuristic in particular when taking into account the component δ reasonably (e.g., $c_\delta = 2$, $c_w = 1$). We use $\mathcal{W}$ as the standard heuristic required by FR, i.e., $\mathrm{FR}(\lambda) = (d(\lambda) + p) \cdot \mathcal{W}(\lambda)$. Here, we always set the coefficients $c_\delta = 0$ and $c_w = 1$, thus ignoring the level for reasons explained in [9]. Based on extensive experiments (cp. [9]) the remaining parameters of FR are set as follows: $p = 20$, $q_1 = 0.75$, $q_2 = 0.25$, $q = 0$.

4.2 Experimental Results

We examined FR in the light of problems LCL040-1 through LCL072-1 (problems 1–33 in [16]) and problems LCL109-1 through LCL116-1 (problems 55–62 in [16]). These problems have the property emphasized in the preceding subsection which is important for learning, namely to offer a varying degree of difficulty that ranges from rather simple to challenging. They are a part of the public TPTP problem library ([22]) version 1.2.1. (Note that the problems in the TPTP are given in CNF, a form suitable for resolution-based or tableaux-oriented theorem provers. The CNF "encoding" of problems of CD is not needed for CoDe.) We consider here mainly those problems that pose some difficulties for CoDe when using $\mathcal{W}$ to control the search. The performance of $\mathcal{W}$ with respect to the problems above (for various settings of the parameters c_δ and c_w) was thoroughly investigated in

Table 1. Case 1.

Target	Source	FR	FFU	RFF	lemmas	lemmas*	$\mathcal{W}$
058	060	23s (519)	89%	100%	—	5s (362)	25s (710)
060	058	8s (328)	95%	93%	—	12s (533)	26s (733)
071	070	4s (235)	75%	94%	—	3s (230)	—
	072	7s (366)	58%	100%	—	3s (254)	
068	067	4s (227)	72%	95%	2s (193)	2s (225)	68s (982)
	069	20s (629)	77%	100%	—	—	
114	113	3s (234)	83%	91%	12s (513)	10s (482)	—
116	113	5s (314)	78%	95%	14s (541)	10s (492)	—

[9] (see also [10]) and compared with the performance of the renowned theorem prover OTTER ([17]) as reported in [16]. This comparison showed that $\mathcal{W}$ can control the search so well that CoDE clearly outperforms OTTER even though CoDE is inferior to OTTER in terms of inferences per second. Therefore, it is not easy at all for `FR` to improve on $\mathcal{W}$.

In the sequel we examine the three interesting cases regarding the axiomatizations and the goals of target and source problems $\mathcal{A}_T = (Ax_T, \lambda_T)$ and $\mathcal{A}_S = (Ax_S, \lambda_S)$:

1. Identical axiomatization ($Ax_T = Ax_S$), but different goals (i.e., neither $\lambda_S \lhd \lambda_T$ nor $\lambda_T \lhd \lambda_S$);
2. Different axiomatizations (Ax_T and Ax_S may share some axioms, but one axiomatization is not a subset of the other), but the same goal ($\lambda_S \lhd \lambda_T$);
3. Different axiomatizations and different goals;

(We omit the fourth case "*the same axiomatization and the same goal*" because then plain re-enactment suffices and always succeeds very fast.)

Note that only for case 1 it is sound to add positive facts of the source proof as lemmas to Ax_T. Therefore, we shall consider this possibility only for case 1. Table 1 displays our experiments concerning case 1. The first two columns list target and source problems, respectively. (The names are abbreviated, e.g., `058` instead of `LCL058-1`.) The source problems were chosen according to some simple, automatable criteria. (This is not a relevant issue in this paper. See [12] instead.) Note that we employed source proofs found by $\mathcal{W}$ whenever possible in order to avoid creating particularly similar proofs with `FR`. For instance, we did *not* use the proof of `LCL058-1` found by `FR` using a proof of `LCL060-1` when proving `LCL060-1` with `FR` (and source `LCL058-1`).

The results of `FR` are given in column '`FR`'. The entries display run-time (approximate CPU time in seconds obtained on a SPARCstation 10) and the number of activation steps (i.e., the length of the search protocol) in parentheses. This number gives us a rough idea of the search effort. (The length of the search protocol usually does not correlate well with the length of the proof eventually found. This is also not an issue here since we are only interested in *some* proof, not necessarily a short one.)

Column 'FFU' shows the "focus fact usage", i.e., the share (relevant) focus facts have of the target proof, given as an approximate percentage. Column 'RFF' shows the share which facts that account for relevant focus facts have of the source proof. This value hence gives us an idea of how many facts of the source proof proved useful for finding the target proof. Consider, for instance, target problem `LCL071-1` and source problem `LCL070-1`. The target proof consists of 20 facts. 15 of the 20 facts are focus facts. Hence, FFU is $15/20 = 75\%$. These 15 relevant focus facts go back to 15 facts in the source proof. Since 16 facts constitute the source proof, RFF is $15/16 \approx 94\%$.

The values of RFF listed in table 1 indicate that it is important that RFF is rather high. Given case 1 (equal source and target axiomatization) this is understandable considering that the share $100\% - \text{RFF}$ of the source proof will definitely give rise to irrelevant focus facts that may disarrange the search. Large values of FFU indicate that a large part of the target proof goes back to re-enactment and hence could be found rather efficiently.

When adding the positive facts of the source proof as lemmas to the axiomatization of the target problem, CoDE employed $\mathcal{W}$ with $c_\delta = 2$ and $c_w = 1$ to guide the search. (This parameter setting turned out to be generally useful during the experiments reported on in [9].) The results of utilizing lemmas in this manner are listed in column 'lemmas'. The entries again show run-time and length of the search protocol. The entry '—' signifies that no proof was found within one hour.

