Generic type-preserving traversal strategies

Ralf Lämmel

CWI, Kruislaan 413, NL-1098 SJ Amsterdam
Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam
Email: `Ralf.Laemmel@cwi.nl`
WWW: `http://www.cwi.nl/~ralf/`

Abstract. A typed model of strategic rewriting is developed. The main result is that generic traversals are covered. The assumed calculus offers, for example, a strategy operator $\Box(\cdot)$, which applies the argument strategy to all children. To provide a typeful model for generic strategies, one has to identify signature-independent, that is, generic types. In the present article, we restrict ourselves to TP—the generic type of all Type-Preserving strategies. TP is easily integrated into a standard many-sorted type system for rewriting. To inhabit TP, we need to introduce a (left-biased) type-driven choice operator $\cdot \leftarrow\!\!\!\!\!\oplus \cdot$. The operator applies its left argument (corresponding to a sort-specific strategy) if the type of the given term fits, and the operator resorts to the right argument (corresponding to a generic default) otherwise. This approach dictates that the semantics of strategy application must be type-dependent to a certain extent. Fortunately, the overhead can be shown to be minimal.

1 Introduction

Strategic rewriting Several frameworks for rewriting offer means to describe strategies (as opposed to frameworks which assume a fixed built-in strategy for normalisation/evaluation). Strategies are supported, for example, by the specification formalisms Maude [13, 7] and ELAN [4, 5]. Also, the ρ-calculus [6] is very suitable for the definition of strategies. The programming language Stratego [17] based on system S [18] is entirely devoted to strategic programming. The idea of rewriting strategies goes back to Paulson's work on higher-order implementation of rewriting strategies [15]. Strategies are useful to describe evaluation and normalisation strategies, e.g., to control rewriting for some rewrite rules which are not confluent or terminating when considered as a standard rewrite system. Strategies are particularly useful for the specification of traversals. To describe traversals in standard rewriting (without extra support for traversals), one has to resort to auxiliary function symbols, and rewrite rules have to be used to perform the actual traversal for the signature at hand (usually one rewrite rule per term constructor). Generic traversal primitives support an important dimension of genericity in programming. In [16], for example, generic traversal strategies are used for language implementation: Algorithms for free variable collection, substitution, unification and others are defined in a generic, that is, language-independent manner by suitably parameterised generic traversals.

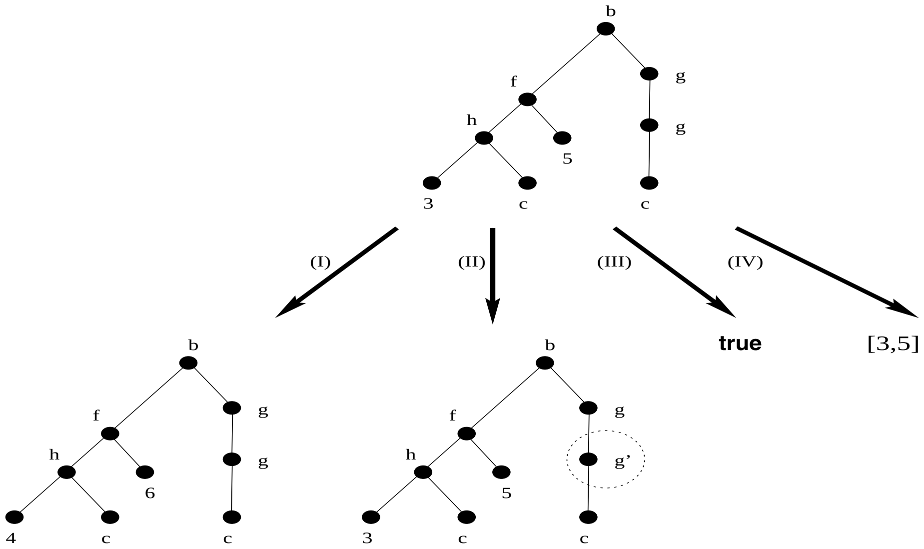

Fig. 1. Illustration of generic traversals

Examples of generic traversals In Figure 1, four examples (I)–(IV) of intentionally generic traversals are illustrated. In (I), all naturals in the given term (say tree) are incremented as modelled by the rewrite rule $N \to succ(N)$. We need to turn this rule into a traversal strategy because the rule on its own is not terminating when considered as rewrite system. The strategy should be generic, that is, it should be applicable to any term. In (II), a particular pattern is rewritten according to the rewrite rule $g(P) \to g'(P)$. Assume that we want to control this replacement so that it is performed in bottom-up manner, and the first matching term is rewritten only. The strategy to locate the desired node in the term is completely generic. The examples (III) and (IV) are examples of type-changing traversals, actually these are type-unifying traversals according to [12, 11]. In (III), we might test some property of the term, e.g., if naturals occur at all. In (IV), we collect all the naturals in the term using a left-to-right traversal.

Value of typing The contribution of the article is a type system which covers generic traversals as needed for the examples above. Actually, in the present article, we restrict ourselves to type-preserving strategies. A more exhaustive treatment including type-changing strategies can be found in [10]. Let us understand why types are valuable. In general, typing should obviously prevent us from constructing ill-typed terms. Generic traversals typically employ specific ingredients (say sort-specific rewrite rules). A type system and the corresponding dynamic semantics should ensure that the specific ingredients are applied in a type-safe manner. Consider, for example, the rewrite rule for incrementation $N \to succ(N)$ as assumed in example (I) above. This one-step rewrite rule should only be applied to naturals during a traversal. On the other hand, the complete traversal must be able to process *any* term. A type system should also prevent the programmer from combining specific and generic strategies in certain undesirable ways. Consider, for example, a left-biased choice $\ell \leftarrow\!\!\!+ \textsc{Id}$ where the sort-specific rewrite rule ℓ is applied if possible, and otherwise the "generic default" $\textsc{Id}$ triggers. One might argue that this strategy is generic because $\textsc{Id}$ is applicable to terms of all possible sorts. Actually, we favour two other possible interpretations. Either we refuse this choice altogether (because we would insist on the types of the argument strategies to be the same), or we favour the intersection of the argument types for the type of the compound strategy. The strategy $\ell \leftarrow\!\!\!+ \textsc{Id}$ will be specifically typed in this case. In fact, strategies should not get generic too easily since we otherwise loose the valuable precision of a many-sorted type system. Even if ill-typed types cannot be constructed, accidentally generic strategies are likely to refuse terms (leading to failure), or they leave terms unchanged in an untraceable manner.

Difficulties of typing Some strategy combinators are easier to type than others. Combinators for different kinds of choice, sequential composition, signature-specific congruence operators and others are easy to type in a many-sorted setting. Some use of overloading and/or parametric types might be necessary. Indeed, ELAN is based on such a many-sorted type system. By contrast, generic traversal primitives (e.g., $\square(s)$ to apply a strategy s to all children of a given term as provided by Stratego or system S) are more challenging since standard many-sorted types are not applicable, and also other well-established concepts like polymorphism are insufficient to model the kind of genericity needed. Generic traversal strategies have to be applicable to terms of any sort (or at least to some class of types). Generic traversals are in a sense type-sensitive (as opposed to polymorphism) since they are usually derived from specific ingredients (say rewrite rules) to deal with some distinguished sorts in a specific manner. We also refer the reader to [16, 11], where it is argued that typing generic traversals is difficult. Typing generic traversals is further complicated if type-changing strategies are covered [10].

