Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Typed generic traversals in S'_γ

R. Lämmel

REPORT SEN-R0122 AUGUST 2001

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Typed Generic Traversals in S'_γ

Ralf Lämmel
Email: Ralf.Laemmel@cwi.nl
WWW: http://www.cwi.nl/~ralf/
Phone: +31 20 592 4090 Fax: +31 20 592 4199

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
A typed model of strategic rewriting is developed. An innovation is that generic traversals are covered. To this end, we define a rewriting calculus S'_γ. The calculus offers a few strategy combinators for generic traversals. There is, for example, a combinator to apply a strategy to all immediate subterms of a given term. This idiom is relevant for generic type-preserving traversals. We also go beyond type-preservation which corresponds to another innovation. There is, for example, a combinator to reduce all the immediate subterms of a term. S'_γ employs a many-sorted type system extended by distinguished signature-independent (say generic) strategy types γ. To inhabit generic types, we need to add a fundamental combinator to lift a many-sorted strategy s to a generic type γ. The reduction semantics for this kind of lifting states that s is only applied if the type of the term at hand fits, otherwise the strategy fails. This approach dictates that the semantics of strategy application must be type-dependent to a certain extent. Typed strategic rewriting with generic traversals is a simple but expressive model of generic programming. It has applications in program transformation and program analysis.

1998 ACM Computing Classification System: D.1.1, D.1.2, D.3.1, D.3.3, F.4.2, I.1.3, I.2.2
Keywords and Phrases: Term rewriting, Generic programming, Term rewriting strategies, Traversal schemes, Type systems, Program transformation

1. Preface

Strategic programming Term rewriting strategies are of prime importance for the implementation of term rewriting systems. In the present article, we focus on another application of strategies, namely on their utility for programming. Strategies can be used to describe evaluation and normalisation strategies, e.g., to explicitly control rewriting for some rewrite rules which are not confluent or terminating when considered as a standard rewrite system. Moreover, strategies can be used to describe generic traversals (or traversal schemes). In fact, the typeful treatment of traversals is the primary subject of the present article. 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 encode the actual traversal for the signature at hand. One usually requires one rewrite rule per term constructor, per traversal. This problem has been identified in [BSV00, LVK00, BKV01, Vis01] (from different points of view). In a framework, where traversal strategies are supported, one can focus on the patterns which have to be specifically handled for the problem at hand. All the other patterns can be covered once and for all by the generic part of a suitable strategy.

Application potential Language concepts for generic traversals support an important dimension of generic programming which is useful, for example, for the implementation of program transformations and program analyses. Such functionality is usually invariant w.r.t. many patterns in the traversed syntax. In [Vis00], (untyped) traversal strategies are used to approach to a library for language definition and implementation: Algorithms for free variable collection, substitution, unification

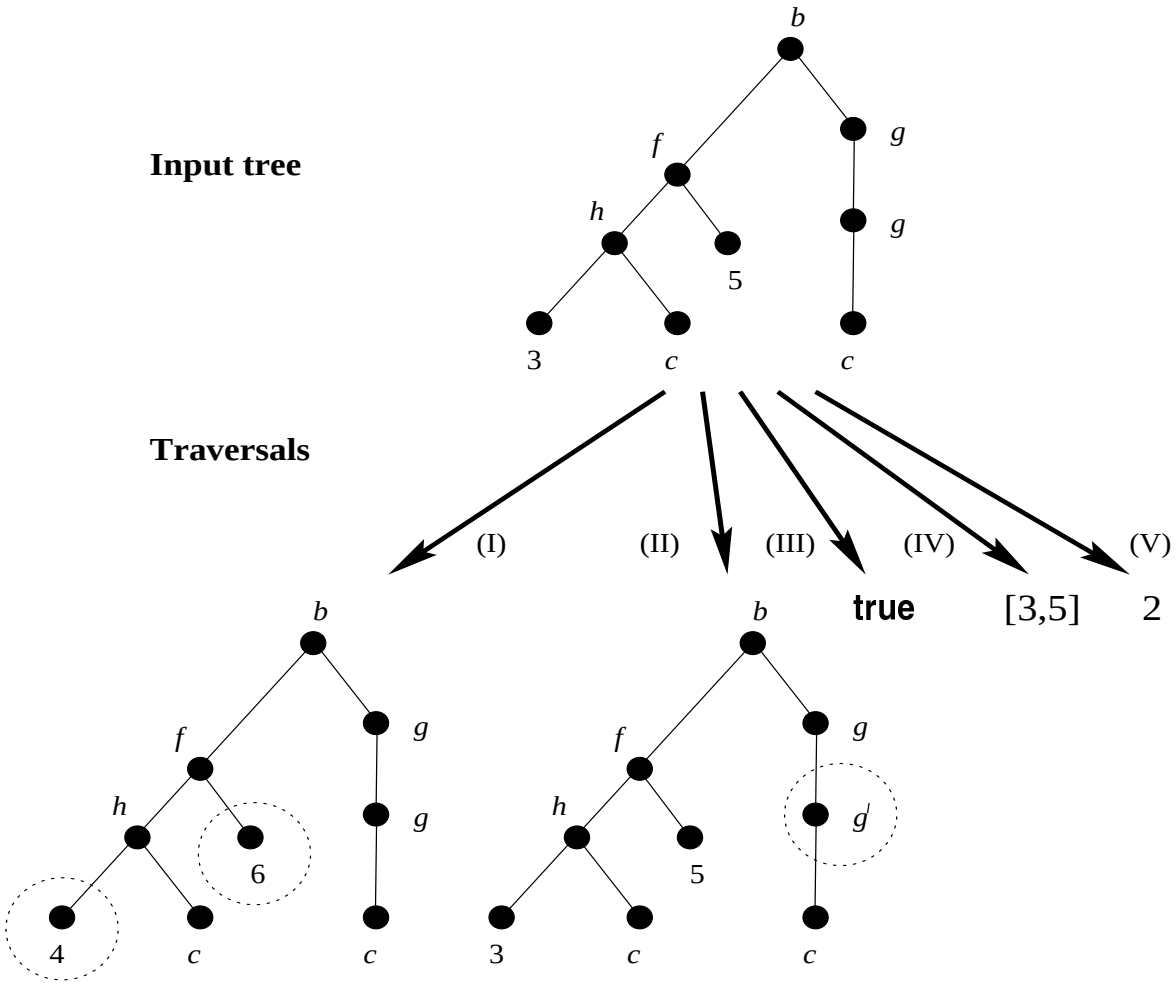

Figure 1: Illustration of generic traversals

and others are defined in a generic, that is, language-independent manner by suitably parameterised traversals. In [LV01], typed traversal strategies are employed for the specification of refactorings for object-oriented programs [Opd92, Fow99]. We also refer to [BKV01] for further applications.

S'_γ and relatives In the present article, the rewriting calculus S'_γ is developed. The calculus corresponds to a simple but expressive language for generic programming. The design of S'_γ was influenced by existing rewriting frameworks with support for 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 [MCLM96, CDE+99] and ELAN [BKK+98, BKKR01]. The ρ-calculus [CK99] provides an abstract model for rewriting including the definition of strategies. The programming language Stratego [VBT98] based on system S [VB98] is entirely devoted to strategic programming. The idea of rewriting strategies goes back to Paulson's work on higher-order implementation of rewriting strategies [Pau83] in the context of the implementation of tactics and tacticals for theorem proving. The original contribution of S'_γ is the *typeful* approach to generic *traversal* strategies in a *many-sorted* setting.

Examples of generic traversals In Figure 1, five examples (I)–(V) 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 of any sort. 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. In fact, in Figure 1, only one g is turned into a g', namely the deeper one. The strategy to locate the desired node in the term is completely generic. While (I)–(II) require type-preserving traversals, (III)–(V) require type-changing ones. In (III), we test some property of the term, namely if naturals occur at all, i.e., we map a term to a Boolean. In (IV), we collect all the naturals in the term using a left-to-right traversal. Finally, in (V), we count all the occurrences of the function symbol g.

Value of typing The typed rewriting calculus S'_γ covers both type-preserving and type-changing strategies. We propose two distinguished generic strategy types, namely the type TP denoting all type-preserving strategies, and the type $\mathsf{TU}(\tau)$ denoting all type-unifying strategies with the unified result type τ. Strategies are statically typed in S'_γ. The type system should obviously prevent us from constructing ill-typed terms. Generic traversals typically employ (many-sorted) rewrite rules. One has to ensure that the rewrite rules are applied in a type-safe manner. Consider, for example, the rewrite rule $\textsc{Inc} = N \to succ(N)$ of type $Nat \to Nat$ for incrementing naturals in the context of example (I) above. The rule should only be applied to naturals during a traversal. On the other hand, the complete traversal must be of type TP, that is, it must be applicable to any term. This implies that the rule Inc must be lifted to TP at some point in the derivation of the traversal strategy for (I). In fact, S'_γ offers a corresponding combinator for so-called strategy extension which is type-safe. The type system should also prevent the programmer from combining specific and generic strategies in certain undesirable ways. Consider, for example, a biased choice where a rewrite rule ℓ is strictly preferred over the identity strategy, and only if ℓ fails, the identity strategy triggers. This choice is controlled by success and failure of ℓ. One could argue that this strategy is generic because the identity strategy is applicable to any term. Actually, we favour two other possible interpretations. Either we refuse this composition altogether (because we would insist on the types of the branches in a choice to be the same), or we favour the intersection of the argument types for the type of the compound strategy. The choice will not be generic in the latter 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 terms cannot be constructed, ill-formed generic strategies are likely to fail or to succeed in some unintended way. Such problems are hard to trace in strategic programming. The type system of S'_γ addresses this issue by an effective separation of generic and ordinary many-sorted strategy types, and by appropriate combinators mediating between these two kinds of strategy types.

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 (although some use of overloading and/or parametric types might be necessary). By contrast, generic traversal primitives (e.g., a combinator to apply a strategy s to all immediate subterms of a given term) are more challenging since standard many-sorted types are not applicable, and also other well-established concepts like parametric polymorphism are insufficient to model the required kind of genericity. Let us consider the type schemes underlying generic traversals:

$$\begin{array}{lcll} \mathsf{TP} & \equiv & \forall\alpha.\ \alpha \to \alpha & \text{(i.e., generic type-preserving traversals)} \\ \mathsf{TU}(\tau) & \equiv & \forall\alpha.\ \alpha \to \tau & \text{(i.e., generic type-unifying traversals with } \tau \text{ as unified result type)} \end{array}$$

In the schemes we point out that α is a universally quantified type variable. It is easy to see that these schemes are appropriate. Generic type-preserving traversals, for example, process terms of any sort (i.e., α), and they return terms of the same sort (i.e., α). If we read the type schemes in the sense

of parametric polymorphism, we can only inhabit them in a trivial way. The first scheme can only be inhabited by the identity function. The second scheme can only be inhabited by a constant function returning some fixed value of type τ. More generally, a model for generic traversals based on solely parametric polymorphism is out of reach since it would be necessarily in conflict with the notion of parametricity [Wad89, MR92, LMS93]. Parametricity does not holds since generic traversals usually employ many-sorted ingredients (say rewrite rules) to deal with some distinguished sorts in a specific manner. This form of genericity implies that the reduction semantics involves type dependencies although the type of strategies and strategy applications is statically known. Note that S'_γ does not enable us to inhabit somewhat arbitrary type schemes. In fact, the above two schemes are the only schemes which can be inhabited with the traversal combinators of S'_γ. This is also the reason that we do not favour type schemes to represent types of generic strategies in the first place but we rather employ the distinguished constants TP and $\mathsf{TU}(\tau)$.

Structure of the article In Section 2, we provide a gentle introduction to the subject of strategic programming, and to the rewriting calculus S'_γ defined in the subsequent sections. Examples of traversal strategies are given. Types are shown to be useful if not essential.[1] In Section 3, we start with a many-sorted fragment of S'_γ. In this phase, we cannot yet cover the traversal primitives. A minor contribution is here that we show how to cope with type-changing strategies. In Section 4, we provide a type system for generic strategies. The two aforementioned schemes of typing are covered, namely type preservation and type unification. A few supplementary issues to complement or to extend S'_γ are addressed in Section 5. Implementation issues and related work are discussed in Section 6 and Section 7. The article is concluded in Section 8.

Objective An important meta-goal of the present article is to develop a simple, typeful and self-contained model of programming with generic traversals. To this end, we basically resort to a many-sorted, first-order setting. We want to clearly identify the necessary machinery to accomplish generic traversals in such a simple setting. We also want to enable a simple implementation of the intriguing concept of typed generic traversals. In the course of the article, we show that our type system is sensible from a strategic programmer's point of view. We envision that the type system of S'_γ disciplines strategic programs (employing generic traversals) in a useful and not too restrictive manner.

Acknowledgement The work of the author was supported, in part, by *NWO*, in the project "*Generation of Program Transformation Systems*". Some core ideas spelled out in the article took shape during a visit of the author to the *Protheo* group at LORIA Nancy. I am particularly grateful for the interaction with my colleague Joost Visser—in Nancy and in general. Many thanks to Christophe Ringeissen who shared an intensive and insightful ELAN session with Joost and me. I want to thank David Basin, Johan Jeuring, Claude Kirchner, Paul Klint, Pierre-Étienne Moreau, Christophe Ringeissen, Ulf Schünemann, Jurgen Vinju, and Eelco Visser for discussions on the subject of the article. Finally, many thanks to the anonymous WRS 2001 workshop referees for their comments on an early fragment of this article (cf. [Läm01]).