A slight modification of lemma usage is to assign the weight 0 to all axioms (including added lemmas) so that they are immediately activated. Column 'lemmas*' shows that this modification is crucial in order to make adding lemmas competitive. The improvements are understandable because we can thus coerce re-enactment and hence simulate the obviously profitable re-enactment part of `FR`. Nonetheless, table 1 does not reveal any significant advantage of adding lemmas compared to `FR`. As a matter of fact, adding lemmas can cause a failure when `FR` still succeeds (cp. target `LCL068-1`, source `LCL069-1`). An opposite observation (failure of `FR`, success of adding lemmas) has so far not been made. This leads us to conclude that the "hierarchical" search induced by the distance measure d is pivotal. (Note, however, that the component δ of $\mathcal{W}$ (switched off for `FR`) also allows for taking into account distance or depth, but in a much cruder way than d does. Basically, the ancestor with maximal depth determines the depth of a descendant. The depth of the other ancestor has no effect.)

The last column of table 1 lists the results of $\mathcal{W}$ as a point of reference. The generally useful parameter setting ($c_\delta = 2$ and $c_w = 1$) fails for all target problems of table 1 (and also of tables 2 and 3). (Recall that the selected problems are the ones that cause difficulties for $\mathcal{W}$.) The successful runs of $\mathcal{W}$ were obtained with $c_\delta = 0$ and $c_w = 1$—the setting that also `FR` employs. These experiments show that `FR` enables CoDE to prove problems quite fast which it could not handle with $\mathcal{W}$ and one of the two mentioned parameter settings. (There are,

Table 2. Case 2.

Target	Source	FR	FFU	RFF	$\mathcal{W}$
054	042	118s	31%	79%	—
058	045	4s	23%	82%	25s
045	058	$<$ 1s	95%	51%	—
042	054	$<$ 1s	84%	55%	—

Table 3. Case 3.

Target	Source	FR	FFU	RFF	$\mathcal{W}$
042	058	—	—	—	—
	060	126s	82%	95%	
045	060	22s	76%	41%	—
058	042	18s	20%	42%	25s
060	042	—	—	—	26s
	045	11s	18%	82%	

however, two different and very problem-specific parameter settings for problems `LCL068-1` and `LCL071-1` that allow $\mathcal{W}$ to prove these problems. See [9].)

Experiments regarding cases 2 and 3 are summarized by tables 2 and 3. These tables are organized like table 1 without the columns concerning lemmas since the soundness of adding lemmas is—as mentioned earlier—not guaranteed for cases 2 and 3. As a matter of fact, for case 2 adding lemmas is completely pointless since the goal itself would be added as a lemma.

Considering tables 2 and 3 we can again observe that `FR` allows CoDe to solve problems rather quickly which were out of reach when using $\mathcal{W}$. The two bottom rows of table 2 indicate that `FR` can also cope with situations where large parts of the target proof go back to quite a small part of the source proof (i.e., large FFU, but rather small RFF). Note that—due to different axiomatizations—it is not necessarily the case that the share 100% $-$ RFF of the source proof entails irrelevant focus facts. The results concerning problems `LCL042-1`, `LCL058-1`, and `LCL060-1` in different roles as source and target problems (cp. tables 1 and 2) reveal that—quite expectedly—`FR` is not "symmetric". That is, a proof of a problem A found by `FR` when using a source problem B does not guarantee to find a proof of B when using A as source.

Due to space limitations we cannot present proof listings which illustrate properties of `FR`. See [13] instead.

5 Discussion

We presented a method called flexible re-enactment (`FR`) that exploits past proof experience with heuristic means. When searching for the (target) proof of a given proof problem, `FR` *attempts* to re-enact a given source proof by *searching* for the facts that constitute the source proof. Flexibility is achieved by also searching in the "neighborhood" of the source proof, i.e., by also preferring descendants of the facts constituting the source proof. Experiments have demonstrated that `FR` allows for solving problems that could not be handled before.

The similarity between the source and target proof required so that `FR` or in other words a "flexible, search-based re-enactment of the source proof" can succeed is basically reflected by the number (or percentage) of facts constituting the source proof that are also useful for a target proof: the more, the better. But such a kind of similarity can only be assessed *a posteriori*. Hence, *a priori* similarity can only be estimated with "heuristic indicators" such as, for instance, the percentage of source axioms that are also present in the target axiomatization. Simple, easy to check criteria like this can already produce remarkable results (cp. [12]) which supports our belief that selecting an appropriate source problem can be automated reasonably (see also [5]). Naturally, in order to be independent of symbol names, methods for finding a suitable renaming of symbols must be available (cf. [5]). Thus, for `FR` similarity is kind of a "loosely" defined notion that perforce cannot and does not play as important a role as it does, e.g., in [14]. There, similarity is a clearly defined central notion, but—probably a common trade-off—restricts applicability.

`FR` is not limited to the use of just one source proof. The design of `FR` allows us to exploit an arbitrary number of source proofs. But the usefulness of more than one source proof is questionable ("a jack of all trades, but master of none"). Redundant search effort might become a problem that outweighs possible advantages like having available more focus facts which undoubtedly increases the chance of having the "right" ones. Nonetheless, exploiting a pool of proof experience rather than one piece also has certain advantages (cp. [6] where sophisticated abstraction techniques are used).

Finally, `FR` can of course also be employed for *proof checking* and *proof completion*. Proof checking corresponds to plain re-enactment. Proof completion means that the given positive facts represent the "skeleton" of a source proof with a few intermediate steps missing. Under these conditions the positive facts will give rise only to relevant focus facts. Thus, `FR` can search for the missing intermediate steps *without* being confused and possibly misguided by irrelevant focus facts which makes things easier than in general applications of `FR`.

In [25] the *hints strategy* (`HS`) has been investigated as to its usefulness for proof checking, proof completion, and also for proof finding. Hints basically are the counterparts of the positive facts of a source proof. `HS` and `FR` have commonalities, but also have different features and design concepts. In order to search for (or complete) a proof with certain properties, `HS` also allows for *avoiding* (instead of focusing on) hints—a feature that was not an objective when designing `FR`. But when it comes to searching for a (target) proof without such constraints, `FR` subsumes `HS` in particular because of its ability to deal sensibly with non-focus facts that are descendants of focus facts, thus exploiting a source proof beyond mere re-enactment—a concept not present in `HS`.