Contribution and structure of the article In Section 2, we shortly recall untyped strategies including primitives for traversals. In Section 3, we discuss standard many-sorted types for type-preserving strategies.[1] In Section 4, we provide a type system which includes a generic strategy type TP for generic type-preserving strategies. To this end, we also need to introduce a type-sensitive choice operator to mediate between sort-specific and generic strategies. In Section 5, we discuss implementation issues. In the course of the article, we show that our type system for strategies is sensible from a strategic programmer's point of view. We envision that the presented type system disciplines strategic programs (employing generic traversals) in a useful and not too restrictive manner. We also show that generic type-preserving strategies can (more or less) easily be implemented. The article is concluded in Section 6.

[1] As for terminology, we use the term "type" even for types of many-sorted terms (as opposed to the term "sort"). The term "type" is more common in the context of type systems. Also, we might easily go beyond just many-sorted terms, and deal with polymorphic datatypes.

2 Untyped strategies

We set up a rewriting calculus very much inspired by ELAN, the ρ-calculus, and system S. We are very brief regarding explanations, examples, and details of the semantics. Some basic knowledge of strategic rewriting (as found in [3, 18, 6]) is a helpful background for reading the present article.

First, we give an overview on the strategy combinators we want to cover, and we explain how to define new ones by means of strategy definitions. Then, we explain the semantic model for strategy application. Finally, we devote a detailed explanation to the generic traversal primitives included in our framework.

2.1 Strategy combinators

We have the following grammar for strategy expressions:

$$s ::= t \to t \mid \textsc{Id} \mid \textsc{Fail} \mid s; s \mid s + s \mid s \nleftarrow s \mid f(s, \ldots, s) \mid \Box(s) \mid \Diamond(s)$$

There is a form of strategy $t \to t'$ for one-step rules to be applied at the root of the term. We adopt some common restrictions for rewrite rules. The left-hand side t determines the bound variables. (Free) variables on the right-hand t' side also occur in t. If substitutions are applied, then we assume α-conversion. Besides rule formation, there are standard primitives for the identity strategy ($\textsc{Id}$), the failure strategy ($\textsc{Fail}$), sequential composition $(\cdot;\cdot)$, symmetric choice $(\cdot + \cdot)$, asymmetric left-biased choice $(\cdot \nleftarrow \cdot)$. The strategy $f(s_1, \ldots, s_n)$ denotes the congruence strategy for the function symbol f.[2] The forms $\Box(\cdot)$ and $\Diamond(\cdot)$ are the traversal primitives. The strategy $\Box(s)$ applies the argument strategy s to all children of the given term. The strategy $\Diamond(s)$ applies the argument strategy s to some child of the given term. We postpone discussing $\Box(\cdot)$ and $\Diamond(\cdot)$ in detail.

Example 1. Let us consider the problem of flipping the top-level subtrees in a binary tree with naturals at the leafs. We assume the following symbols to construct such trees:

$$\begin{aligned} \mathit{zero} &: \mathit{Nat} \\ \mathit{succ} &: \mathit{Nat} \to \mathit{Nat} \\ \mathit{leaf} &: \mathit{Nat} \to \mathit{Tree} \\ \mathit{fork} &: \mathit{Tree} \times \mathit{Tree} \to \mathit{Tree} \end{aligned}$$

N and T—optionally primed or subscripted—are used as variables of sort *Nat* and *Tree*, respectively. We can specify the problem of flipping top-level subtrees with a standard term rewriting system. We need to use an auxiliary function symbol *flip-top* in order to obtain a terminating rewrite system:

$$\begin{aligned} \mathit{flip\text{-}top}(\mathit{leaf}(N)) &\to \mathit{leaf}(N) \\ \mathit{flip\text{-}top}(\mathit{fork}(T_1, T_2)) &\to \mathit{fork}(T_2, T_1) \end{aligned}$$

[2] I.e., the strategy $f(s_1, \ldots, s_n)$ applies the argument strategies to the parameters of a term with f as outermost symbol, otherwise the strategy fails unconditionally.

Alternatively, we can use a strategy *flip-top* without introducing additional function symbols. We combine two one-step rewrite rules via $\cdot + \cdot$. Rule ℓ_1 models preservation of leafs, whereas rule ℓ_2 flips top-level subtrees:

$$\begin{aligned} \ell_1 &= \mathit{leaf}(N) \to \mathit{leaf}(N) \\ \ell_2 &= \mathit{fork}(T_1, T_2) \to \mathit{fork}(T_2, T_1) \\ \mathit{flip\text{-}top} &= \ell_1 + \ell_2 \end{aligned}$$

New strategy combinators can be defined by means of an abstraction mechanism which we call strategy definitions. Similar mechanisms are provided by ELAN, system S and Stratego. A definition $\varphi(\nu_1, \ldots, \nu_n) = s$ introduces an n-ary strategy combinator φ. An application $\varphi(s_1, \ldots, s_n)$ of φ denotes the instantiation $s[\nu_1 := s_1, \ldots, \nu_n := s_n]$ of the body s of the definition of φ. Strategy definitions can be recursive.[3]

Example 2. Some sample definitions are the following:

$$\begin{aligned} \mathit{try}(\nu) &= \nu \nleftarrow\!\!+ \textsc{Id} \\ \mathit{repeat}(\nu) &= \mathit{try}(\nu; \mathit{repeat}(\nu)) \\ \mathit{flip\text{-}top} &= \mathit{try}(\ell_2) \end{aligned}$$

The first two strategies are generic. $\mathit{try}(s)$ applies s, but behaves like $\textsc{Id}$ if s fails. $\mathit{repeat}(s)$ repeatedly applies s as often as possible. The strategy *flip-top* is specific in nature. It reconstructs the strategy of the same name introduced earlier in Example 1. The reconstruction illustrates the use of *try*.

2.2 The semantic model

The application of a strategy s to a term t is denoted by $\langle s \rangle t$. As for the dynamic semantics, we employ a judgement for strategy application $\vdash \langle s \rangle t \rightsquigarrow r$ where r is the reduct which is either a term t' or failure denoted by "$\uparrow$". This model has been adopted from system S. Note that the given judgement is not sufficient to define the semantics of applications of strategy definitions. For that purpose, we had to propagate the definitions via a context parameter. Note also that we assume that strategies are only applied to ground terms, and then also yield ground terms.[4] We employ a certain style for the specification of the deduction rules. One gives positive rules for cases when the reduct is a term, and one also gives negative rules for the remaining cases with failure as the reduct.

We show an excerpt of the evaluation judgement in Figure 2. It covers the positive and negative rules for asymmetric left-biased choice. These rules also illustrate why we need to include failure as reduct. Otherwise, the semantics could not query whether a certain application did *not* succeed (cf. [$\mathsf{lchoice}^+$.2]).

[3] Recursive definitions are common in ELAN, whereas system S and official Stratego employ a special recursion operator $\mu \cdot . \cdot$.

[4] The assumption is well in line with standard rewriting, especially for ordinary first-order rewrite rules. It is certainly possible to give up the assumption.

Semantics of strategy application $\boxed{\vdash \langle s \rangle t \leadsto r}$

Positive rules

$$\frac{\vdash \langle s \rangle t \leadsto t'}{\vdash \langle s \nleftarrow s' \rangle t \leadsto t'} \quad [\mathsf{lchoice}^+.1]$$

$$\frac{\begin{array}{c}\vdash \langle s \rangle t \leadsto \uparrow \\ \wedge \ \vdash \langle s' \rangle t \leadsto t'\end{array}}{\vdash \langle s \nleftarrow s' \rangle t \leadsto t'} \quad [\mathsf{lchoice}^+.2]$$

Negative rule

$$\frac{\begin{array}{c}\vdash \langle s \rangle t \leadsto \uparrow \\ \wedge \ \vdash \langle s' \rangle t \leadsto \uparrow\end{array}}{\vdash \langle s \nleftarrow s' \rangle t \leadsto \uparrow} \quad [\mathsf{lchoice}^-]$$

Fig. 2. Positive and negative rules for application of $s \nleftarrow s'$

2.3 Generic traversal primitives

Let us take a closer look at the generic traversal primitives to apply a strategy s to all children ($\Box(s)$), or to some child ($\Diamond(s)$). The operators $\Box(\cdot)$ and $\Diamond(\cdot)$ are defined like in system S. For brevity, we do not consider the hybrid operator $\varoast$ from system S which applies a strategy to one or more children. Similar operators can also be defined in the ρ-calculus (cf. $\Psi(s)$ and $\Phi(s)$ in [6] corresponding to $\Box(s)$ and $\Diamond(s)$).