2. Rationale

We set up a rewriting calculus S'_γ inspired by ELAN [BKK+98, BKKR01], the ρ-calculus [CK99], and system S [VB98]. The "S" in S'_γ points to system S which was most influential in the design of S'_γ. The "$'$" in S'_γ indicates that even the untyped part of S'_γ does not coincide with system S. The "γ" in S'_γ stands for the syntactical domain γ of generic types. Some basic knowledge of strategic rewriting is a helpful background for the present article (cf. [BKK96, VB98, CK99]). First, we give an overview on the primitive strategy combinators of S'_γ. Then, we explain how to define new combinators by means of strategy definitions. Afterwards, we pay special attention to generic traversals, that is, we

[1] We use the term "type" for types of variables, constant symbols, function symbols, terms, strategies, and combinators. We also use the term "sort" in the many-sorted sense if it is more suggestive.

explain the meaning of traversal primitives, and we illustrate their expressiveness. Ultimately, we show the impact and the utility of types in strategic programming. Finally, we relate S'_γ to some of the aforementioned strategic rewriting calculi. The upcoming explanations are informal. A formal definition of S'_γ is developed in Section 3, Section 4, and Section 5.

2.1 Primitive combinators

In an abstract sense, a (term rewriting) strategy is a (partial) mapping from a term to a term, or to a set of terms. In an extreme case, a strategy performs normalisation, that is, it maps a term to a normal form. We use s and t (possibly subscripted or primed) to range over strategy expressions, or terms, respectively. The application of a strategy s to a term t is denoted by $s \mathbin{@} t$. The result r of strategy application is called a reduct. It is either a term or "$\uparrow$" to denote failure. The primitive combinators of the rewriting calculus S'_γ are shown in Figure 2.[2]

s	$::=$	$t \to t$	(Rewrite rule)
	$\mid$	ϵ	(Identity)
	$\mid$	δ	(Failure)
	$\mid$	$s; s$	(Sequential composition)
	$\mid$	$s + s$	(Non-deterministic choice)
	$\mid$	$\neg\, s$	(Negation)
	$\mid$	c	(Congruence for constant symbol)
	$\mid$	$f(s, \ldots, s)$	(Congruence for function symbol)
	$\mid$	$\Box(s)$	(Apply strategy to all children)
	$\mid$	$\Diamond(s)$	(Apply strategy to one child)
	$\mid$	$\bigcirc^s(s)$	(Reduce all children)
	$\mid$	$\sharp(s)$	(Select one child)
	$\mid$	$\underline{\perp}$	(Build $\langle\rangle$)
	$\mid$	$s \parallel s$	(Spawn two strategies)
	$\mid$	$s \lhd \gamma$	(Extend strategy)

Figure 2: Primitives of S'_γ

Rewrite rules as strategies There is a form of strategy $t_l \to t_r$ for first-order, one-step rules to be applied at the root of the term. The idea is that if the term at hand and the left-hand side t_l do not match, then the rewrite rule (considered as a strategy) fails. Otherwise, the input is rewritten to the right-hand side t_r with the variables in t_r bound according to the initial match. We adopt some common restrictions for rewrite rules. The left-hand side t_l determines the bound variables. (Free) variables on the right-hand t_r side also occur in t_l. If substitution is performed, then we assume α-conversion.

Basic combinators Besides rule formation, there are standard primitives for the identity strategy (ϵ), the failure strategy (δ), sequential composition ($\cdot;\cdot$), non-deterministic choice ($\cdot + \cdot$), and negation ($\neg\, \cdot$). Non-deterministic choice means that there is no prescribed order in which the two argument strategies are considered. Negation (by failure) means that $\neg\, s$ fails if and only if s succeeds. In case of success of $\neg\, s$, the input term is simply preserved. Let us define some syntactic sugar for so-called left- and right-biased choice (as opposed to non-deterministic choice):

$$s_1 \mathbin{\not\leftarrow} s_2 \quad \equiv \quad s_1 + (\neg\, s_1; s_2)$$
$$s_1 \mathbin{\not\rightarrow} s_2 \quad \equiv \quad s_2 \mathbin{\not\leftarrow} s_1$$

[2] We use the term "combinator" for all kinds of operators on strategies, even for constant strategies like ϵ and δ.

That is, in $s_1 \Leftarrow s_2$, the left argument has higher priority than the right one. s_2 will only be applied if s_1 fails. For any kind of choice it holds that if both argument strategies fail, then choice also fails.

Congruences Recall that rewrite rules (when considered as strategies) are applied at the top of a term. From here on, we use the term "child" to denote an immediate subterm of a term, i.e., one of the t_i in a term of the form $f(t_1, \ldots, t_n)$. The congruence strategy $f(s_1, \ldots, s_n)$ provides a convenient way to apply strategies to the children of a term with f as outermost symbol. More precisely, the argument strategies $s_1, \ldots, s_n$ are applied to the parameters $t_1, \ldots, t_n$ of a term of the form $f(t_1, \ldots, t_n)$. If all these strategy applications deliver proper term reducts $t'_1, \ldots, t'_n$, then the term $f(t'_1, \ldots, t'_n)$ is constructed, i.e., the outermost function symbol is preserved. If any child cannot be processed successfully, or if the outermost function symbol of the input term is different from the f in the congruence, then the strategy fails. The congruence c for a constant c can be regarded as a test for the constant c.

Example 1 *We can already illustrate a bit of strategic rewriting with the combinators which we have explained so far. Let us consider the following problem. We want to flip the top-level subtrees in a binary tree with naturals at the leafs. We assume the following symbols to construct such trees:*

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

N and T (optionally subscripted or primed) are used as variables of sort Nat and Tree, respectively. We can specify the problem of flipping top-level subtrees with a standard rewrite system. We need to use an auxiliary function symbol fliptop in order to operate at the top-level.

$$\mathit{fliptop}(\mathit{fork}(T_1, T_2)) \to \mathit{fork}(T_2, T_1)$$

Note that there is no rewrite rule which eliminates fliptop when applied to a leaf. We could favour the invention of an error tree for that purpose. Now, let us consider a strategy FlipTop *to flip top-level subtrees:*[3]

$$\textsc{FlipTop} = \mathit{fork}(T_1, T_2) \to \mathit{fork}(T_2, T_1)$$

That is, we define a strategy (namely a rewrite rule) FlipTop *which rewrites a fork-tree by flipping the subtrees. Note that this rule is non-terminating when considered as a standard rewrite system. However, when considered as strategy, the rewrite rule is applied at the root of the input term, and application is not iterated (unless explicitly expressed in another strategy). Note also that an application of the strategy* FlipTop *to a leaf will simply fail. There is no need to invent an error element. If we want* FlipTop *to succeed on a leaf, we can define the following variant of* FlipTop*. Actually, we show two equivalent variants:*

$$\begin{array}{rcl} \textsc{FlipTop}' & = & \textsc{FlipTop} \Leftarrow \epsilon \\ & = & \textsc{FlipTop} + \mathit{leaf}(\epsilon) \end{array}$$

In the first formulation, we employ left-biased choice and the identity ϵ to recover from failure if FlipTop *is not applicable. In the second formulation, we use a case discrimination such that* FlipTop *handles the constructor* fork*, and the constructor* leaf *is covered by a separate congruence for* leaf*.* $\diamond$

[3] *Slanted* type style is used for constant/function symbols and sorts. The former start in lower case, the latter in upper case. Small Caps type style is used for names of strategies.

Generic traversal combinators Congruences can be used for type-specific traversals. S'_γ provides a few combinators to deal with generic traversals. These traversal combinators have with congruences in common that they operate on the children of a term. 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 exactly one child of the given term. Strategies derived in terms of $\Box(\cdot)$ and $\Diamond(\cdot)$ are intentionally type-preserving as they preserve the outermost function symbol. By contrast, the remaining operators deal with type-unifying traversals. The strategy $\bigcirc^{s_\circ}(s)$ reduces all children. Here, s is used to pre-process the children, and $s_\circ$ is used for the pairwise composition of the intermediate reducts.[4] We will later discuss the utility of different orders for processing children. The strategy $\sharp(s)$ selects one child via s. At a first glance, one might view the aforementioned traversal combinators as list-processing functions. However, note that the list of children is heterogeneously typed. This constrains the formal model for the combinators, especially if we aim at a typed calculus.

There are two trivial combinators which are needed for a typeful treatment of type-unifying strategies. They do not quite perform traversals but they are helpers. The strategy $\underline{\bot}$ builds the empty tuple $\langle\rangle$ (corresponding to a distinguished term in S'_γ) regardless of the input term. It is useful to encode success of a generic strategy by $\langle\rangle$ rather than by a term of any sort. The strategy $s_1 \parallel s_2$ applies two strategies in parallel to the input term, and forms a pair from the results. It is useful for decomposition of traversals.

Strategy extension The last combinator in Figure 2 is crucial for the S'_γ model of typed strategic programming. A strategy of the form $s \lhd \gamma$ models the extension of the strategy s to be applicable to terms of all sorts. In fact, γ is a generic type, and the strategy s must be of a many-sorted type $\tau \to \tau'$. The extension of s is performed in the most basic way, namely $s \lhd \gamma$ fails for (at least) all terms of sorts which are different from τ. Of course, the types $\tau \to \tau'$ and γ must be related in a certain way, namely the type scheme underlying γ has to cover the many-sorted type $\tau \to \tau'$. The reduction semantics of $s \lhd \gamma @ t$ is truly type-dependent. One has to check if the type of t coincides with the domain of s to enable the application of s. The combinator $\cdot \lhd \cdot$ is essential for the type-safe application of many-sorted ingredients of generic traversals.

2.2 Strategy definitions

New strategy combinators can be defined by means of an abstraction mechanism which we call strategy definitions. We use ν (possibly subscripted) for formal strategy parameters in strategy definitions. 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 $s[\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n]$ of the body s of the definition of φ. Strategy definitions can be recursive.

$\textsc{Try}(\nu)$	$=$	$\nu \nleftarrow \epsilon$	(Apply ν if possible, succeed otherwise)
$\textsc{Repeat}(\nu)$	$=$	$\textsc{Try}(\nu; \textsc{Repeat}(\nu))$	(Apply ν as often as possible)
$\textsc{Chi}(\nu, \nu_t, \nu_f)$	$=$	$(\nu; \nu_t) \nleftarrow (\underline{\bot}; \nu_f)$	("Characteristic function")

Figure 3: Reusable strategy definitions

In Figure 3, three simple strategy definitions are shown. These definitions embody idioms which are useful in strategic programming. Firstly, $\textsc{Try}(s)$ denotes the idiom to try s but to succeed via ϵ if s fails. Secondly, $\textsc{Repeat}(s)$ denotes exhaustive iteration in the sense that s is performed as many times as possible. Thirdly, $\textsc{Chi}(s, s_t, s_f)$ is intended to map success and failure of s to "constants" s_t and s_f, respectively. To this end, s is supposed to compute $\langle\rangle$ (if it succeeds), while s_t and s_f map

[4] The "$\bigcirc$" in $\bigcirc^{s_\circ}(s)$ hints on the binary operator $s_\circ$ which is central to reduction.

the "content-free" $\langle\rangle$ to some term. Here, we see that $\bot$ is useful to encode constant functions (in the second branch of CHI).

Example 2 *Recall Example 1 where we defined a strategy* FLIPTOP *for flipping top-level subtrees. Let us define a strategy* FLIPALL *which flips subtrees at all levels:*

$$\textsc{FlipAll} = \textsc{Try}(\textsc{FlipTop}; \mathit{fork}(\textsc{FlipAll}, \textsc{FlipAll}))$$

Note how the congruence for fork-trees is used to apply FLIPALL *to the subtrees of a fork-tree.* ◇

Polyadic strategies Many strategies need to work on several terms rather than just on one term. Consider, for example, a strategy for addition. It is supposed to take two naturals (and to return one). There are several ways to accomplish strategies with multiple term arguments. Firstly, the programmer could be required to define function symbols for grouping. Secondly, we could introduce a special notation to allow a kind of polyadic strategy application with multiple term positions. Thirdly, we could consider curried strategy application (immediately leading to a higher-order calculus). Fourthly, polyadic strategies could be based on tuple types. We choose the last option because a simple notation of strategy application can be retained, and higher-order strategies are not needed. In S'_γ, there are distinguished constructors for tuples. The constant symbol $\langle\rangle$ represents the empty tuple, and a pair is represented by $\langle t_1, t_2\rangle$. The notions of rewrite rules and congruence strategies are immediately applicable to tuples. Note that strategies, which take a tuple but return a many-sorted term (as opposed to a tuple), are type-changing.