References

1. **Bachmair, L.; Dershowitz, N.; Plaisted, D.:** *Completion without Failure*, Coll. on the Resolution of Equations in Algebraic Structures, Austin, TX, USA (1987), Academic Press, 1989.
2. **Brock, B.; Cooper, S.; Pierce, W.:** *Analogical Reasoning and Proof Discovery*, Proc. CADE-9, Argonne, IL, USA, 1988, Springer LNCS 310, pp. 454–468.
3. **Bundy, A.:** *The Use of Explicit Plans to Guide Inductive Proofs*, Proc. CADE-9, Argonne, IL, USA, 1988, Springer LNCS 310, pp. 111–120.

4. **Chang, C.L.; Lee, R.C.:** *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973.
5. **Denzinger, J.; Fuchs, M.; Fuchs, Marc:** *High Performance ATP Systems by Combining Several AI Methods*, SEKI Report SR-96-09, University of Kaiserslautern, 1996, `http://www.uni-kl.de/AG-AvenhausMadlener/fuchs.html`.
6. **Denzinger, J.; Schulz, S.:** *Learning Domain Knowledge to Improve Theorem Proving*, Proc. CADE-13, New Brunswick, NJ, USA, 1996, Springer LNAI 1104, pp. 62–76.
7. **Fuchs, D.; Fuchs, M.:** *CODE: A Powerful Prover for Problems of Condensed Detachment*, Proc. CADE-14, Townsville, AUS, 1997, Springer LNAI, to appear.
8. **Fuchs, M.:** *Learning Proof Heuristics by Adapting Parameters*, Proc. 12^{th} ICML, Tahoe City, CA, USA, 1995, Morgan Kaufmann, pp. 235–243.
9. **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, SEKI Report SR-95-10, University of Kaiserslautern, 1996, obtainable via WWW at `http://www.uni-kl.de/AG-AvenhausMadlener/fuchs.html`.
10. **Fuchs, M.:** *Powerful Search Heuristics Based on Weighted Symbols, Level, and Features*, Proc. FLAIRS-96, Key West, FL, USA, 1996, pp. 449–453.
11. **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, Proc. CADE-13, New Brunswick, NJ, USA, 1996, Springer LNAI 1104, pp. 523–537.
12. **Fuchs, M.:** *Towards Full Automation of Deduction: A Case Study*, SEKI Report SR-96-07, University of Kaiserslautern, 1996, obtainable via WWW at `http://www.uni-kl.de/AG-AvenhausMadlener/fuchs.html`.
13. **Fuchs, M.:** *Flexible Re-enactment of Proofs*, SEKI Report SR-97-01, Univ. of Kaiserslautern, 1997, `http://www.uni-kl.de/AG-AvenhausMadlener/fuchs.html`.
14. **Kolbe, T.; Walther, C.:** *Reusing Proofs*, Proc. 11^{th} ECAI '94, Amsterdam, HOL, 1994, pp. 80–84.
15. **Łukasiewicz, J.:** *Selected Works*, L. Borkowski (ed.), North-Holland, 1970.
16. **McCune, W.; Wos, L.:** *Experiments in Automated Deduction with Condensed Detachment*, Proc. CADE-11, Saratoga Springs, NY, USA, 1992, Springer LNAI 607, pp. 209–223.
17. **McCune, W.:** *OTTER 3.0 reference manual and guide*, Techn. report ANL-94/6, Argonne Natl. Laboratory, 1994.
18. **Melis, E.:** *A Model of Analogy-driven Proof-plan Construction*, Proc. 14^{th} IJCAI, Montreal, CAN, AAAI Press, 1995, pp. 182–189.
19. **Peterson, G.J.:** *An Automatic Theorem Prover for Substitution and Detachment Systems*, Notre Dame J. of Formal Logic, Vol. 19, No. 1, Jan. 1976, pp. 119–122.
20. **Slagle, J.R.; Farrell, C.D.:** *Experiments in Automatic Learning of a Multipurpose Heuristic Program*, Comm. of the ACM, Vol. 14, No. 2, 1971, pp. 91–99.
21. **Slaney, J.:** *SCOTT: A Model-guided Theorem Prover*, Proc. IJCAI '93, Chambery, FRA, 1993, AAAI Press, pp. 109–114.
22. **Sutcliffe, G.; Suttner, C.; Yemenis, T.:** *The TPTP Problem Library*, Proc. CADE-12, Nancy, FRA, 1994, Springer LNAI 814, pp. 252–266.
23. **Suttner, C.; Ertel, W.:** *Automatic Acquisition of Search-guiding Heuristics*, Proc. CADE-10, Kaiserslautern, FRG, 1990, Springer LNAI 449, pp. 470-484.
24. **Tarski, A.:** *Logic, Semantics, Metamathematics*, Oxford University Press, 1956.
25. **Veroff, R.:**, *Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies*, JAR **16:**223–239, 1996.
26. **Wos, L.:** *Meeting the Challenge of Fifty Years of Logic*, JAR **6:**213–232, 1990.
27. **Wos, L.:** *Searching for Circles of Pure Proofs*, JAR **15:**279–315, 1995.

Equalizing Terms by Difference Reduction Techniques

Dieter Hutter

German Research Center for Artificial Intelligence GmbH, Stuhlsatzenhausweg 3,
D-66123 Saarbrücken, Germany
hutter@dfki.de, Tel. -49-681-302-5317

Abstract. Based on the success of difference reduction techniques and especially of rippling techniques in the field of inductive theorem proving we propose the use of difference reduction (rippling) and proof planning techniques also for non-inductive theorem proving. While in inductive theorem proving a proof idea is given by the task of enabling the induction hypothesis, in general theorem proving we use abstractions in order to decompose the given task of proving a theorem into different subgoals. This paper illustrates how these different techniques can be combined to an integrated approach in order to deal with equational problems.

1 Introduction

Heuristics for judging similarities between formulas and subsequently reducing differences have been applied to automated deduction since the 1950s, when Newell, Shaw, and Simon built their first "logic machine" [NeShSi63]. Since the later 60s, a similar theme of difference identification and reduction appears in the field of automated reasoning (e.g. [QuHu68, Mor69, Dig85]). But none of these approaches have been generally accepted and saturation-based theorem provers are still the state of the art.

