HAL
open science

Rewriting Strategies and Strategic Rewrite Programs

Hélène Kirchner

To cite this version:

Hélène Kirchner. Rewriting Strategies and Strategic Rewrite Programs. Logic, Rewriting, and Concurrency (LRC 2015). Festschrift Symposium in Honor of José Meseguer, Urbana, Illinois, USA, September 23-25, 2015,, 2015. hal-01143486v2

HAL Id: hal-01143486

https://inria.hal.science/hal-01143486v2

Submitted on 11 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Rewriting Strategies and Strategic Rewrite Programs

Hélène Kirchner

Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex France
e-mail:helene.kirchner@inria.fr

Abstract. This survey aims at providing unified definitions of strategies, strategic rewriting and strategic programs. It gives examples of main constructs and languages used to write strategies. It also explores some properties of strategic rewriting and operational semantics of strategic programs. Current research topics are identified.

1 Introduction

Since the 80s, many aspects of rewriting have been studied in automated deduction, programming languages, equational theory decidability, program or proof transformation, but also in various domains such as chemical or biological computing, plant growth modelling, security policies, etc. Facing this variety of applications, the question arises to understand rewriting in a more abstract way, especially as a logical framework to encode different logics and semantics. Discovering the universal power of rewriting, in particular through its matching and transformation power, led first to the emergence of Rewriting Logic and Rewriting Calculus.

On the other hand, with the development of rewrite frameworks and languages, more and more reasoning systems have been modeled, for proof search, program transformation, constraint solving, SAT solving. It then appeared that straightforward rule-based computations or deductions are often not sufficient to capture complex computations or proof developments. 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 and this leads to the design and study of strategy constructs and strategy languages also in these contexts.

A common understanding is that rules describe local transformations and strategies describe the control of rule application. Most often, it is useful to distinguish between rules for computations, where a unique normal form (i.e. syntactic expressions which cannot be rewritten anymore) is required and where the strategy is fixed, and rules for deductions, in which case no confluence nor termination is required but an application strategy is necessary. Due to the

strong correlation of rules and strategy in many applications, claiming the universal character of rewriting also requires the formalisation of its control. This is achieved through strategic rewriting.

This survey aims at providing unified definitions of strategies, strategic rewriting and strategic programs, with the goal to show the progression of ideas and definitions of the concept, as well as their correlations. It gives examples of main constructs and languages used to write strategies, together with the definition of an operational semantics for strategic programs. Well-studied properties of strategic rewriting are reviewed and current research topics are identified.

Accordingly, following this introduction, the paper is organised as follows: after a brief history of the notion of strategy in rewriting and automated deduction in Section 2, we first explain in Section 3, what are strategic rewriting and strategic programs. In Section 4, several approaches to describe strategies and strategic rewriting are reviewed. In order to catch the higher-order nature of strategies, a strategy is first defined as a proof term expressed in rewriting logic then as a ρ term in rewriting calculus. Looking at a strategy as a set of paths in a derivation tree, the extensional description of strategies, defined as a subset of derivations, is briefly explored. Then a strategy is considered as a partial function that associates to a reduction-in-progress, the possible next steps in the reduction sequence. Last, positional strategies that choose where rules apply are studied. Section 5 presents a few strategy languages, and extracts comon constructs with their variants. We propose an operational semantics for strategic programs in Section 6, study properties of their executions together with correctness and completeness results. We then address in Section 7 various properties, namely termination, confluence and normalizing properties of strategic rewriting. The conclusion points out further work and possible improvements.

This survey is an extended version of [45], and of a lecture given at ISR2014. Although this research on strategies has been largely influenced by related works on proof search, automated deduction and constraint solvers, this paper does not cover these domains and restricts to the area of rewriting.

2 Historical considerations

In programming languages, strategies have been primarily provided to describe the operational semantics of functional languages and the related notions of call by value, call by name, call by need. In languages such as Clean [62], OBJ [31], ML [6], and more recently Haskell [39] or Curry [36], strategies are used to improve efficiency of interpreters or compilers, and are not directly accessible to the programmer. In relation with the operational semantics of functional and algebraic languages, strategies have been studied from a long time in λ-calculus [47,9], in the classical setting of first-order term and graph rewriting, or in abstract rewriting systems.

In the context of functional languages, the notions of termination and confluence of reductions to provide normal forms are meaningful to ensure the existence and unicity of results. Significant research was devoted to design computable and

efficient strategies that are guaranteed to find a normal form for any input term, whenever it exists. Motivated by the need to avoid useless infinite computations in functional programming languages, local strategies were used in eager languages such as Lisp (with its lazy cons), in the OBJ family of languages (OBJ, CafeOBJ, Maude,...) to guide the evaluation using local strategies for functions, in lazy functional programming, via different kinds of syntactic annotations on the program (strictness annotations, or global and local annotations). For instance, Haskell allows for syntactic annotations on the arguments of datatype constructors.

Besides functional or logic programming, strategies also frequently occur in automated deduction and reasoning systems which have been developed in a different community. Beginning with the ML meta-language of LCF [32], strategies are fundamental in several proof environments, such as Coq [20], TPS [4], PVS [61] but also in automated theorem proving [56], constraint solving [15], SAT or SMT solvers [19]. In these contexts, they are more often called tactics, action plans, search plans or priorities.

From the 1990s, attempts have been made to look at the concept of strategy per se, with the intent to confront point of views and to look at computation and deduction in a logical and uniform approach. Already in [43], the notion of computational system, defined as a rewrite theory and a strategy describing the control for rules application, was applied in a uniform way to straightforward computations with rewrite rules, to constraint solving and to combination of these paradigms in the same logical framework. Since 1997, there has been two series of worshops whose goal was to address the concept of strategy and mix different point of views. The Strategies workshops held by the CADE-IJCAR community[1] and the Workshops on Reduction Strategies held by the RTA-RDP community[2]. More recently in 2013, the International Workshop on Strategic Reasoning is emerging from the game theory community[3].

Once the idea was there, the challenge was to propose good descriptions of the concept of strategy. The approach followed in the rewriting community was to formalize a notion of strategy relying on rewriting logic [53] and rewriting calculus [17] that are powerful formalisms to express and study uniformly computations and deductions in automated deduction and reasoning systems. Briefly speaking, rules describe local transformations and strategies describe the control of rule application. Most often, it is useful to distinguish between rules for computations, where a unique normal form is required and where the strategy is fixed, and rules for deductions, in which case no confluence nor termination is required but an application strategy is necessary. Regarding rewriting as a relation and considering abstract rewrite systems leads to consider derivation tree exploration: derivations are computations and strategies describe selected computations.

[1] See `http://www.logic.at/strategies`
[2] `http://users.dsic.upv.es/~wrs/`
[3] See `http://www.strategicreasoning.net/`

With the idea to understand and unify strategies in reduction systems and deduction systems, abstract strategies are defined in [41] and in [14] as a subset of the set of all derivations (finite or not). Another point of view is to see a strategy as a partial function that, at each step of reduction, gives the possible next steps. Strategies are thus considered as a way of constraining and guiding the steps of a reduction. So at any step in a derivation, it should be possible to say which next step obeys the strategy.

In the 1990s, inspired by tactics in proof systems and by constraint programming, the idea came up to provide a *strategy language* to specify which derivations we are interested in. Various approaches have followed, yielding different strategy languages such as Elan [44,12,13], APS [46], Stratego [70,71], Tom [7] or Maude [18,54,55]. For such languages, rules are the basic strategies and additional constructs are provided to combine them and express the control.