Example 3 *To map a pair of naturals to the first component, the rewrite rule* $\langle N_1, N_2\rangle \to N_1$ *is appropriate. To flip the top-level subtrees of a pair of fork-trees, the congruence* $\langle \textsc{FlipTop}, \textsc{FlipTop}\rangle$ *is appropriate.* ◇

Example 4 *The following confluent and terminating rewrite system defines addition of naturals in the common manner:*

$$\begin{aligned}
&\mathit{add} : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}\\
&\mathit{add}(N, \mathit{zero}) \to N\\
&\mathit{add}(N_1, \mathit{succ}(N_2)) \to \mathit{succ}(\mathit{add}(N_1, N_2))
\end{aligned}$$

In this reading, add *is a function symbol to represent addition (say to group two naturals to be added). We rely on a normalisation strategy (such as innermost) to actually perform addition. By contrast, we can also define a polyadic strategy* ADD *which takes a pair of naturals:*

$$\begin{aligned}
\textsc{Dec} &= \mathit{succ}(N) \to N\\
\textsc{Inc} &= N \to \mathit{succ}(N)\\
\textsc{Add}_{\mathit{base}} &= \langle N, \mathit{zero}\rangle \to N\\
\textsc{Add}_{\mathit{step}} &= \langle \epsilon, \textsc{Dec}\rangle; \textsc{Add}; \textsc{Inc}\\
\textsc{Add} &= \textsc{Add}_{\mathit{base}} + \textsc{Add}_{\mathit{step}}
\end{aligned}$$

For clarity of exposition, we defined a number of auxiliary strategies. DEC *attempts to decrement a natural.* INC *increments a natural. Actual addition is performed according to the scheme of primitive recursion with the helpers* $\textsc{Add}_{\mathit{base}}$ *and* $\textsc{Add}_{\mathit{step}}$ *for the base and the step case. Both cases are mutually exclusive. The base case is applicable if the second natural is a* zero*. The step case is applicable if the second natural is a non-zero value (since* DEC *will otherwise fail). Notice how a congruence for pairs is employed in the step case.* ◇

Where-clauses To make the definition of (strategies which involve) rewrite rules more convenient, we generalise the concept of rewrite rules as follows. A rewrite rule is of the form $t \to b$ where t is the term of the left-hand side (as before), and b is the right-hand side body of the rule. In the simplest case, a body b is a term t' (as before). A body can also involve where-clauses. Then b is of the following form:

$$b' \text{ where } x = s \mathbin{@} t'$$

The meaning of such a body with a where-clause is that the term reduct (if any) resulting from the strategy application $s \mathbin{@} t'$ is bound to x for the evaluation of the remaining body b'.

Example 5 *We illustrate the utility of where-clauses by a concise reconstruction of the strategy* Add *from Example 4:*

$$\begin{array}{lcl} \textsc{Add} & = & \langle N, zero \rangle \to N \\ & + & \langle N_1, succ(N_2) \rangle \to succ(N_3) \ \textit{where}\ N_3 = \textsc{Add} \mathbin{@} \langle N_1, N_2 \rangle \end{array}$$

◇

2.3 Generic traversals

Let us discuss the core asset of S'_γ, namely its combinators for generic traversals in some detail. To prepare the explanation of the corresponding primitives, we start with a discussion for encoding (not yet generic) traversals. Afterwards we will define a number of reusable schemes for generic traversals in terms of the S'_γ primitives. Ultimately, we will provide the encodings for the traversal problems posed in the introduction.

Traversal functions vs. traversal strategies Let us compare the coding style for (not yet generic) traversals in strategic rewriting and in standard rewriting.[5] Suppose we want to traverse terms of a certain sort. Then, we soon will have to process the subterms of these terms. In general, these subterms are of different sorts. The corresponding result types might differ for the various traversed sorts. If we want to encode traversals in standard rewriting, we basically need an auxiliary function symbol for each traversed sort to map it to the result type. Usually, one has to define one rewrite rule per constructor in the signature at hand. Traversals based on such auxiliary function symbols and rewrite rules get very cumbersome when larger signatures are considered. An application domain which deals with large signatures is program transformation. Here, signatures correspond to language syntaxes. In strategic rewriting, we do not employ auxiliary function symbols, but we rather describe traversals as strategies. As a by-product, we can even cope with type-changing rewrite rules serving as ingredients of a traversal strategy. The many rewrite rules in the above approach can be avoided if strategy combinators are provided to generically process children of any term. Consequently, generic traversal strategies provide a better scalability for traversal problems compared to standard rewriting. We refer the reader to [BSV00, LVK00, BKV01, Vis01] for a discussion of traversal approaches in the rewriting context, and corresponding considerations of scalability of traversal technology.

Example 6 *Let us define a traversal to count leafs in a tree. Note that a function from trees to naturals is obviously type-changing. Consider the following rewrite rule:*

$$\textsc{Count}_{leaf} = leaf(N) \to succ(zero)$$

This rule directly models the essence of counting leafs, namely it says that a leaf is mapped to 1*, i.e.,* $succ(zero)$*. In standard rewriting, we cannot employ the above rewrite rule (since it is type-changing). Instead, we have to organise a traversal with rewrite rules for an auxiliary function symbol* count*:*

$$\begin{array}{l} count : Tree \to Nat \\ count(leaf(N)) \to succ(zero) \\ count(fork(T_1, T_2)) \to add(count(T_1), count(T_2)) \end{array}$$

[5] By "standard rewriting", we mean many-sorted, first-order rewriting based on a fixed normalisation strategy.

Note the scheme for this traversal. We basically have to write one rewrite rule for each constructor. A strategy Count *which does not rely on auxiliary function symbols can be defined as follows:*

$$\begin{aligned} \textsc{Out}_{fork} &= fork(T_1, T_2) \to \langle T_1, T_2 \rangle \\ \textsc{Count}_{fork} &= \textsc{Out}_{fork}; \langle \textsc{Count}, \textsc{Count} \rangle; \textsc{Add} \\ \textsc{Count} &= \textsc{Count}_{leaf} + \textsc{Count}_{fork} \end{aligned}$$

Note that the strategy employs the aforementioned rewrite rule $\textsc{Count}_{leaf}$*. The helper* $\textsc{Count}_{fork}$ *specifies how to count the leafs of a proper fork-tree. That is, we first turn the fork-tree into a pair of its subtrees, then we perform counting for the subtrees by means of a type-changing congruence on pairs, and finally the resulting pair is fed to the strategy for addition. Generic traversal combinators enable us to abstract from the concrete constructor in* $\textsc{Count}_{fork}$*. Here is a variant of* Count *which can cope with any constructor with one or more children:*

$$\begin{aligned} \textsc{Count}_{any} &= \bigcirc^{\textsc{Add}}(\textsc{Count}) \\ \textsc{Count} &= \textsc{Count}_{leaf} + \textsc{Count}_{any} \end{aligned}$$

That is, we use the combinator $\bigcirc^{\cdot}(\cdot)$ *to reduce children accordingly.* ◇

Derived combinators In Figure 4, we show some useful derived combinators for generic traversals. The corresponding strategy definitions immediately illustrate the potential of the generic traversal combinators. Several of the definitions from the type-preserving group are adopted from [VB98]. We postpone discussing typing issues for a minute.

Type-preserving combinators

$\textsc{Con}$	$=$	$\Box(\delta)$	(Test for a constant)
$\textsc{Fun}$	$=$	$\Diamond(\epsilon)$	(Test for a compound term)
$\textcircled{*}(\nu)$	$=$	$\Box(\textsc{Try}(\nu))$	(Apply ν to as many children as possible)
$\textcircled{+}(\nu)$	$=$	$\neg\,\Box(\neg\,\nu); \textcircled{*}(\nu)$	(Apply ν to at least one child)
$\mathrm{TD}(\nu)$	$=$	$\nu; \Box(\mathrm{TD}(\nu))$	(Top-down traversal)
$\mathrm{BU}(\nu)$	$=$	$\Box(\mathrm{BU}(\nu)); \nu$	(Bottom-up traversal)
$\textsc{OnceTD}(\nu)$	$=$	$\nu \nleftarrow \Diamond(\textsc{OnceTD}(\nu))$	(Top-down traversal with single application)
$\textsc{OnceBU}(\nu)$	$=$	$\nu \nrightarrow \Diamond(\textsc{OnceBU}(\nu))$	(Bottom-up traversal with single application)
$\textsc{Innermost}(\nu)$	$=$	$\textsc{Repeat}(\textsc{OnceBU}(\nu))$	(Innermost evaluation strategy)
$\textsc{StopTD}(\nu)$	$=$	$\nu \nleftarrow \Box(\textsc{StopTD}(\nu))$	(Top-down traversal with "cut")

Type-unifying combinators

$\textsc{Any}(\nu)$	$=$	$\nu + \sharp(\textsc{Any}(\nu))$	(Deep match)
$\textsc{Tm}(\nu)$	$=$	$\nu \nleftarrow \sharp(\textsc{Tm}(\nu))$	(Topmost match)
$\textsc{Bm}(\nu)$	$=$	$\nu \nrightarrow \sharp(\textsc{Bm}(\nu))$	(Bottommost match)
$\textsc{Flatten}(\nu, \nu_u, \nu_\circ)$	$=$	$(\textsc{Con}; \underline{\perp}; \nu_u) + (\textsc{Fun}; \bigcirc^{\nu_\circ}(\nu))$	(Reduction with unit)
$\textsc{Crush}(\nu, \nu_u, \nu_\circ)$	$=$	$(\nu \parallel \textsc{Flatten}(\textsc{Crush}(\nu, \nu_u, \nu_\circ), \nu_u, \nu_\circ)); \nu_\circ$	(Deep reduction)
$\textsc{StopCrush}(\nu, \nu_u, \nu_\circ)$	$=$	$\nu \nleftarrow \textsc{Flatten}(\textsc{StopCrush}(\nu, \nu_u, \nu_\circ), \nu_u, \nu_\circ)$	(Deep reduction with cut)

Figure 4: Derived combinators for generic traversals

Let us read a few of the definitions in Figure 4. The strategy $\mathrm{TD}(s)$ applies s to each node in top-down manner. This is expressed by sequential composition such that s is first applied to the current node, and then we recurse into the children. It is easy to see that if s fails for any node, the traversal fails entirely. A similar derived combinator is StopTD. However, left-biased choice (instead of sequential composition) is used to transfer control to the recursive part. Thus, if the strategy succeeds for the node at hand, the children will not be processed anymore. Another insightful (intentionally)

type-preserving example is INNERMOST which directly models the innermost normalisation strategy known from standard rewriting. The first three type-unifying combinators ANY, TM, and BM deal with the selection of a subterm. They all have in common that they resort to the selection combinator $\sharp(\cdot)$ to determine a suitable child. They differ in the sense that they perform search either non-deterministically, or in top-down manner, or in bottom-manner. One might wonder if it is sensible to vary the horizontal order as well. We will discuss this issue later. The combinator FLATTEN performs reduction with a unit. In general, we do not impose the existence of a unit for the most basic form of reduction via $\bigcirc(\cdot)$. Hence, FLATTEN is able to reduce constants whereas $\bigcirc(\cdot)$ is not. The combinators CRUSH and STOPCRUSH model deep reduction w.r.t. to the same kind of monoid-like argument strategies as FLATTEN.[6] The combinator CRUSH evaluates each node in the tree, and hence, it needs to succeed for each node. The reduction of the current node and the recursion into the children is done in parallel with $\cdot \parallel \cdot$ leading to a pair of intermediate results. The corresponding pair is reduced with the binary monoid operation. STOPCRUSH is more similar to STOPTD in the sense that the current node is first evaluated, and only if evaluation fails, then we recurse into the children.

		Combinators on Booleans	
FALSE	=	$\langle\rangle \to \mathit{false}$	(Build "false")
TRUE	=	$\langle\rangle \to \mathit{true}$	(Build "true")
		Combinators on naturals	
NAT	=	$\mathit{zero} + \mathit{succ}(\epsilon)$	(Test by congruences)
ZERO	=	$\langle\rangle \to \mathit{zero}$	(Build "0")
ONE	=	$\langle\rangle \to \mathit{succ}(\mathit{zero})$	(Build "1")
INC	=	$N \to \mathit{succ}(N)$	(Increment)
ADD	=	...	(Addition)
		Combinators on lists of naturals	
NIL	=	$\langle\rangle \to \mathit{nil}$	(Build "nil")
SINGLETON	=	$N \to \mathit{cons}(N, \mathit{nil})$	(Construct a singleton list)
APPEND	=	...	(Append two lists)
(I)	=	STOPTD(NAT; INC)	
(II)	=	ONCEBU($g(P) \to g'(P)$)	
(III)	=	CHI(ANY(NAT; $\underline{\perp}$), TRUE, FALSE)	
(IV)	=	STOPCRUSH(NAT; SINGLETON, NIL, APPEND)	
(V)	=	CRUSH(CHI($g(\epsilon)$; $\underline{\perp}$, ONE, ZERO), ZERO, ADD)	

Figure 5: Untyped encodings for traversals from Figure 1

Example 7 *Let us solve the five problems (I)–(V) illustrated in Figure 1 in the introduction of the article. In Figure 5, we first define some auxiliary strategies on naturals, Booleans, and lists of naturals, and then, the ultimate traversals (I)–(V) are defined in terms of the combinators from Figure 3 and Figure 4. Note that the encodings are not yet fully faithful w.r.t. typing. We will later revise these encodings accordingly. Let us explain the traversals in detail.*

(I) *We are supposed to increment all naturals. The combinator* STOPTD *is employed to descend into the given term as long as we do not find a natural (recognised via* NAT*). When we encounter a natural in top-down manner, we apply the rule* INC *for incrementing naturals. Note that we*

[6] The term crushing has been coined in the related context of polytypic programming [Mee96].

must not further descend into the term. In fact, if we used TD *instead of* StopTD*, we describe a non-terminating strategy. Also note that a bottom-up traversal is not an option either. If we used* BU *instead of* StopTD*, we will model the replacement of a natural N by $2N+1$.*

(II) *We want to replace terms of the form $g(P)$ by $g'(P)$. As we explained in the introduction, the replacement must not be done exhaustively. We only want to perform one replacement where the corresponding redex should be identified in bottom-up manner. These requirements are met by the combinator* OnceBU*.*

(III) *We want to find out if naturals occur in the term. The result should be encoded as a Boolean (hence, the two branches* True *and* False *in* Chi*). We look for naturals again via the auxiliary strategy* Nat*. The kind of deep matching we need is provided by the combinator* Any *which non-deterministically looks for a child where* Nat *succeeds.* Nat *is followed by $\perp$ to express that we are not looking for actual naturals but only for the property if there are naturals at all. The application of* Chi *turns success and failure into a Boolean.*

(IV) *To collect all naturals in a term, we need to perform a kind of deep reduction. Here, it is important that reduction with cut (say,* StopCrush*) is used because a term representing a non-zero natural N "hosts" the naturals $N-1,\ldots,0$ due to the representation of naturals via the constructors* succ *and* zero*. These hosted naturals should not be collected. Recall that crushing (say reduction) uses monoid-like arguments. In this example,* Append *is the associative operation of the monoid, and* Nil *(i.e., the strategy to build the empty list) represents the unit of* Append*.*

(V) *Finally, we want to count all occurrences of g. In order to locate these occurrences, we use the congruence $g(\epsilon)$. In this example, it is important that we perform crushing exhaustively (i.e., without cut) since terms rooted by g might indeed host further occurrences of g to be counted (as illustrated in Figure 1).*

$\diamond$

Note the genericity of the defined traversals (I)–(V). They can be applied to any term. Of course, the strategies are somewhat specific because they rely on some constant or function symbols, namely *true*, *false*, *zero*, *succ*, g, and g'.