During the last decade a variety of industrial strength formal methods have emerged and have been applied to industrial test cases to demonstrate their adequacy and scalability. Formal techniques require a sufficient tool support especially when dealing with proof obligations. But saturation-based provers are apparently not appropriate for these purposes: specifying case studies of industrial strength results in large databases containing thousands of axioms and at the same time only a small percentage of these axioms are used to prove a concrete proof obligation. These proofs are usually longish but in general not tricky. Furthermore, dealing with inductively defined datatypes in general, proofs require induction which implies an infinite branching in each node of the search space [Kre65]. In general we cannot expect that each proof obligation will be proven fully automatically. Thus, there is a need for a user interaction allowing one to guide the prover on a strategic level. Finally in order to increase the automation, knowledge about inbuilt datatypes like integers or sets, has to be integrated into the prover.

Based on the success of difference reduction techniques and especially of rippling [BvHS91, Hut90, Hut96a] in the field of inductive theorem proving we propose the use of difference reduction (rippling) and proof planning techniques also for non-inductive theorem proving. While in inductive theorem proving a proof idea is given by the task of enabling the induction hypothesis, in general theorem proving we use abstractions in order to decompose the given task of proving a theorem into different subgoals. This paper illustrates how these different techniques can be combined to an integrated approach in order to deal with equational problems. Throughout the paper we illustrate the described techniques with the help of the simple example of proving

$$(a + b) \times (a - b) = (a \times a) - (b \times b) \tag{1}$$

2 Rippling to Maintain Differences

Difference reduction techniques compare an actual situation with the intended goal and try to minimize these differences with respect to some measure by applying step by step possible operations to the actual situation. In terms of proving an equation $q = r$ we try to minimize the syntactical differences of q and r by applying given rewrite rules until we finally reach a goal situation $q' = q'$. To maintain the similarities of q and r achieved so far we use a rippling calculus as given in [Hut90, Hut96a].

Rippling is based on the notion of annotated terms where each occurrence of a symbol can be attached by additional information (encoded into the color of an occurrence) which is maintained during the deduction by the rippling calculus. E.g. $(a + b) \times (a - b)$ is an example of such an annotated term. The colors are used to represent the similarities explicitly. While the white parts denote the common parts of both sides of the equation to be proven, the grey symbols represent their differences. Thus, instead of describing differences by identifying minimal subterms which contain them (as in the RUE-approach), the notion of annotated terms allows one to represent the differences explicitly and therefore, seems to be a more adequate representation for implementing difference reduction techniques. Also, the explicit representation of differences is an important requirement in computing a measure of how far two terms or formulas differ which is a central aid in guiding (parts of) the proof search.

Using the rippling calculus the white parts of an annotated term stay — as a property of the calculus — invariant while the grey parts are subjects to change. In order to manipulate annotated terms correctly, the equations given as axioms are also colored such that the white parts on both sides coincide while the grey parts represent their differences. Examples of such annotated equations are given by:

$$(X + Y) \times Z = (X \times Z) + (Y \times Z) \tag{2}$$

$$X \times (Y - Z) = (X \times Y) - (X \times Z) \tag{3}$$

For example using (2) and (3) we are able to perform the following rewriting:[1]

$$\begin{aligned}(a+b)\times(a-b) &\rightarrow (a\times(a-b))+(b\times(a-b))\\ &\rightarrow ((a\times a)-(a\times b))+(b\times(a-b))\end{aligned}$$

Notice that in all these annotated terms the white parts — namely $a \times a$ — stay fixed. These white parts are called the *skeleton* $skel(q)$ of an annotated term q while the grey parts are called the *wave-fronts*. Precisely, the skeleton of such an annotated term q is a *set* of un-annotated terms which are obtained by successively replacing each subterm q' of q with a grey top-level symbol by some biggest subterm of q' with a white top-level symbol. We extend the notion of a skeleton to a equality by $skel(q = r) = \{q' = r' \mid q' \in skel(q) \text{ and } r' \in skel(r)\}$.

Removing all color-information from an annotated term q results in a term q' the so-called erasure $erase(q)$ of q.

In order to enable the use of speculated lemmata we use difference matching (c.f. [BaWa93]). We say a term q difference-matches a term t iff there is an annotated term t' with $erase(t) = erase(t')$ and a substitution σ such that $\{\sigma(q)\} = skel(t')$. For example $(X-Y)+Y$ difference matches $((a\times a)-(a\times b))+((a\times b)-(b\times b))$ since there is an annotated term $t' = ((a\times a)-(a\times b))+((a\times b)-(b\times b))$ and a substitution $\sigma = \{X \leftarrow a\times a, Y \leftarrow a\times b\}$ such that $\{\sigma(q)\} = skel(t')$ holds.

Suppose, we want to prove an goal equation $q = r$ with the help of a set of axioms yielding a chain of intermediate equations $q = r, q_1 = r_1, \ldots, q_n = r_n$. We start with no explicit known similarities between both sides and obtain an empty skeleton on both sides which results in the annotated equation $q = r$. In order to manipulate an equation $q_i = r_i$ we allow two different kinds of modifications:

First, annotated axioms (rewrite rules like (2) or (3)) are used to rewrite an equation $q_i = r_i$ to a new goal $q_{i+1} = r_{i+1}$. Using the rippling calculus, such a step will preserve the skeleton of the equation, i.e. $skel(q_i = r_i) = skel(q_{i+1} = r_{i+1})$. This manipulation step corresponds to a usual rewrite step $erase(q_i = r_i) \rightarrow erase(q_{i+1} = r_{i+1})$ in the underlying first-order calculus.

Second, there are skeleton changing steps which do not change the underlying equation, i.e. $erase(q_i = r_i) = erase(q_{i+1} = r_{i+1})$, but which redye arbitrarily the equation such that still $skel(q_{i+1}) = skel(r_{i+1})$ holds. This step has no effect in the underlying first-order calculus since the un-annotated equations remain unchanged.