Strategy constructs are also present in graph transformation tools such as PROGRES [67], AGG [23], Fujaba [58], GROOVE [66], GrGen [27], GP [64] and Porgy [2,24,25]. Graph rewriting strategies are especially useful in Porgy, an environment to facilitate the specification, analysis and simulation of complex systems, using port graphs. In Porgy, a complex system is represented by an initial graph, a collection of graph rewriting rules, and a user-defined strategy to control the application of rules. The Porgy strategy language includes constructs to deal with graph traversal and management of rewriting positions in the graph. Indeed in the case of graph rewriting, top-down or bottom-up traversals do not make sense. There is a need for a strategy language which includes operators to select rules and the positions where the rules are applied, and also to change the positions along the derivation.

All these languages share the concern to provide abstract ways to express control of rule applications. In these flexible and expressive strategy languages, elaborated strategies are defined by combining a small number of primitives.

3 What are strategic rewriting and strategic programs?

Strategic rewrite programs considered in this paper combine the general concept of rewriting applied to syntactic structures (like terms, graphs, propositions, states, etc.) with a strategy to express the control on rule application. In this way, strategic programming follows the separation of concerns principle [21] since different strategies can be designed and experimented with a same rewrite system. Strategic rewrite programs so contribute to improve agility and modularity in programming.

This section reminds notions of rewriting and abstract rewrite systems and introduces related definitions of strategic rewrite programs and strategic rewriting.

3.1 Rewriting

In the various domains where rewrite rules are applied, rewriting definitions have the same basic ingredients. Rewriting transforms syntactic structures that may

be words, terms, propositions, dags, graphs, geometric objects like segments, and in general any kind of structured objects. In order to emphasize this fact, we use t, G or a to denote indifferently terms, graphs of any other syntactic structure, used for instance to abstractly model the state of a complex system.

Transformations are expressed with patterns called rules. Rules are buit on the same syntax but with an additional set of variables, say $\mathcal{X}$, and with a binder $\Rightarrow$, relating the left-hand side and the right-hand side of the rule, and optionally with a condition or constraint that restricts the set of values allowed for the variables. Performing the transformation of a syntactic structure t is applying the rule labelled ℓ on t, which is basically done in three steps: (1) match to select a redex of t at position p denoted $t_{|p}$ (possibly modulo some axioms, constraints,...); (2) instantiate the rule variables by the result(s) of the matching homomorphism (or substitution) σ; (3) replace the redex by the instantiated right-hand side.

Formally, t rewrites to t' using the rule $\ell : l \Rightarrow r$ if $t_{|p} = \sigma(l)$ and $t' = t[\sigma(r)]_p$. This is denoted $t \longrightarrow_{p,\ell,\sigma} t'$.

The transformation process is similar on graphs (see for instance [34,65]) and many other structured objects can be encoded by terms or graphs.

When $\mathcal{R}$ is a set of rules, this transformation generates a relation $\longrightarrow_{\mathcal{R}}$ on the set of syntactic structures. Its (reflexive) transitive closure is denoted $(\overset{*}{\longrightarrow}_{\mathcal{R}})$ $\overset{+}{\longrightarrow}_{\mathcal{R}}$.

Given a set of rewrite rules $\mathcal{R}$, a *derivation*, or computation from G is a sequence of rewriting steps $G \rightarrow_{\mathcal{R}} G' \rightarrow_{\mathcal{R}} G" \rightarrow_{\mathcal{R}} \ldots$

In this transformation process, there are many possible choices: for the rule itself, the position(s) in the structure, the matching homomorphism(s). For instance, one may choose to apply a rule concurrently at all disjoint positions where it matches, or using matching modulo an equational theory like associativity-commutativity, or also according to some probability. Since in general, there is more than one way of rewriting a structure, the set of rewrite derivations can be organised as a derivation tree. The *derivation tree* of G, written $DT(G, \mathcal{R})$, is a labelled tree whose root is labelled by G, the children of which being all the derivation trees $DT(G_i, \mathcal{R})$ such that $G \rightarrow_{\mathcal{R}} G_i$. The edges of the derivation tree are labelled with the rewrite rule and the morphism used in the corresponding rewrite step. A derivation tree may be infinite, if there is an infinite reduction sequence out of G.

3.2 Strategic Rewrite Programs

Intuitively, a strategic program consists of an initial structure G (or t when it is a term), together with a set of rules $\mathcal{R}$ that will be used to reduce it, according to a given strategy expression S, used to decide which rewrite steps should be performed on G. This amounts to identify the branches in G's derivation tree that satisfy the strategy S and to view strategic rewriting derivations as selected computations.

Formally, a *strategic rewrite program* consists of a finite set of rewrite rules $\mathcal{R}$, a strategy expression S, built from $\mathcal{R}$ using a strategy language, and a given structure G.

We denote it $[S_{\mathcal{R}}, G]$, or simply $[S, G]$ when $\mathcal{R}$ is clear from the context.

Several questions come up with this definition: how to describe a strategy expression S, how to characterize strategic rewriting derivations, how to design a language for strategy expressions, how to define an operational semantics for strategic programs?

3.3 Abstract Reduction System

Dealing with a general notion of rewriting is well addressed in abstract reduction systems. There, rewriting is considered as an abstract relation on structured objects. Even if different variants of the definition of Abstract Reduction System have been given in the literature [69,41,42], they agree on the following basis. An *Abstract Reduction System (ARS)* is a labelled oriented graph $(\mathcal{O}, \mathcal{S})$ with a set of labels $\mathcal{L}$. The nodes in $\mathcal{O}$ are called *objects*. The oriented labelled edges in $\mathcal{S}$ are called *steps*: $a \xrightarrow{\phi} b$ or (a, ϕ, b), with *source* a, *target* b and *label* ϕ. Two steps $a \xrightarrow{\phi} b$ and $c \xrightarrow{\phi'} d$ can be composed if b and c are the same object. Derivations are composition of steps and may be finite or infinite.

For a given ARS $\mathcal{A}$, a *finite derivation* is denoted $\pi : a_0 \xrightarrow{\phi_1} a_1 \ldots \xrightarrow{\phi_{n-1}} a_n$ or $a_0 \xrightarrow{\pi} a_n$, where $n \in \mathbb{N}$ is the length of the derivation. The *source* of π is a_0 and its domain $Dom(\pi) = \{a_0\}$. The *target* of π is a_n and applying π to a_0 gives the singleton set $\{a_n\}$, which is denoted $\pi \bullet \{a_0\} = \{a_n\}$, or $\pi \bullet a_0 = a_n$ by abusively identifying elements and singletons. The *concatenation* of two finite derivations $\pi_1; \pi_2$ is defined as $a \xrightarrow{\pi_1} b \xrightarrow{\pi_2} c$ if $\{a\} = Dom(\pi_1)$ and $\pi_1 \bullet a = Dom(\pi_2) = \{b\}$. Then $(\pi_1; \pi_2) \bullet \{a\} = \pi_2 \bullet (\pi_1 \bullet \{a\}) = \{c\}$, or more simply $(\pi_1; \pi_2) \bullet a = \pi_2 \bullet (\pi_1 \bullet a) = c$.

Termination and confluence properties for ARS are then expressed as follows. For a given ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$:

- An object a in $\mathcal{O}$ is *irreducible* if a is the source of no edge.
- A derivation is *normalizing* when its target is irreducible.
- An ARS is *weakly terminating* if every object a is the source of a normalizing derivation.
- $\mathcal{A}$ is *terminating* (or *strongly normalizing*) if all its derivations are of finite length.
- An ARS $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is *confluent* if

$$\begin{gathered}\text{for all objects } a, b, c \text{ in } \mathcal{O}, \text{ and all } \mathcal{A}\text{-derivations } \pi_1 \text{ and } \pi_2, \\ \text{when } a \xrightarrow{\pi_1} b \text{ and } a \xrightarrow{\pi_2} c, \\ \text{there exist } d \text{ in } \mathcal{O} \text{ and two } \mathcal{A}\text{-derivations } \pi_3, \pi_4 \text{ such that} \\ c \xrightarrow{\pi_3} d \text{ and } b \xrightarrow{\pi_4} d.\end{gathered}$$

3.4 Strategic rewriting

Given a rewrite system $\mathcal{R}$, defined on a set of objects $\mathcal{O}$ that may be terms, equivalence classes of terms, graphs, or states, we can consider the rewrite relation $\longrightarrow_{\mathcal{R}}$ defined in Section 3.1 to get the ARS $\mathcal{A} = (\mathcal{O}, \longrightarrow_{\mathcal{R}})$.

