Strategic Computation and Deduction

Claude Kirchner, Florent Kirchner, Hélène Kirchner

> I'd like to conclude by emphasizing what a wonderful field this is to work in. Logical reasoning plays such a fundamental role in the spectrum of intellectual activities that advances in automating logic will inevitably have a profound impact in many intellectual disciplines. Of course, these things take time. We tend to be impatient, but we need some historical perspective. The study of logic has a very long history, going back at least as far as Aristotle. During some of this time not very much progress was made. It's gratifying to realize how much has been accomplished in the less than fifty years since serious efforts to mechanize logic began.
>
> Peter B. Andrews [2]

1 Introduction

Strategies, tactics, tacticals, proof plans, are terms widely used in artificial intelligence, in automated or interactive reasoning, in semantics of programming languages as well as in every day life. But what do they mean? What are these concepts used for and why is there so many different points of view? We do not claim to address all these questions here, but will contribute to define in a uniform way what strategies are for computation and deduction.

The complementarity between deduction and computation, as emphasized in particular in deduction modulo [18], allows us to now envision a completely new generation of proof assistants where customized deductions are performed modulo appropriate and user definable computations [11, 12]. This has in particular the advantage to allow for a uniform implementation of higher-order and first-order logics [15] making possible the safe use of existing dedicated proof environments [34, 19, 9]. This generalizes the approaches typical in first-order theorem proving [43], as well as higher-order ones like PVS [47], TPS [3, 4], Omega [8, 51] or Coq [20], to mention just a few.

To encompass this view of computation and deduction uniformly, we start from a rule-based view point. Rule-based reasoning is present in many domains of computer science: in formal specifications, rewriting is used for prototyping specifications; in theorem proving, for dealing with equality, simplifying the formulas and pruning the search space; in programming languages, the rule object can be explicit like in PROLOG, OBJ or ML, or hidden in the operational semantics; expert systems use rules to describe actions to perform; in constraint logic programming, solvers are described via rules transforming constraint systems. XML document transformations, access-control policies or bio-chemical reactions are a few examples of application domains.

But deterministic computations or deductions based on inference rules are often not sufficient to capture every computation or proof development. A formal mechanism is needed, for instance, to sequentialize the search for different solutions, to check context conditions, to request user input to instantiate variables, to process subgoals in a particular order, etc. This is the place where the notion of strategy comes in.

Reduction strategies in term rewriting study which expressions should be selected for evaluation and which rules should be applied. These choices usually increase efficiency of evaluation but may affect fundamental properties of computations such as confluence or (non-)termination. Programming languages like ELAN, Maude and Stratego allow for the explicit definition of the evaluation strategy, whereas languages like Clean, Curry, and Haskell allow for its modification.

In theorem proving environments, including automated theorem provers, proof checkers, and logical frameworks, strategies (also called tacticals in some contexts) are used for various purposes, such as proof search and proof planning, restriction of search spaces, specification of control components, combination of different proof techniques and computation paradigms, or meta-level programming in reasoning systems.

Strategies are thus ubiquitous in automated deduction and reasoning systems, yet only recently have they been studied in their own right. In the two communities of automated deduction and rewriting, workshops have been launched to make progress towards a deeper understanding of the nature of strategies, their descriptions, their properties, and their usage.

In this paper we would like to contribute to the theoretical foundations of strategies and to the convergence of different points of view, namely rewriting-based computations on one hand, rule-based deduction and proof-search on the other hand. We will rely on previous works and strategy languages that have been recently designed and studied. In rewriting, from elementary strategies expressions directly issued from a term rewrite system

R, more elaborated strategies expressions can be built using a strategy language like in ELAN [32, 10], Stratego [54], TOM [6] or more recently Maude [42]. The semantics of such a language is naturally described in the rewriting calculus [13, 14]. A similar mechanism also exists in proof systems, where a set of core strategies, historically derived from the language LCF, is used to program sophisticated proof search patterns. The semantics of such a language appears in [17, 29, 33].

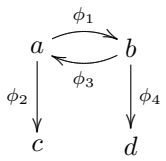
Building upon the definition of abstract reduction systems (recalled in Section 2), the main contributions of this paper are as follows: we propose in Section 3 a notion of abstract strategies together with adequate properties of termination, confluence and normalization under strategy. Thanks to this abstract concept and to the rewriting calculus (recalled in Section 4), we are able to draw a parallel between strategies for computation and strategies for deduction. While strategies for computation, developed in Section 5, essentially rely on the largely explored and well-known domain of term reduction by rewriting or narrowing, strategies for deduction, developed in Section 6, require to introduce an original point of view: we define deduction rules as rewrite rules, a deduction step as a rewriting step, a deduction system as an abstract reduction system. Strategic deductions are there of interest for developing complete proof trees. The same vision allows us to introduce proof construction as narrowing derivation. Computation, deduction and proof search are then captured as the foundational concept of abstract strategy.

2 Abstract reduction systems

When abstracting the notion of strategies, one important preliminary remark is that we need to start from an appropriate notion of *abstract reduction system* (ARS) based on the notion of graph instead of relation. This is due to the fact that, speaking of derivations, we need to make a difference between "being in relation" and "being connected". Typically modeling ARS as relations as in [5] allows us to say that, e.g, a and b are in relation but not that there may be two ways to derive b from a. Consequently, we need to use the more expressive approach of [52, 36], based on a notion of oriented graph instead of that of a relation.

DEFINITION 1. An *abstract reduction system* (ARS) is a labelled oriented graph $(\mathcal{O}, \mathcal{S})$. The nodes in $\mathcal{O}$ are called *objects*, the oriented edges in $\mathcal{S}$ are called *steps*.

EXAMPLE 2. We use the standard graphical representation of binary oriented graph.

1. A first standard example: $\mathcal{A}_{lc} =$ 

2. The interest of using graph instead of binary relations is exemplified by $\mathcal{A}_c =$ 

The next definitions can be seen as a renaming of usual ones in graph theory. Their interest is to allow us to define uniformly term rewriting derivations and strategies. If the concepts are not original, their presentation is.

DEFINITION 3 (Derivation). For a given ARS $\mathcal{A}$:

1. A *reduction step* is a labelled edge ϕ together with its source a and target b. This is written $a \to\!\!\!\triangleright^{\phi}_{\mathcal{A}} b$, or simply $a \to\!\!\!\triangleright^{\phi} b$ when unambiguous.

2. An $\mathcal{A}$-*derivation* or $\mathcal{A}$-*reduction sequence* is a path π in the graph $\mathcal{A}$.

3. When it is finite, π can be written $a_0 \to\!\!\!\triangleright^{\phi_0} a_1 \to\!\!\!\triangleright^{\phi_1} a_2 \ldots \to\!\!\!\triangleright^{\phi_{n-1}} a_n$ and we say that a_0 reduces to a_n by the derivation $\pi = \phi_0\phi_1 \ldots \phi_{n-1}$; this is also denoted $a_0 \to\!\!\!\triangleright^{\phi_0\phi_1\ldots\phi_{n-1}} a_n$ or simply $a_0 \to\!\!\!\triangleright^{\pi} a_n$; n is the *length* of π.

   (a) The *source* of π is the object a_0 and $dom(\pi) = \{a_0\}$.

   (b) The *target* of π is the object a_n and $(\pi\ a_0) = \{a_n\}$. This is also denoted simply πa_0 when there is no syntactic ambiguity.

4. The set of all derivations is denoted $\mathcal{D}(\mathcal{A})$.

5. A derivation is *empty* when its length is zero, in which case its source and target are the same. The empty derivation issued from a is denoted id_a.

6. The *concatenation* of two derivations π_1 and π_2 is defined if there exist objects $a \in dom(\pi_1)$ and $b \in (\pi_1 a) \cap dom(\pi_2)$. Then $\pi_1;\pi_2$ denotes the new $\mathcal{A}$-derivation $a \to\!\!\!\triangleright^{\pi_1}_{\mathcal{A}} b \to\!\!\!\triangleright^{\pi_2}_{\mathcal{A}} c$, such that $a \in dom(\pi_1;\pi_2)$ and $(\pi_1;\pi_2 a) = (\pi_2(\pi_1 a)) = \{c\}$.