2.4 Typed strategies

Let us now motivate the typeful model of strategic programming underlying S'_γ. The ultimate challenge is to assign types to generic traversal strategies like TD, StopTD, or Crush. Recall our objective for S'_γ to stay as close in a first-order many-sorted setting as possible. The type system we envision should be easy to reason about, and to implement.

Many-sorted types Let us start with a basic fragment of S'_γ without support for generic traversals. Clearly, many-sorted types are sufficient for strategies which only involve specific sorts. We use τ and π (possibly subscripted or primed) to range over term types or strategy types, respectively. Term types are sorts and tuple types (i.e., products over term types). A strategy type π is a first-order function type, that is, π is of the form $\tau \to \tau'$. Here, τ is the type of the input term, and τ' is the type of the term reduct (if any). We also use the terms domain and co-domain for τ or τ', respectively. The type declaration for a strategy combinator φ which does not take any strategy arguments is of the form $\varphi : \pi$. We use $\langle \tau_1, \tau_2 \rangle$ to denote the product type for pairs $\langle t_1, t_2 \rangle$ where the components t_1 and t_2 are of type τ_1 and τ_2, respectively. The type of the empty tuple $\langle\rangle$ is simply denoted as $\langle\rangle$. The type declaration for a strategy combinator φ with $n \geq 1$ arguments is represented in the following format:

$$\varphi : \pi_1 \times \cdots \times \pi_n \to \pi_0$$

Here, $\pi_1, \ldots, \pi_n$ denote the strategy types for the argument strategies, and π_0 denotes the strategy type of an application of φ. All the π_i are again of the form $\tau_i \to \tau_i'$. Consequently, strategy combinators (with argument strategies) correspond to second-order functions on terms. This can be checked by counting the level of nesting of arrows "$\to$" in a combinator type.

Example 8 *We show the many-sorted types for a few strategy combinators, namely the type of* FLIPALL *from Example 2, the type of the congruence combinator* $fork(\cdot,\cdot)$ *for the function symbol* $fork$ *used in Example 2, and the type of* ADD *from Example 4.*

$$\begin{array}{lcl} \textsc{FlipAll} & : & \mathit{Tree} \to \mathit{Tree} \\ \mathit{fork} & : & (\mathit{Tree} \to \mathit{Tree}) \times (\mathit{Tree} \to \mathit{Tree}) \to (\mathit{Tree} \to \mathit{Tree}) \\ \textsc{Add} & : & \langle \mathit{Nat}, \mathit{Nat} \rangle \to \mathit{Nat} \end{array}$$

◇

For simplicity, we assume that the types of all function and constant symbols, variables, and strategies are explicitly declared. This is well in line with standard practice in term rewriting and algebraic specification. Declarations for variables, rewriting functions and strategies are common in several frameworks for rewriting, e.g., in CASL, ASF+SDF, and ELAN. Note however that this assumption is not essential. Inference of types for constant and function symbols, variables, and strategies is feasible. As an aside, type inference is simple because S_γ' basically deals with (at most) second-order function types, and the polymorphism we consider (see below) is restricted to top-level universal quantification (cf. [Mil78]). The distinguished generic types of S_γ' do not challenge type inference since they are like constant types, and inhabitation is done explicitly via the combinator $\cdot \lhd \cdot$.

Example 9 *We define a strategy* APPEND *to append two lists of naturals. We declare all the constant and function symbols (i.e., nil and cons), and variables for lists (namely* L_1, L_2, L_3*) and naturals (namely* N*) as list elements.*

$$\begin{array}{rcl} \mathit{nil} & : & \mathit{NatList} \\ \mathit{cons} & : & \mathit{Nat} \times \mathit{NatList} \to \mathit{NatList} \\ N & : & \mathit{Nat} \\ L_1, L_2, L_3 & : & \mathit{NatList} \\ \textsc{Append} & : & \langle \mathit{NatList}, \mathit{NatList} \rangle \to \mathit{NatList} \\ \textsc{Append} & = & \langle \mathit{nil}, L \rangle \to L \\ & + & \langle \mathit{cons}(N, L_1), L_2 \rangle \to \mathit{cons}(N, L_3) \ \textit{where}\ L_3 = \textsc{Append} \mathbin{@} \langle L_1, L_2 \rangle \end{array}$$

◇

Generic types In order to provide types for generic traversals, we need to extend our basically many-sorted type system. To this end, we identify distinguished generic types for strategies which are applicable to all sorts. We use γ to range over generic strategy types. There are two generic strategy types. The type TP models generic type-preserving strategies, while the type $\mathsf{TU}(\tau)$ models type-unifying strategies where all types are mapped to τ. These two forms correspond to the main characteristics of S_γ'. As we will see, TP and $\mathsf{TU}(\cdot)$ can be integrated into a basically many-sorted system in a simple manner.

Example 10 *The following types are the intended ones for the traversals defined in Figure 5.*

$$\begin{array}{rcl} \text{(I)} & : & \mathsf{TP} \\ \text{(II)} & : & \mathsf{TP} \\ \text{(III)} & : & \mathsf{TU}(\mathit{Boolean}) \\ \text{(IV)} & : & \mathsf{TU}(\mathit{NatList}) \\ \text{(V)} & : & \mathsf{TU}(\mathit{Nat}) \end{array}$$

◇

Type-parameterised strategy definitions Generic strategy types capture the kind of genericity needed for generic traversals with different behaviour for different term types. In order to turn S'_γ in a somewhat complete programming language, we also need to enable parametric polymorphism. Consider, for example, the combinator CRUSH for deep reduction in Figure 4. The result type of reduction should be a parameter. The scheme of crushing is in fact not sensitive w.r.t. the parameter. The arguments passed to CRUSH are the only strategies to operate on the parametric type for unification. We employ a very simple form of parametric polymorphism [Mil78, CW85]. Types of strategy combinators may contain type variables which are explicitly quantified at the top level. We use α (possibly subscripted) for term type parameters. Thus, in general, a type of a strategy combinator is of the following form:[7]

$$\varphi : \forall\alpha_1.\ \ldots.\ \forall\alpha_m.\ \pi_1 \times \cdots \times \pi_n \to \pi_0$$

We assume that the type parameters in $\pi_0, \ldots, \pi_n$ are contained in the set $\{\alpha_1, \ldots, \alpha_m\}$ of formal parameters. Throughout the article, we assume explicit type application, that is, the application of universally quantified strategy combinators involves type application. Application of a type-parameterised combinator is denoted by $\varphi[\tau_1, \ldots, \tau_m](s_1, \ldots, s_n)$. For convenience, an actual implementation of S'_γ is likely to support implicit type application.

Example 11 *Here are the types for the strategy combinators from Figure 3 and Figure 4. As we can see, all traversal schemes which involve a type-unifying facet, need to be parameterised.*

$$\begin{array}{rcl}
\textsc{Try}, \textsc{Repeat} & : & \mathsf{TP} \to \mathsf{TP} \\
\textsc{Chi} & : & \forall\alpha.\mathsf{TU}(\langle\rangle) \to (\langle\rangle \to \alpha) \to (\langle\rangle \to \alpha) \to \mathsf{TU}(\alpha) \\
\textsc{Con}, \textsc{Fun} & : & \mathsf{TP} \\
\textsc{All}^*, \ldots, \textsc{StopTd} & : & \mathsf{TP} \to \mathsf{TP} \\
\textsc{Any}, \textsc{Tm}, \textsc{Bm} & : & \forall\alpha.\mathsf{TU}(\alpha) \to \mathsf{TU}(\alpha) \\
\textsc{Flatten}, \textsc{Crush}, \textsc{StopCrush} & : & \forall\alpha.\mathsf{TU}(\alpha) \times (\langle\rangle \to \alpha) \times (\langle\alpha, \alpha\rangle \to \alpha) \to \mathsf{TU}(\alpha)
\end{array}$$

We update all definitions which involve type parameters:

$$\begin{array}{rcl}
\textsc{Chi}[\alpha](\nu, \nu_t, \nu_f) & = & (\nu; \nu_t) \Leftarrow (\underline{\bot}; \nu_f) \\
\textsc{Any}[\alpha](\nu) & = & \nu + \sharp(\textsc{Any}[\alpha](\nu)) \\
\textsc{Tm}[\alpha](\nu) & = & \nu \Leftarrow \sharp(\textsc{Tm}[\alpha](\nu)) \\
\textsc{Bm}[\alpha](\nu) & = & \nu \Rightarrow \sharp(\textsc{Bm}[\alpha](\nu)) \\
\textsc{Flatten}[\alpha](\nu, \nu_u, \nu_\circ) & = & (\textsc{Con}; \underline{\bot}; \nu_u) + (\textsc{Fun}; \bigcirc^{\nu_\circ}(\nu)) \\
\textsc{Crush}[\alpha](\nu, \nu_u, \nu_\circ) & = & (\nu \parallel \textsc{Flatten}[\alpha](\textsc{Crush}[\alpha](\nu, \nu_u, \nu_\circ), \nu_u, \nu_\circ)); \nu_\circ \\
\textsc{StopCrush}[\alpha](\nu, \nu_u, \nu_\circ) & = & \nu \Leftarrow \textsc{Flatten}[\alpha](\textsc{StopCrush}[\alpha](\nu, \nu_u, \nu_\circ), \nu_u, \nu_\circ)
\end{array}$$

$\diamond$

Example 12 *Let us also give an example of a polymorphic strategy definition which does not rely on the generic strategy types* TP *and* TU(·) *at the same time. To this end, let us first consider the declaration* TRY : TP → TP *in Example 11. This type was motivated by the use of* TRY *in the definition of generic traversals, e.g., in the definition of* ALL*(·) in Figure 4. However, the generic type of* TRY *invalidates the application of* TRY *in Example 2 where it was used to complete, in a sense, a many-sorted strategy. We resolve this conflict of interests by the introduction of a polymorphic combinator* TRY′ *for many-sorted strategies. We use this new combinator to revise Example 2 as follows:*

$$\begin{array}{lcl}
\textsc{Try}' & : & \forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha) \\
\textsc{Try}'[\alpha] & = & \nu \Leftarrow \epsilon \\
\textsc{FlipAll} & = & \textsc{Try}'[\mathit{Tree}](\textsc{Flip}; \mathit{fork}(\textsc{FlipAll}, \textsc{FlipAll}))
\end{array}$$

[7] For brevity, we omit the case that the combinator φ does not have any strategy arguments.

At a first glance, the consideration of both Try *and* Try′[α] *(which are defined equally) might appear redundant. However, the point is that both modes of making a strategy total are in fact different (in a typeful setting). Either we say that a many-sorted strategy is made total (for the corresponding sort), or we say that a generic strategy is made total.* ◇

Parameterised data types For convenience, we should also enable parameterised data types (such as parameterised lists as opposed to lists of naturals in Example 9). Basically, we have to add support for parameterised algebraic data types (as opposed to purely many-sorted constructors). Then, type-parameterised strategy definitions would be sufficient to describe operations on parameterised data types. For brevity, we do not formalise parameterised data types in the present article since they are not strictly needed to develop a typeful model of generic traversals. A corresponding extension is routine. Parameterised (algebraic) data types are well-understood in the context of algebraic specification and rewriting. The instantiation of parameterised specifications (or modules) is typically based on signature morphisms as supported, e.g., in CASL [ABK+01] or ELAN [BKK+98]. A different approach to support parameterised data types could be based on a language design with full support for polymorphic functions and parameterised data types like in the functional languages SML and Haskell.

Strategy extension The remaining problem with generic strategies is the mediation between many-sorted strategy types (such as the type $Nat \to Nat$ of a rewrite rule like Inc), and generic strategy types. If we look back to the simple-minded definition of, for example, (I) in Example 7, we see that Nat; Inc is used as a parameter of StopTD. By contrast, the derived combinator StopTD intentionally should be invoked with a generic strategy as it is potentially applied to nodes of any sort (via $\Box(\cdot)$). From an operational perspective, Nat; Inc will fail when faced with a term of a sort other than *Nat*. In this case, we are lucky since the strategy performs a kind of dynamic type check via Nat (defined in terms of the congruences for sort *nat*) so that Inc will not be applied to terms of other types. In general, we argue as follows:

> A programmer has to explicitly turn many-sorted strategies into generic ones. The type system (and the corresponding reduction semantics) is responsible for the type-safe application of strategies.