In Figure 3, we show some useful derivable generic traversal strategies defined in terms of $\Box(\cdot)$ and $\Diamond(\cdot)$. The definitions are adopted from [18] except the last one. Note that ν is a meta-variable for strategies in all these definitions.

$\mathit{topdown}(\nu)$	$= \nu; \Box(\mathit{topdown}(\nu))$	(Apply ν in top-down manner)
$\mathit{bottomup}(\nu)$	$= \Box(\mathit{bottomup}(\nu)); \nu$	(Apply ν in bottom-up manner)
$\mathit{oncetd}(\nu)$	$= \nu \nleftarrow \Diamond(\mathit{oncetd}(\nu))$	(Apply ν once in top-down manner)
$\mathit{oncebu}(\nu)$	$= \Diamond(\mathit{oncebu}(\nu)) \nleftarrow \nu$	(Apply ν once in bottom-up manner)
$\mathit{innermost}(\nu)$	$= \mathit{repeat}(\mathit{oncebu}(\nu))$	(Innermost evaluation strategy for ν)
$\mathit{stoptd}(\nu)$	$= \nu \nleftarrow \Box(\mathit{stoptd}(\nu))$	(Apply ν in top-down manner with "cut")

Fig. 3. Some derived generic traversal strategies

Example 3. Let us solve the first two problems (I) and (II) illustrated in Figure 1 in the introduction of the article:

$$\begin{aligned} \mathit{nat} &= \mathit{zero} + \mathit{succ}(\textsc{Id}) \\ \mathit{traverse}_{(I)} &= \mathit{stoptd}(\mathit{nat}; N \to \mathit{succ}(N)) \\ \mathit{traverse}_{(II)} &= \mathit{oncebu}(g(P) \to g'(P)) \end{aligned}$$

nat is an auxiliary strategy testing for naturals in terms of the congruence strategies for *zero* and *succ*. We use *nat* as type check to enable the applicability of the rewrite rule $N \to \mathit{succ}(N)$ in the definition of $\mathit{traverse}_{(I)}$. Recall, $\mathit{traverse}_{(I)}$ is meant to increment all naturals. The corresponding strategy is defined in terms of the generic strategy *stoptd* (cf. Figure 3) which descends into the given term as long as the argument strategy ν does not succeed. This is exactly the traversal scheme we need to increment all naturals in a tree. Encountering naturals in a top-down manner we apply the incrementation rule, but then we do not further descend into the term. If we used *topdown* instead of *stoptd*, we describe a non-terminating strategy. The definition of $traverse_{(II)}$ is also very easy to read. $traverse_{(II)}$ finds the first pattern of form $g(x)$ in bottom-up-manner (as required), and replaces it by $g'(x)$ (as expressed by the rewrite rule). Note the genericity of the traversals $traverse_{(I)}$ and $traverse_{(II)}$. They can be applied to any term. Of course, the strategies are somewhat specific because they rely on some constant or function symbols, namely *zero*, *succ*, g, and g'.

As an aside, since the present article only covers type-preserving strategies, we cannot implement the two other problems from the introduction.

The positive semantics rules for $\Box(\cdot)$ and $\Diamond(\cdot)$ are shown in Figure 4. The rule [all^+.1] says that $\Box(s)$, when applied to a constant, immediately succeeds because there are no children which s has to be applied to. The rule [all^+.2] directly encodes what it means to apply s to all children of a term $f(t_1, \ldots, t_n)$. Note that the function symbol f is preserved in the result. The definition of $\Diamond(s)$ is similar. The rule [one^+.1] says that s is applied to some subterm t_i of $f(t_1, \ldots, t_n)$.

Semantics of strategy application $\boxed{\vdash \langle s \rangle t \leadsto r}$

$$\vdash \langle \Box(s) \rangle c \leadsto c \qquad [\mathsf{all}^+.1]$$

$$\frac{\begin{array}{ll} & \vdash \langle s \rangle t_1 \leadsto t'_1 \\ \wedge & \cdots \\ \wedge & \vdash \langle s \rangle t_n \leadsto t'_n \end{array}}{\vdash \langle \Box(s) \rangle f(t_1, \ldots, t_n) \leadsto f(t'_1, \ldots, t'_n)} \qquad [\mathsf{all}^+.2]$$

$$\frac{\exists i \in \{1, \ldots, n\}.\ \vdash \langle s \rangle t_i \leadsto t'_i}{\vdash \langle \Diamond(s) \rangle f(t_1, \ldots, t_n) \leadsto f(t_1, \ldots, t'_i, \ldots, t_n)} \qquad [\mathsf{one}^+.1]$$

Fig. 4. Evaluation of $\Box(s)$ and $\Diamond(s)$

3 Simply-typed strategies

As a warm-up, we provide a type system for (non-generic, say sort-specific) type-preserving strategies. First, we will explain the model for the type system. Then, we discuss the actual deduction rules in some detail. Finally, we discuss some design properties of the type system, mainly to prepare it for an extension to cover generic strategies (as developed in main part of the paper, that is, Section 4).

3.1 The type model

There are two levels of types. We have types for many-sorted terms and types for strategies. We use σ to range over sorts, τ to range over term types, and π

to range over strategy types. The forms of type expressions are trivially defined by the following grammar:[5]

$$\begin{array}{ll} \tau ::= \sigma & \text{(Term types)} \\ \pi ::= \tau \to \tau & \text{(Strategy types)} \end{array}$$

In the typing judgements, we use a context parameter Γ to keep track of sorts σ, to map constant symbols c, function symbols f, term variables x, strategy variables ν, and combinators φ to types. We use the following grammar for contexts:

$$\begin{array}{lll} \Gamma ::= & \emptyset \mid \Gamma \cup \Gamma & \text{(Contexts as sets)} \\ & \mid\ \sigma \mid c : \sigma \mid f : \sigma \times \cdots \times \sigma \to \sigma & \text{(Signature part)} \\ & \mid\ x : \tau \mid \nu : \pi & \text{(Term and strategy variables)} \\ & \mid\ \varphi : \pi \times \cdots \times \pi \to \pi & \text{(Strategy combinators)} \end{array}$$

Thus, Γ contains a many-sorted term signature, variable declarations (for term variables and strategy variables), and combinator type declarations originating from strategy definitions. Let us state a few well-formedness requirements. We assume that the various kinds of symbols and variables are not confused (i.e., there are different name spaces), and the symbols and variables are not associated with different types in Γ (in particular, we do not consider overloading). All sorts used in some type declaration have also to be introduced in Γ. All type declarations have to be well-formed (w.r.t. the well-formedness judgements defined below). Note that Γ is assumed to be static (say given) in all upcoming judgements. Thus, we assume explicit type declarations for the various kinds of variables and symbols.[6] It is easy to infer Γ instead.

The principal judgement of the type system is the type judgement for strategies. It is of the form $\Gamma \vdash s : \pi$, and it holds if the strategy s is of strategy type π in the context of Γ.

3.2 Deduction rules

The deduction rules for the various judgements are shown in Figure 5. For brevity, we omit the typing rules for strategy definitions.