Note that an $\mathcal{A}$-derivation is the concatenation of its reduction steps.

EXAMPLE 4. Following the previous examples, we have:

1. $\mathcal{D}(\mathcal{A}_{lc}) \supset \{id_a, \phi_1, \phi_1\phi_3, \phi_1\phi_4, \phi_1\phi_3\phi_1, (\phi_1\phi_3)^n, (\phi_1\phi_3)^\omega, \ldots\}$, where ϕ^n denotes the n-steps iteration of ϕ and ϕ^ω denotes the infinite iteration of ϕ;
2. $\mathcal{D}(\mathcal{A}_c) \supset \{\phi_1, \phi_1, \phi_1\phi_2, \ldots, (\phi_1)^\omega, (\phi_2)^\omega, \ldots\}$.

The following definitions state general properties of an ARS.

DEFINITION 5 (Termination). For a given ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$:

- $\mathcal{A}$ is *terminating* (or *strongly normalizing*) if all its derivations are of finite length;
- An object a in $\mathcal{O}$ is *normalized* when the empty derivation is the only one with source a (e.g., a is the source of no edge);
- A derivation is *normalizing* when its target is normalized;
- An ARS is *weakly terminating* if every object a is the source of a normalizing derivation.

DEFINITION 6 (Confluence). An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is *confluent* if for all objects a, b, c in $\mathcal{O}$, and all $\mathcal{A}$-derivations π_1 and π_2, when $a \twoheadrightarrow^{\pi_1} b$ and $a \twoheadrightarrow^{\pi_2} c$, there exist d in $\mathcal{O}$ and two $\mathcal{A}$-derivations π_3, π_4 such that $c \twoheadrightarrow^{\pi_3} d$ and $b \twoheadrightarrow^{\pi_4} d$.

3 Abstract strategies

We use a general definition, compliant with [32] and slightly different from the one used in, e.g., [52].

DEFINITION 7 (Abstract Strategy). For a given ARS $\mathcal{A}$:

1. An *abstract strategy* ζ is a subset of the set of all derivations (finite or not) of $\mathcal{A}$.
2. Applying the strategy ζ on an object a is denoted ζa. It denotes the set of all objects that can be reached from a using a derivation in ζ:

$$\zeta a = \{b \mid \exists \pi \in \zeta \text{ such that } a \twoheadrightarrow^{\pi} b\} = \{\pi a \mid \pi \in \zeta\}$$

When no derivation in ζ has source a, we say that the strategy application on a fails.

3. Applying the strategy ζ on a set of objects consists in applying ζ to each element a of the set. The result is the union of ζa for all a in the set of objects.

4. The *domain* of a strategy is the set of objects that are source of a derivation in ζ:
$$dom(\zeta) = \bigcup_{\delta \in \zeta} dom(\delta)$$

5. The strategy that contains all the empty derivations is denoted Id:
$$Id = \{id_a \mid a \in \mathcal{O}\}$$

Note that, with this definition, a strategy is not defined on all objects of the ARS and becomes a partial function. Indeed, a strategy that contains only infinite derivations from source a is undefined on a. This has to be distinguished from the case where a strategy contains no derivations from source a and returns the empty set $\emptyset$, i.e., the strategy application fails. The empty set of derivations is a strategy called *Fail*; its application always fails. Notice finally that instead of returning *sets* of results, we might define strategies to return *multisets* or even *list* or any appropriate data structure.

EXAMPLE 8 (Example 2 continued). For $\mathcal{A}_{lc}$, let us define and examine a few strategies:

1. $\zeta_1 = \mathcal{D}(\mathcal{A}_{lc})$, i.e., all the derivations. So we have for example: $\zeta_1 a = \{a, b, c, d\}$.
   If multisets of results are considered, $\zeta_1 a = \{a, a, ..., b, c, d\}$ in which a occurs for each finite derivation which approximates $(\phi_1\phi_3)^\omega$.

2. $\zeta_2 = \emptyset (= \mathit{Fail})$: failure, no applicable derivation, i.e., for all x in $\mathcal{O}_{lc}$, $\zeta_2 x = \emptyset$.

3. $\zeta_3 = \{(\phi_1\phi_3)^*\phi_4\}$,
   a always converges to d: $\zeta_3 a = \{d\}$;
   b is not transformed (as well as c and d): $\zeta_3 b = \emptyset$.

4. The result of $((\phi_1\phi_3)^\omega\ a)$ is the empty set.

The standard notions of ARS termination and confluence must be carefully extended to abstract strategies.

DEFINITION 9 (Termination under strategy). For a given ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ and strategy ζ:

- $\mathcal{A}$ is ζ-*terminating* if all derivations in ζ are of finite length;
- An object a in $\mathcal{O}$ is ζ-normalized when the empty derivation is the only one in ζ with source a;
- A derivation is ζ-*normalizing* when its target is ζ-normalized;
- An ARS is *weakly* ζ-*terminating* if every object a is the source of a ζ-normalizing derivation.

EXAMPLE 10 (Example 2 continued). Given $\mathcal{A}_{lc}$ and the strategy ζ defined as $a \twoheadrightarrow^{\phi_1} b \twoheadrightarrow^{\phi_4} d$, b is ζ-normalized since there is no derivation in ζ with source b.

It could be tempting to generalize Definition 6 as follows:
An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ *is* confluent *under strategy* ζ *if for all objects* a, b, c *in* $\mathcal{O}$, *and all* $\mathcal{A}$*-derivations* π_1 *and* π_2 *in* ζ, *when* $a \twoheadrightarrow^{\pi_1} b$ *and* $a \twoheadrightarrow^{\pi_2} c$ *there exists* d *in* $\mathcal{O}$ *and two* $\mathcal{A}$*-derivations* π_3, π_4 *in* ζ *such that* $c \twoheadrightarrow^{\pi_3} d$ *and* $b \twoheadrightarrow^{\pi_4} d$.
However this generalization is not correct since nothing ensures that the derivations $c \twoheadrightarrow^{\pi_3} d$ and $b \twoheadrightarrow^{\pi_4} d$ belong to the strategy. This leads to two possible definitions of respectively weak and strong confluence under strategy.

DEFINITION 11 (Weak Confluence under strategy). An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is *weakly confluent* under strategy ζ if for all objects a, b, c in $\mathcal{O}$, and all $\mathcal{A}$-derivations π_1 and π_2 in ζ, when $a \twoheadrightarrow^{\pi_1} b$ and $a \twoheadrightarrow^{\pi_2} c$ there exists d in $\mathcal{O}$ and two $\mathcal{A}$-derivations π_3', π_4' in ζ such that $\pi_3' : a \twoheadrightarrow b \twoheadrightarrow d$ and $\pi_4' : a \twoheadrightarrow c \twoheadrightarrow d$.

DEFINITION 12 (Strong Confluence under strategy). An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is *strongly confluent* under strategy ζ if for all objects a, b, c in $\mathcal{O}$, and all $\mathcal{A}$-derivations π_1 and π_2 in ζ, when $a \twoheadrightarrow^{\pi_1} b$ and $a \twoheadrightarrow^{\pi_2} c$ there exists d in $\mathcal{O}$ and two $\mathcal{A}$-derivations π_3, π_4 in ζ such that:

1. $b \twoheadrightarrow^{\pi_3} d$ and $c \twoheadrightarrow^{\pi_4} d$;
2. $\pi_1; \pi_3$ and $\pi_2; \pi_4$ belong to ζ.