In a typical deduction, rewrite steps and skeleton changing steps alternate with each other until we finally end up with a trivial equation $q' = q'$ in which both sides coincide and $skel(q') = \{q'\}$. We achieve this final situation if we succeed to remove all occurring wave-fronts either by applying rewrite-rules or by increasing the common skeleton (i.e. insertion of common wave-fronts into the skeleton). We define a deduction step $q_i = r_i \rightarrow q_{i+1} = r_{i+1}$ to be *skeleton embedding* if for each $q' = r' \in skel(q_i = r_i)$ there is some $q'' = r'' \in skel(q_{i+1} =$

[1] For sake of simplicity we omit the details of matching and unification in the rippling calculus. The reader is referred to [Hut90, Hut96a] for details.

r_{i+1}) such that $q' = r'$ is homomorphic embedded [Kru60] in $q'' = r''$. Since rewrite steps do not alter the skeleton they are trivially skeleton embedding.

Following the paradigm of difference reduction for the present, we are interested in deductions in which each step is skeleton embedding. Each deduction in the first-order calculus can be trivially simulated by a skeleton-embedding deduction since we may perform the complete deduction using an empty skeleton and add a final step in which we enlarge the skeleton to cover the total equation. But from a strategic point of view we only gain advantages if we impose a non-empty skeleton as soon as possible in order to restrict the search space. Since the skeleton remains invariant, admissible deduction steps will only change the wave-front parts of the annotated term. Thus, declaring parts of a term to be parts of the skeleton prevents them from being modified but may also hamper the proof if it relies on a change of skeleton parts. Thus, one has to carefully plan *which* common parts of the theorem should be used as skeleton.

Using abstractions as proposed in section 3 we have to map abstract solutions into the ground space of the rippling calculus. Typically this refinement involves the problem of enabling the use of "bridge" lemmata speculated in the abstract space. Thus, besides the overall problem of equalizing the left- and right-hand side of a goal equation we also obtain the task of manipulating some subterm of the goal equation in order to get it matched with the left-hand side of the speculated lemma. This difference reduction problem is handled in the same manner as the original problem. Hence, we introduce annotations to the designated subterm and the left-hand side of the speculated equation specifying the common skeleton. While enabling the use of the speculated lemma we have to manipulate the designated subterm in such a way that, at the end, the skeleton covers totally the left-hand side of the lemma. Additionally, the other skeleton, denoting the similarities between left- and right-hand side of the goal equation, has also to be preserved.

Consider our example of proving theorem (1) and suppose we have already done some rewriting in order to identify the common subterms $a \times a$ and $b \times b$. Thus, we obtain the following intermediate equation

$$((a \times a) - (a \times b)) + ((a \times b) - (b \times b)) = (a \times a) - (b \times b). \tag{4}$$

Abstractions as they will be described in section (3) suggest the use of the lemma

$$(X - Y) + Y = X \tag{5}$$

to eliminate some of the wave-fronts on the left-hand side. Difference matching between the left-hand side of (5) and the left-hand side of the intermediate equation results in a new annotation of the left-hand side:

$$((a \times a) - (a \times b)) + ((a \times b) - (b \times b)) \tag{6}$$

If we are able to remove the wave-fronts inside the skeleton of (6) we could apply lemma (5) in order to remove the occurrences $a \times b$. Notice that we aim at two different goals. First, we want to equalize both sides of (4) and second in order to do so, we have to enable the use of (5). Both tasks possess their own annotations (as given in (4) and (6)) and in order to manipulate the goal equation we have to bear in mind both skeletons. Hence, if we manipulate the equation, e.g. in order to enable the use of (5) we have also to guarantee that the skeleton given in (4) will be preserved. This is done by using different wave-rules denoting the same axiom in order to rewrite the different annotated versions of the intermediate goal:

$$((a \times a) - (a \times b)) + ((a \times b) - (b \times b)) \text{ with } X + (Y - Z) \to (X + Y) - Z \; :$$
$$(((a \times a) - (a \times b)) + (a \times b)) - (b \times b)$$
$$((a \times a) - (a \times b)) + ((a \times b) - (b \times b)) \text{ with } X + (Y - Z) \to (X + Y) - Z \; :$$
$$(((a \times a) - (a \times b)) + (a \times b)) - (b \times b)$$

Thus, the more we speculate subgoals to explore the search space the more we collect constraints given by the hierarchy of skeletons which have to be preserved during possible rewrite steps.

3 Abstractions

Our approach uses abstractions in order to guide the proof search. Following [GW92a] an abstraction is a pair of formal systems $\langle \Sigma_1, \Sigma_2 \rangle$ with languages Λ_1 and Λ_2 and a total function $f_\Lambda : \Lambda_1 \to \Lambda_2$. In our setting Λ_1 is the underlying annotated first-order calculus we use for equality reasoning while Λ_2 depends on the abstraction we choose to tackle the actual problem (see below).

$$\begin{tikzcd}
\Phi(a,b) \arrow[r, "\Lambda_2"] & \Phi(a_1,b_1) \arrow[d] & \ldots \arrow[r, "\Lambda_2"] & \Phi(a_n,b_n) \arrow[d] \\
q = r \arrow[u, "f_\Lambda"'] \arrow[r, "*"', "\Lambda_1"] & q_1 = r_1 & \ldots \arrow[r, "*"', "\Lambda_1"] & q_n = r_n
\end{tikzcd}$$

Fig. 1. Abstracted proof search

Using abstractions we transform the given problem of equalizing two annotated terms q and r into a problem of manipulating two objects a and b in a formal system Λ_2 in order to obtain a so-called *solved form* $\Phi(a_n, b_n)$. Each step in the abstract space has to correspond to a sequel of steps in the ground space (Λ_1).

Once we find a solution in the abstract space, we have to transfer it to the given (first-order) calculus which imposes a set of different subtasks to be tackled. For each intermediate result $\Phi(a_{i+1}, b_{i+1})$ of the deduction in the abstract space we have to find an appropriate equation $q_{i+1} = r_{i+1}$ such that $q_i = r_i \to^*_{\Lambda_1} q_{i+1} = r_{i+1}$ holds.

Thus, abstractions are used to speculate intermediate goals during the proof search. Typically the refinement of abstract steps results in tasks to enable the use of specific bridge lemmata which are tackled by the use of difference unification (difference matching, respectively) and appropriate rippling techniques.

In the following we sketch some abstractions which we use in our approach. We illustrate these abstractions with the help of our example in order to give the reader an impression of the different subgoals arising during the proofs. Some of the heuristics above create subtasks to enable the use of some bridge lemma.