Based on the ARS concept, we can consider strategic rewriting in two dual ways:

- The first one emphasizes the selective purpose of strategies among the set of rewriting derivations. Abstract strategies are defined in [41] and in [14] as follows: for a given ARS $\mathcal{A}$, an *abstract strategy* ζ is a subset of the set of all derivations (finite or not) of $\mathcal{A}$. To relate this definition to the functional aspect of strategies, the notions of domain and application are then defined as follows: $Dom(\zeta) = \bigcup_{\pi \in \zeta} Dom(\pi)$ and $\zeta \bullet a = \{b \mid \exists \pi \in \zeta \text{ such that } a \xrightarrow{\pi} b\} = \{\pi \bullet a \mid \pi \in \zeta\}$.
- The second way emphasizes the reduction relation itself and relies on a restriction of the rewrite relation. Instead of the rewrite relation $\longrightarrow_{\mathcal{R}}$ we may consider on the same set of objects, another relation (induced by strategic steps) corresponding to strategic rewriting. A *strategic reduction step* is a relation $\xrightarrow{S}$ such that $\xrightarrow{S} \subseteq \xrightarrow{+}_{\mathcal{R}}$.

  This leads to consider another ARS $\mathcal{A}' = (\mathcal{O}, \xrightarrow{\mathcal{S}})$ and to compare it with the previous one $\mathcal{A} = (\mathcal{O}, \longrightarrow_{\mathcal{R}})$.

The derivation tree defined in 3.1 is a representation of the ARS $\mathcal{A} = (\mathcal{O}, \longrightarrow_{\mathcal{R}})$. The selected branches in the derivation tree is then a representation of the ARS$=\mathcal{A}' = (\mathcal{O}, \xrightarrow{\mathcal{S}})$.

Indeed termination, confluence and irreducible objects are in general different for the two ARS. In the term rewriting approach of strategic reduction described in [10], it is required that moreover $NF(\xrightarrow{S}) = NF(\mathcal{R})$ where NF denotes the set of terms which are not reducible any more by the considered relation. We will come back later in Section 7 on these properties.

But first, in the following Section 4, we consider different ways to describe strategies and strategic rewriting.

4 Strategy Description: different points of view

Different definitions of strategy have been given in the rewriting community in the last twenty years, when strategies began to be studied per se. We review them in this section, making clear that they all actually define either selected sets of rewriting derivations, or selected sets of positions where rules should be applied.

4.1 Rewriting logic

The Rewriting Logic is due to Meseguer [57,53]: *Rewriting logic (RL) is a natural model of computation and an expressive semantic framework for concurrency,*

parallelism, communication, and interaction. It can be used for specifying a wide range of systems and languages in various application fields. It also has good properties as a metalogical framework for representing logics. In recent years, several languages based on RL (ASF+SDF, CafeOBJ, ELAN, Maude) have been designed and implemented. [4]

In Rewriting Logic, the syntax is based on a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{Y})$ built with an alphabet $\mathcal{F}$ of function symbols with arities and with variables in $\mathcal{Y}$. A theory is given by a set $\mathcal{R}$ of labeled rewrite rules denoted $\ell(x_1, \ldots, x_n) : l \Rightarrow r$, where labels $\ell(x_1, \ldots, x_n)$ record the set of variables occurring in the rewrite rule. Formulas are sequents of the form $\pi : t \to t'$, where π is a *proof term* recording the proof of the sequent: $\mathcal{R} \vdash \pi : t \to t'$ if $\pi : t \to t'$ can be obtained by finite application of equational deduction rules [57] given in Figure 1. In this context, a proof term π encodes a sequence of rewriting steps called a derivation.

Reflexivity For any $t \in \mathcal{T}(\mathcal{F}, \mathcal{Y})$:

$$\mathbf{t} : t \to t$$

Transitivity

$$\frac{\pi_{\mathbf{1}} : t_1 \to t_2 \qquad \pi_{\mathbf{2}} : t_2 \to t_3}{\pi_{\mathbf{1}}; \pi_{\mathbf{2}} \ : \ t_1 \to t_3}$$

Congruence For any $f \in \mathcal{F}$ with $arity(f) = n$:

$$\frac{\pi_{\mathbf{1}} : t_1 \to t'_1 \quad \ldots \quad \pi_{\mathbf{n}} : t_n \to t'_n}{\mathbf{f}(\pi_{\mathbf{1}}, \ldots, \pi_{\mathbf{n}}) : f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n)}$$

Replacement For any $\ell(x_1, \ldots, x_n) : l \Rightarrow r \in \mathcal{R}$,

$$\frac{\pi_{\mathbf{1}} : t_1 \to t'_1 \quad \ldots \quad \pi_{\mathbf{n}} : t_n \to t'_n}{\ell(\pi_{\mathbf{1}}, \ldots, \pi_{\mathbf{n}}) : l(t_1, \ldots, t_n) \to r(t'_1, \ldots, t'_n)}$$

Fig. 1. Deduction rules for Rewriting Logic

Let us consider the following example of sorting a list of natural numbers, where natural numbers are a subsort of lists of natural numbers, which is denoted as "`Nat < List`"; the concatenation operateur "`_ _ : List x List -> List`" is associatif with the empty list "`nil : -> List`" as identity; operators profiles are "`sort, rec, fin : List -> List`"; natural numbers are denoted as "`1, 2, 3,...`" for simplicity and compared with the usual ordering "<". The rules are expressed as follows:

```
rules for List
  X, Y : Nat ; L L' L'' : List;
  rec :  sort (L X L' Y L'') => sort (L Y L' X L'')
           if Y < X
  fin :  sort (L) => L
```

[4] `http://wrla2012.lcc.uma.es/`

```
end
```

For the derivation:

```
sort(3 1 2) -> sort(1 3 2) -> sort(1 2 3) -> (1 2 3)
```

the proof term is

```
rec(nil,3,nil,1,(2));rec((1),3,nil,2,nil);fin((1 2 3)).
```

The Elan language, designed in the 1990's, introduced the concept of strategy by giving explicit constructs for expressing control on the rule application [43,11]. Beyond labeled rules and concatenation denoted ";", other constructs for choice, failure, iteration, were also defined in Elan. A strategy is there defined as a set of proof terms in rewriting logic and can be seen as a higher-order function : if the strategy ζ is a set of proof terms π, applying ζ to the term t means finding all terms t' such that $\pi : t \to t'$ with $\pi \in \zeta$. Since rewriting logic is reflective, strategy semantics can be defined inside the rewriting logic by rewrite rules at the meta-level. This is the approach followed by Maude in [54,55].

4.2 Rewriting Calculus

The rewriting calculus, also called ρ-calculus, has been introduced in 1998 by Horatiu Cirstea and Claude Kirchner [17]. *The rho-calculus has been introduced as a general means to uniformly integrate rewriting and λ-calculus. This calculus makes explicit and first-class all of its components: matching (possibly modulo given theories), abstraction, application and substitutions.*

The rho-calculus is designed and used for logical and semantical purposes. It could be used with powerful type systems and for expressing the semantics of rule based as well as object oriented paradigms. It allows one to naturally express exceptions and imperative features as well as expressing elaborated rewriting strategies. [5]

Some features of the rewriting calculus are worth emphasizing here: first-order terms and λ-terms are ρ-terms ($\lambda x.t$ is $(x \Rightarrow t)$); a rule is a ρ-term as well as a strategy, so rules and strategies are abstractions of the same nature and "first-class concepts"; application reduction generalizes $\beta-$reduction; composition of strategies is function composition and is denoted explicitely here by the operator $\bullet$; recursion can be for example expressed as in λ calculus with a recursion operator μ.