EXAMPLE 13 (Example 2 continued). Let us again consider $\mathcal{A}_{lc}$ and the following various strategies:

1. $\zeta_1 = \mathcal{D}(\mathcal{A}_{lc})$, i.e., all the derivations. $\mathcal{A}_{lc}$ is neither weakly nor strongly confluent under ζ_1: just consider $\pi_1 : a \twoheadrightarrow^{\phi_1} b \twoheadrightarrow^{\phi_4} d$ and $\pi_2 : a \twoheadrightarrow^{\phi_2} c$.
2. $\zeta_2 = \emptyset (= \mathit{Fail})$: $\mathcal{A}_{lc}$ is trivially both weakly and strongly confluent under ζ_2.

3. $\zeta_3 = \{(\phi_1\phi_3)^*\phi_4\}$: $\mathcal{A}_{lc}$ is also weakly and strongly confluent under ζ_3.
4. For a different reason, this is also the case for $\zeta_4 = (\phi_1\phi_3)^\omega$ whose result is the empty set.

To understand the difference between weak and strong confluence let us consider $\mathcal{O} = \{a, b, c, d\}$ and reduction steps $\phi_1, \phi_2, \phi_3, \phi_4, \phi_1', \phi_2', \phi_3', \phi_4'$. This ARS $\mathcal{A}_{lc}$ is weakly and strongly confluent under the strategy $\zeta = \{a \to^{\phi_1} b, a \to^{\phi_2} c, b \to^{\phi_3} d, c \to^{\phi_4} d, a \to^{\phi_1} b \to^{\phi_3} d, a \to^{\phi_2} c \to^{\phi_4} d\}$, but is not under $\zeta = \{a \to^{\phi_1} b, a \to^{\phi_2} c, b \to^{\phi_3} d, c \to^{\phi_4} d\}$. A is weakly but not strongly confluent under the strategy $\zeta = \{a \to^{\phi_1} b, a \to^{\phi_2} c, b \to^{\phi_3} d, c \to^{\phi_4} d, a \to^{\phi_1'} b \to^{\phi_3'} d, a \to^{\phi_2'} c \to^{\phi_4'} d\}$.

4 The rewriting calculus

The rewriting calculus or ρ-calculus generalizes term rewriting and lambda-calculus. It has been introduced in [13]. We recall here the main syntactic ingredients necessary to set-up the framework developed in this paper, using a syntax that restricts patterns to be only algebraic, a case simpler than the general one where pattern may contain abstractions (see [13]).

Definition 1 (Syntax). We consider the symbols "$\_ \twoheadrightarrow \_$" (abstraction operator), "$\_ \wr \_$" (structure operator), the (hidden) application operator, a set $\mathcal{X}$ of variables and a set $\mathcal{K}$ of constants, each of them having an arity denoted $ar(\mathcal{K})$. The syntax of the basic rewriting calculus is the following:

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{K} \mid \mathcal{P} \twoheadrightarrow \mathcal{T} \mid \mathcal{T}\,\mathcal{T} \mid \mathcal{T} \wr \mathcal{T} \qquad \text{Terms}$$

$$\mathcal{P} ::= \mathcal{X} \mid (\dots((\mathcal{K}\,\mathcal{P}_1)\,\mathcal{P}_2)\dots\mathcal{P}_{ar(\mathcal{K})} \qquad \text{Algebraic Patterns}$$

A *linear* pattern is a pattern where every variable occurs at most once.

We assume that the application operator associates to the left, while the other operators associate to the right. The priority of the application is higher than that of "$\_ \twoheadrightarrow \_$" which is, in turn, of higher priority than the "$\_ \wr \_$". In the following, the symbols $A, B, C, \dots$ range over the set $\mathcal{T}$ of terms, the symbols $x, y, z, \dots$ range over the set $\mathcal{X}$ of variables ($\mathcal{X} \subseteq \mathcal{T}$), the symbols $a, b, c, \dots, f, g, h, \dots$ and strings built from them range over a set $\mathcal{K}$ of constants ($\mathcal{K} \subseteq \mathcal{T}$). Finally, the symbols P, Q range over the set $\mathcal{P}$ of patterns, ($\mathcal{X} \subseteq \mathcal{P} \subseteq \mathcal{T}$). Vectors of terms $(A_1, \dots, A_n)$ are denoted by $\overline{A}$. We usually denote a term of the form $(\dots((f\ A_1)\ A_2)\dots\ A_n)$ by $f(A_1, A_2, \dots, A_n)$. Identity of terms is denoted by $\equiv$.

A term of the form $P \twoheadrightarrow B$ is an *abstraction* (or *rule*) with pattern P and body B; intuitively, the free variables of P are bound in B. The term $A \wr B$ is called a *structure*.

As for the lambda-calculus for instance, the reduction relation $\mapsto\!\!\!\!\rightarrow$ of the rewriting calculus is defined as the smallest congruence generated by the following top-level reductions:

$$(P \twoheadrightarrow A)\, B \quad \rightarrow_p \quad A\theta \qquad\qquad \text{if } \exists\theta.\ P\theta \equiv B$$

$$(A \wr B)\, C \quad \rightarrow_\delta \quad A\, C \wr B\, C$$

The one step reduction relation is denoted by $\mapsto_p$ or $\mapsto_\delta$ according to the reduction rule which is used. Notice that in the main reduction ($\rightarrow_p$), the condition consists in checking the existence of a matching substitution θ. When it exists, this substitution is applied to A, taking care of free and bounded variables as usual in a higher-order setting (remember that $\twoheadrightarrow$ is an abstractor).

5 Strategic reduction

Let us now come back to the more classical context of first-order terms and term rewriting to see how it fits to the general notions presented above. Basic definitions on term rewriting can be found in [31, 35, 5]. The following standard notations will be used in the following. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of first-order terms built from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variables. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. A substitution σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, with a finite domain $\{x_1, \ldots, x_k\}$ and is written $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$.

5.1 Term rewriting systems

A rewrite rule is an ordered pair of terms $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted $l \to r$, where it is often required that l is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. We will drop these restrictions later. The terms l and r are respectively called the left-hand side and the right-hand side of the rule. A *rewrite system* is a (finite or infinite) set of rewrite rules. Rules can be labelled to easily talk about them.

DEFINITION 14 (One step rewriting). Given a rewrite system R, an algebraic term t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to an algebraic term t' if there exists a rewrite rule $l \to r$ of R and a position ω in t, such that in the ρ-calculus, applying the ρ-term $(l \twoheadrightarrow r)$ to t at position ω evaluates to t', which is denoted by

$$t[(l \twoheadrightarrow r)t_{|\omega}]_\omega \mapsto_p t'$$

This is denoted $t \longrightarrow_\omega^{l \to r} t'$ or $t \longrightarrow_\omega^{R} t'$ when we do not need to make precise which rewrite rule is used.

Indeed this means that there exists a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t[\sigma(r)]_\omega$. A subterm $t_{|\omega}$ where the rewriting step is applied is called *redex*. A term that has no redex is said to be irreducible for R or to be in *R-normal form*.

To a set of rewrite rules corresponds directly a unique abstract reduction system that can be seen as a generic way to describe the set of all derivations.

DEFINITION 15 (Reduction system). Given a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ and a set of rewrite rules R, a reduction system is an abstract reduction system $\mathcal{R} = (\mathcal{O}_R, \mathcal{S}_R)$ with:

- $\mathcal{O}_R = \mathcal{T}(\mathcal{F}, \mathcal{X})$, and
- $\mathcal{S}_R = \{t \to t' | t \longrightarrow_\omega^R t' \text{ for } \omega \text{ a position in } t\}$.

It is now possible to give the definition of strategic rewriting:

DEFINITION 16 (Strategic rewriting).

Given an abstract reduction system $\mathcal{R} = (\mathcal{O}_R, \mathcal{S}_R)$ generated by a term rewrite system R, and a strategy ζ of $\mathcal{R}$, a strategic rewriting derivation (or rewriting derivation under strategy ζ) is an element of ζ. A strategic rewriting step under ζ is a rewriting step $t \longrightarrow_\omega^R t'$ that occurs in a derivation of ζ.