3.1 Wave-front Positions

While we introduced rippling as a basic mechanism to maintain the similarities of terms during the proof and to focus the interest of the prover to specific parts of the theorem, rippling denotes also a kind of abstraction which allows one to formulate abstract goals. The idea is to abstract from the concrete shape of the wave-fronts (grey parts) and to specify the positions in which these wave-fronts may occur in the abstracted goal state.

Suppose, we want to equalize the two (annotated) terms $(a+b) \times (a-b)$ and $(a \times a) - (b \times b)$, then we abstract from the shape of the wave-fronts and obtain the following abstractions: $(\ldots a \ldots) \times (\ldots a \ldots)$ and $\ldots (a \times a) \ldots$ In their abstractions both terms disagree in the positions of the wave-fronts wrt. the skeleton. While in the first term the wave-fronts occur inside both arguments of $\times$, in the second term the only wave-front occurs at top-level. Equalizing both terms requires also to equalize the positions in which wave-fronts occur. Thus, an abstract solution will fix the positions of the wave-fronts while the refinement of the abstract solution will ripple the wave-fronts from their current positions to the desired target positions. Consider our example: if we commit ourselves to the top-level as target position for wave-fronts we have to ripple out all wave-fronts in the first term while the second term already meets the constraints of the abstract solution. The concrete refinement of our example is given as an rippling example in section 2.

3.2 Adaptation of Maximal Symbols

Dealing with large case studies the specifications have to be structured into theories and small modules in order to ease the proof search. This structuring also imposes a partial ordering on the function and predicate symbols occurring in a specification denoting some kind of definition hierarchy.

Similarly to techniques in rewriting we define such a definition hierarchy on function symbols by: $f < g$ holds if f occurs in the definition of g. $<^*$ is defined as the transitive closure of $<$. A symbol f occurring in some wave-front c of an annotated term is called a *maximal* symbol of c iff there is no symbol g in c with $f <^* g$. Furthermore, specific formulas give rise to a $<$-relation, like $\forall x, y\; f(x,y) = g(h(x), k(y))$ implies $g < f$, $k < f$, and $h < f$.

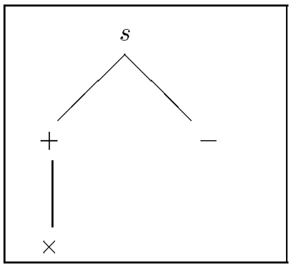


Fig. 2. Symbol Hierarchy

Thus, in order to equalize the wave-fronts of two terms q and r we map them into the multi-sets of maximal symbols of both wave-fronts and modify both multi-sets in order to equalize them. Once we come up with a common multi-set of maximal symbols we map this solution into the task of modifying the original wave-fronts such that these maximal symbols occur in the wave-fronts of both sides.

Consider for instance our example of proving

$$(a+b)\times(a-b)=(a\times a)-(b\times b).$$

Due to the definition hierarchy (cf. Figure 2) $\times > +$ holds while $-$ and $\times$ (or $+$) are incompatible. After initial steps of enlarging the skeleton we obtain the equation:

$$((a\times a)-(a\times b))+((a\times b)-(b\times b)) = (a\times a)-(b\times b)$$

The abstraction results in a form $\Phi(\{\times,\times,-,-\},\{-\})$ which suggests the removal of occurrences of $\times$ and $-$ in order to identify the multisets of maximal symbols on both sides.

Exploring the database will result in some speculations about useful "bridge lemmata". Given the annotated equation

$$(X-Y)+Y=X \tag{7}$$

we are able to remove at least an occurrence of $-$ in our example if we could apply the equation from left to right. Since the equation is not applicable we have to do some equality reasoning in order to refine the first abstract step on the ground space (as we have already described in section 2). Hence, we use difference matching between the left-hand side of the lemma and a subterm t containing the wave-front to be modified which results in:

$$((a\times a)-(a\times b))+((a\times b)-(b\times b)) \tag{8}$$

Using rippling out to remove the wave-front inside the skeleton uses an annotated equation

$$X+(Y-Z)=(X+Y)-Z \tag{9}$$

and results in

$$(((a\times a)-(a\times b))+(a\times b))-(b\times b)$$

We can now apply (7) and its use identifies the multi-sets of maximal symbols on both sides of the original goal:

$$(a\times a)-(b\times b)=(a\times a)-(b\times b)$$

As we have seen in the example above, the abstraction of maximal symbols will usually allow only to plan one step ahead in the abstract space since the effects of applying equations to the goal term can be hardly traced on the abstract level. For instance removing $\times$ may result in $+$ being now a maximal symbol but in the abstract space $+$ has not been monitored as long as $\times$ occurred in the goal. Thus, in the abstract space there is no information available whether $+$ still occurs inside the term to be modified.

3.3 Paths to Common Parts of the Skeleton

Since the abstraction to maximal symbols preserves to little information to plan in advance several deduction steps in the abstract space, the next abstraction [AuHu97] suggests to equalize the paths in both terms from the top-level to a specific part of the skeleton. In case of success the symbols on this path will be added to the skeleton in the next step. This heuristic produces a proof sketch which suggests the manipulation of "paths" to specific occurrences of terms. Expanding this proof sketch to a first order proof will create a chain of bridge lemmas to be applied on the goal.

Consider again the given goal in our previous example:

$$((a \times a) - (a \times b)) + ((a \times b) - (b \times b)) = (a \times a) - (b \times b) \tag{10}$$

Extracting the function symbols occurring on the way from top-level to the occurrences of the common skeleton-part $a \times a$, we obtain the list $\langle +, - \rangle$ for the left-hand and $\langle - \rangle$ for the right-hand side. If we add also the argument positions in which the skeleton part occurs as subscripts to the symbols then we obtain the lists $\langle +_1, -_1 \rangle$ and $\langle -_1 \rangle$ respectively. Analogously, we may abstract the equations given in our database: from (7) we obtain

$$\langle +_1, -_1 \rangle = \langle \rangle \tag{11}$$

and the abstraction of the erased version of (9) results in:

$$\langle +_1 \rangle = \langle -_1, +_1 \rangle, \tag{12}$$

$$\langle +_2, -_1 \rangle = \langle -_1, +_2 \rangle, \tag{13}$$

$$\langle +_2, -_2 \rangle = \langle -_2 \rangle, \tag{14}$$

Thus, the problem of equating wave-fronts is reduced to the problem of equalizing the strings of indexed symbols with the help of these abstracted equations on strings of indexed symbols. A solution of the above problem is given by the following deduction in the abstract space:

$$\langle +_1, -_1 \rangle \; =_{(12)} \; \langle -_1, +_1, -_1 \rangle \; =_{(11)} \langle -_1 \rangle$$

which suggests the use of a version of equation (9) followed by the use of (7). Refining this abstract solution to the ground space results in the same deduction steps as in the previous section. But in contrast to the previous abstraction we have planned the complete adaptation on the abstract space before we refined the obtained proof sketch to the ground space.