To illustrate the notion of ρ-term on a simple example, let us come back to the list sorting algorithm. For the derivation:

```
sort (3 1 2) -> sort (1 3 2) -> sort (1 2 3) -> (1 2 3)
```

the corresponding ρ-term can be written :

$$fin \bullet rec_2 \bullet rec_1 \bullet \ sort(312)$$

[5] http://rho.loria.fr/index.html

with $fin = (sort(L_3) \Rightarrow L_3), rec_2 = (sort(L_2X_2L'_2Y_2L''_2) \Rightarrow sort(L_2Y_2L'_2X_2L''_2))$ and $rec_1 = (sort(L_1X_1L'_1YL''_1) \Rightarrow sort(L_1Y_1L'_1X_1L''_1))$.

In the ρ-calculus, strategies expressed by a well-typed ρ-term of type $term \mapsto term$ evaluates to a set of rewrite derivations [16].

The Abstract Biochemical Calculus (or ρ_{Bio}-calculus) [3] illustrates a useful instance of the ρ-calculus. The ρ_{Bio}-calculus models autonomous systems as *biochemical programs* which consist of the following components: collections of molecules (objects and rewrite rules), higher-order rewrite rules over molecules (that may introduce new rewrite rules in the behaviour of the system) and strategies for modelling the system's evolution. A visual representation via *port graphs* and an implementation are provided by the Porgy environment described in [2]. In this calculus, strategies are abstract molecules, expressed with an arrow constructor ($\Rightarrow$ for rule abstraction), an application operator $\bullet$ and a constant operator `stk` (for *stuck*) for explicit failure.

4.3 Extensional strategies

The *extensional* definition of abstract strategies as a set of derivations of an abstract reduction system is given in [14]. The concept is useful to understand and unify reduction systems and deduction systems as explored in [41].

The extensional approach is also useful to address infinite elements. Since abstract reduction systems may involve infinite sets of objects, of reduction steps and of derivations, we can schematize them with constraints at different levels: (i) to describe the objects occurring in a derivation (ii) to describe, via the labels, requirements on the steps of reductions (iii) to describe the structure of the derivation itself (iv) to express requirements on the histories. The framework developed in [42] defines a strategy ζ as all instances $\sigma(D)$ of a derivation schema D such that σ is solution of a constraint C involving derivation variables, object variables and label variables. As a simple example, the infinite set of derivations of length one that transform a into $f(a^n)$ for all $n \in \mathbb{N}$, where $a^n = a * \ldots * a$ (n times), is simply described by: $(a \to f(X) \mid X * a =_A a * X)$, where $=_A$ indicates that the constraint is solved modulo associativity of the operator $*$.

4.4 Intensional strategies

Extensional strategies do not capture the idea that a strategy is a partial function that associates to each step in a reduction sequence, the possible next steps. Here, the strategy as a function may depend on the object and the derivation so far. This notion of strategy coincides with the definition of strategy in sequential path-building games, with applications to planning, verification and synthesis of concurrent systems [22]. This remark leads to the following *intensional* definition given in [14]. Again, the essence of the definition is that strategies are considered as a way of constraining and guiding the steps of a reduction. So at any step in a derivation, it should be possible to say which is the next step that obeys the strategy ζ. In order to take into account the past derivation steps to decide the next possible ones, the history of a derivation has to be memorized and available

at each step. Through the notion of traced-object $[\alpha]\, a = [(a_0, \phi_0), \ldots, (a_n, \phi_n)]\, a$ in $\mathcal{O}^{[\mathcal{A}]}$, each object a memorizes how it has been reached with the trace α.

An *intensional strategy* for $\mathcal{A} = (\mathcal{O}, \mathcal{S})$ is a partial function λ from $\mathcal{O}^{[\mathcal{A}]}$ to $2^{\mathcal{S}}$ such that for every traced object $[\alpha]\, a$, $\lambda([\alpha]\, a) \subseteq \{\pi \in \mathcal{S} \mid Dom(\pi) = a\}$. If $\lambda([\alpha]\, a)$ is a singleton, then the reduction step under λ is deterministic.

As described in [14], an intensional strategy λ naturally generates an abstract strategy, called its *extension*: this is the abstract strategy ζ_λ consisting of the following set of derivations:

$\forall n \in \mathbb{N}, \pi : a_0 \xrightarrow{\phi_0} a_1 \xrightarrow{\phi_1} a_2 \ldots \xrightarrow{\phi_{n-1}} a_n \ \in \zeta_\lambda$

iff $\forall j \in [0, n-1], \quad (a_j \xrightarrow{\phi_j} a_{j+1}) \in \lambda([\alpha]\, a_j)$.

This extension may obviously contain infinite derivations; in such a case it also contains all the finite derivations that are prefixes of the infinite ones, and so is closed under taking prefixes.

A special case are memoryless strategies, where the function λ does not depend on the history of the objects. This is the case of many strategies used in rewriting systems, as shown in the next example. Let us consider an abstract rewrite system $\mathcal{A}$ where objects are terms, reduction is term rewriting and labels are positions where the rewrite rules are applied. Let us consider an order $<$ on the labels which is the prefix order on positions. Then the intensional strategy that corresponds to innermost rewriting is $\lambda_{inn}(t) = \{\pi : t \xrightarrow{p} t' \mid p = max(\{p' \mid t \xrightarrow{p'} t' \in \mathcal{S}\})\}$. When a lexicographic order is used, the classical *rightmost-innermost* strategy is obtained.

Another example, to illustrate the interest of traced objects, is the intensional strategy that restricts the derivations to be of bounded length k. Its definition makes use of the size of the trace α, denoted $|\alpha|$: $\lambda_{ltk}([\alpha]\, a) = \{\pi \mid \pi \in \mathcal{S},\ Dom(\pi) = a,\ |\alpha| < k-1\}$. However, as noticed in [14], the fact that intensional strategies generate only prefix closed abstract strategies prevents us from computing abstract strategies that look straightforward: there is no intensional strategy that can generate a set of derivations of length exactly k. Other solutions are provided in [14].

4.5 Positional Strategies

In order to build the function that gives the next possible steps in a reduction sequence, mechanisms to choose the positions in the syntactic structure where a rule or a set of rules can be applied. This can be done in two different ways: either by traversing the syntactic structure, or by using annotations to select a set of positions.

In term rewriting, the first way is illustrated by leftmost-innermost (resp. outermost) reduction strategies on terms that choose the rewriting position according to suffix (resp. prefix) ordering on the set of positions in the term. The second way inspired from OBJ, uses local annotations. Informally, a *strategy annotation* is a list of argument positions and rule names [26,1]. The argument positions indicate the next argument to evaluate and the rule names indicate

rules to apply. For instance, the leftmost-innermost strategy for a function symbol C corresponds to an annotation $strat(C) = [1, 2, , .., k, R_1, R_2, ...R_n]$ that indicates that all its arguments should be evaluated from left to right and that the rules R_i should be tried. This is also called *on-demand rewriting*. Note that including (labels of) rules is not allowed in such strategy annotations. It is, however, allowed in the so-called *just-in-time strategies* developed in [68].

Context-sensitive rewriting is a rewriting restriction which can be associated to every term rewriting system [48]. Given a signature $\mathcal{F}$, a mapping $\mu : \mathcal{F} \mapsto \mathcal{P}(N)$, called the *replacement map*, discriminates some argument positions $\mu(f) \subseteq \{1, ..., k\}$ for each k-ary symbol f. Given a function call $f(t_1, ..., t_k)$, the replacements are allowed on arguments t_i such that $i \in \mu(f)$ and are forbidden for the other argument positions. Examples are given in [48,50].