Notice that the previous definitions extend immediatly to the case where matching is performed modulo an equational theory like associativity-commutativity of some fonction symbols. Strategic rewriting can therefore be defined for rewriting modulo relations as defined in [49, 30].

Besides the rewriting relation, another reduction relation, called narrowing, is worth considering to develop an ARS. This well-known process, introduced in [21, 28], is quite similar to rewriting but there, matching is replaced by unification. Let us recall its usual definition:

DEFINITION 17 (One step narrowing). Given a term rewrite system R, an algebraic term t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ narrows to an algebraic term t' if there exists a rewrite rule $l \to r$ of R and a position ω in t, such that $t|_\omega$ and l are unifiable with a most general unifier σ. Then $t' = \sigma(t[r]_\omega)$. This is denoted $t \rightsquigarrow_{\omega,\sigma}^{l \to r} t'$ where either ω or σ may be omitted, or simply $t \rightsquigarrow^R t'$ when we do not need to make precise which rewrite rule is used.

For instance, using the rewrite rule $f(x, x) \to x$ the term $f(g(y), z)$ narrows to the term $g(y)$ with the substitution $\sigma = \{x \mapsto g(y), z \mapsto g(y)\}$.

Then, we can also consider that a rewrite system R generates in a similar way an abstract reduction system $\mathcal{N} = (\mathcal{O}_R, \mathcal{S}_R)$ with:

- $\mathcal{O}_R = \mathcal{T}(\mathcal{F}, \mathcal{X})$, and

- $\mathcal{S}_R = \{t \rightsquigarrow t' | t \rightsquigarrow_\omega^{l \to r} t'$ for ω a position in $t\}$.

As for the rewriting relation, we will therefore use the definitions given in the abstract setting and we can define strategic narrowing:

DEFINITION 18 (Strategic narrowing).

Given an abstract reduction system $\mathcal{N} = (\mathcal{O}_R, \mathcal{S}_R)$ generated by a term rewrite system R, and a strategy ζ of $\mathcal{N}$, a strategic narrowing derivation (or narrowing derivation under strategy ζ) is an element of ζ. A strategic narrowing step under ζ is a narrowing step $t \rightsquigarrow_\omega^R t'$ that occurs in a derivation of ζ.

5.2 Term rewriting strategies

In the classical setting of first-order term rewriting, strategies have been used to determine at each step of a derivation which is the next redex. Thus they have often been defined as functions on the set of terms like in [52]. In our general setting of abstract strategies, the definitions of these standard strategies express a property that a derivation must satisfy to belong to the strategy of interest. Let us illustrate this point of view by a few examples of strategies that have been primarily provided to describe the operational semantics of functional programming languages and the related notions of call by value, call by name, call by need.

Leftmost-innermost and outermost reduction

Let R be a rewrite system on $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A rewriting derivation under the innermost (resp. outermost) strategy verifies : for a given term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t \rightarrow_\omega^R t'$ and the rewriting position ω in t is such that there is no suffix (resp. prefix) position ω' of ω such that t rewrites at position ω'.

Lazy reduction

Lazy reduction, as performed in Haskell for instance, or described in [22], combines innermost and outermost strategies, in order to avoid redundant rewrite steps. For that, operators of the signature have labels specifying which arguments are lazy or eager. Positions in terms are then annotated as lazy or eager, and the strategy consists in reducing the eager subterms only when their reduction allows a reduction step higher in the term [46].

Lazy reduction can also be performed with other mechanisms, such as local strategies or context-sensitive rewriting.

Reduction under local strategies

Local strategies on operators, used for instance in the OBJ-like languages [24], introduce a function LS from $\mathcal{F}$ to the set $\mathcal{L}(\mathbb{N})$ of lists of integers to specify which and in which order the arguments of each function symbol have to be reduced. At each step of the derivation, for the current

term t, the next redex is chosen according to $LS(f)$ where f is the top operator of the term t.

Context-sensitive reduction

Local strategies are close to context-sensitive rewriting [39, 40, 1], where rewriting is also allowed only at some specified positions in the terms. The former specify an ordering on these rewriting positions, so they are more specific than context-sensitive rewriting where a redex is chosen in a set of positions. More precisley, here, a replacement map is a mapping $\mu : \mathcal{F} \mapsto \mathcal{P}(\mathbb{N})$ satisfying $\mu(f) \subseteq \{1, \ldots, k\}$, for each k-ary symbol f of a signature $\mathcal{F}$ [38]. Replacement maps discriminate the argument positions on which the rewriting steps are allowed.

Needed and standard reduction

In [26, 27], the notions of needed and strongly needed redexes are defined for orthogonal rewrite systems. The main idea here is to find the optimal way, when it exists, to reach the normal form a term. A redex is needed when there is no way to avoid reducing it to reach the normal form. Reducing only needed redexes is clearly the optimal reduction strategy, as soon as needed redexes can be decided, which is not the case in general.

5.3 Strategy language for reduction

This section is devoted to expressing rewriting strategies; exactly the same principles could be used to express other strategies as we will see later.

In the 1990s, generalizing OBJ's concept of local strategies, the idea has emerged to better formalize the control on rewriting which was performed implicitly in interpreters or compilers of rule-based programming languages. Then instead of describing a strategy by a property of its derivations, the idea is now to provide a *strategy language* to specify which derivations we are interested in.

Various approaches have followed, yielding different strategy languages such as ELAN [32, 10], APS [37], Stratego [54], TOM [6] or more recently Maude [42]. All these languages share the concern to provide abstract ways to express control of rule applications, by using reflexivity and the meta-level for Maude, or a strategy language for ELAN, Stratego or ASF+SDF. Strategies such as bottom-up, top-down or leftmost-innermost are high level ways to describe how rewrite rules should be applied. TOM, ELAN, Maude and Stratego provide flexible and expressive strategy languages where high-level strategies are defined by combining low-level primitives. In this section, we choose TOM [7][1] to illustrate the construction of strategy languages of this family and we describe below the main elements of the language. The

[1] `http://tom.loria.fr`

semantics of the TOM strategy language as well as others are naturally described in the rewriting calculus [13, 14].

We can distinguish two classes of constructs in the strategy language: the first one allows construction of derivations from the basic elements, namely the rewrite rules. The second one corresponds to constructs that express the control, especially deterministic and undeterministic choice. Moreover, the capablility of expressing recursion in the language brings even more expressive power.

Elementary strategies

An elementary strategy is either $Identity$ which corresponds to the set Id of all empty derivations, $Fail$ which denotes the empty set of derivations $Fail$, or a set of rewrite rules R which represents one-step derivations with rules in R at the root position. $Sequence(s_1, s_2)$, also denoted $s_2; s_1$ is the concatenation of s_1 and s_2 whenever it exists: $Sequence(s_1, s_2)t = s_2(s_1 t)$.

Control strategies

A few constructions are needed to build derivations, branching and take into account the structure of elements (terms).

choice $Choice(s_1, s_2)$ selects the first strategy that does not fail; it fails if both fail:

$$Choice(s_1, s_2)t = s_1 t$$

if $s_1 t$ does not fail else $s_2 t$.

all subterms On a term t, $All(s)$ applies the strategy s on all immediate subterms:

$$All(s)f(t_1, ..., t_n) = f(t'_1, ..., t'_n)$$

if $st_1 = t'_1, ..., st_n = t'_n$; it fails if there exists i such that st_i fails.

one subterm On a term t, $One(s)$ applies the strategy s on the first immediate subterm where s does not fail:

$$One(s)f(t_1, ..., t_n) = f(t_1, ..., t'_i, ..., t_n)$$

if for all $j < i$, st_j fails, and $st_i = t'_i$; it fails if for all i, st_i fails.

fixpoint The μ recursion operator (comparable to rec in OCaml) is introduced to allow the recursive definition of strategies. $\mu x.s$ applies the derivation in s with the variable x instantiated with $\mu x.s$:

$$\mu x.s = s[x \leftarrow \mu x.s]$$

These strategies are then composed to build more elaborated ones. For instance $Try(s) = choice(s, Identity)$: $Try(s)$ applies s if it can, and performs the $Identity$ otherwise. The All and One combinators are used in combination with the fixpoint operator to define tree traversals. For example, we have $TopDown(s) = \mu x.Sequence(s, All(x))$: the strategy s is first applied on top of the considered term, then the strategy $TopDown(s)$ is recursively called on all immediate subterms of the term.