Note that type-preservation is enforced by the well-formedness judgement for strategy types (cf. rule [pi.1]), and also by some other rules (cf. rules [comp.1], [apply], [rule], [id], [fail], and [congr]). The type system for simply-typed strategies should not be regarded as a contribution of the present article. It is rather straightforward, and it corresponds very much to the kind of type system assumed for ELAN. Let us read some inference rules for convenience. Rule [apply] says that a strategy application $\langle s \rangle t$ is well-typed if the strategy s is of type

[5] These grammar rules are subject to extension when we wanted to accomplish more expressiveness than sort-specific type-preserving strategies, especially polymorphic datatypes and generic strategies.

[6] Declarations for variables, rewriting functions and strategies are common in several frameworks for rewriting, e.g., in ASF+SDF and ELAN.

Term types $\boxed{\Gamma \vdash \tau}$

$$\frac{\sigma \in \Gamma}{\Gamma \vdash \sigma} \quad [\mathsf{tau.1}]$$

Strategy types $\boxed{\Gamma \vdash \pi}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \to \tau} \quad [\mathsf{pi.1}]$$

Terms $\boxed{\Gamma \vdash t : \tau}$

$$\frac{c : \sigma \in \Gamma}{\Gamma \vdash c : \sigma} \quad [\mathsf{const}]$$

$$\frac{\begin{array}{l} f : \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \Gamma \\ \wedge\ \Gamma \vdash t_1 : \sigma_1 \\ \wedge\ \cdots \\ \wedge\ \Gamma \vdash t_n : \sigma_n \end{array}}{\Gamma \vdash f(t_1, \ldots, t_n) : \sigma} \quad [\mathsf{fun}]$$

$$\frac{x : \tau \in X}{\Gamma \vdash x : \tau} \quad [\mathsf{var}]$$

Composable types $\boxed{\pi_1 \circ_\Gamma \pi_2 \leadsto \pi}$

$$\frac{\Gamma \vdash \tau}{\tau \to \tau \circ_\Gamma \tau \to \tau \leadsto \tau \to \tau} \quad [\mathsf{comp.1}]$$

Strategy application $\boxed{\Gamma \vdash \langle s \rangle t : \tau}$

$$\frac{\begin{array}{l} \Gamma \vdash s : \tau \to \tau \\ \wedge\ \Gamma \vdash t : \tau \end{array}}{\Gamma \vdash \langle s \rangle t : \tau} \quad [\mathsf{apply}]$$

Strategies $\boxed{\Gamma \vdash s : \pi}$

$$\frac{\begin{array}{l} \Gamma \vdash t : \tau \\ \wedge\ \Gamma \vdash t' : \tau \end{array}}{\Gamma \vdash t \to t' : \tau \to \tau} \quad [\mathsf{rule}]$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \textsc{Id} : \tau \to \tau} \quad [\mathsf{id}]$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \textsc{Fail} : \tau \to \tau} \quad [\mathsf{fail}]$$

$$\frac{\begin{array}{l} \Gamma \vdash s_1 : \pi_1 \\ \wedge\ \Gamma \vdash s_2 : \pi_2 \\ \wedge\ \pi_1 \circ_\Gamma \pi_2 \leadsto \pi \end{array}}{\Gamma \vdash s_1 ; s_2 : \pi} \quad [\mathsf{seq}]$$

$$\frac{\begin{array}{l} \Gamma \vdash s_1 : \pi \\ \wedge\ \Gamma \vdash s_2 : \pi \end{array}}{\Gamma \vdash s_1 + s_2 : \pi} \quad [\mathsf{choice}]$$

$$\frac{\Gamma \vdash s_1 + s_2 : \pi}{\Gamma \vdash s_1 \not\leftarrow s_2 : \pi} \quad [\mathsf{lchoice}]$$

$$\frac{\begin{array}{l} f : \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \Gamma \\ \wedge\ \Gamma \vdash s_1 : \sigma_1 \to \sigma_1 \\ \wedge\ \cdots \\ \wedge\ \Gamma \vdash s_n : \sigma_n \to \sigma_n \end{array}}{\Gamma \vdash f(s_1, \ldots, s_n) : \sigma \to \sigma} \quad [\mathsf{congr}]$$

Fig. 5. Sort-specific type-preserving strategies

$\tau \to \tau$, and the term t is of type τ. The strategies Id and Fail have many types, namely any type $\tau \to \tau$ where $\Gamma \vdash \tau$ holds (cf. rules [id] and [fail]).[7] The strategy types for compound strategies are regulated by the rules [seq], [choice], [lchoice], and [congr]. The compound strategy $s_1; s_2$ refers to an auxiliary judgement for composable types. As for sort-specific type-preserving strategies, composable types are trivially defined according to rule [comp.1]. The compound strategies $s_1 + s_2$ and $s_1 \not\leftarrow s_2$ are well-typed if both strategies s_1 and s_2 are of a common type π which also determines the type of the compound strategy.

[7] Indeed, in ELAN, Id is defined by overloading. It is available for each sort in the signature at hand.

3.3 Discussion

On the positive side, we can assign types to certain strategies as illustrated by the following example.

Example 4. The strategy *flip-top* from Example 1 is type-preserving. Thus, the type *flip-top* : *Tree* $\rightarrow$ *Tree* can be approved for its definition.

On the negative side, there is no way to assign types to certain other strategies we have seen so far. Certainly, we cannot assign types to generic traversals as they are not restricted to a specific sort. But we cannot even assign types to some strategies which do not involve traversals. What are, for example, the types of *try* and *repeat* defined in Example 2? In a sense, these combinators take a type-preserving strategy, and return a type-preserving strategy. Like ID and FAIL, the combinators *try* and *repeat* could be associated with many types. However, this view interferes with the ideal of unicity of typing. Actually, the typing rules for ID and FAIL also violate unicity of typing, but as these are primitives, this violation can be regarded as an acceptable formulation of overloading, or as an encoding of an parametric type.[8]

4 Generically typed strategies

An important property of $\Box(s)$ and $\Diamond(s)$ is that they are supposed to be applicable to terms of *any* sort, i.e., they are generic. Clearly, this is also the case for ID and FAIL. Contrast that with a rewrite rule. It is only applicable to a term of a specific sort because of the way it is constructed from specifically-typed terms. Thus, we need to add a form of generic strategy type to our type model. Then, we are able to assign types to strategies involving $\Box(\cdot)$ and $\Diamond(\cdot)$ (and, at the same time, we also resolve the unicity problems with ID and FAIL, and we can assign types to strategies like *try* and *repeat*).

Note that the parameters of the traversal primitives have to be generically typed, too. Consider, for example, $\Box(s)$. The argument s must be potentially applicable to subterms of any sort. $\Box(s)$ might fail because some proper applicability conditions for s are not met by some child. By contrast, $\Box(s)$ should not fail just because s is not applicable to the children it encounters for pure typing reasons.

First, we will introduce a type to model generic type-preserving strategies. Then, the problem of *mediation* between sort-specific and generic strategies is considered. Finally, we point out some convenient qualities of the resulting type system.

4.1 The type of all type-preserving strategies

We extend our syntax for strategy types π, namely we add one case for generic types γ. In this article, we only consider one particular generic type, namely TP

[8] We will later discuss the possible employment of parametric types.

representing the type of all *T*ype-*P*reserving strategies. In [10], we also consider type-changing strategies. Our grammar of types is extended as follows:

$$\pi ::= \cdots \mid \gamma$$
$$\gamma ::= \mathsf{TP}$$

Example 5. All the strategies in Figure 3 are generic type-preserving strategies. Also, the argument strategy ν for all the definitions is of type TP. The same holds for the strategies *try* and *repeat* defined in Example 2. Thus, we assume the type $\mathsf{TP} \to \mathsf{TP}$ for all these strategy definitions.