In case we choose the lists of function symbols occurring on the path to the second part $b \times b$ of the skeleton we obtain a solution

$$\langle +_2, -_2 \rangle \; =_{(14)} \; \langle -_2 \rangle$$

suggesting only the application of equation (9) which would already equalize the paths to this skeleton part $b \times b$.

3.4 Built-In Theories

Integrating decision procedures for built-in theories like the sup-inf method or other methods for linear arithmetic in a first-order setting usually causes very complex interactions between the decision procedure and the first-order calculus in order to handle conditional equations or user-defined functions properly.

In contrast to saturation-based provers, our approach supports the use of these decision procedures as it builds up step by step a common skeleton which is basically a set of common subterms of the theorem $q = r$. In case corresponding wave-fronts consist only of symbols of our built-in theory we may abstract from the shape of the skeleton and replace each member of the skeleton in both terms by a new constant. Thus, we obtain a problem in the in-built theory which can be tackled by the decision procedure. We then use the obtained solution in the abstract space to equalize the wave-fronts and to enlarge the skeleton by the identical wave-fronts.

In our example of proving

$$\boxed{((a \times a) - (a \times b)) + ((a \times b) - (b \times b))} = \boxed{(a \times a) - (b \times b)}$$

we abstract the skeleton terms $a \times a$ to c and $b \times b$ to d and obtain the problem of proving the following equation on integers:

$$\boxed{(c - (a \times b)) + ((a \times b) - d)} = \boxed{c - d}$$

Once we have proven this equation we end up with the trivial equation

$$\boxed{c - d} = \boxed{c - d}$$

Expanding the definitions of c and d we obtain

$$\boxed{(a \times a) - (b \times b)} = \boxed{(a \times a) - (b \times b)}.$$

Since both wave-fronts are identical we enlarge the skeleton to

$$(a \times a) - (b \times b) = (a \times a) - (b \times b)$$

which proves the theorem.

4 Beyond Difference Reduction

As already mentioned above, the skeleton guiding the proof has mainly two effects. Suppose there is a proof of an annotated equation $q = r$ by a chain of equation $q = r \rightarrow q_1 = r_1 \rightarrow \ldots \rightarrow q_n = r_n$ with $q_n = r_n$. Since q and r share the same skeleton and all deduction steps are homomorphic embedding, $q_n = r_n$ still contain the skeleton of $q = r$. Thus, the skeleton of $q = r$ is also a pattern of the goal term $q_n = r_n$.

On the other hand enlarging the skeleton by a common function symbol imposes also a link between the two selected occurrences of the symbol. During the ongoing deduction this link has to be inherited and in case the proof succeeds it will finally link two identical subterms.

There are cases in which we want to impose such a link between different subterms and inherit it until it finally links two identical subterms but in which the first property that the skeleton stays invariant (or will only be increased) is to restrictive to guide a successful proof search.

An example in which a rewriting of the initial skeleton is required is the proof of the **lim**-example [Ble90]:

$$lim(\lambda x.f(x), a, l) \wedge lim(\lambda x.g(x), a, m) \rightarrow lim(\lambda x.(f(x) + g(x)), a, l + m)$$

Although this example is not an equality problem we can use the same techniques described above to equalize either one or both premises and the conclusion of the implication. Enlargement of the skeleton will result in an annotated implication.

$$lim(\lambda x.f(x), a, l) \wedge lim(\lambda x.g(x), a, m) \rightarrow lim(\lambda x.(\boxed{f(x) + g(x)}), a, \boxed{l + m})$$

In order to prove this statement we have to open up the definition of lim

$$\begin{array}{l} \lim(f, a, l) \equiv \\ (\forall \epsilon \; (0 < \epsilon \rightarrow \exists \delta \; (0 < \delta \wedge \forall x \; (x \neq a) \wedge |x - a| < \delta \rightarrow |f(x) - l| < \epsilon))) \end{array}$$

After unfolding this definition the occurring wave-fronts can be easily equalized in the reformulated theorem (see [YBGWB94] for a complete proof of this theorem).

The rewriting has to be done on both sides of the goal equation simultaneously and we have to take care that each step on one side can be also performed on the other side. In order to ensure this, we only use equations for rewriting with the property that each function symbol on the left-hand side of the equation matches with some skeleton part. Thus, wave-fronts may only match with some instantiation of variables occurring on the left-hand side of the equation. Also all variables occurring on the right-hand side of the rewrite-equation have to occur on the left-hand side. Since we apply the same equation on both sides at the same position wrt. the skeleton both replacing subterms differ at most in the instantiations of the right-hand side variables. Hence, we add all function symbols of the non-instantiated right-hand side to the skeleton and relate corresponding symbols to each other. The annotation of the instantiated variables is inherited from the previous term. In case each variable occurs only once in the left-hand side there is a unique annotation for all instantiated occurrences of the variables on the right-hand side. But in case the variables occur several times on the left-hand side we have to choose which occurrence we select to annotate the instantiation of the right-hand side appropriately.

5 Results and Conclusion

The strategies presented in this paper are implemented and successfully tested within the INKA-system and are the basis of the equality reasoning part of this system. The INKA-System is itself integrated into a tool called VSE [HLSSSW96] for the formal development of software. Within this tool several large industrial case studies have been performed and up to now more than 20,000 lines of code have been verified which causes that thousands of first order proof obligations were tackled automatically or with minor user interaction by the INKA-system.