6 Strategic deduction

In the previous section, we have seen how the abstract strategy concept instantiates to control rule-based computation. Let us now explore the deduction side.

In proof assistants such as Coq [20] and PVS [47], the interaction language (or *proof language*) contains two kinds of *proof commands*: *tactics*, which construct the proof tree by applying logical rules, and *strategies* which provide proof search control[2]. In this section, we provide innovative definitions of these two concepts, starting with tactics.

6.1 Proof rewriting systems

We describe the proof construction mechanism using the abstract reduction system framework. Our purpose in this section is not to formalize any particular instance of a deduction system, such as classical natural deduction [50] or intuitionistic sequent calculus [23]: our model is set-theoretic. Roughly, sequents/typing judgements are composed of a finite set of labelled hypotheses, a finite set of labelled conclusions, and a proof term.

DEFINITION 19 (Labelled proposition). Given a proof term language $\mathcal{T}$ and a logical language $\mathcal{L}$, a labelled proposition is a pair consisting of a proof term $\pi \in \mathcal{T}$ and a logical proposition $A \in \mathcal{L}$, noted $\pi : A$.

DEFINITION 20 (Sequent). Given a proof term language $\mathcal{T}$ and a logical language $\mathcal{L}$, a sequent is a triple $s = (\pi, \Gamma, \Delta)$, where π is a proof term of $\mathcal{T}$, and where Γ and Δ (respectively the *antecedent* and the *consequent*) are finite sets of labelled propositions of $\mathcal{L}$. In the following, this triple will be denoted: $\pi \therefore (\Gamma \vdash \Delta)$, or in short $\pi \therefore s$. The (infinite) set of sequents will be denoted $\mathcal{S}(\mathcal{T}, \mathcal{L})$.

This representation of sequents differs from the usual definition, given in natural deduction, where all proof terms are attached to a proposition in the antecedent or in the consequent, and a sequent is just the pair $\Gamma \vdash \Delta$.

[2] Various other terminologies exist in this field: for instance, PVS tactics are called *proof rules*, while Coq strategies are names *tacticals*. We use the words "tactic" and "strategy" because they appropriately convey a notion of locality.

The usual representation can be encoded in our formalism using a reserved formula label '$*$', as illustrated by the following example:

EXAMPLE 21. Let us take Λ, the set of simply typed lambda-terms, for $\mathcal{T}$, and FOL, the set of all first-order propositions, for $\mathcal{L}$. The sequent stating that under the assumption $(t : A)$, one of the statements $(\lambda x.t : B \Rightarrow A)$ or $(u : C)$ is provable, is denoted in natural deduction:

$$t : A \vdash \lambda x.t : B \Rightarrow A, u : C$$

Assume a proof of this sequent begins with the deconstruction of the λ-proof term $\lambda x.t$, using the "$\Rightarrow$-intro" deduction rule. Since it is the central element of the next deduction step, this proof term has a particular status, and can be qualified as *pivotal*. In order to express the aforementioned sequent within the framework set by Definition 19, the pivotal proof term is separated from its proposition, and the '$*$' placeholder put in its place to label the proposition. Hence the sequent is written:

$$\lambda x.t \therefore (t : A \vdash * : B \Rightarrow A, u : C)$$

Since a pivotal proof term can always be pinpointed, such a transformation can be made on any sequent of a proof in natural deduction.

On the other hand, separating the proof term from the antecedent and consequent allows us to capture more involved logical settings, where the pivotal proof terms might appear, not only at the level of formulas, but also at the level of sequents. For instance, in the case of sequent calculus *à la* Curien-Herbelin [16], the sequents $\Gamma \vdash \lambda x.v : A \Rightarrow B; \Delta$ and $\langle v|e\rangle : (\Gamma \vdash \Delta)$, where $\lambda x.v$ and $\langle v|e\rangle$ are pivotal $\bar{\lambda}\mu\tilde{\mu}$-proof terms, are well-formed. In our formalism, while the first sequent is expressed using the aforementioned '$*$' label using the same process as in Example 20, the second sequent is trivially represented by the isomorphic triple $\langle v|e\rangle \therefore (\Gamma \vdash \Delta)$, in which no '$*$' label appear. The same holds for Urban's sequent calculus [53], which places *all* proof terms on the same level as the antecedents and consequents.

Finally we need to add a notion of *list of sequents*: therefore we introduce the usual list constructors *cons* and *nil*, noted ';' and '$\varnothing$'. For typesetting purposes and when unambiguous, we abbreviate the list $\phi; \varnothing$ to ϕ, and we omit all list constructors for unary lists. Finally, the notion of substitution carried by terms and propositions is easily extended through simple subterm propagation to range over sequents and lists of sequents. The set of lists of sequents is denoted $\mathcal{LS}(\mathcal{T}, \mathcal{L})$.

A deduction system can be seen as a rewrite system operating on typed proof terms, i.e., sequents. The following definitions lay the groundwork

and lead to the expression of deduction systems as instances of the abstract reduction system framework.

DEFINITION 22 (Deduction rule). A rewrite rule, also called *deduction rule*, connects a single sequent to a list of sequents:

$$l \therefore (\Gamma_l \vdash \Delta_l) \to r_1 \therefore (\Gamma_{r_1} \vdash \Delta_{r_1}); \ldots; r_n \therefore (\Gamma_{r_n} \vdash \Delta_{r_n})$$

In order to capture the expressivity of logical rules, rewrite rules in this context may have a variable as left-hand side and may introduce in the right-hand side some variables that do not occur in the left-hand side. Indeed, relaxing these syntactic restrictions makes even more crucial the need of controlling rule application thanks to strategies.

DEFINITION 23 (One step deduction). We say that the sequent list $\phi = \pi_1 \therefore s_1; \ldots; \pi_m \therefore s_m$ reduces in one step to the sequent list ϕ' by the deduction rule $l \therefore s_l \to r_1 \therefore s_1; \ldots; r_n \therefore s_n$ if and only if there exists a substitution σ such that $\pi_i \therefore s_i = \sigma(l \therefore s_l)$ and

$$\begin{aligned}\phi' = \pi_1 \therefore s_1; \ldots; \pi_{i-1} \therefore s_{i-1};& \\ \sigma(r_1 \therefore s_1; \ldots; r_n \therefore s_n);& \\ \pi_{i+1} \therefore s_{i+1}; \ldots; \pi_m \therefore s_m&\end{aligned}$$

When appropriate, the matching step used in the previous definition can be applied modulo an equational theory, like in the following example.