In Figure 6, we extend the typing judgements. We use a subtype-equality-relationship to characterise generic types. Rule [subtypeeq.2] axiomatises TP. The rule says that $\tau \to \tau \leq_\Gamma \mathsf{TP}$ for all well-formed τ. This directly encodes the idea of type-preserving strategies. A strategy of type TP can be applied to a term of any sort. A (proper) subtype of TP is a sort-specific strategy type.[9] ID and FAIL are defined to be (generic) type-preserving strategies in rules [id] and [fail]. $\Box(s)$ and $\Diamond(s)$ and their argument strategy s are also defined to be type-preserving in rules [all] and [one].

Strategy types $\boxed{\Gamma \vdash \pi}$

$\Gamma \vdash \mathsf{TP}$ [pi.2]

Subtyping $\boxed{\pi \leq_\Gamma \pi'}$

$$\frac{\Gamma \vdash \pi}{\pi \leq_\Gamma \pi} \quad \text{[subtypeeq.1]}$$

$$\frac{\Gamma \vdash \tau}{\tau \to \tau \leq_\Gamma \mathsf{TP}} \quad \text{[subtypeeq.2]}$$

Composable types $\boxed{\pi_1 \circ_\Gamma \pi_2 \leadsto \pi}$

$\mathsf{TP} \circ_\Gamma \mathsf{TP} \leadsto \mathsf{TP}$ [comp.2]

Application $\boxed{\Gamma \vdash \langle s \rangle t : \tau}$

$$\frac{\begin{array}{l}\Gamma \vdash s : \pi \\ \wedge\ \Gamma \vdash t : \tau \\ \wedge\ \tau \to \tau \leq_\Gamma \pi\end{array}}{\Gamma \vdash \langle s \rangle t : \tau} \quad \text{[apply]}$$

Strategies $\boxed{\Gamma \vdash s : \pi}$

$\Gamma \vdash \mathrm{ID} : \mathsf{TP}$ [id]

$\Gamma \vdash \mathrm{FAIL} : \mathsf{TP}$ [fail]

$$\frac{\Gamma \vdash s : \mathsf{TP}}{\Gamma \vdash \Box(s) : \mathsf{TP}} \quad \text{[all]}$$

$$\frac{\Gamma \vdash s : \mathsf{TP}}{\Gamma \vdash \Diamond(s) : \mathsf{TP}} \quad \text{[one]}$$

Fig. 6. Generic type-preserving strategies

The type system strictly separates sort-specific strategies (such as rewrite rules), and generic strategies (such as applications of $\Box(\cdot)$). As for the moment being, we cannot turn sort-specific strategies into generic ones, neither the other way around. We will provide a corresponding refinement of the strategy calculus soon. The well-typedness rule for strategy application (cf. [apply]) certainly clarifies how a generic strategy can be applied to a term of a specific sort.

[9] π is a proper subtype of π' (notation: $\pi <_\Gamma \pi'$) if $\pi \leq_\Gamma \pi' \wedge \pi \neq \pi'$.

4.2 Mediation between specificity and genericity

Now that we have typed generic traversal operators, the question is how we *inhabit* TP. So far, we only have two trivial constants of type TP, namely ID and FAIL. We would like to construct generic strategies from sort-specific rewrite rules. It turns out that we lack a construct to perform inhabitation in a typeful manner. We also need to relax the typing rules for some existing combinators in order to make it easy to apply generic strategies in a specific context.

One approach to the inhabitation of TP is to use a generic default (initially ID and FAIL, but not just these) if the specific strategy is not applicable (for typing reasons). We might attempt to turn, for example, a rewrite rule ℓ into a generic strategy using the forms $\ell \nleftarrow\!\!+ \ \mathrm{ID}$ or $\ell \nleftarrow\!\!+ \ \mathrm{FAIL}$. This is not a good idea since the operator $\cdot \nleftarrow\!\!+ \cdot$ is concerned with choice controlled by success and failure. What we are looking for in the context of qualification of specific strategies to become generic, is a different form of choice. We need a *type-sensitive form of choice* where the specific strategy is chosen if the actual term is covered by its domain. Otherwise the generic strategy (serving as a kind of default) should be chosen. We introduce a corresponding operator:

$$s ::= \cdots \mid s \leftarrow\!\!\oplus\, s$$

The left argument is the sort-specific strategy whereas the right argument is the generic default. The static and dynamic semantics of the operator are defined in Figure 7.

Example 6. We recall the solution to the first problem from the introduction as given in Example 3. The original (Stratego-like) solution is not typeable because a sort-specific strategy, namely $nat; N \rightarrow succ(N)$, is passed to *stoptd* which expects a generic argument. We recover typeability by the following redefinition of $traverse_{(I)}$:

$$\begin{aligned} traverse_{(I)} &= stoptd((nat; N \rightarrow succ(N)) \leftarrow\!\!\oplus\, \mathrm{FAIL}) \\ &= stoptd((N \rightarrow succ(N)) \leftarrow\!\!\oplus\, \mathrm{FAIL}) \end{aligned}$$

This solution illustrates that we can qualify specific rewrite rules to become generic by $\cdot \leftarrow\!\!\oplus \cdot$. As an aside, the simplification to eliminate the test for naturals is enabled by the typed model. The type of the rewrite rule sufficiently restricts its applicability. The other strategy $traverse_{(II)}$ from Example 3 can be made fit in the same manner.

The operator $\cdot \leftarrow\!\!\oplus \cdot$ serves for asymmetric left-biased choice of strategies based on the type of the term in the application. As the deduction rules detail, if $s_1 \leftarrow\!\!\oplus\, s_2$ is applied to a term t of type τ, and the type of s_1 coincides with $\tau \rightarrow \tau$, then s_1 is chosen (cf. [lplus$^+$.1]). Otherwise, we resort to the generic strategy s_2 (cf. [lplus$^+$.2]). Thus, $\cdot \leftarrow\!\!\oplus \cdot$ is different from $\cdot \nleftarrow\!\!+ \cdot$ in the sense that $\cdot \leftarrow\!\!\oplus \cdot$ is left-biased w.r.t. type-sensitivity, whereas $\cdot \nleftarrow\!\!+ \cdot$ is left-biased w.r.t. success and failure. This separation makes it explicit where we want to become generic. There is no hidden way how specific ingredients can get generic accidentally. Without separating the two kinds of choice, strategies get too easily (say accidentally) generic and typeable.

Well-typedness $\boxed{\Gamma \vdash s : \pi}$

$$\frac{\Gamma \vdash s_1 : \tau \to \tau \quad \wedge\ \Gamma \vdash s_2 : \gamma \quad \wedge\ \tau \to \tau \leq_\Gamma \gamma}{\Gamma \vdash s_1 \leftarrow\!\!\!\!\oplus\, s_2 : \gamma} \quad [\mathsf{lplus}]$$

Semantics $\boxed{\Gamma \vdash \langle s \rangle t \rightsquigarrow r}$

Positive rules

$$\frac{\Gamma \vdash s_1 : \tau \to \tau \quad \wedge\ \Gamma \vdash t : \tau \quad \wedge\ \Gamma \vdash \langle s_1 \rangle t \rightsquigarrow t'}{\Gamma \vdash \langle s_1 \leftarrow\!\!\!\!\oplus\, s_2 \rangle t \rightsquigarrow t'} \quad [\mathsf{lplus}^+.1]$$

$$\frac{\Gamma \vdash s_1 : \tau' \to \tau' \quad \wedge\ \Gamma \vdash t : \tau \quad \wedge\ \tau \neq \tau' \quad \wedge\ \Gamma \vdash \langle s_2 \rangle t \rightsquigarrow t'}{\Gamma \vdash \langle s_1 \leftarrow\!\!\!\!\oplus\, s_2 \rangle t \rightsquigarrow t'} \quad [\mathsf{lplus}^+.2]$$