S'_γ offers the combinator $\cdot \lhd \cdot$ to turn a many-sorted strategy into a generic one. Given a strategy s of some specific type $\tau \to \tau'$, and a generic strategy type γ, the type of $s \lhd \gamma$ is γ, that is, s is lifted to γ. We call this process strategy extension because the lifted strategy $s \lhd \gamma$ is applicable to terms of all sorts and not just to terms of sort τ. It is a trivial form of extension because $s \lhd \gamma$ fails for all terms which are not of sort τ. In a sense, failure is the generic default for the extended strategy. This trivial form of extension is completely sufficient because other extensions can be achieved by subsequent application of $\cdot + \cdot$ (and friends). That is, one can recover from failure, and one can resort to a more useful generic default, e.g., ϵ, or a catch-all case of a generic traversal. In fact, many traversal schemes are encoded in a way that failure at the current node triggers recursion, e.g., in StopTD.

Example 13 *We revise Example 7 to finally supply the typeful solutions for the traversals (I)–(V) from the introduction. The following definitions are in full compliance with the S'_γ type system:*

$$
\begin{array}{rcl}
\text{(I)} & = & \textsc{StopTD}(\textsc{Inc} \lhd \mathsf{TP}) \\
\text{(II)} & = & \textsc{OnceBU}(g(P) \to g'(P) \lhd \mathsf{TP}) \\
\text{(III)} & = & \textsc{Chi}[\mathit{Boolean}](\textsc{Any}[\langle\rangle](\textsc{Nat} \lhd \mathsf{TP}; \underline{\bot}), \textsc{True}, \textsc{False}) \\
\text{(IV)} & = & \textsc{StopCrush}[\mathit{NatList}](\textsc{Nat} \lhd \mathsf{TU}(\mathit{Nat}); \textsc{Singleton}, \textsc{Nil}, \textsc{Append}) \\
\text{(V)} & = & \textsc{Crush}[\mathit{Nat}](\textsc{Chi}[\mathit{Nat}](g(\epsilon) \lhd \mathsf{TP}; \underline{\bot}, \textsc{One}, \textsc{Zero}), \textsc{Zero}, \textsc{Add})
\end{array}
$$

The changes concern the inserted applications of $\cdot \lhd \cdot$, and the actual type parameters for type-unifying combinators. In the definition of (I)*, the strategy* Inc *clearly needs to be lifted to* TP*; similarly for the*

rewrite rule in (II). *Note that the original test for naturals (via strategy* NAT *in Example 7) is gone in* (I). *The type of* INC *sufficiently restricts its applicability. In the definition of* (III), *the strategy* NAT *is used to check for naturals, and it is lifted to* TP. *The type-unifying facet of* (III) *is enforced by the subsequent application of* $\underline{\perp}$, *and it is also pointed out by the application of* CHI. *In the definition of* (IV), *the strategy* NAT *is used to select naturals, and it is lifted to* $\mathsf{TU}(Nat)$. *The subsequent application of* SINGLETON *converts naturals to (singleton) lists of naturals. The extension performed in* (V) *can be justified by similar arguments as for* (III). *In both cases,* CHI *is applied to map the success and failure behaviour of a strategy to distinguished constants.* ◇

Static type safety The resulting typed calculus S'_γ obeys a number of convenient properties. Firstly, each strategy expression is strictly either many-sorted or generic. Secondly, many-sorted strategies cannot become generic just by accident, say due to the context in which they are used. By contrast, strategies only get generic via $\cdot \lhd \cdot$. Thirdly, the type-dependent facet of the reduction semantics is clearly restricted to $\cdot \lhd \cdot$. The semantics of all other strategy combinators does not involve type-dependency. Hence, S'_γ is statically type-safe, and no dynamic type checking is due. An application of the combinator $\cdot \lhd \cdot$ implies a type inspection at run time (i.e., reduction time), but this inspection is concerned with the treatment of different behaviours depending on the actual term type. The inspection is enforced by the programmer as a kind of type-case. Such expressiveness has also been integrated into other statically typed languages (cf. [CGL95, DRW95, CWM99]).

2.5 Bibliography

Let us relate the calculus S'_γ to those frameworks for strategic programming which were most influential for its design, namely system S underlying Stratego [VB98, VBT98], and ELAN [BKK+98, BKKR01]. A more general discussion of related work is due in Section 7.

S'_γ vs. system S and Stratego Our typed rewriting calculus S'_γ adopted the untyped system S to a large extent. We did not just adopt important combinators like $\Box(\cdot)$ and $\Diamond(\cdot)$. We also stick to the same semantic model. The main limitation of S'_γ compared to system S is that we favour standard first-order rewrite rules with where-clauses as primitive form of strategy, whereas system S provides less standard primitives which are sufficient to model rewrite rules as syntactic sugar. These primitives are matching (to bind variables), building terms (relying on previous bindings), and scoping of variables. The additional flexibility which one gains by this separation is that arbitrary strategies can be performed between matching and building. One can simulate this style by using where-clauses in S'_γ. On the other hand, S'_γ also introduces a few combinators which are not expressible in system S, namely the combinators $\bigcirc^{\cdot}(\cdot)$ and $\sharp(\cdot)$ for intentionally type-unifying traversals. In Stratego, a related language construct "$\cdot\#\cdot$" was introduced to traverse the children of a term (and also to access the surrounding function symbol as a string if needed). The construct is borrowed from Prolog, where terms can be generically destructed and constructed via the so-called univ operator denoted as "=..". The "$\cdot\#\cdot$' construct is not covered by system S. A crucial problem with the "$\cdot\#\cdot$" is that it leads to a hopelessly untyped model of traversal since the children are treated as lists. In general, system S and Stratego have not been designed with typing in mind. The S'_γ combinator $\cdot \lhd \cdot$ for strategy extension was especially introduced for typeful strategic programming with generic traversals. As an aside, system S is not minimal in the sense that one can reconstruct two primitives of system S in terms of others, namely left-biased choice (as we have shown in Section 2.1), and the hybrid traversal combinator $\textcircled{\scriptsize o}(\cdot)$ of system S (cf. $\textcircled{\scriptsize o}^+(\cdot)$ in Figure 4). One might argue that our definitions of these combinators are unnecessarily inefficient, and primitive combinators as in system S are more appropriate.

S'_γ vs. ELAN The influence of ELAN is also traceable in S'_γ. We adopted the model of rewrite rules with where-clauses from ELAN. We also adopted recursive strategy definitions from ELAN (while system S employs a special recursion operator $\mu\, \cdot\, .\, \cdot$). In the initial design of a basically many-sorted

type system we also received inspiration from the ELAN specification language. ELAN and S'_γ differ in the semantic model assumed for reduction. ELAN offers a faithful model of non-determinism by sets (or lists) of possible results where the empty set represents failure. The type system of S'_γ does not rely on the simple model of system S. In fact, the typeful S'_γ approach to generic traversals could be integrated with the ELAN-like semantic model without changing any detail in the type system.

3. Many-sorted strategies

We start the formal definition of S'_γ. As a warm-up, we discuss many-sorted strategies. Both type-preserving and type-changing strategies will be covered in the present section. For simplicity, we postpone formalising strategy definitions until Section 5.1. First, we will define a basic calculus S'_0 corresponding to an initial untyped fragment of S'_γ. Then, we develop a simple type system starting with type-preserving strategies. We will discuss some standard properties of the type system. Afterwards, we elaborate the type system to cover type-changing strategies and tuples for polyadic strategies. We use inference rules (say deduction rules) in the style of Natural semantics [Kah87] for both the reduction semantics and the type system. In fact, the reduction semantics of S'_γ is a big-step semantics.

3.1 The basic calculus S'_0

Judgement for reduction As for the dynamic semantics of strategies (say the reduction semantics), we employ the judgement $s @ t \rightsquigarrow r$ for strategy application.[8] Here, r is the reduct of rewriting. We assume that strategies are only applied to ground terms, and then also yield ground terms. The latter assumption is not essential but it is well in line with standard rewriting.

Positive and negative rules In Figure 6, we define the reduction of strategy applications for the initial calculus S'_0. The inference rules formalise our informal explanations from Section 2.1. We employ a certain style for the specification of the deduction rules. We give positive rules for cases when the reduct is a term whereas we give negative rules for the remaining cases with failure as the reduct. The rules for ϵ and δ are trivial. Let us read, for example, the rules for negation. The application $\neg\, s @ t$ returns t if the application $s @ t$ returns "↑" (cf. [neg^+]). If $s @ t$ results in a proper term reduct, then $\neg\, s @ t$ evaluates to "↑" (cf. [neg^-]). These rules also illustrate why we need to include failure as reduct. Otherwise, a judgement could not query whether a certain strategy application did not succeed. In strategic programming, negation is not used that often, but syntactic sugar employing negation is essential, e.g., the combinators $\cdot \Leftarrow \cdot$ and $\cdot \Rightarrow \cdot$ for biased choice as defined in Section 2.1. Let us also look at the rules for the other combinators. The rule [seq^+] directly encodes the idea of sequential composition where the intermediate term t' obtained via s_1 is further reduced via s_2. Sequential composition fails if one of the two ingredients s_1 or s_2 fails (cf. [$\mathsf{seq}^-.1$], [$\mathsf{seq}^-.2$]). As for choice, there is one positive rule for each operand of the choice (cf. [$\mathsf{choice}^+.1$], [$\mathsf{choice}^+.2$]). Choice fails if both options do not admit success (cf. [choice^-]). The congruences for constants are trivially defined (cf. [$\mathsf{congr}^+.1$], [$\mathsf{congr}^-.1$]). The congruences for function symbols are defined in a schematic manner to cover arbitrary arities (cf. [$\mathsf{congr}^+.2$], [$\mathsf{congr}^-.2$], [$\mathsf{congr}^-.3$]).

Where-clauses In Figure 6, rule bodies were assumed to be terms. In Figure 7, an extension to cope with where-clauses as motivated earlier is supplied. The semantics of rewrite rules as covered in Figure 6 is surpassed by the new rules in Figure 7. Essentially, we resort to a new judgement for the evaluation of rule bodies. A rule body, which consists of a term, evaluates trivially to this term (cf. [$\mathsf{body}^+.1$]). A rule body of the form b where $x = s @ t$ is evaluated by first performing the strategy application $s @ t$, and then binding the intermediate term reduct t' (if any) to x in the remaining body b (cf. [$\mathsf{body}^+.2$]). Obviously, a rewrite rule can now fail for two reasons, either because of an infeasible match (cf. [$\mathsf{rule}^-.1$]), or due to a failing subcomputation in a where-clause (cf. [$\mathsf{rule}^-.2$], [$\mathsf{body}^-.1$], and

[8] Note that "reduction" has two meanings in the present article, namely reduction in the sense of the reduction semantics for strategies, and reduction of the children of a term by monoid-like combinators.

Syntax

$$
\begin{array}{lcll}
c & & & \text{(Constant symbols)} \\
f & & & \text{(Function symbols)} \\
x & & & \text{(Term variables)} \\
t & ::= & c \mid f(t,\ldots,t) \mid x & \text{(Terms)} \\
s & ::= & t \to b \mid \epsilon \mid \delta \mid s;s \mid s+s \mid c \mid f(s,\ldots,s) & \text{(Strategies)} \\
b & ::= & t & \text{(Rule bodies)}
\end{array}
$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow r}$

Positive rules

$$\frac{\exists\theta.(\theta(t_l) = t \wedge \theta(t_r) = t')}{t_l \to t_r \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{rule}^+]$$

$$\epsilon \mathbin{@} t \rightsquigarrow t \quad [\mathsf{id}^+]$$

$$\frac{s \mathbin{@} t \rightsquigarrow \uparrow}{\neg\, s \mathbin{@} t \rightsquigarrow t} \quad [\mathsf{neg}^+]$$

$$\frac{\begin{array}{ll} & s_1 \mathbin{@} t \rightsquigarrow t' \\ \wedge & s_2 \mathbin{@} t' \rightsquigarrow t'' \end{array}}{s_1; s_2 \mathbin{@} t \rightsquigarrow t''} \quad [\mathsf{seq}^+]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow t'}{s_1 + s_2 \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{choice}^+.1]$$

$$\frac{s_2 \mathbin{@} t \rightsquigarrow t'}{s_1 + s_2 \mathbin{@} t \rightsquigarrow t'} \quad [\mathsf{choice}^+.2]$$

$$c \mathbin{@} c \rightsquigarrow c \quad [\mathsf{congr}^+.1]$$

$$\frac{\begin{array}{ll} & s_1 \mathbin{@} t_1 \rightsquigarrow t_1' \\ \wedge & \cdots \\ \wedge & s_n \mathbin{@} t_n \rightsquigarrow t_n' \end{array}}{f(s_1,\ldots,s_n) \mathbin{@} f(t_1,\ldots,t_n) \rightsquigarrow f(t_1',\ldots,t_n')} \quad [\mathsf{congr}^+.2]$$

Negative rules

$$\frac{\nexists\theta.\theta(t_l) = t}{t_l \to t_r \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{rule}^-]$$

$$\delta \mathbin{@} t \rightsquigarrow \uparrow \quad [\mathsf{fail}^-]$$

$$\frac{s \mathbin{@} t \rightsquigarrow t'}{\neg\, s \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{neg}^-]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow \uparrow}{s_1; s_2 \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{seq}^-.1]$$

$$\frac{\begin{array}{ll} & s_1 \mathbin{@} t \rightsquigarrow t' \\ \wedge & s_2 \mathbin{@} t' \rightsquigarrow \uparrow \end{array}}{s_1; s_2 \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{seq}^-.2]$$

$$\frac{\begin{array}{ll} & s_1 \mathbin{@} t \rightsquigarrow \uparrow \\ \wedge & s_2 \mathbin{@} t \rightsquigarrow \uparrow \end{array}}{s_1 + s_2 \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{choice}^-]$$

$$\frac{c \neq t}{c \mathbin{@} t \rightsquigarrow \uparrow} \quad [\mathsf{congr}^-.1]$$

$$\frac{f \neq g \vee n \neq m}{f(s_1,\ldots,s_n) \mathbin{@} g(t_1,\ldots,t_m) \rightsquigarrow \uparrow} \quad [\mathsf{congr}^-.2]$$

$$\frac{\exists i \in \{1,\ldots,n\}.\ s_i \mathbin{@} t_i \rightsquigarrow \uparrow}{f(s_1,\ldots,s_n) \mathbin{@} f(t_1,\ldots,t_n) \rightsquigarrow \uparrow} \quad [\mathsf{congr}^-.3]$$

Figure 6: Reduction semantics for the basic calculus S_0'

[$\mathsf{body}^-.2$]). For brevity, we will abstract from where-clauses in the formalisation of the type system for S_γ'. As the reduction semantics indicates, where-clause do not pose any challenge for formalisation.