EXAMPLE 24. First-order minimal natural deduction is perhaps the most well-known logical framework to support the proof-as-terms morphism. It considers first-order propositions as types for lambda-terms, and has typing judgements of the form $\Gamma \vdash t : A$. In our formalism, we have $\mathcal{T} = \Lambda$, $\mathcal{L} = \text{FOL}$ and sequents of the form $t \therefore (\Gamma \vdash * : A)$. The deduction rules for minimal natural deduction can be expressed as the following rewrite rules, where Γ is a variable representing a multiset of propositions (i.e., the separator "," is associative and commutative), t, u are proof term variables, A, B are proposition variables.

$$I \therefore (\Gamma \vdash * : \top) \to \varnothing \tag{D_1}$$

$$t \therefore (\Gamma, t : A \vdash * : A) \to \varnothing \tag{D_2}$$

$$\lambda x.t \therefore (\Gamma \vdash * : A \Rightarrow B) \to t \therefore (\Gamma, x : A \vdash * : B) \tag{D_3}$$

$$tu \therefore (\Gamma \vdash * : B) \to t \therefore (\Gamma \vdash * : A \Rightarrow B); u \therefore (\Gamma \vdash * : A) \tag{D_4}$$

$$\lambda x.t \therefore (\Gamma \vdash * : \forall x, B) \to t \therefore (\Gamma \vdash * : B) \tag{D_5}$$

$$tu \therefore (\Gamma \vdash * : B[u]_\omega) \to t \therefore (\Gamma \vdash * : \forall x, B[x]_\omega) \tag{D_6}$$

where I is a proof term constant. Notice the use of a free variable A (*resp.* x) in the right hand-side of rule (D_4) (*resp.* (D_6)). Based on this set of rewrite rules, a series of four reduction steps could be:

$$\begin{aligned}\lambda x.\lambda y.\,(yx) \therefore (\vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B) \\ \to^{(D_3)} \lambda y.\,(yx) \therefore (x : A \vdash * : (A \Rightarrow B) \Rightarrow B) \\ \to^{(D_3)} (yx) \therefore (x : A, y : A \Rightarrow B \vdash * : B) \\ \to^{(D_4)} y \therefore (x : A, y : A \Rightarrow B \vdash * : A \Rightarrow B); \\ x \therefore (x : A, y : A \Rightarrow B \vdash * : A) \\ \to^{(D_2)} \varnothing; \varnothing\end{aligned}$$

Note that in the previous example, the rule (D_2) can be applied because we assume the symbol "," to be associative and commutative. Another possibility could be to add a commutation rule, with the risk to inherit of the explicit treatment of commutativity at the proof design level.

DEFINITION 25 (Deduction system). Given a logical language $\mathcal{L}$ and a term algebra $\mathcal{T}$, a deduction system, also called a *proof rewriting system*, is an abstract reduction system $\mathcal{R} = (\mathcal{O}_R, \mathcal{S}_R)$ where:

- $\mathcal{O}_R$ contains elements of $\mathcal{LS}(\mathcal{T}, \mathcal{L})$, and
- $\mathcal{S}_R = \{\phi \to \phi' \mid \phi \longrightarrow_{\omega}^{\phi_l \to \phi_r} \phi'\}$ where ω is a position in ϕ (i.e., one of its sequents) and $\phi_l \to \phi_r$ is a deduction rule.

As for term rewriting, we will hereafter use all the definitions given in the abstract case for abstract reduction systems.

For a given deduction system $\mathcal{DS}$, a *completed proof* of a sequent ϕ is a derivation issued from ϕ and reaching the list $\varnothing; \ldots; \varnothing$ using the rules in $\mathcal{DS}$.

It is important to notice that the notion of derivation considered here concerns a full proof tree and not only one of its branches. This is due to the fact that we want to define strategies able to consistently develop complete proof trees and not only part of them. Therefore, the notions of deduction systems and of strategies are defined accordingly on *lists* of sequents representing proof trees.

Following these definitions, completed proofs are caracterized as reduction sequences, or equivalently as paths into the ARS formed by a deduction system. However, this leaves the question of the *construction* of a proof, i.e., of a proof term, unanswered. Yet this is a fundamental mechanism in the design of proof assistants. In order to address this question, various

authors [41, 45, 29] have shown that the syntax of proof terms and sequents can be extended with *proof metavariables*, that represent the still unknown parts of a proof.

DEFINITION 26 (Proof metavariables). Consider the syntax of proof terms extended with a special set of variables $\mathcal{M}$, called *metavariables*, noted $X, Y, Z \ldots$. We note $\mathcal{T}_{\mathcal{M}}$ the algebra of proof terms with metavariables, and meta(π) the set of metavariables appearing in π. We call *open sequent* a triple $\pi \therefore (\Gamma \vdash \Delta)$ where π contains at least one metavariable.

As shown in [45], the use of metavariables in logical settings featuring dependent types or polymorphism requires the addition of *constraints* to the representation of sequents. Since this extension is more elaborated, interesting but largely orthogonal to our current purpose, it will not be further detailed in this paper.

DEFINITION 27 (Proof grafting rule). A proof grafting rule is an ordered pair of sequents, denoted $L \therefore s \to r \therefore s$, where L is a proof metavariable and r may or may not contain metavariables.

A proof construction step can then be simply defined as the application of a proof grafting rule to a list of sequents, similarly to Definition 22. The sequent featuring the newly introduced proof term is then reduced using deduction rules, until the proof ends or a new grafting step is necessary.

EXAMPLE 28. Recall the minimal natural deduction formalism presented in Example 23. Assume the following rules for metavariable instantiation:

$$X \therefore (\Gamma \vdash * : \top) \to I \therefore (\Gamma \vdash * : \top) \tag{C_1}$$
$$X \therefore (\Gamma, x : A \vdash * : A) \to x \therefore (\Gamma, x : A \vdash A) \tag{C_2}$$
$$X \therefore (\Gamma \vdash * : A \Rightarrow B) \to \lambda x^A.Y \therefore (\Gamma \vdash * : A \Rightarrow B) \tag{C_3}$$
$$X \therefore (\Gamma \vdash * : A) \to (Y\,Z) \therefore (\Gamma \vdash * : A) \tag{C_4}$$
$$X \therefore (\Gamma \vdash * : \forall x, B) \to \lambda x.t \therefore (\Gamma \vdash * : \forall x, B) \tag{C_5}$$
$$X \therefore (\Gamma \vdash * : B[u]_\omega) \to tu \therefore (\Gamma \vdash * : B[u]_\omega) \tag{C_6}$$

While the proof grafting rules can be arbitrarily chosen, remark that these six rules each describe the construction of a λ-proof term head symbol. Furthermore, it is easy to verify that they build proofs terms that can be typechecked using the deduction rules for natural deduction. Based on this set of rewrite rules, the construction of the proof term of example 23 is

conducted by an alternation of deduction and grafting rules:

$$
\begin{aligned}
X_1 \therefore (\vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B)
&\to^{(C_3)} \lambda x^A.X_2 \therefore (\vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B)\\
&\to^{(D_3)} X_2 \therefore (x : A \vdash * : (A \Rightarrow B) \Rightarrow B)\\
&\to^{(C_3)} \lambda y^{A \Rightarrow B}.X_3 \therefore (x : A \vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B)\\
&\to^{(D_3)} X_3 \therefore (x : A, y : A \Rightarrow B \vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B)\\
&\to^{(C_4)} (X_4 X_5) \therefore x : A, y : A \Rightarrow B \vdash A \Rightarrow (A \Rightarrow B) \Rightarrow B\\
&\to^{(D_4)} X_4 \therefore (x : A, y : A \Rightarrow B \vdash * : A \Rightarrow B);\\
&\qquad\quad X_5 \therefore (x : A, y : A \Rightarrow B \vdash * : A)\\
&\to^{(C_2)} y \therefore (x : A, y : A \Rightarrow B \vdash * : A \Rightarrow B);\\
&\qquad\quad X_5 \therefore (x : A, y : A \Rightarrow B \vdash * : A)\\
&\to^{(D_2)} \varnothing; X_5 \therefore (x : A, y : A \Rightarrow B \vdash * : A)\\
&\to^{(C_2)} \varnothing; x \therefore (x : A, y : A \Rightarrow B \vdash * : A)\\
&\to^{(D_2)} \varnothing; \varnothing
\end{aligned}
$$

The synthesis of the grafting step instances then yields the final proof term. Remark that the alternate $C_i; D_i$ reductions correspond to the LCF definition of a tactic, i.e., a grafting step that instantiates a proof term metavariable followed by a series of deduction steps that infers new goals from the resulting sequent.

Yet this approach using metavariables and grafting can be significantly condensed by using a mechanism specific to the term rewriting domain. Indeed, the instantiation of a metavariable by a term, such that the latter becomes reducible by a rule of the deduction system, is exactly a narrowing step, as defined in Definition 16.