Negative rules

$$\frac{\Gamma \vdash s_1 : \tau \to \tau \quad \wedge\ \Gamma \vdash t : \tau \quad \wedge\ \Gamma \vdash \langle s_1 \rangle t \rightsquigarrow \uparrow}{\Gamma \vdash \langle s_1 \leftarrow\!\!\!\!\oplus\, s_2 \rangle t \rightsquigarrow \uparrow} \quad [\mathsf{lplus}^-.1]$$

$$\frac{\Gamma \vdash s_1 : \tau' \to \tau' \quad \wedge\ \Gamma \vdash t : \tau \quad \wedge\ \tau \neq \tau' \quad \wedge\ \Gamma \vdash \langle s_2 \rangle t \rightsquigarrow \uparrow}{\Gamma \vdash \langle s_1 \leftarrow\!\!\!\!\oplus\, s_2 \rangle t \rightsquigarrow \uparrow} \quad [\mathsf{lplus}^-.2]$$

Fig. 7. Turning specific strategies into generic ones

Note that the semantics judgement needs to be able to determine the type of the specific strategy, and the type of the given term. For that reason, we add the typing context Γ to the judgement for strategy application. The negative semantics rules for $\cdot \leftarrow\!\!\!\!\oplus \cdot$ are also shown in Figure 7 since they are instructive. One can clearly see how the type of the term is used to determine the applicability of the left operand s_1 in $s_1 \leftarrow\!\!\!\!\oplus\, s_2$ during strategy application. This type-sensitivity might be regarded as a paradigm shift. We postpone discussing a way to eliminate the premises to determine the types of the ingredients of $\langle s_1 \leftarrow\!\!\!\!\oplus\, s_2 \rangle t$ in the semantics rules.

So far, we only considered one direction of mediation. We should also refine our type system so that generic strategies can be easily applied in sort-specific contexts. This requirement amounts to a simple relaxation of the typing for argument strategies of the strategy combinators. Basically, we want to state that the type of a compound strategy like $s_1; s_2$ and $s_1 + s_2$ is dictated by a sort-specific argument (if any). As for congruence strategies, we simply employ subtyping to cope with the case of a generic strategy applied to some child. There are no further specific contexts for the available combinators. A corresponding refinement of our type system is defined in Figure 8.

As for $\cdot ; \cdot$, we relax the definition of composable types to cover composition of a specific and a generic type in both possible orders (cf. rules [comp.3] and [comp.4]). As for $\cdot + \cdot$ (and hence for $\cdot \leftarrow\!\!\!+ \cdot$ as well), we do not insist on equal argument types anymore, but we employ an auxiliary judgement for intersection (cf. rules [glb.1], [glb.2], and [choice]). Finally, we relax the argument types for congruence strategies via the $\leq_\Gamma$ relation. The refinement in a sense, *automates* the cast from TP to sort-specific strategy types. This is not considered as problem (as opposed to hidden ways for sort-specific strategy to become generic) since an accidentally sort-specific strategy would be easily realised by the programmer

Composable types $\boxed{\pi_1 \circ_\Gamma \pi_2 \rightsquigarrow \pi}$

$$\frac{\Gamma \vdash \tau}{\tau \to \tau \circ_\Gamma \mathsf{TP} \rightsquigarrow \tau \to \tau}\ [\mathsf{comp.3}]$$

$$\frac{\Gamma \vdash \tau}{\mathsf{TP} \circ_\Gamma \tau \to \tau \rightsquigarrow \tau \to \tau}\ [\mathsf{comp.4}]$$

Intersection $\boxed{\pi_1 \sqcap_\Gamma \pi_2 \rightsquigarrow \pi}$

$$\frac{\begin{array}{l} \Gamma \vdash \pi_1 \\ \wedge\ \Gamma \vdash \pi_2 \\ \wedge\ \pi_1 \leq_\Gamma \pi_2 \end{array}}{\pi_1 \sqcap_\Gamma \pi_2 \rightsquigarrow \pi_1}\ [\mathsf{glb.1}]$$

$$\frac{\begin{array}{l} \Gamma \vdash \pi_1 \\ \wedge\ \Gamma \vdash \pi_2 \\ \wedge\ \pi_2 \leq_\Gamma \pi_1 \end{array}}{\pi_1 \sqcap_\Gamma \pi_2 \rightsquigarrow \pi_2}\ [\mathsf{glb.2}]$$

Well-typedness $\boxed{\Gamma \vdash s : \pi}$

$$\frac{\begin{array}{l} \Gamma \vdash s_1 : \pi_1 \\ \wedge\ \Gamma \vdash s_2 : \pi_2 \\ \wedge\ \pi_1 \sqcap_\Gamma \pi_2 \rightsquigarrow \pi \end{array}}{\Gamma \vdash s_1 + s_2 : \pi}\ [\mathsf{choice}]$$

$$\frac{\begin{array}{l} f : \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \Gamma \\ \wedge\ \Gamma \vdash s_1 : \pi_1\ \wedge\ \sigma_1 \to \sigma_1 \leq_\Gamma \pi_1 \\ \wedge\ \cdots \\ \wedge\ \Gamma \vdash s_n : \pi_n\ \wedge\ \sigma_n \to \sigma_n \leq_\Gamma \pi_n \end{array}}{\Gamma \vdash f(s_1, \ldots, s_n) : \sigma \to \sigma}\ [\mathsf{congr}]$$

Fig. 8. Application of generic strategies in a potentially specific context

when he or she attempts to apply the strategy in a generic context, that is, the type system will catch such accidents.

4.3 Properties of the calculus

Our ultimate typed strategy calculus is obtained by starting from simply-typed strategies (cf. Figure 5), and updating it with the $\cdot \leftarrow\!\!\!\oplus \cdot$ operator (cf. Figure 7) and the relaxations from above (cf. Figure 8). We call this calculus S'_{TP} (to point out its close relation to system S). The following theorem summarises desirable properties of S'_{TP}.

Theorem 1. *The type system of S'_{TP} obeys the following:*

1. *Strategies satisfy unicity of typing.*
2. *Strategy application satisfies unicity of typing.*
3. *The semantics for strategy application satisfies subject reduction.*

Proof.
1. By induction on the strategy in the well-typedness judgement: *Base cases*: The type of a rewrite rule (cf. rule [rule]) is uniquely defined by the involved terms. Unicity of typing holds, of course, for the type judgement for terms. The types for the constant strategies ID and FAIL are uniquely defined (cf. rules [id] and [fail]). *Induction step*: The type of all argument strategies of all combinators are unique by the induction hypothesis. As for the binary operator $\cdot;\cdot$, the result type is defined as the type composed from the argument types. As for $\cdot + \cdot$, and $\cdot \leftarrow\!\!\!+ \cdot$, the result type is defined as the intersection of the argument types. The corresponding judgements $\cdot \circ_\Gamma \cdot \rightsquigarrow \cdot$ and $\cdot \sqcap_\Gamma \cdot \rightsquigarrow \cdot$ obviously encode functions. The type of a congruence strategy (cf. rule [congr]) is dictated by the well-formed context which is used to lookup the sort of the

function symbol at hand. The types of $\Box(s)$ and $\Diamond(s)$ are uniquely defined as TP. The type of $s_1 \mathrel{\leftarrow\!\!\!\oplus} s_2$ is the type of s_2.

2. Follows immediately from unicity of typing for terms, and the fact that the type of the resulting term (say the type of strategy application) is the type of the input term (cf. [apply]).