3.2 Type-preserving strategies

We want to provide a type system for the basic calculus S_0'. We first focus on type-preserving strategies. We use S_{tp}' to denote the resulting calculus. In fact, type-changing strategies are not standard in rewriting. So we shall consider them in a separate step in Section 3.3. In general, the typed calculus S_γ' is developed in a stepwise and modular fashion.

Syntax

$$b \quad ::= \quad \cdots \mid b \text{ where } x = s \mathbin{@} t$$

Rule bodies $\boxed{b \rightsquigarrow r}$

Positive rules

$$t \rightsquigarrow t \qquad [\mathsf{body}^+.1]$$

$$\frac{s \mathbin{@} t \rightsquigarrow t' \quad \wedge \quad b[x \mapsto t'] \rightsquigarrow t''}{b \text{ where } x = s \mathbin{@} t \rightsquigarrow t''} \qquad [\mathsf{body}^+.2]$$

Negative rules

$$\frac{s \mathbin{@} t \rightsquigarrow \uparrow}{b \text{ where } x = s \mathbin{@} t \rightsquigarrow \uparrow} \qquad [\mathsf{body}^-.1]$$

$$\frac{s \mathbin{@} t \rightsquigarrow t' \quad \wedge \quad b[x \mapsto t'] \rightsquigarrow \uparrow}{b \text{ where } x = s \mathbin{@} t \rightsquigarrow \uparrow} \qquad [\mathsf{body}^-.2]$$

Rules $\boxed{s \mathbin{@} t \rightsquigarrow r}$

Positive rule

$$\frac{\exists \theta.\ (\theta(t_l) = t \ \wedge \ \theta(b) \rightsquigarrow t')}{t_l \to b \mathbin{@} t \rightsquigarrow t'} \qquad [\mathsf{rule}^+]$$

Negative rules

$$\frac{\nexists \theta.\ \theta(t_l) = t}{t_l \to b \mathbin{@} t \rightsquigarrow \uparrow} \qquad [\mathsf{rule}^-.1]$$

$$\frac{\exists \theta.\ (\theta(t_l) = t \ \wedge \ \theta(b) \rightsquigarrow \uparrow)}{t_l \to b \mathbin{@} t \rightsquigarrow \uparrow} \qquad [\mathsf{rule}^-.2]$$

Figure 7: Extension for where-clauses

Type expressions There are three levels of types. We have types for many-sorted terms, types for strategies, and types for strategy combinators. We already sketched the type syntax in Section 2.4. As for purely many-sorted strategies, the forms of term and strategy types are trivially defined by the following grammar:

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

Contexts In the upcoming type judgements, we use a context parameter Γ to keep track of sorts σ, and to map constant symbols c, function symbols f and term variables x to types. We will have to consider richer contexts when we formalise strategy definitions in Section 5.1. Initially, we use the following grammar for contexts:

$$\begin{array}{lcll} \Gamma & ::= & \emptyset \mid \Gamma, \Gamma & \text{(Contexts as sets)} \\ & \mid & \sigma \mid c : \sigma \mid f : \sigma \times \cdots \times \sigma \to \sigma & \text{(Signature part)} \\ & \mid & x : \tau & \text{(Term variables)} \end{array}$$

Thus, a context Γ contains a many-sorted term signature, and variable declarations (for term variables). Let us state the requirements for a well-formed context Γ. We assume that there are different name spaces for the various kinds of symbols and variables. Also, we assume that symbols and variables are not associated with different types in Γ (in particular, we do not consider overloading of symbols and variables). All sorts used in some type declaration in Γ also have to be declared themselves in Γ. Finally, when contexts are composed via Γ_1, Γ_2 we require that the sets of symbols and variables in Γ_1 and Γ_2 are disjoint.[9]

[9]Note that disjoint union of contexts will not be used before Section 5.1. In fact, our contexts are completely static until then.

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

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

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

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

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

$$\frac{c : \sigma \in \Gamma}{\Gamma \vdash c : \sigma} \quad \text{[con]}$$

$$\frac{\begin{array}{ll} & f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in \Gamma \\ \wedge & \Gamma \vdash t_1 : \sigma_1 \\ \wedge & \cdots \\ \wedge & \Gamma \vdash t_n : \sigma_n \end{array}}{\Gamma \vdash f(t_1, \ldots, t_n) : \sigma_0} \quad \text{[fun]}$$

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

Negatable types $\boxed{\neg_\Gamma \pi \rightsquigarrow \pi'}$

$$\frac{\Gamma \vdash \tau}{\neg_\Gamma \tau \to \tau \rightsquigarrow \tau \to \tau} \quad \text{[negt.1]}$$

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

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

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

$$\frac{\begin{array}{ll} & \Gamma \vdash s : \tau \to \tau \\ \wedge & \Gamma \vdash t : \tau \end{array}}{\Gamma \vdash s @ t : \tau} \quad \text{[apply]}$$

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

$$\frac{\begin{array}{ll} & \Gamma \vdash t_l : \tau \\ \wedge & \Gamma \vdash t_r : \tau \end{array}}{\Gamma \vdash t_l \to t_r : \tau \to \tau} \quad \text{[rule]}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \epsilon : \tau \to \tau} \quad \text{[id]}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \delta : \tau \to \tau} \quad \text{[fail]}$$

$$\frac{\begin{array}{ll} & \Gamma \vdash s : \pi \\ \wedge & \neg_\Gamma \pi \rightsquigarrow \pi' \end{array}}{\Gamma \vdash \neg s : \pi'} \quad \text{[neg]}$$

$$\frac{\begin{array}{ll} & \Gamma \vdash s_1 : \pi_1 \\ \wedge & \Gamma \vdash s_2 : \pi_2 \\ \wedge & \pi_1 \circ_\Gamma \pi_2 \rightsquigarrow \pi \end{array}}{\Gamma \vdash s_1 ; s_2 : \pi} \quad \text{[seq]}$$

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

$$\frac{c : \sigma \in \Gamma}{\Gamma \vdash c : \sigma \to \sigma} \quad \text{[congr.1]}$$

$$\frac{\begin{array}{ll} & f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in \Gamma \\ \wedge & \Gamma \vdash s_1 : \sigma_1 \to \sigma_1 \\ \wedge & \cdots \\ \wedge & \Gamma \vdash s_n : \sigma_n \to \sigma_n \end{array}}{\Gamma \vdash f(s_1, \ldots, s_n) : \sigma_0 \to \sigma_0} \quad \text{[congr.2]}$$

Figure 8: Many-sorted type-preserving strategies

Typing judgements 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 Γ. Here is a complete list of all well-formedness and well-typedness judgements:

- $\Gamma \vdash \tau$ (Well-formedness of term types)
- $\Gamma \vdash \pi$ (Well-formedness of strategy types)
- $\Gamma \vdash t : \tau$ (Well-typedness of terms)
- $\Gamma \vdash s @ t : \tau$ (Well-typedness of strategy applications)
- $\Gamma \vdash s : \pi$ (Well-typedness of strategies)

The corresponding deduction rules are shown in Figure 8. Since we assume explicit type declarations for the various kinds of variables and symbols, it suffices to propagate a fixed context Γ (as opposed to its synthesis). It is easy to define the type system in a different style. The type system for many-sorted type-preserving strategies should not be regarded as a contribution of the present article. It is rather straightforward. Let us read some inference rules for convenience. Type-preservation is postulated by the well-formedness judgement for strategy types (cf. [pi.1]). Rule [apply] says that a strategy application $s @ t$ is well-typed if the strategy s is of type $\tau \to \tau$, and the term t is of type τ. The strategies ϵ and δ have many types, namely any type $\tau \to \tau$ where $\Gamma \vdash \tau$ holds (cf. [id] and [fail]). In turn, compound strategies can also have many types. The strategy types for compound strategies are regulated by the rules [neg], [seq], [choice], and [congr.2]. The typing rules for negation and sequential composition (cf. [neg] and [seq]) refer to auxiliary judgements for negatable and composable types. There definition is straightforward for the initial case of many-sorted type-preserving strategies (cf. [negt.1] and [comp.1]). The compound strategy $s_1 + s_2$ for choice is well-typed if both strategies s_1 and s_2 are of a common type π. This common type constitutes the type of the choice.

Type annotations Strategies can be annotated by types. In Figure 9, the trivial extension for type annotations is formalised. An annotated strategy is of the form $s : \pi$. Well-typedness means that s is indeed of type π. As for the semantic level, reduction of $s : \pi$ directly resorts to s (cf. [type$^{+/-}$]). Thus, we say that the corresponding reduction rule is "neutral".

Syntax

$$s \quad ::= \quad \cdots \mid s : \pi$$

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

$$\frac{\Gamma \vdash s : \pi}{\Gamma \vdash s : \pi : \pi}\ [\mathsf{type}]$$

Reduction $\boxed{\Gamma \vdash s @ t \rightsquigarrow r}$

$$\frac{\Gamma \vdash s @ t \rightsquigarrow r}{\Gamma \vdash s : \pi @ t \rightsquigarrow r}\ [\mathsf{type}^{+/-}]$$

Figure 9: Type-annotated strategies

Properties We use S'_{tp} to denote the composition of S'_0 defined in Figure 6, and the type system from Figure 8. The following theorem is concerned with properties of S'_{tp}. It says that strategy types adhere to the scheme of type preservation, strategy applications are uniquely typed, and the reduction semantics is properly constrained by the type system.

Theorem 1 *The calculus S'_{tp} for many-sorted type-preserving strategies obeys the following properties:*

1. *Types of strategies adhere to the scheme of type-preservation, i.e.,*
   for all well-formed contexts Γ, strategies s and strategy types π:
   $\Gamma \vdash s : \pi$ implies π is of the form $\tau \to \tau$.

2. *Strategy applications satisfy unicity of typing, i.e.,*
   for all well-formed contexts Γ, strategies s, term types τ, τ' and terms t:
   $\Gamma \vdash s @ t : \tau \ \wedge\ \Gamma \vdash s @ t : \tau'$ implies $\tau = \tau'$.

3. *Reduction of strategy applications satisfies subject reduction, i.e.,*
   for all well-formed contexts Γ, strategies s, term types τ and terms t, t':
   $\Gamma \vdash s : \tau \to \tau \ \wedge\ \Gamma \vdash t : \tau \ \wedge\ s @ t \rightsquigarrow t'$ implies $\Gamma \vdash t' : \tau$.

$\diamond$

In the further development of S'_γ, we will use (refinements of) these properties to prove the formal status of the evolving type system. Unicity of typing and subject reduction are basic desirable properties of type systems (cf. [Bar92, Geu93, Sch94]). We can only claim unicity of typing for strategy applications but not for strategies themselves because of the typing rules for the constant combinators ϵ and δ. We believe that this is acceptable, for the moment. Informally, non-unique strategy types can be regarded as an encoding of parametric types for the combinators. Later we will recover unicity of typing by resorting to generic types. Unicity of typing for strategy application means that the result type of a strategy application is determined by the type of the input term (although the type of the strategy itself is not necessarily unique). Subject reduction means that if we initiate a reduction of a well-typed strategy application, then we can be sure that the resulting term reduct (if any) is of the prescribed type. The following proof is very verbose to prepare for the elaboration of the proof in the context of generic types.