EXAMPLE 29. The deduction system of Example 23 can be used to *find* a proof of the proposition $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ by narrowing the initial query in the following way.

$$
\begin{aligned}
U \therefore (\vdash * : A \Rightarrow (A \Rightarrow B) \Rightarrow B) &\rightsquigarrow^{(D_3)}_{U \mapsto \lambda x.X} X \therefore (x : A \vdash * : (A \Rightarrow B) \Rightarrow B)\\
&\rightsquigarrow^{(D_3)}_{X \mapsto \lambda y.Y} Y \therefore (x : A; y : A \Rightarrow B \vdash * : B)\\
&\rightsquigarrow^{(D_4)}_{Y \mapsto TZ} T \therefore (x : A; y : A \Rightarrow B \vdash * : \alpha \Rightarrow B);\\
&\qquad\quad Z \therefore (x : A; y : A \Rightarrow B \vdash * : \alpha)\\
&\rightsquigarrow^{(D_2)}_{\alpha \mapsto A, T \mapsto y, Z \mapsto x} \varnothing; \varnothing
\end{aligned}
$$

The proof grafting rules can be recovered by annotating the unification assignments with the appropriate sequents:

$$\begin{aligned} X \therefore (\Gamma \vdash * : A \Rightarrow B) &\to \lambda x^A.Y \therefore (\Gamma \vdash * : A \Rightarrow B) \\ Y \therefore (\Gamma \vdash * : A) &\to (TZ) \therefore (\Gamma \vdash * : A) \\ T \therefore (\Gamma, x : A \vdash * : A) &\to x \therefore (\Gamma, x : A \vdash A) \end{aligned}$$

and the computed proof term is simply obtained by composing the narrowing substitutions:

$$\begin{aligned} U &= \{\alpha \mapsto A, T \mapsto y, Z \mapsto x\}(\{Y \mapsto (TZ)\}(\{X \mapsto \lambda y.Y\}(\lambda x.X))) \\ &= \lambda x.\lambda y.\,(yx) \end{aligned}$$

Notice in the previous example the use made of proof term variables (i.e., U, X, Y, Z, T) and the use of one proposition variable, i.e., α.

Hence, while traditionally a proof construction system was defined as an alternation of metavariable grafting and deduction steps, what emerges here is a more concise definition, based on the narrowing relation derived from a deduction system.

DEFINITION 30 (Proof construction system). Given a logical language $\mathcal{L}$, a term algebra $\mathcal{T}$ and a deduction system $\mathcal{DS}$ defined from a set D of proof rewrite rules, a *proof construction system* is an abstract reduction system $\mathcal{N} = (\mathcal{O}_D, \mathcal{S}_D)$ where:

- $\mathcal{O}_D$ contains elements of $\mathcal{LS}(\mathcal{T}, \mathcal{L})$, and
- $\mathcal{S}_D = \{\phi \rightsquigarrow \phi' \mid \phi \rightsquigarrow^D_\omega \phi'$ for ω a position in $\phi\}$.

This definition provides an explicit link between the deduction rules, which are usually well-known and documented as part of the underlying logical framework of a proof construction system, and the metavariable instantiation steps, which often are left implicit. As such, this formalism can be seen as an innovative contribution to the semantics of proof systems.

6.2 Proof rewriting strategies

The need for the creation of a mechanism for proof strategies was acknowledged along with the first theorem provers implementations. In [25], the authors remark that their initial attempts at using LCF [44], one of the first deduction software, were

> "limited by the fixed, and rather primitive, nature of its repertoire of commands",

and compared the experience to using a top-level interactive assembly language. What was needed, they analyzed, was a way to specify "recipes for proofs", i.e., strategies.

Interestingly, expressing a proof construction and deduction system as an instance of an abstract reduction system, gives us the possibility to extend the notion of abstract strategy (as defined in Section 3) to proof rewriting systems. The result is the following definition of a proof strategy.

DEFINITION 31 (Proof strategy). A proof strategy is a subset of the set of all possible proof derivations.

The application and domain of a proof strategy, as well as the strategy containing all the empty derivations, are defined accordingly to Definition 7. However, this definition in extension does not provide much guidance regarding the actual implementation of proof strategies in a theorem prover.

In modern LCF, the strategies are introduced as combinators whose behavior depends on the *state* of the proof after the application of its arguments. The state of the proof usually contains non-logical information that signals whether the previous tactic application has, e.g., solved the goal, or failed. In an interactive proof development, this information is fed back to the user, who uses it to further his understanding of the proof. An example of LCF strategy can be found in the `orelse` combinator, which when provided two tactics t_1 and t_2, applies t_2 only if the application of t_1 has either failed, or did not modify the proof. All major implementations of proof strategy languages are based on a core set of commands inherited from the LCF prover [48], namely: `then`, `thenl`, `orelse`, `idtac`, `fail`, and `repeat`.

The natural question to ask is whether it is possible to bridge the divide between a LCF-style proof strategy language and the concept of a proof strategy as given by Definition 30. More generally, because of the uniform approach proposed in this paper, in the upcoming section we investigate the possibility of having an *abstract strategy language*, i.e., a language for building strategies for abstract reduction systems that can be instantiated into rewriting and proof strategy languages. The LCF proof strategy language will be a starting point for the design of this abstract strategy language.

6.3 Strategy language for deduction

The strategy language of modern procedural theorem provers is based on the language of modern LCF, as summarized in Section 6. With time, this core has been extended to include a variety of commands, either to enrich the interaction with the proof systems (postpone goals, undo changes, etc.), or to build more complex and automated deduction rules (including unification, narrowing, etc.) by combining simple rules using strategic constructs.

In the following, we propose a classification of the different strategic com-

mands into the two aforementioned branches: programming and interaction. As a guideline through this taxonomy, we pay particular attention to the place of strategies in the *representation* (that is, the final format) of proofs and in the role they play with regard to the proof *construction* mechanism.

Strategies as programming constructs

Strategies are used to build other commands, i.e., they act as a programming language at the level of the proof language. Unlike deduction and proof construction rules, they are not used directly to construct proofs per se, but rather to generate more complex rules (build the proof builders). In general, strategies are not necessary to represent the proofs. Indeed, because they are in essence programming constructs, only the *results* of the strategic programs are to be retained. As an example, take the following proof script:

```
orelse (apply lemma_1) (apply lemma_2)
```

that attempts to perform a deduction by applying the result of a first lemma, and that applies a second lemma if the first one is unsuccessful. Whichever branch of the `orelse` strategy gets selected should be recorded in the final representation of the proof. However, operationally, there is no point in storing the whole script. We call these *programming strategies*.

Strategies as proof structuring commands

There are two strategic constructs that, although used to build other commands, are important to distinguish from the programming strategies. Indeed, they constitute exceptions to the non representativity of the commands in the previous category. On the contrary they are *fundamental* in the representation of proofs:

`thens,` which combines commands in a tree structure, isomorphic to the structure of the proof. It is the "glue" that holds the building bricks of the proof together, and thus it is necessary to the representation of non-trivial proofs.

`idtac,` that "does nothing" when applied to a sequent. It is used to represent incomplete proofs, and in the tree of commands, fills the place where later tactics might be employed.

Together with tactics, these two strategies are all that are needed to represent proofs. In reference to their tree-structuring functionality, we call these special programming strategies *bark strategies*.

Strategies as interaction controls

Building proofs has become a largely interactive process, with goals being presented one by one, postponed, changes undone, etc.. The second group

of strategic commands are the ones that control such interactions. They do not contribute directly to the construction of the proof, neither should they appear in a finished proof script. Examples of such commands include the `postpone` construct evoked in the introduction, PVS's `(hide)` / `(reveal)` or Coq's `focus`. Also, Coq's unassuming '`.`' command, that (literally) punctuates command applications, should be seen as both an evaluation trigger and an interactive control, returning the first subgoal of the active subtree once the evaluation is complete. We call these controls *interactive commands*.