A different approach is followed on graphs. Motivated by the need to apply rules on huge graphs, Porgy [24] introduces annotations to focus on or to avoid part of the graph. A *located graph* G_P^Q consists of a port graph G and two distinguished subgraphs P and Q of G, called respectively the *position subgraph*, or simply *position*, and the *banned subgraph*. In a located graph G_P^Q, P represents the subgraph of G where rewriting steps may take place (i.e., P is the focus of the rewriting) and Q represents the subgraph of G where rewriting steps are forbidden. The intuition is that subgraphs of G that overlap with P may be rewritten, if they are outside Q. The subgraph P generalises the notion of rewrite position in a term: if G is the tree representation of a term t then we recover the usual notion of rewrite position p in t by setting P to be the node at position p in the tree G, and Q to be the part of the tree above P (to force the rewriting step to apply from P downwards). When applying a port graph rewrite rule, not only the underlying graph G but also the position and banned subgraphs may change. A *located rewrite rule* specifies two disjoint subgraphs M and N of the right-hand side r that are used to update the position and banned subgraphs, respectively. If M (resp. N) is not specified, r (resp. the empty graph) is used as default. In general, for a given located rule and located graph G_P^Q, several rewriting steps at P avoiding Q might be possible. Thus, the application of the rule at P avoiding Q produces a *set of located graphs*.

The precise definitions and details are given in [25]. Such definitions of forbidden positions are quite useful to formalize deduction process that for instance prevents rewriting in the parts brought by instantiating rules variables, or needs to always apply at some interface nodes.

5 Strategy languages

A *strategy language* gives syntactic means to describe strategies. Various *strategy languages* have been proposed by different teams, giving rise to different families. Five of them, representative of these families, are reviewed in this section: Elan [13] puts emphasis on rules and strategies as a paradigm to combine deduction and computation by rewriting[6], and its successor Tom [7] is strongly

[6] `http://elan.loria.fr/elan.html`

based on the ρ-calculus [7]. Stratego [70] is a successor of ASF+SDF, mainly dedicated to program transformation [8]. Maude [55] inherits from the OBJ family, order-sorted equational rewriting and strategic annotations of operators, and is strongly based on rewriting logic [9]. Porgy [25] took inspiration, partly from the aforementioned languages and also from graph transformation languages, and puts emphasis on strategies which can be useful for modeling and analysing big graphs [10].

Language design is largely a matter of choice and the idea here is not to give a catalogue of constructs present in these languages, but rather extract from them some comon features and understand how they address the two main purposes of strategies: on one hand, build derivation steps and derivations; on the other hand, operationaly compute the next strategic reduction steps.

Let us classify the constructs to see which ones are commonly agreed and which ones are specific to one language. Remind that t or G denotes a syntactic expression (term, graph,...) and S is a strategy expression in a strategy language on a rewrite rule system $\mathcal{R}$ with rules $R_1, \ldots, R_n$. Application of S to G is denoted $S{\bullet}G$.

Elementary strategies are the basis of all languages. The most basic strategy is a labelled rule $\ell : l \Rightarrow r$ ($\ell \triangleq l \Rightarrow r$). `id` and `fail` are two strategies that respectively denote success and failure. They can be encoded either as constant or as rules $\texttt{id} \triangleq X \Rightarrow X$ and $\texttt{fail} \triangleq X \Rightarrow \texttt{stk}$ where `stk` denotes a special constant denoting failure.

However, even for a single rule, rewriting can be performed in various ways, according to redexes or homomorphisms. There are mainly two options there: $\texttt{all}(R)$ denotes all possible applications of the transformation R on the current object, creating a new one for each application. In the derivation tree, this creates as many children as there are possible applications. Instead $\texttt{one}(R)$ chooses only one of the possible applications of the transformation and ignores the others; again there are some variations here, in the way to choose, either by taking the first found application, of by making a random choice between all the possible applications, with equal probabilities.

Note however that the `all` and `one` constructs are not available in all strategy languages and are sometimes implicit.

Building derivations is always present under different syntaxes. Composition of two strategies S_1 and S_2 is primarily done by sequential application of S_1 followed by S_2. It is denoted $\texttt{Sequence}(S_1, S_2)$ or $\texttt{seq}(S_1, S_2)$ or S_1 `Then` S_2 or S_1 ; S_2.

Selection of branches in the derivation tree is obviously needed and present in all languages although with different syntaxes: $\texttt{first}(S_1, S_2)$, $(S_1)\texttt{orelse}(S_2)$

[7] `https://gforge.inria.fr/projects/tom/`
[8] `http://strategoxt.org/Stratego/WebHome`
[9] `http://maude.cs.uiuc.edu/`
[10] `tulip.labri.fr/TulipDrupal/?q=porgy`

or $S_1 <^+ S_2$ selects the first strategy that does not fail; it fails if both fail. As a variant, $\texttt{try}(S)$ tries the strategy S but never fails and $\texttt{try}(S) \triangleq \texttt{first}(S, \texttt{id})$.

While `first` selects the strategy according to the order of its arguments, in the Elan language, the don't care construct $\texttt{dc}(R_1, \ldots, R_n)$ randomly chooses one of the rules for application. In its implementation however, the first rule that is applicable is chosen and the `dc` construct is actually a `first`.

Probabilistic choice is provided in Porgy. When probabilities $p_1, \ldots, p_n \in [0, 1]$ are associated to strategies $S_1, \ldots, S_n$ such that $p_1 + \ldots + p_n = 1$, the construct $\texttt{ppick}(S_1, p_1, \ldots, S_n, p_n)$ picks one of the strategies for application, according to the given probabilities.

Conditionals and Tests again are present in all languages but with some variations. $\texttt{if}(S)\texttt{then}(S')\texttt{else}(S'')$ checks if application of S is successful (i.e. returns `id`), in which case S' is applied, otherwise S'' is applied. In Elan, Tom, Stratego and Porgy, in case the application of S to the current object G succeeds, S' is applied to S, while in Maude, S' is applied to $S{\bullet}G$. Maude also provides the consctruct $\texttt{match}(S)$ that matches the term S to G and returns G if success or `fail` otherwise. As a derived operator, $\texttt{not}(S) \triangleq \texttt{if}(S)\texttt{then}(\texttt{fail})\texttt{else}(\texttt{id})$ fails if S succeeds and succeeds if S fails.

Recursive strategies and iterations are essential due to the functional aspect of strategies. Expressed in Tom with a fixpoint operator $\mu x.S = S[x \leftarrow \mu x.S]$, $\texttt{repeat}(S)$ keeps on sequentially applying S until it fails and returns the last result: $\texttt{repeat}(S) = \mu x.\texttt{first}(\texttt{Sequence}(S, x), \texttt{id})$. As a variant, $\texttt{while}(S)\texttt{do}(S')$ keeps on sequentially applying S' while the expression S is successful; if S fails, then `id` is returned.

Stratego [70] instead introduces recursive closure strategies. The recursive closure $recx(S)$ of the strategy S attempts to apply S to the entire subject term and the strategy $recx(S)$ to each occurrence of the variable x in S. Iterators are provided based on this construction.

$$
\begin{array}{ll}
try(S) & = S <^+ id \\
repeat(S) & = recx(try(S; x)) \\
while(c, S) & = recx(try(c; S; x)) \\
do - while(S, c) & = recx(S; try(c; x)) \\
while - not(c, S) & = recx(c <^+ S; x) \\
for(i, c, S) & = i; while - not(c, S)
\end{array}
$$

Exploiting the structure of objects Traversal strategies are useful to traverse structures, be terms or graphs. They are based on local neighbourhood exploration and iteration.