Proof 1 *(IH abbreviates induction hypothesis in all the upcoming proofs.)*

1. *We show adherence to the scheme of type preservation by induction on s in $\Gamma \vdash s : \pi$.*

   Base cases*: Type preservation is directly enforced for rewrite rules, ϵ, δ, and congruences for constants by the corresponding typing rules (cf.* [rule], [id], [fail], *and* [congr.1]*), that is, the type position in the conclusion is instantiated according to the type-preserving form of strategy types.*

   Induction step*: Type preservation for $\neg\cdot$ (cf.* [neg]*) is implied by the rule* [negt.1] *for negatable types. Strictly speaking, we do not need to employ the IH since the type-preserving shape of the result type is enforced by* [negt.1] *regardless of the input type. As for $s_1; s_2$, the auxiliary judgement for composable types enforces type preservation (cf. $\cdots \circ_\Gamma \cdots \rightsquigarrow \tau \to \tau$ in* [comp.1]*). Again, the IH does not need to be employed. As for $s_1 + s_2$, the result type coincides with the argument types, and hence, type-preservation is implied by the IH. Finally, type preservation for congruences $f(s_1, \ldots, s_n)$ is directly enforced by the corresponding typing rule (cf. the type position in the conclusion of* [congr]*).*

2. *Let us first point out that unicity of typing obviously holds for terms because the inductive definition of $\Gamma \vdash t : \tau$ enforces a unique type τ for t. Here, it is, of course, essential that we ruled out overloading of function and constant symbols, and variables. According to the rule* [apply]*, the result type of a strategy application is equal to the type of the input term. Hence, $s @ t$ is uniquely typed.*

3. *In the type-preserving setting, subject reduction actually means that the reduction semantics for strategy applications is type-preserving (as prescribed by the type system), i.e., if the reduction of a strategy application $s @ t$ with $s : \tau \to \tau$, $t : \tau$ yields a proper term reduct t', then t' is also of type τ. We show this property by induction on s in $s @ t \rightsquigarrow r$ while we assume $\Gamma \vdash s : \tau \to \tau$ and $\Gamma \vdash t : \tau$. To this end, it is crucial to maintain that the IH can only be employed for a premise $s_i @ t_i \rightsquigarrow t'_i$ and a corresponding type τ_i, if we can prove the following side condition:*

   $$\Gamma \vdash s : \tau \to \tau \ \wedge\ \Gamma \vdash t : \tau \text{ implies } \Gamma \vdash s_i : \tau_i \to \tau_i \ \wedge\ \Gamma \vdash t_i : \tau_i$$

   The judgements $\Gamma \vdash s_i : \tau_i \to \tau_i$ and $\Gamma \vdash t_i : \tau_i$ have to be approved by consulting the corresponding typing rules (relating t and t_i, or s and s_i, respectively), and by other means. Note there are no proof obligations for reduction rules which do not yield a proper term reduct, namely for negative rules. In particular, there is no case for δ in the sequel.

   Base cases*: As for rewrite rules, we know that both the left-hand side t_l and the right-hand side t_r are of type τ as prescribed by* [rule]*. The substitution θ in* [rule$^+$] *preserves the type of the right-hand side (implied by basic properties of many-sorted unification and substitution). Hence, rule application is type-preserving. ϵ preserves the very input term, and hence, it is type-preserving. The same holds for congruences for constants.*

Induction step: *Negation is type-preserving because the very input term is preserved as for ϵ. Thus, we do not even need to employ the IH for s in $\neg\, s$. In fact, the IH tells us here that we do not even attempt to apply s in an ill-typed manner. Let us consider sequential composition $s_1; s_2 \,@\, t$. By the rules* [seq] *and* [comp.1]*, we know that the types of s_1, s_2 and $s_1; s_2$ coincide, namely $\tau \to \tau$. We want to show that t'' in $s_1; s_2 \,@\, t \rightsquigarrow t''$ is of the same type as t. Since s_1 must be of the same type as $s_1; s_2$, the IH is enabled for $s_1 \,@\, t \rightsquigarrow t'$. Thereby, we know that t' is of type τ, Since we also know that s_2 must be of the same type as $s_1; s_2$, the IH is enabled for the second premise $s_2 \,@\, t' \rightsquigarrow t''$. Hence, t'' is of the same type as t, and sequential composition is type-preserving. As for choice, reduction of $s_1 + s_2 \,@\, t$ directly resorts to either $s_1 \,@\, t$ or $s_2 \,@\, t$ (cf.* [choice$^+$.1] *and* [choice$^+$.2]*). We also know that s_1, s_2 and $s_1 + s_2$ have to be of the same type (cf.* [choice]*). Hence, the IH is enabled for the reduction of the chosen strategy, be it s_1 or s_2. Finally, let us consider congruence strategies where $f(s_1, \ldots, s_n) \,@\, f(t_1, \ldots, t_n)$ is reduced to $f(t'_1, \ldots, t'_n)$ while the t'_i are obtained by the reduction of the $s_i \,@\, t_i$ (cf.* [congr.2]*). Let $f : \sigma_1 \times \cdots \times \sigma_n \to \sigma$ be in Γ. Then, we know that for a well-typed term $f(t_1, \ldots, t_n)$, the t_i must be of type σ_i (cf.* [fun]*). We also know that for a well-typed strategy $f(s_1, \ldots, s_n)$, the s_i must be of type $\sigma_i \to \sigma_i$ (cf.* [congr.2]*). Hence, the IH is enabled for the various $s_i \,@\, t_i \rightsquigarrow t'_i$. Then, the type of $f(t'_1, \ldots, t'_n)$ is the same as of $f(t_1, \ldots, t_n)$.*

$\diamond$

As an aside, the proof of subject reduction was certainly simplified by the fact that (possibly recursive) strategy definitions were omitted. The use of simple induction on s is enabled by the strong normalisation of the purely inductive reduction semantics for strategy applications. If (recursive) strategy definitions were included, proof by induction on the depth of computations is needed. The use of static contexts also simplifies our proofs.

3.3 Type-changing strategies

If a fixed evaluation or normalisation strategy is assumed in rewriting, type-preservation is indispensable. It does not make sense to repeatedly look for a redex and apply some rewrite rule, if this would change the type of the redex location. In strategic rewriting, it is no longer necessary to insist on type-preserving rewrite rules (or strategies in general). Type-changing strategies (including rewrite rules) can be applied in a disciplined manner making sure that changed subterms are properly combined without causing ill-typed terms in between.

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

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

Application $\boxed{\Gamma \vdash s \,@\, t : \tau}$

$$\frac{\Gamma \vdash s : \tau \to \tau' \;\wedge\; \Gamma \vdash t : \tau}{\Gamma \vdash s \,@\, t : \tau'} \quad [\mathsf{apply}]$$

Negatable types $\boxed{\neg_\Gamma\, \pi \;\rightsquigarrow\; \pi'}$

$$\frac{\Gamma \vdash \tau \;\wedge\; \Gamma \vdash \tau'}{\neg_\Gamma\, \tau \to \tau' \;\rightsquigarrow\; \tau \to \tau} \quad [\mathsf{negt.1}]$$

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

$$\frac{\Gamma \vdash t_l : \tau \;\wedge\; \Gamma \vdash t_r : \tau'}{\Gamma \vdash t_l \to t_r : \tau \to \tau'} \quad [\mathsf{rule}]$$

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

$$\frac{\Gamma \vdash \tau \;\wedge\; \Gamma \vdash \tau' \;\wedge\; \Gamma \vdash \tau''}{\tau \to \tau' \;\circ_\Gamma\; \tau' \to \tau'' \;\rightsquigarrow\; \tau \to \tau''} \quad [\mathsf{comp.1}]$$

Figure 10: Refinement of S'_{tp} to enable type-changing strategies

Type system update In Figure 10, the type system for type-preserving strategies is updated to enable type-changing strategies. To this end, we replace rule [pi.1] to characterise potentially type-changing strategies as well-formed. We also replace the rule [apply] for strategy application, and the rule [rule] to embrace type-changing strategies. Furthermore, the auxiliary judgements for negatable and composable strategy types have to be generalised accordingly (cf. [negt.1] and [comp.1]). All the other typing rules carry over from S'_{tp}. We use S'_{tc} to denote the refinement of S'_{tp} according to Figure 10.

Theorem 2 *The calculus S'_{tc} for potentially type-changing strategies obeys the following properties:*

1. *Co-domains of strategies are determined by domains. i.e.,*
   for all well-formed contexts Γ, strategies s and term types τ_1, τ'_1, τ_2, τ'_2:
   $\Gamma \vdash s : \tau_1 \to \tau'_1 \ \wedge \ \Gamma \vdash s : \tau_2 \to \tau'_2 \ \wedge \ \tau'_1 \neq \tau'_2$ *implies* $\tau_1 \neq \tau_2$.
2. *Strategy applications satisfy unicity of typing, i.e., ... (cf. Theorem 1).*
3. *Reduction of strategy applications satisfies subject reduction, i.e.,*
   for all well-formed contexts Γ, strategies s, term types τ, τ' and terms t, t':
   $\Gamma \vdash s : \tau \to \tau' \ \wedge \ \Gamma \vdash t : \tau \ \wedge \ s \mathbin{@} t \rightsquigarrow t'$ *implies* $\Gamma \vdash t' : \tau'$.

$\diamond$

The first property is the necessary generalisation of adherence to the scheme of type preservation in Theorem 1. We require that the co-domain of a strategy type is uniquely determined by its domain, that is, there might be different types for a strategy, but once the type of the input term is fixed, the type of the result is determined. We also generalised the subject reduction property compared to Theorem 1 in order to cover type-changing strategies.

Proof 2

1. *Note that the property trivially holds for type-preserving strategies. We show the property by induction on s in $\Gamma \vdash s : \pi$.*

   Base cases*: The co-domain of a rewrite rule is even uniquely defined (regardless of the domain) as an implication of unicity of typing for terms. The remaining base cases are type-preserving, and hence, they are trivial.*

   Induction step*: Negation is type-preserving, and hence, unicity of co-domains holds trivially. As for sequential composition, the domain of $s_1; s_2$ coincides with the domain of s_1, the co-domain of s_1 coincides with the domain of s_2, and the co-domain of s_2 coincides with the co-domain of $s_1; s_2$ (cf.* [comp.1]*). Since unicity of co-domains holds for s_1 and s_2 by the IH, the co-domain of $s_1; s_2$ is (transitively) determined by its domain. As for choice, the property follows from the IH and from the strict coincidence of the types of s_1, s_2, and $s_1 + s_2$ (cf.* [choice]*). Congruences $f(s_1, \ldots, s_n)$ are type-preserving, and hence, unicity of co-domains holds trivially.*

2. *The simple argument from Proof 1 regarding rule* [apply] *can be generalised as follows. The domain of the strategy in $s \mathbin{@} t$ needs to coincide with the type of t. Thus, by unicity of co-domains, we know that the type of the reduct is determined.*

3. *We need to elaborate our induction proof for Proof 1 where we argued that subject reduction (for type-preserving strategies) can be proved by showing that the reduction semantics is type-preserving, too. Now, for strategies, which are potentially type-changing, we need to show that reduction obeys the strategy types. Hence, the side condition for the employment of the IH also has to be revised. That is, the IH can be employed for a premise $s_i \mathbin{@} t_i \rightsquigarrow t'_i$ and corresponding types τ_i and τ'_i, if we can prove the following side condition:*

   $$\Gamma \vdash s : \tau \to \tau' \ \wedge \ \Gamma \vdash t : \tau \text{ implies } \Gamma \vdash s_i : \tau_i \to \tau'_i \ \wedge \ \Gamma \vdash t_i : \tau_i$$

Base cases*: Subject reduction for rewrite rules is implied by basic properties of many-sorted unification and substitution. That is, the substitution of variables on the right-hand side according to the obtained unifier θ preserves the type of the right-hand side (which can of course be different from the type of the left-hand side and the type of the input term). The remaining base cases are type-preserving, and hence, they are covered by Proof 1.*

Induction step*: The strategy s in $\neg\, s$ is not necessarily type-preserving anymore but the positive rule for $\neg\, s$ still exposes type-preservation as prescribed by the type system (cf.* [negt.1]*). Let us consider sequential composition. We start from the assumptions $\Gamma \vdash s_1; s_2 : \tau \to \tau''$ and $\Gamma \vdash t : \tau$. We want to show that t'' in $s_1; s_2 @ t \rightsquigarrow t''$ is of type τ''. There must exist a τ' such that $\Gamma \vdash s_1 : \tau \to \tau'$ and $\Gamma \vdash s_2 : \tau' \to \tau''$ (by* [comp.1] *and* [seq]*). In fact, unicity of co-domains implies that τ' is uniquely defined. We apply the IH for $s_1 @ t$, and hence, we obtain that the reduction of $s_1 @ t$ delivers a term t' of type τ'. This enables the IH for the second operand of sequential composition. Hence, we obtain that the reduction of $s_2 @ t'$ delivers a term t'' of type τ''. As for choice, the arguments from Proof 1 are still valid since we did not rely on type-preservation. That is, we know that the reduction of the choice directly resorts to one of the argument strategies, and the type of the choice has the same type as the two argument strategies. Hence, subject reduction for the choice follows from the IH. Congruences $f(s_1, \ldots, s_n)$ and the involved arguments are type-preserving, and hence, subject reduction carries over from Proof 1.*

$\diamond$

Syntax

$$
\begin{array}{lcl}
t & ::= & \cdots \mid \langle\rangle \mid \langle t_1, t_2\rangle \\
s & ::= & \cdots \mid \langle\rangle \mid \langle s_1, s_2\rangle \\
\tau & ::= & \cdots \mid \langle\rangle \mid \langle \tau_1, \tau_2\rangle
\end{array}
$$

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

$$\Gamma \vdash \langle\rangle \qquad \text{[tau.2]}$$

$$\frac{\Gamma \vdash \tau_1 \ \wedge\ \Gamma \vdash \tau_2}{\Gamma \vdash \langle \tau_1, \tau_2\rangle} \qquad \text{[tau.3]}$$

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

$$\Gamma \vdash \langle\rangle : \langle\rangle \qquad \text{[empty-tuple]}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \ \wedge\ \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \langle t_1, t_2\rangle : \langle \tau_1, \tau_2\rangle} \qquad \text{[pair]}$$

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

$$\Gamma \vdash \langle\rangle : \langle\rangle \qquad \text{[congr.3]}$$

$$\frac{\Gamma \vdash s_1 : \tau_1 \to \tau_1' \ \wedge\ \Gamma \vdash s_2 : \tau_2 \to \tau_2'}{\Gamma \vdash \langle s_1, s_2\rangle : \langle \tau_1, \tau_2\rangle \to \langle \tau_1', \tau_2'\rangle} \qquad \text{[congr.4]}$$

Reduction of congruences is defined precisely as for ordinary many-sorted constant and function symbols

Figure 11: Tuple types, tuples, and tuple congruences

3.4 Polyadic strategies

As we motivated in Section 2, we want to employ tuples to describe polyadic strategies, that is, strategies which process several terms. In principle, the following extension for tuples can be composed with both S'_{tp} and S'_{tc}. However, tuples are only vital in S'_{tc} where type-changing strategies are enabled. In

fact, strategies which operate on several terms are typically type-changing. In particular, congruences on pairs are very useful to perform type-changing strategies on a number of terms simultaneously.