7 Conclusion and further work

We have given in this paper the definition of abstract strategies and formalized the properties of termination, confluence and normalization under strategy. We then provided instantiations of this framework both in the field of computation and deduction systems. Finally, we exposed and classified the notions of strategies used in these two fields.

We believe that our description of strategies for computation and deduction, combined with our use of a similar framework for expressing their underlying derivation system, makes them ripe for a comparison, and that their similarities are manifest. As a nice bonus, in the process, we have proposed a new definition of the concept of proof construction system, which sheds new light on the semantics of the mechanisms involved.

Further work involves two directions: first, the definition of abstract strategy has to be refined according to the objects on which they are applied: this would lead to more operational definitions and properties for strategies on terms. Second, a proposal for an abstract strategy language should emerge from this work and could be the basis either for future program and proof assistants, or for cooperation between existing ones.

Acknowledgements

This paper beneficiate of many interactions we had within the Protheo team in Nancy since 15 years and from our previous works on Coq, PVS, OBJ, ELAN and TOM. Many thanks also to Dan Dougherty for constructive interaction on strategies and to Christoph Benzmüller for his precise reading and comments.

BIBLIOGRAPHY

[1] María Alpuente, Santiago Escobar, and Salvador Lucas. Correct and Complete (Positive) Strategy Annotations for OBJ. In *Proceedings of the 5th International Workshop on Rewriting Logic and its Applications (RTA)*, volume 71 of *Elecronic Notes In Theoretical Computer Science*, pages 70–89, 2004.

[2] Peter B. Andrews. Herbrand Award Acceptance Speech. *journal of automated deduction*, 31:169–187, 2003.

[3] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3):321–353, June 1996.
[4] Peter B. Andrews and Chad E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *J. Applied Logic*, 4(4):367–395, 2006.
[5] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
[6] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom Manual. LORIA, Nancy (France), version 2.4 edition, October 2006.
[7] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.
[8] Christoph Benzmüller, Matthew Bishop, and Volker Sorge. Integrating Tps and Omega. *J. UCS*, 5(3):188–207, 1999.
[9] Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building Decision Procedures in the Calculus of Inductive Constructions. In Jacques Duparc and Thomas Henziger, editors, *16th Annual Conference on Computer Science and Logic - CSL 2007*, volume 4646 of *Lecture Notes in Computer Science*, Lausanne, Suisse, 2007. Springer Verlag.
[10] Peter Borovanský, Claude Kirchner, Helene Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001.
[11] Paul Brauner, Clément Houtmann, and Claude Kirchner. Principle of superdeduction. In Luke Ong, editor, *Proceedings of LICS*, pages 41–50, jul 2007.
[12] Paul Brauner, Clément Houtmann, and Claude Kirchner. Superdeduction at work. In Hubert Comon, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof. Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600. Springer, jun 2007.
[13] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:427–498, May 2001.
[14] Horatiu Cirstea, Claude Kirchner, Luigi Liquori, and Benjamin Wack. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
[15] Denis Cousineau and Gilles Dowek. Embedding Pure Type Systems in the lambda-Pi-calculus modulo. Available on author's web page, 2007.
[16] Pierre-Louis Curien and Hugo Herbelin. The Duality of Computation. *ACM sigplan notices*, 35(9), 2000.
[17] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Proc. 7th Int. Conf. on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in computer Science*, pages 85–95. Springer-Verlag, November 2000.
[18] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.
[19] Gilles Dowek and Benjamin Werner. Arithmetic as a Theory Modulo. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 423–437. Springer-Verlag, 2005.
[20] Bruno Barras et al. *The Coq Proof Assistant Reference Manual*, 2006.
[21] Michael Fay. First Order Unification in Equational Theories. In *Proceedings 4th Workshop on Automated Deduction, Austin (Tex., USA)*, pages 161–167, 1979.
[22] Wan Fokkink, Jasper Kamperman, and Pum Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.
[23] Gerhard Gentzen. Untersuchungen über das Logisches Schließen. *Mathematische Zeitschrift*, 1:176–210, 1935.

[24] Joseph Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Tim Winkler. An Introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.

[25] Michael Gordon, Robin Milner, Lockwood Morris, Malcolm Newey, and Christopher Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages 119–130. ACM, ACM Press, January 1978.

[26] Gérard Huet and Jean-Jacques Lévy. Computations in Non-ambiguous Linear Term Rewriting Systems. Technical Report, INRIA Laboria, 1979.

[27] Gérard Huet and Jean-Jacques Lévy. *Computational Logic – Essays in Honor of Alan Robinson*, chapter Computations in orthogonal rewriting systems, Part I+II, pages 349–405. mitpress, 1991.

[28] Jean-Marie Hullot. Canonical Forms and Unification. In *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, pages 318–334, July 1980.

[29] Gueorgui Jojgov. Holes with Binding Power. In *Proc. 2002 Int. Workshop on Proofs and Programs*. Springer-Verlag, 2003.

[30] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of Equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

[31] Claude Kirchner and Hélène Kirchner. Rewriting, Solving, Proving. A preliminary version of a book available at `http://www.loria.fr/~ckirchne/=rsp/rsp.pdf`, 1999.

[32] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[33] Florent Kirchner. *Interoperable proof systems*. PhD thesis, École Polytechnique, 2007.

[34] Florent Kirchner and Claudio Sacerdoti Coen. The Fellowship proof manager. `www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`, 2007.

[35] Jan Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[36] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Reduction Strategies and Acyclicity. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600. Springer, jun 2007.

[37] Alexander Letichevsky. Development of Rewriting Strategies. In Maurice Bruynooghe and Jaan Penjam, editors, *PLILP*, volume 714 of *Lecture Notes in Computer Science*, pages 378–390. Springer, 1993.

[38] Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1:1–61, January 1998.

[39] Salvador Lucas. Termination of on-demand rewriting and termination of OBJ programs. In H. Sondergaard, editor, *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93, Firenze, Italy, September 2001. ACM Press, New York.

[40] Salvador Lucas. Termination of Rewriting With Strategy Annotations. In A. Voronkov and R. Nieuwenhuis, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684, La Habana, Cuba, December 2001. Springer-Verlag, Berlin.

[41] Lena Magnusson. *The Implementation of ALF: A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[42] Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a Strategy Language for Maude. In Narciso Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

[43] William McCune. Semantic Guidance for Saturation Provers. *Artificial Intelligence and Symbolic Computation*, pages 18–24, 2006.

[44] Robin Milner. Implementation and applications of Scott's logic for computable functions. *ACM SIGPLAN Notices*, 7(1):1–6, January 1972.

[45] César Muñoz. Proof-term Synthesis on Dependent-Type Systems via Explicit Substitutions. *Theoretical Computer Science*, 266(1-2):407–440, 2001.

[46] Quang-Huy Nguyen. Compact Normalisation Trace via Lazy Rewriting. In S. Lucas and B. Gramlich, editors, *Proceedings of the 1st International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Elecronic Notes In Theoretical Computer Science*. Elsevier, 2001.

[47] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[48] Lawrence Paulson. *Logic and Computation : Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[49] G. Peterson and M. E. Stickel. Complete Sets of Reductions for Some Equational Theories. *Journal of the ACM*, 28:233–264, 1981.

[50] Dag Prawitz. *Natural Deduction: a Proof-Theoretical Study*, volume 3 of *Stockholm Studies in Philosophy*. Almqvist & Wiksell, 1965.

[51] Jörg H. Siekmann, Christoph Benzmüller, and Serge Autexier. Computer supported mathematics with Omega. *J. Applied Logic*, 4(4):533–559, 2006.

[52] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[53] Christian Urban and Gavin Bierman. Strong Normalisation of Cut-elimination in Classical Logic. In *Proc. 4th Int. Conf. on Typed Lambda Calculi and Applications*, pages 365–380. Springer-Verlag, 1999.

[54] Eelco Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.