We have also successfully tested our method to standard academic examples in lattice ordered groups given in the TPTP problem library [SuSu95]. For example, SAM's lemma [GOBS69] is fully automatically proven by the system in less than a second. The ternary Boolean algebra problem EQ-3 ($f(x, g(x), y) = y$ holds if axiom 3 is removed) [LuMC93] is proven using the path-abstraction on y and the described DR-techniques to refine the abstract solution. This example illustrates how abstractions are used to overcome local minima of the difference reduction process since they guide the deduction process how to blow up terms which is for instance necessary in case of this example.

The explicit notion of differences and the corresponding mechanism to maintain them allows us to divide an equality problem more flexible into subproblems than for instance [QuHu68, Mor69, Dig85] (which are restricted to a top-down approach). Abstractions are used to cope with situations in which no sufficient control knowledge is available in order to plan a deduction in ground space.

References

[AuHu97] S. Autexier, D. Hutter. Parameterized Abstractions used for Proof Planning *Research Report RR 97-04*, DFKI GmbH, Saarbrücken, 1997

[BaWa93] D. Basin and T. Walsh. Difference unification. In *13th IJCAI*, Morgan Kaufmann, 1993.

[Ble90] W. Bledsoe. Challenge problems in elementary calculus *Journal of Automated Reasoning*, Vol. 6(3), pp. 341-359, 1990

[BvHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, Vol. 7, 1991.

[Dig85] V.J. Digricoli. The management of heuristic search in boolean experiments with RUE resolution. *9th IJCAI*, pp. 1154 – 1161, Los Angeles, California, 1985.

[GOBS69] J. Guard, and F. Oglesby, and J. Bennett, and L. Settle. Semi-automated mathematics. In *Journal of the ACM*, pp. 49-62, Vol. 16, 1969

[GW92a] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 56(2–3), pp. 323–390, 1992. Also available as DAI Research Paper No 516, Dept. of Artificial Intelligence, Edinburgh.

[Hut90] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th CADE*, Springer, LNAI 449, 1990

[Hut96] D. Hutter. Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, forthcoming, accepted 1996.

[Hut96a] D. Hutter. Using rippling for equational reasoning. *KI-96, Advances in Artificial Intelligence*, LNAI 1137, Springer, 1996

[HS96] D. Hutter, and C. Sengler. INKA - The Next Generation In J. Slaney, M.McRobbie, editor, *13th CADE*, Springer, LNAI 1104, 1996

[HLSSSW96] D. Hutter, B. Langenstein, C. Sengler, J. Siekmann, W. Stephan, and A. Wolpers. Deduction in verification support environment (vse). In *Formal Method Europe*, LNCS 1051, Springer, Oxford, Great Britain, 1996.

[Kre65] G. Kreisel. Lectures on Modern Mathematics. *Mathematical Logic Vol.3*, pp. 95-195, J.Wiley, Sons, 1965

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc. 95*, pp. 210-255, May 1960

[LuMC93] E.L. Lusk, W.M. McCune. Uniform Strategies: The CADE-11 Theorem Proving Contest. In *Journal of Automated Reasoning*, Vol. 11, pp. 317-331, 1993

[Mor69] J. Morris. E-resolution: an extension of resolution to include the equality relation. In *IJCAI-69*, 1969.

[NeShSi63] A. Newell, J.C. Shaw, and H.A. Simon. The logic theory machine. In Feigenbaum and Feldman, editors, *Computers and Thought*, pp. 61–79. McGraw-Hill, 1963.

[Pla81] D. Plaisted. Theorem proving with abstractions. *Artificial Intelligence*, 16:47–108, 1981.

[QuHu68] J. Quinlan and E. Hunt. A formal deductive problem-solving system. *Journal of the ACM*, 15(4):625–646, 1968.

[SuSu95] G. Sutcliff, C.B. Suttner. The TPTP Problem Library Technical Report AR-95-03, Institut für Informatik, TU München, Germany, 1995

[YBGWB94] T. Yoshida, A. Bundy, I. Green, T. Walsh, D. Basin. Coloured Rippling: An extension of a theorem proving heuristic In A.G.Cohn, editor, *ECAI-94*, pp. 85-89, John Wiley & Sons, 1994

List of Participants

Bonacina, Maria Paola
Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419, USA
e-mail: bonacina@cs.uiowa.edu

Clavel, Manuel
Computer Science Laboratory, SRI International
Menlo Park, CA 94025, USA
e-mail: clavel@csl.sri.com

Dahn, Bernd Ingo
Institut für Reine Mathematik, Humboldt University Berlin
Unter den Linden 6, PSF 1297, 10099 Berlin, Germany
e-mail: dahn@mathematik.hu-berlin.de

Dennis, Louise
Department of Artificail Intelligence, University of Edinburgh
80, South Bridge, Edinburgh EH1 1HN, Scotland
e-mail: louised@dai.ed.ac.uk

Fuchs, Matthias
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern, Germany
e-mail: fuchs@informatik.uni-kl.de

Gramlich, Bernhard
INRIA-Lorraine & CRIN
615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy Cedex, France
e-mail: Bernhard.Gramlich@loria.fr

Harland, James
Department of Computer Science, Royal Melbourne Institute of Technology
GPO Box 2476V, Melbourne, 3001, Australia
e-mail: jah@cs.rmit.edu.au

Hutter, Dieter
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
e-mail: hutter@dfki.uni-sb.de

Kirchner, Claude
INRIA-Lorraine & CRIN
615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy Cedex, France
e-mail: Claude.Kirchner@loria.fr

Kirchner, Hélène
CRIN-CNRS & INRIA-Lorraine
Campus Scientifique, Bâtiment LORIA, BP 239, 54506 Vandoeuvre-lès-Nancy, France
e-mail: Helene.Kirchner@loria.fr

Middeldorp, Aart
Institute of Information Sciences and Electronics, University of Tsukuba
Tsukuba 305, Japan
e-mail: ami@score.is.tsukuba.ac.jp

Zinn, Claus
Department of Artificial Intelligence, University of Erlangen-Nürnberg
Am Weichselgarten 9, 91058 Erlangen, Germany
e-mail: zinn@immd8.informatik.uni-erlangen.de