- On a term $t = f(t_1, ..., t_n)$, $\texttt{AllSuc}(S)$ applies the strategy S on all immediate subterms: $\texttt{AllSuc}(S){\bullet}f(t_1, ..., t_n) = f(t'_1, ..., t'_n)$ if $S{\bullet}t_1 = t'_1, ..., S{\bullet}t_n = t'_n$; it fails if there exists i such that $S{\bullet}t_i$ fails. $\texttt{OneSuc}(S)$ applies the strategy S on the first immediate subterm (if it exists) where S does not fail:

$\texttt{OneSuc}(S)\bullet f(t_1, ..., t_n) = f(t_1, ..., t'_i, ..., t_n)$ if for all $1 \leq j < i$, $S\bullet t_j$ fails, and $S\bullet t_i = t'_i$; it fails if f is a constant or if for all i, $S\bullet t_i$ fails.

– On a graph G, $\texttt{AllNbg}(S)$ applies the strategy S on all immediate successors of the nodes in G, where an immediate successor of a node v is a node connected to v. $\texttt{OneNbg}(S)$ applies the strategy S on one immediate successor of a node in G, randomly chosen.

Traversal strategies are expressed in Tom with the following fixpoint equations:

$$\begin{aligned}
&OnceBottomUp(S) &&= \mu x.First(OneSuc(x), S)\\
&BottomUp(S) &&= \mu x.Sequence(AllSuc(x), S)\\
&TopDown(S) &&= \mu x.Sequence(S, AllSuc(x))\\
&Innermost(S) &&= \mu x.Sequence(AllSuc(x), Try(Sequence(S, x)))
\end{aligned}$$

Focusing strategies Instead of traversing the structure through a systematic exploration, one may want to focus on or to avoid on sub-structures. Strategy annotations may be seen as precursors of this idea. Porgy allows combining applications of rewrite rules and position updates, using *focusing expressions*. The direct management of positions in strategy expressions, via the distinguished subgraphs P and Q in the target graph and the distinguished graphs M and N in a located port graph rewrite rule are original features of the language. The grammar generates expressions that are used to define positions for rewriting in a graph, or to define positions where rewriting is not allowed. They denote functions used in strategy expressions to change the positions P and Q in the current located graph (e.g. to specify graph traversals). The constructs `CrtGraph` (current graph), `CrtPos` (current positions) and `CrtBan` (current banned positions), applied to a located graph G^Q_P, return respectively the graphs G, P and Q. To generate traversal strategies on graphs, Porgy uses neighbourhood constructs `Nbg`() that returns the neighbours of a set of nodes possibly satisfying some user-defined properties.

6 Operational semantics of strategic programs

There are several ways to describe the operational semantics of a programming language. Due to the fact that rewriting logic is reflexive, it is tempting to describe the operational semantics of a strategy language with a set of rewrite rules. This has been done for instance for Elan [11], Maude [18] and Porgy [2] at least. We sketch below another way by defining a transition relation on configurations using semantic rules in the SOS style of [63].

Let us consider a strategic rewrite program consisting of a finite set of rewrite rules $\mathcal{R}$, a strategy expression S (built from $\mathcal{R}$ using a strategy language $\mathcal{L}(\mathcal{R})$) and a given structure G. The intuition behind a strategic program is to use the strategy expression S to decide which rewrite steps should be performed on G. As already said, in general, there may be more than one way of rewriting

a structure according to S. In order to keep track of the various rewriting alternatives, we introduce the notion of a *configuration* as a multiset of strategic rewrite programs. A *configuration* C is a multiset $\{O_1, \ldots, O_n\}$ where each O_i is a strategic program $[S_i, G_i]$. The *initial configuration* is $\{[S, G]\}$.

The transition relation $\longmapsto$ is a binary relation on configurations defined as follows:

$$\{O_1, \ldots, O_k, \ldots, O_n\} \longmapsto \{O_1, \ldots, O'_{k_1}, \ldots, O'_{k_m}, \ldots, O_n\}$$

if $O_k \mapsto \{O'_{k_1}, \ldots, O'_{k_m}\}$, for $1 \leq k \leq n$. The transition relation $\mapsto$ is defined through semantic rules. For instance, a few semantic rules are given in Figure 2 coming from the Porgy operational semantics. More are given in [25].

$$\frac{}{[\texttt{one}(l \Rightarrow r), G] \mapsto \{[\texttt{id}, G']\}}\ G' \in LS_{l \Rightarrow r}(G)$$

where LS is the legal set of reducts

$$\frac{}{[\texttt{one}(l \Rightarrow r), G] \mapsto \{[\texttt{fail}, G]]\}}\ LS_{l \Rightarrow r}(G) = \emptyset$$

$$\frac{}{[\texttt{all}(l \Rightarrow r), G] \mapsto \{[\texttt{id}, G_1], \ldots, [\texttt{id}, G_k]\}}\ \{G_1, \ldots, G_k\} = LS_{l \Rightarrow r}(G)$$

$$\frac{}{[\texttt{all}(l \Rightarrow r), G] \mapsto \{[\texttt{fail}, G]]\}}\ LS_{l \Rightarrow r}(G) = \emptyset$$

$$\frac{}{[\texttt{id}; S, G] \mapsto \{[S, G]\}}$$

$$\frac{}{[\texttt{fail}; S, G] \mapsto \{[\texttt{fail}, G]\}}$$

$$\frac{[S_1, G] \mapsto \{[S_1^1, G_1], \ldots, [S_1^k, G_k]\}}{[S_1; S_2, G] \mapsto \{[S_1^1; S_2, G_1], \ldots, [S_1^k; S_2, G_k]\}}$$

$$\frac{\exists G', M \ s.t. \ \{[S_1, G]\} \mapsto^* \{[\texttt{id}, G'], M\}}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G] \mapsto \{[S_2, G]\}}$$

$$\frac{\nexists G', M \ s.t. \ \{[S_1, G]\} \mapsto^* \{[\texttt{id}, G'], M\}}{[\texttt{if}(S_1)\texttt{then}(S_2)\texttt{else}(S_3), G] \mapsto \{[S_3, G]\}}$$

$$\frac{}{[\texttt{while}(S_1)\texttt{do}(S_2), G] \mapsto \{[\texttt{if}(S_1)\texttt{then}(S_2; \texttt{while}(S_1)\texttt{do}(S_2))\texttt{else}(\texttt{id}), G]\}}$$

Fig. 2. Examples of semantic rules for Strategy Language

Given a configuration $\{O_1, \ldots, O_k, \ldots, O_n\}$, there may be several strategic programs O_k where a $\mapsto$-step can be applied, so there is also a $\longmapsto$-derivation tree whose nodes are configurations. Intuitively these configurations provide another

view of the derivation tree of the strategic program, or equivalently of the ARS of the relation $\xrightarrow{S}$, with root G. One can recover it by projecting a strategic program $O = [S, G]$ on its second component G and by associating to a $\mapsto$-step $O_k \mapsto \{O'_{k_1}, \ldots, O'_{k_m}\}$, for $1 \leq k \leq n$, a set of m strategic reduction steps $G_k \xrightarrow{S} G'_{k_i}$ for $1 \leq i \leq m$.

For a given configuration $C = \{O_1, \ldots, O_k, \ldots, O_n\}$, where each O_i is a strategic program $[S_i, G_i]$, let $Reach(C) = \{G_1, \ldots, G_k, \ldots, G_n\}$ be the set of associated reachable structures. For a derivation $T = C_1 \longmapsto \ldots \longmapsto C_n$ let $Reach(T) = \bigcup_{1 \leq k \leq n} Reach(C_k)$ be the set of associated reachable structures.