3. By induction on the strategy in the semantics judgement. Note that we only deal with type-preserving strategies which simplifies matters because we basically have to show that strategy application preserves term types as well. *Base cases*: The treatment of rewrite rules is standard. We can ignore FAIL since we are only interested in proper term reducts. Subject reduction holds for ID since the resulting term coincides with the input term, and hence, type-preservation holds. *Induction step*: Subject reduction holds for $s_1 + s_2$ and $s_1 \mathrel{\leftarrow\!\!\!+} s_2$ since reduction simply resorts in all cases to one of the arguments which satisfy subject reduction by the induction hypothesis. As for $s_1; s_2$, subject reduction follows from the induction hypothesis, and transitivity of subject reduction. Note that the types of the argument strategies for the aforementioned combinators are at least as general as the type of the compound strategy. Thus, redirection to s_1 or s_2, and the sequential composition of s_1 and s_2 are type-safe. Subject reduction holds for congruence strategies and for the generic traversal combinators because the outermost function symbol is preserved and the types of all children are preserved by the induction hypothesis. The interesting case is $s_1 \mathrel{\leftarrow\!\!\!\oplus} s_2$. Rule [lplus$^+$.2] (where we resort to the generic default s_2 dictating the type of the compound strategy) can be covered using the same arguments used for $s_1 + s_2$. As for rule [lplus$^+$1], we basically resort to s_1 although its type is more specific than the type of the compound strategy. Still, subject reduction holds because reduction according to rule [lplus$^+$.1] is guarded by the typing premises to ensure that s_1 is applicable.

4.4 Beyond TP

A note on generality is maybe in place. The presented type system (especially Figure 7 and Figure 8) is really geared towards generic type-preserving strategies, and we assume that we have only two levels: sort-specific and generic strategies. Type-changing strategies (especially rewrite rules) are also sensible since strategies can control that type changes are performed consistently. This is different in conservative rewriting where type-changing rewrite rules are incompatible with the idea of a fixed strategy (like innermost). Especially, if we talk about generic strategies, one important subclass of type-changing strategies follows the scheme of type-unification [12], that is, the result type of the generic strategy is of a fixed type (such as Boolean values or lists of naturals for the problems (III) and (IV) in the introduction) regardless of the type of the input term. In addition to sort-specific and generic strategies, one might also consider strategies with a finite set of term types covered by them. These could be called sum-typed strategies. In this context, we might think of a more general form of $s_1 \mathrel{\leftarrow\!\!\!\oplus} s_2$ where the types of s_1 and s_2 are solely constrained by $<_\Gamma$. One can also think of a symmetric combinator $\cdot \oplus \cdot$ to perform a kind of disjoint union. We refer to [10] for a thorough treatment of all the aforementioned classes of strategies.

5 Implementation

The calculus S'_{TP} (and generalisations of it) can be implemented without major problems. We have done simple experiments based on Prolog which allowed us to execute the judgements for typing and reduction almost as is.

There is one concern which needs to be addressed in order to obtain a practical implementation, namely the separation of typing and reduction. The reduction rules for $\langle s_1 \leftarrow\!\oplus\, s_2\rangle t$ involve premises to determine the type of the term t, and the type of the strategy s_1 (cf. Figure 7). Conceptually, this is fine because we point out in the most direct way that $\cdot \leftarrow\!\oplus \cdot$ is about type-sensitive choice. Still this type-dependent reduction might be regarded as a debatable paradigm shift, and, in particular, as an obstacle for efficient implementation of S'_{TP}. Fortunately, there is a simple way to eliminate the typing premises. The elimination is considered in this section in some detail.

We conclude the implementation section with an indication why TP can be integrated into existing rewriting environments in a rather simple manner.

5.1 Static elaboration

To eliminate the typing premise determining the type of the sort-specific strategy s_1 in $s_1 \leftarrow\!\oplus\, s_2$, the following approach is appropriate. We statically perform an elaboration step which follows the very scheme of the type judgement for strategies, but *transforms* strategies. We want to turn strategy expressions of the form $s_1 \leftarrow\!\oplus\, s_2$ into $s_1 \;{}_{\tau}\!\!\leftarrow\!\oplus\, s_2$ where we propagate the term type covered by s_1 via the subscript τ. During strategy application, the annotation can be used for a "dispatch" to decide what strategy needs to be applied, either s_1 or s_2.[10] The updated rules for strategy application will be shown in a second. The relevant elimination rule takes the following form:

$$\frac{\begin{array}{l} \quad\; \Gamma \vdash s_1 : \tau \to \tau \\ \wedge\; \Gamma \vdash s_2 : \gamma \\ \wedge\; \tau \to \tau \leq_{\Gamma} \gamma \end{array}}{\Gamma \vdash s_1 \leftarrow\!\oplus\, s_2 \leadsto s_1 \;{}_{\tau}\!\!\leftarrow\!\oplus\, s_2} \qquad [\mathsf{lplus}]$$

5.2 Tagged terms

We also do not want to determine the type of the term at hand at rewriting time (as it is the case in the original rules for $\cdot \leftarrow\!\oplus \cdot$). In some way or another, we should tag terms with types. We show a replacement for the positive rules for $\cdot \leftarrow\!\oplus \cdot$. The replacement relies on the elaboration described above, and on tagged terms in strategy application.

$$\frac{\Gamma \vdash \langle s_1\rangle t : \tau \leadsto t' : \tau}{\Gamma \vdash \langle s_1 \;{}_{\tau}\!\!\leftarrow\!\oplus\, s_2\rangle t : \tau \leadsto t' : \tau} \qquad [\mathsf{lplus}^{+}.1]$$

[10] We consciously use the term "dispatch" (very much used in the object-oriented setting for virtual method dispatch according to the class of the object at hand).

$$\frac{\tau \neq \tau' \quad \wedge \; \Gamma \vdash \langle s_2 \rangle t : \tau \rightsquigarrow t' : \tau}{\Gamma \vdash \langle s_1 \,{}_{\tau'}\!\!\leftarrow\!\!\oplus\, s_2 \rangle t : \tau \rightsquigarrow t' : \tau} \quad [\mathsf{lplus}^{+}.2]$$

As we can see, the static typing context is not needed anymore. Instead the reduction of $\langle s_1 \,{}_{\tau'}\!\!\leftarrow\!\!\oplus\, s_2 \rangle t : \tau$ relies on the annotations τ and τ'. To be precise, the context is definitely not needed for the semantics of $\cdot \,{}_{\tau}\!\!\leftarrow\!\!\oplus\, \cdot$, but the combinators $\Box(\cdot)$ and $\Diamond(\cdot)$ deserve an additional comment. As these combinators descend into terms, the types of the subterms of a term also need to be known. Some options to accomplish this knowledge are the following:

1. We assume that all subterms are tagged by their types at any level of nesting.
2. We can determine the type of *any* (well-typed) term via its outermost function symbol. The declarations of function symbols (as being part of the typing context parameter Γ) are sufficient for that purpose.
3. We use specialised (signature-aware) variants of $\Box(\cdot)$ and $\Diamond(\cdot)$, that is, we had to instantiate the scheme for $\Box(\cdot)$ and $\Diamond(\cdot)$ for all function symbols.

All these formulations lead—more or less directly—to an efficient implementation. Option (1.) has an impact on the representation of terms. Option (2.) relies on an extra lookup per application of $\cdot \,{}_{\tau}\!\!\leftarrow\!\!\oplus\, \cdot$. Option (3.) requires program generation.