As presented in [55], it is expected from a strategy language to satisfy the properties of correctness and completeness w.r.t. rewriting derivations.
Correctness: If T is the derivation $C_0 = \{[S, G]\} \longmapsto \ldots \longmapsto C_k = \{...[S'_k, G'_k]...\}$ and if $G' \in Reach(T)$, then $G \rightarrow^*_{\mathcal{R}} G'$.
Completeness: If $G \rightarrow^*_{\mathcal{R}} G'$, there exists $S \in \mathcal{L}(\mathcal{R})$ and a derivation T of the form $C_0 = \{[S, G]\} \longmapsto \ldots \longmapsto C_k = \{...[S'_k, G'_k]...\}$ such that $G' \in Reach(T)$.

Special strategic programs called *results* in [25], are those of the form $[\texttt{id}, G']$ or $[\texttt{fail}, G']$. For a given configuration $C = \{O_1, \ldots, O_k, \ldots, O_n\}$, where each O_i is a strategic program $[S_i, G_i]$, let $Results(C)$ (respectively $Results(T)$) be the subset of $Reach(C)$ (respectively $Reach(T)$) that are results. The result set associated to a configuration or a derivation can be empty, which can be the case for non-terminating programs.

A configuration is *terminal* if no transition can be performed. A meaningful property to prove is that all terminal configurations consist of *results* of the form $[\texttt{id}, G']$ or $[\texttt{fail}, G']$. This is expressed through the following *Progress property: Characterisation of Terminal Configurations.* For every strategic rewrite program $[S, G]$ that is not a result (i.e., $S \neq \texttt{id}$ and $S \neq \texttt{fail}$), there exists a configuration C such that $\{[S, G]\} \mapsto C$. In other words, in this case, there are no blocked programs: the transition system ensures that, for any configuration, either there are transitions to perform, or we have reached results.

Strategic programs are not terminating in general, however it may be suitable to identify a terminating sublanguage (i.e. a sublanguage for which the transition relation is terminating). For instance, it is not difficult (but not surprising) to prove that in Porgy, the sublanguage that excludes iterators (such as the while/repeat construct) is strongly terminating.

Last, with respect to the computation power of the language, it is easy to state, as in [35], the Turing completeness property.
Computational Completeness property: The set of all strategic programs $[S_R, G]$ is Turing complete, i.e. can simulate any Turing machine. (Sequential composition and iteration are enough) [35].

7 Properties of strategic rewriting

Since strategic rewriting restricts the set of rewriting derivations, it needs careful definitions of termination and confluence under strategies, explored in [41,42].

These properties of confluence or termination for rewriting under strategies have been largely addressed in the rewriting community for specific term rewriting strategies. Different approaches have been explored, either based on schematization of derivation trees, as in [30], or by tuning proof methods to handle specific strategies (innermost, outermost, lazy strategies) as in [28,29]. Termination of on-demand rewriting in the context of OBJ programs is studied in [50,49,1]. Other approaches as [8] use strategies transformation to equivalent rewrite systems to be able to reuse well-known methods.

When the concept of normal form is important, like in the context of term rewriting systems (TRS for short) where rewriting strategies look for efficient ways to compute normal forms, a relevant question is: which (computable) strategies are guaranteed to find a normal form for any term whenever it exists? Having in mind that, given a set of rules $\mathcal{R}$, a strategic term rewriting reduction *normalizes the term* t if there is no infinite $\xrightarrow{S}$-rewrite sequence starting from t, a strategic rewriting reduction is *normalizing* or *complete* if it normalizes every term that has an $\mathcal{R}$-normal form. Proving completeness of strategic rewriting w.r.t. normal forms is actually a difficult problem and results have been most often obtained in the context of orthogonal systems (i.e. with left-linear non-overlapping left-hand sides). Innermost and outermost reduction are studied for instance in [59,10] where it is shown that the leftmost outermost strategy is normalizing for orthogonal left-normal TRS, but not in general [37,38]. Innermost strategy is complete for terminating TRS and some other restricted class as explored in [60].

Special efforts have been devoted to needed reductions. Needed reduction is interesting for orthogonal term rewriting systems occurring in combinatory logic, λ-calculus, functional programming. Already in 1979, later published in [38], Huet and Lévy defined the notions of needed and strongly needed redexes for orthogonal rewrite systems. The main idea here is to find the optimal way, when it exists, to reach the normal form of 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. In an orthogonal TRS, every reducible term contains a needed redex and repeated contraction of needed redexes results in a normal form, if it exists. Unfortunately neededness of a redex is not decidable [69] except for some classes of rewrite systems: in *sequential* TRS [5], every term which is not in normal form contains a needed redex [10]. Strong sequentiality is decidable for left-linear TRS. External redexes (outermost until contracted) are needed. But outermost redexes may fail to be needed if the TRS is not orthogonal. For instance, with $\mathcal{R} = \{f(a) \Rightarrow b,\ a \Rightarrow b\}$, the term $f(a)$ contains two redexes, but the outermost one is not needed: the rewriting step $f(a) \longrightarrow_{\mathcal{R}} f(b)$ normalizes the term without contracting the outermost redex[11]. Again combinatory logic and λ-calculus satisfy these conditions and have motivated their study.

[11] Remark due to an external referee.

Sufficient conditions to ensure that context-sensitive rewriting is able to compute head-normal forms (terms that do not rewrite into a redex) have been established in [48]. In fact, for a given TRS, it is possible to automatically provide replacement maps supporting such computations. In this setting, the *canonical replacement map* (denoted by μ_{can}) specifies the most restrictive replacement map which can be automatically associated to a TRS $\mathcal{R}$ in order to achieve completeness of context-sensitive computations, whenever the TRS is left-linear. So left-linear, confluent, and μ_{can}-terminating TRS admit a computable normalizing strategy to head-normal forms.

8 Conclusion and further work

A lot of questions about strategies are yet open, going from the definition of this concept and the interesting properties we may expect to prove, up to the definition of domain specific strategy languages. As further research topics, several directions seem really worth exploring. The first one is the connection with game theory strategies. In the fields of system design and verification, *games* have emerged as a key tool. Such games have been studied since the first half of 20th century in descriptive set theory [40], and they have been adapted and generalized for applications in formal verification; introductions can be found in [33,72]. The coincidence of the term "strategy" in the domains of rewriting and games is more than a pun. It should be fruitful to explore further the connection and to be guided in the study of strategies by some of the insights in the literature of games.

The second research direction is related to proving properties of strategies and strategic reductions. A lot of work has already begun in the rewriting community and have been presented in journals, workshops or conferences of this domain. Properties of confluence, termination, or completeness for rewriting under strategies have been largely addressed. However, as mentioned in Section 3.1, the application of rules to the considered objects can optionally be restricted by conditions or constraints, and this generalization has to be carefully studied. When conditional rules are allowed, a number of concepts and computational properties that are mentioned here may crucially depend on the conditional part of the rules. For instance, regarding termination, the notion of operational termination (defined as the absence of infinite proof trees), studied in [51] for conditional term rewriting (CTRS) systems, is different from the notion of termination considered here (the absence of infinite reduction sequences). As another example, a discussion about how irreducible terms and normal forms are also different for CTRSs can be found in [52]. Taking into account these phenomena could provide more insights on strategies.

In addition, other properties of strategies such as fairness or loop-freeness could be worthfully explored, again by making connections between different communities (functional programming, proof theory, verification, game theory,...).

Acknowledgements The results presented here are based on pioneer work in the Elan language designed in the Protheo team from 1990 to 2002. They rely on

joint work with many people, in particular Marian Vittek and Peter Borovanský, Claude Kirchner and Florent Kirchner, Dan Dougherty, Horatiu Cirstea and Tony Bourdier, Oana Andrei, Maribel Fernandez and Olivier Namet. I am grateful to the members of the Protheo and the Porgy teams, for many inspiring discussions on the topics of this paper. I sincerely thank the reviewers for their careful reading and pertinent and constructive remarks. A special tribute is given to José Meseguer, for his inspiring works on rewriting logic and strategies, and this paper is dedicated to him.