5.3 Integration with rewriting environments

The calculus S'_{TP} fits very well into the setting of a many-sorted strategic rewriting framework as ELAN. Generic type-preserving traversals are particularly simple to implement in the characterised form in such a setting. As for the frontend, we only need to add a dedicated sort TP, and we need to overload $\cdot \leftarrow\!\!\oplus \cdot$ for all sorts in the signature at hand. The kind of elaboration we discussed above can then even be omitted since parsing will just resolve the overloading and convey the actual instance to the abstract syntax tree. As for the rewriting engine, we only need to be able to inspect the type of the term, and the instance of the overloaded $\cdot \leftarrow\!\!\oplus \cdot$. These requirements do not pose any challenges for the parser and the rewriting machinery underlying rewriting frameworks. In fact, the architecture of ELAN, for example, satisfies these requirements immediately. Thus, we consider S'_{TP} together with the suggested implementational scheme as a lightweight proposal for typeful generic (type-preserving) traversals.

6 Concluding remarks

An alternative model A natural question about generic strategies is whether generic strategies are in a sense polymorphic. The strategy Id, for example, certainly is polymorphic in a reasonable sense. However, in general, a generic strategy is not truly polymorphic since it can inspect the terms at hand in generic manner, whereas a polymorphic function cannot assume *anything* about the polymorphic positions in its type. By saying that generic strategies are not polymorphic, we do not imply that *parametric types* are inappropriate for representing generic strategy types at the language level. Let us consider different candidate types for the strategy definition *try* from Example 2:[11]

$$
\begin{array}{ll}
\mathit{try} : \mathsf{TP} \to \mathsf{TP} & (\text{Type in } S'_{\mathsf{TP}}) \\
\mathit{try} : X \to X & (X \text{ is a strategy type parameter}) \\
\mathit{try} : \langle X \to X \rangle \to \langle X \to X \rangle & (X \text{ is a term type parameter})
\end{array}
$$

The difference between the second and the third formulation is a bit subtle. The type $\mathit{try} : \langle X \to X \rangle \to \langle X \to X \rangle$ explicitly requires a "generic" strategy argument, and then also returns a "generic" strategy. The same is true for the type of *try* in S'_{TP}. By contrast, the type $\mathit{try} : X \to X$ is generic at the strategy type level rather than the term type level. With this interpretation in mind, we might apply the operator *try* to a sort-specific strategy. This is indeed a very sensible application of *try* ruled out by the other two types. Note that this kind of type does not make sense for $\Box(\cdot)$ since the argument strategy must be generic. In the present paper, we favoured the generic type TP instead of type parameters mainly for reasons of simplicity. The type-parametric approach triggers some questions and complications:

- How do we separate genericity as in traversals from truly parametric polymorphism as in parametric datatypes? Is a separation desirable from a semantic perspective?
- It seems that we really need parameters for both strategy types and term types. If we only consider parameters for term types, than we cannot write strategy definitions which apply to both sort-specific and generic strategies.
- In $\mathit{try} : X \to X$, we assume that X is a placeholder for a type-preserving strategy. If we started to consider type-changing strategies, we need to qualify strategy type parameters to point out the relevant class of strategies.

Related work In the present article, we developed a type system for term rewriting strategies. The contribution of the article is that generic traversals are covered, namely generic type-preserving ones. We have designed another model for typed strategies in the context of functional programming [11]. The latter approach originated in turn from our research on (dynamically updatable) generalised monadic folds for systems of datatypes [12].

There is no previous work on statically typed generic strategies in the narrow context of rewriting. In [5], dynamic types [1] are employed to cope with some generic (traversal) strategies in ELAN. A universal datatype *any* is used to represent terms of "any" sort. For that purpose, a parameterised module `any[X]` is offered which can be imported for any sort which is subject to generic programming via the *any* datatype. The module offers an injection and a projection to mediate between *any* and the terms of sort `X`. As for generic traversals, there are *explode* and *implode* functions to destruct and construct terms (say to access the children of a term). The actual implementation employs a (transparent)

[11] We use $\langle \ldots \rangle$ to surround strategy types.

pre-processing approach to obtain a sort-specific instantiation of the interface of `any[X]` according to `X`, and program schemes for *explode* and *implode*. To conclude, specifications relying on *any* are type-safe. However, if during rewriting the manipulated terms of sort *any* do not represent terms of the "intended sorts", at some point projection and term implosion is going to fail. This problem is irrelevant for S'_{TP} since types are statically enforced, and there is no universal (and hence imprecise) sort like *any*.

The presented concepts were inspired by the generic programming paradigm [2], and especially by polytypic programming [8]. A polytypic function is defined by induction on its argument type (with cases for sums, products, and others). Generic traversals in S'_{TP} are performed in a somewhat similar manner. Generic traversals are defined in terms of $\Box(\cdot)$ and $\Diamond(\cdot)$ (corresponding to the polytypic cases for sums and products), usually by recursive strategy definitions (roughly corresponding to induction). While polytypic programming is placed in the context of higher-order functional programming, our approach contributes to the field of strategic, (not necessarily) first-order rewriting. An idea which is central to our approach is that we want to have simple but flexible means to mix genericity and specificity in the context of many-sorted signatures (e.g., language syntaxes), while the bulk of polytypic programming focuses on statically defined, polytypic values for (parameterised) datatypes.

Perspective The many-sorted types in strategic rewriting provide one important dimension of static information. Another dimension entirely ignored in the present article is failure analysis or determinism analysis. Generic traversal strategies are presumably accessible for such an analysis. It should be possible to capture this analysis in a type system. We think that this kind of type information would be extremely beneficial for actual strategic programming. Failure is a highly overloaded concept. It is used intentionally to force local backtracking in a choice. In many applications, it also triggers backtracking to go back to a remote choice point. It might also be used to force a kind of strict error handling subject to global failure triggered somewhere deep in a program. Finally, unintended applicability problems of strategies are also just manifested as failure. Consequently, debugging strategic programs is sometimes a pain.

The ρ-calculus provides an ambitious rewriting calculus. Part of the ρ-cube is typed (but not generic traversals expressiveness). One of the challenging properties of the ρ-calculus is that rewrite rules are higher-order. The developed typed calculus S'_{TP} could be rephrased to cover some first-order fragment of the calculus (however with generic traversals!). The question is how richer fragments in the ρ-cube can be covered, e.g., specifications with higher-orderness, full polymorphism, and dependent types (in the presence of generic traversals). It is not clear what the interaction of these concepts and generic strategies yields.

References

1. M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 ACM Workshop on ML and its Applica-*

tions, pages 92–103, San Francisco, U.S.A., June 1992. Association for Computing Machinery.
2. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *LNCS*, pages 28–115, Braga, Portugal, Sept. 1999. Springer-Verlag.
3. P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In Meseguer [14].
4. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, Sept. 1998. Elsevier Science.
5. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.
6. H. Cirstea and C. Kirchner. Introduction to the rewriting calculus. Rapport de recherche 3818, INRIA, Dec. 1999.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. The Maude System. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag LNCS 1631. System Description.
8. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
9. J. Jeuring, editor. *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.
10. R. Lämmel. Typed Generic Traversal. Technical report, CWI, 2001. To appear.
11. R. Lämmel and J. Visser. Type-safe Functional Strategies. In *Draft proc. of SFP'00, St Andrews*, July 2000.
12. R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [9], pages 46–59. available at `http://www.cwi.nl/~ralf/`.
13. S. E. M. Clavel, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [14].
14. J. Meseguer, editor. *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Sept. 1996.
15. L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149 (or 119–150??), Aug. 1983.
16. E. Visser. Language independent traversals for program transformation. In Jeuring [9], pages 86–104.
17. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98), Baltimore, Maryland. ACM SIGPLAN*, pages 13–26, Sept. 1998.
18. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15, September 1998. In C. Kirchner and H. Kirchner, editors, Proceedings of the Second International Workshop on Rewriting Logic and its Applications (WRLA'98), Pont-à-Mousson, France.