References

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 (WRLA)*, volume 71 of *Elecronic Notes In Theoretical Computer Science*, pages 70–89, 2004.
2. Oana Andrei, Maribel Fernandez, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In Rachid Echahed, editor, *TERMGRAPH, 6th Int. Workshop on Computing with Terms and Graphs*, volume 48 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 54–68, 2011.
3. Oana Andrei and Hélène Kirchner. A Port Graph Calculus for Autonomic Computing and Invariant Verification. *Electronic Notes In Theoretical Computer Science*, 253(4):17–38, march 2009.
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. Sergio Antoy and Aart Middeldorp. A sequential reduction strategy. *Theoretical Computer Science*, 165(1):75–95, 1996.
6. Lennart Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 218–227, New York, NY, USA, 1984. ACM.
7. Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.
8. Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Effective strategic programming for java developers. *Software: Practice and Experience*, 2012.
9. Henk Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.
10. M. Bezem, J.W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.
11. Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, July 2002.
12. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15:55–70, 1998.

13. Peter Borovanský, Claude Kirchner, Hélène 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.
14. Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty, and Hélène Kirchner. Extensional and intensional strategies. In *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming*, volume 15 of *Electronic Proceedings In Theoretical Computer Science*, pages 1–19, 2009.
15. Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3):263–293, 1998.
16. H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
17. Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.
18. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Reriting Logic*, volume 4350. Springer, 2007.
19. Leonardo de Moura and Grant Olney Passmore. The Strategy Challenge in SMT Solving. In Maria-Paola Bonacina and Mark Stickel, editors, *Automated Reasoning and Mathematics*, volume 7788 of *Lecture Notes in Computer Science*, pages 15–44. Springer Berlin Heidelberg, 2013.
20. 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.
21. Edsger W. Dijkstra. *Selected writings on computing - a personal perspective*. Texts and monographs in computer science. Springer, 1982.
22. Daniel J. Dougherty. Rewriting strategies and game strategies. Internal report, August 2008.
23. Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG approach: Language and environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 551–603. World Scientific, 1997.
24. Maribel Fernández, Hélène Kirchner, and Olivier Namet. A strategy language for graph rewriting. In Germán Vidal, editor, *Logic-Based Program Synthesis and Transformation, 21st International Symposium, LOPSTR 2011, Odense, Denmark, July 18-20, 2011. Revised Selected Papers.*, volume 7225 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2012.
25. Maribel Fernández, Hélène Kirchner, and Olivier Namet. Strategic portgraph rewriting: an interactive modelling and analysis framework. In Alberto Lluch Lafuente Dragan Bosnacki, Stefan Edelkamp and Anton Wij, editors, *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering (GRAPHITE 2014), Grenoble, France, 5th April 2014*, volume 159 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–29, 2014.
26. K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 52–66. ACM Press, 1985.

27. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer Verlag, 2006.
28. Jürgen Giesl and Aart Middeldorp. Innermost termination of context-sensitive rewriting. In *Proceedings of the 6th International Conference on Developments in Language Theory (DLT 2002)*, volume 2450 of *Lecture Notes in Computer Science*, pages 231–244, Kyoto, Japan, 2003. Springer.
29. Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, 2011.
30. Isabelle Gnaedig and Hélène Kirchner. Termination of rewriting under strategies. *ACM Trans. Comput. Logic*, 10(2):1–52, 2009.
31. Joseph Goguen and Grant Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*, volume 2 of *Advances in Formal Methods*. Kluwer Academic Publishers, Boston, 2000. ISBN 0-7923-7757-5.
32. 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.
33. Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.
34. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
35. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In Furio Honsell and Marino Miculan, editor, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001.
36. Michael Hanus. Curry: A multi-paradigm declarative language (system description). In *Twelfth Workshop Logic Programming, WLP'97, Munich*, 1997.
37. Gérard Huet and Jean-Jacques Lévy. Computations in non-ambiguous linear term rewriting systems. Technical report, INRIA Laboria, 1979.
38. Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, I and II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, chapter 11, 12, pages 395–414. MIT press, 1991.
39. Simon L. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
40. Alexander S. Kechris. *Classical Descriptive Set Theory*, volume 156 of *Graduate Texts in Mathematics*. Springer, 1995.
41. Claude Kirchner, Florent Kirchner, and Hélène Kirchner. Strategic computations and deductions. In Christoph Benzmüller, Chad E. Brown, Jörg Siekmann, and Richard Statman, editors, *Reasoning in Simple Type Theory. Festchrift in Honour of Peter B. Andrews on His 70th Birthday*, volume 17 of *Studies in Logic and the Foundations of Mathematics*, pages 339–364. College Publications, 2008.
42. Claude Kirchner, Florent Kirchner, and Hélène Kirchner. Constraint based strategies. In *Proceedings 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2009), Brasilia*, volume 5979 of *Lecture Notes in Computer Science*, pages 13–26, 2010.

43. Claude Kirchner, Hélène Kirchner, and Marian Vittek. Implementing computational systems with constraints. In *Principles and Practice of Constraint Programming*, pages 156–165, 1993.
44. 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.
45. Hélène Kirchner. A rewriting point of view on strategies. In Fabio Mogavero, Aniello Murano, and Moshe Y. Vardi, editors, *Proceedings 1st International Workshop on Strategic Reasoning, SR 2013, Rome, Italy, March 16-17, 2013.*, volume 112 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 99–105, 2013.
46. Alexander Letichevsky. Development of rewriting strategies. In M. Bruynooghe and J. Penjam, editors, *PLILP*, volume 714 of *Lecture Notes in Computer Science*, pages 378–390. Springer, 1993.
47. Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.
48. Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1:1–61, January 1998.
49. 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.
50. 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.
51. Salvador Lucas, Claude Marché, and José Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2015.
52. Salvador Lucas and José Meseguer. Strong and weak operational termination of order-sorted rewrite theories. In *Selected Papers of International Workshop on Rewriting Logic and its Applications (WRLA)*, volume 8663 of *Lecture Notes in Computer Science*, pages 178—194. Springer, 2014.
53. Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.
54. 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 Science Publishers B. V. (North-Holland), 2005.
55. Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. *Electronic Notes in Theoretical Computer Science*, 238(3):227–247, 2008.
56. William McCune. Semantic guidance for saturation provers. *Artificial Intelligence and Symbolic Computation*, pages 18–24, 2006.
57. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

58. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *ICSE*, pages 742–745, 2000.
59. Michael J. O'Donnell, editor. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer, 1977.
60. Koji Okamoto, Masahiko Sakai, Naoki Nishida, and Toshiki Sakabe. Weakly-innermost strategy and its completeness on terminating right-linear TRSs. In *Proceedings 5th International Workshop on Reduction Strategies in Rewriting and Programming April 22, 2005, Nara, Japan*, ENTCS, 2005.
61. 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.
62. M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
63. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
64. Detlef Plump. The Graph Programming Language GP. In Symeon Bozapalidis and George Rahonis, editors, *CAI*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.
65. Detlef Plump and Sandra Steinert. The semantics of graph programs. In Ian Mackie and Anamaria Martins Moreira, editors, *Proceedings Tenth International Workshop on Rule-Based Programming, RULE 2009, Brasília, Brazil, 28th June 2009.*, volume 21 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 27–38, 2009.
66. Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003.
67. Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*, pages 479–546. World Scientific, 1997.
68. Jaco Van de Pol. Just-in-time: On strategy annotations. In *Proceedings of WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming*, volume 57 of *Elecronic Notes In Theoretical Computer Science*, pages 41–63, 2001.
69. Vincent van Oostrom and Roel de Vrijer. *Term Rewriting Systems*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*, chapter 9: Strategies. Cambridge University Press, 2003.
70. 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, May 2001.
71. Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.
72. Igor Walukiewicz. A landscape with games in the background. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 356–366, 2004.