In Figure 11, we extend the basic calculus S_0' with concepts for polyadic strategies in a straightforward manner. There are distinguished symbols $\langle\rangle$ for the empty tuple, and $\langle\cdot,\cdot\rangle$ for pairing of terms. We use the same notation for tuple types, tuples, and congruences on tuples. That is, we introduce two special forms of term types namely $\langle\rangle$ and $\langle\tau_1,\tau_2\rangle$ (cf. [tau.2] and [tau.3]). The well-typedness judgement for terms is extended accordingly (cf. [empty−tuple] and [pair]). We also introduce special typing rules for congruences on tuples (cf. [congr.3] and [congr.4]). Note that the typing rules for congruences for ordinary symbols relied on a context lookup (cf. [congr.1] and [congr.2] in Figure 8) while this is not the case for congruences on tuples. Moreover, congruences on pairs can be type-changing (cf. [congr.4]) whereas this is not an option for many-sorted congruences. Finally, note the way how well-formed terms are defined in the extended type system (i.e., S_{tc}' + Figure 11). Many-sorted terms cannot contain tuples, but only the other way around. This restriction models that tuples are solely intended for argument and result lists in the sense of Cartesian products while the actual data should be purely many-sorted.

Example 14 *The strategy* ADD *from Example 4 and the strategy* COUNT *from Example 6 are well-typed as type-changing strategies.*

$$\begin{aligned} \textsc{Add} &: \langle \mathit{Nat}, \mathit{Nat}\rangle \to \mathit{Nat} \\ \textsc{Count} &: \mathit{Tree} \to \mathit{Nat} \end{aligned}$$

As for ADD*, we rely on tuple types since addition is encoded as a strategy which takes a pair of naturals (and returns the sum). As for* COUNT*, the definition from Example 6 involves a type-changing congruence on pairs, namely* $\langle\textsc{Count},\textsc{Count}\rangle$*. This congruence applies* COUNT *to the two subtrees of a fork-tree independently.* ◇

4. GENERIC STRATEGIES

Recall the distinguishing property of generic traversal strategies. They are supposed to be applicable to terms of any sort. 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 many-sorted terms. In fact, ϵ and δ are also generic in the aforementioned sense. Note however, that the parameters of the traversal primitives have to be generic, too. Consider, for example, $\Box(s)$. The argument s must be potentially applicable to subterms of any sort. Hence, a rewrite rule is not appropriate for s.

In the present section, we extend our basic calculus for many-sorted strategies by types and combinators for generic strategies. First, we spell out the reduction rules for type-preserving combinators, and we formalise the corresponding generic type TP. Then, the problem of mediation between many-sorted and generic strategies is addressed. There are two directions for mediation. When we qualify a many-sorted strategy to become generic, then we perform extension. When we make the type of a generic strategy more specific, then we perform restriction. Afterwards, we define the type-unifying traversal combinators and the corresponding generic type (constructor) TU(·).

4.1 Strategies of type TP

Combinators In Figure 12, we define the reduction semantics of the generic traversal primitives $\Box(\cdot)$ and $\Diamond(\cdot)$ adopted from system S. The rule [all$^+$.1] says that $\Box(s)$ 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 reduction scheme for $\Diamond(s)$ is similar. The rule [one$^+$] says that s is applied to some subterm t_i of $f(t_1,\ldots,t_n)$. The negative rule [one$^-$.1] says that $\Diamond(s)$ applied to a constant fails because there are no children which s could be applied to. The negative rule [one$^-$.2] says that $\Diamond(s)$ fails if s fails for all children of $f(t_1,\ldots,t_n)$. Dually, $\Box(s)$ fails if s fails for some child (cf. [all$^-$]).

Syntax

$$s \quad ::= \quad \cdots \mid \Box(s) \mid \Diamond(s)$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow t'}$

Positive rules

$$\Box(s) \mathbin{@} c \rightsquigarrow c \qquad [\mathsf{all}^+.1]$$

$$\frac{\begin{array}{rl} & s \mathbin{@} t_1 \rightsquigarrow t'_1 \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow t'_n \end{array}}{\Box(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow f(t'_1, \ldots, t'_n)} \qquad [\mathsf{all}^+.2]$$

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

Negative rules

$$\frac{\exists i \in \{1, \ldots, n\}. s \mathbin{@} t_i \rightsquigarrow \uparrow}{\Box(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \qquad [\mathsf{all}^-]$$

$$\Diamond(s) \mathbin{@} c \rightsquigarrow \uparrow \qquad [\mathsf{one}^-.1]$$

$$\frac{\begin{array}{rl} & s \mathbin{@} t_1 \rightsquigarrow \uparrow \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow \uparrow \end{array}}{\Diamond(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \qquad [\mathsf{one}^-.2]$$

Figure 12: Generic (intentionally type-preserving) traversal combinators

The generic type TP We establish a syntactical domain γ of generic types. We integrate γ into the grammar for types by stating that γ corresponds to another form of strategy types π (complementing strategy types of the form $\tau \to \tau'$). We start the definition of γ with the generic type TP denoting all *T*ype-*P*reserving strategies.

Syntax

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

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

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

Genericity $\boxed{\pi \prec_\Gamma \pi'}$

$$\frac{\Gamma \vdash \tau}{\tau \to \tau \prec_\Gamma \mathsf{TP}} \qquad [\mathsf{less}.1]$$

Negatable types $\boxed{\neg_\Gamma \pi \rightsquigarrow \pi'}$

$$\frac{\Gamma \vdash \gamma}{\neg_\Gamma \gamma \rightsquigarrow \mathsf{TP}} \qquad [\mathsf{negt}.2]$$

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

$$\frac{\Gamma \vdash \gamma}{\mathsf{TP} \circ_\Gamma \gamma \rightsquigarrow \gamma} \qquad [\mathsf{comp}.2]$$

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

$$\Gamma \vdash \epsilon : \mathsf{TP} \qquad [\mathsf{id}]$$

$$\Gamma \vdash \delta : \mathsf{TP} \qquad [\mathsf{fail}]$$

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

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

Figure 13: TP—The type of generic type-preserving strategies

In Figure 13, we extend the typing judgements to formalise TP, and to employ TP for the relevant combinators. The constant combinators ϵ and δ are defined to be (generic) type-preserving strategies

(cf. [id] and [fail]). As for negation, we add another rule to the auxiliary judgement $\neg_\Gamma \pi \rightsquigarrow \pi'$ for negatable types. The enabled form of negation is concerned with generic strategies. The rule [negt.2] states that any strategy of a generic type γ can be negated. As for sequential composition, we also add a rule to the auxiliary judgement $\pi_1 \circ_\Gamma \pi_2 \rightsquigarrow \pi$ to cover the case that a generic type-preserving strategy and another generic strategy are composed (cf. [comp.2]). The typing rules for the traversal combinators $\Box(\cdot)$ and $\Diamond(\cdot)$ expose that they can be used to derive a strategy of type TP from an argument strategy of type TP (cf. [all] and [one]).

We use a relation $\prec_\Gamma$ on types to characterise generic types, say the genericity of types. The relation $\preceq_\Gamma$ denotes the reflexive closure of $\prec_\Gamma$. By $\pi \prec_\Gamma \pi'$, we mean that π' is more generic than π. If we view generic types as type schemes, we can say that $\pi \prec_\Gamma \pi'$ means the following: The type π is an instance of the type scheme π'. Rule [less.1] axiomatises TP. The rule says that $\tau \to \tau$ is an instance of TP for all well-formed τ. This reading indeed suggests to consider TP as the type scheme $\forall\alpha.\ \alpha \to \alpha$. In general, we assume that a proper generic type should admit an instance for every possible term type since a generic strategy should be applicable to terms of all sorts. To be precise, there should be exactly one instance per term type. This property obviously holds for TP. For convenience, we summarise the requirements for generic types in the following definition.

Definition 1 *γ is a* proper generic type *if for all well-formed Γ such that $\Gamma \vdash \gamma$, and for all τ such that $\Gamma \vdash \tau$, there is a precisely one τ' with $\Gamma \vdash \tau'$ such that $\tau \to \tau' \prec_\Gamma \gamma$.* $\diamond$

It is an essential property that there is only one instance per term type. Otherwise, the type of the application of a generic strategy would be ambiguous. Note that there are now two levels in our type system, that is, there are many-sorted types and generic types. The type system strictly separates many-sorted strategies (such as rewrite rules) and generic strategies (such as applications of $\Box(\cdot)$). Since there are no further intermediate levels of genericity, there are only chains of length 1 in the partial order $\preceq_\Gamma$. In Section 5.4, we will consider a possible sophistication of S'_γ to accomplish overloaded strategies. This elaboration will make the relation $\preceq_\Gamma$ more vital.

As the type system stands, we cannot turn many-sorted strategies into generic ones, nor the other way around. Also, strategy application, as it was defined for S'_{lp} and S'_{lc}, only copes with many-sorted strategies s in $s @ t$. The type system should allow us to apply a generic strategy to any term. We will now develop the corresponding techniques for extension and restriction.

4.2 Strategy extension

Now that we have typed generic traversal combinators at our disposal, we also want to inhabit the generic type TP. So far, we only have two trivial constants of type TP, namely ϵ and δ. We would like to construct generic strategies from rewrite rules. We will formalise the corresponding combinator $\cdot \lhd \cdot$. To this end, we also examine other approaches, and we justify the design of $\cdot \lhd \cdot$.

Infeasible approaches One (infeasible) approach to the inhabitation of generic strategy types like TP is that the typing requirements for generic strategy parameters (of primitive or derived strategy combinators) would be relaxed via $\cdot \preceq_\Gamma \cdot$. Whenever we require a generic strategy argument, e.g., for s in $\Box(s)$, we also would accept a many-sorted one. Striking problems with this implicit approach are that the corresponding type dependency is in conflict with static type safety, and that applicability failures of many-sorted strategies in generic contexts are not approved by the strategic programmer. Consider, for example, the strategy $\Box(s)$ where s is of some type $\tau \to \tau$. For the sake of subject reduction (say, type-safe application of strategies), the semantics of $\Box(s)$ had to ensure that every child at hand is of type τ before it even attempts to apply s to it. If a certain child is not of type τ, $\Box(s)$ should fail. From the programmer's point of view, the approach makes it indeed too easy for many-sorted strategies to get applied in a generic context. Many-sorted strategies are likely to fail or to succeed in a trivial way when they are applied in a generic context. By contrast:

> We envision a statically type-safe style of strategic programming, where the employment of many-sorted strategies (such as rewrite rules) in generic contexts is approved by the

programmer. Moreover, the corresponding calculus should correspond to a conservative extension of S'_{tp} (or S'_{tc}).

Another (infeasible) approach to the inhabitation of generic strategy types is to resort to a choice combinator ($\cdot + \cdot$ or $\cdot \Leftarrow \cdot$) to compose a many-sorted strategy and a generic default. We might attempt to turn, for example, a rewrite rule ℓ into a generic strategy using the forms $\ell \Leftarrow \epsilon$ or $\ell \Leftarrow \delta$. We could assume that the result type of a choice corresponds to the least upper bound (w.r.t. $\preceq_\Gamma$) of the argument types. One possible argument to refuse such an style arises from the following simple derivation:

$$s \Leftarrow \delta \quad \rightsquigarrow \quad s + \neg\, s; \delta \quad \rightsquigarrow \quad s + \delta \quad \rightsquigarrow \quad s$$

That is, δ is the right unit of $\cdot \Leftarrow \cdot$.[10] This derivation complies with the reduction semantics for strategies. Since the derivation resembles desirable algebraic laws of choice and failure, we should assume that all strategies in the derivation are of the same type. This is in conflict with the idea to use $\cdot \Leftarrow \cdot$ or $\cdot + \cdot$ to inhabit generic types. Furthermore, the approach is also not convenient because of its impact on the reduction semantics. We had to redefine the semantics of $\cdot + \cdot$ to make sure that the argument strategies are applied only if their type and the type of the term at hand fits. This is again in conflict with the idea of a conservative extension. More generally, the combinators $\cdot + \cdot$ and $\cdot \Leftarrow \cdot$ are concerned with choice controlled by success and failure, and we should not confuse this notion with the problem of creating generic strategies. This confusion is unavoidable in untyped settings such as Stratego or system S.

Inhabitation by extension The combinator $\cdot \lhd \cdot$ serves for the explicit qualification of a many-sorted strategy. In a sense, the domain of the strategy s in $s \lhd \mathsf{TP}$ is extended to all possible term types. Suppose the type of s is $\tau \to \tau$. Then, of course, s can only be applied in a type-safe manner to terms of sort τ. It is the very meaning of $s \lhd \mathsf{TP} \mathbin{@} t$ to apply s if and only if t is of sort τ. Otherwise, $s \lhd \mathsf{TP} \mathbin{@} t$ fails. Hence, s is extended in a trivial sense, that is, to behave like δ for all sorts different from τ. Well-typedness and the reduction semantics of the combinator $\cdot \lhd \cdot$ are defined in Figure 14. As an aside, the definitions are somewhat more general that one might expect for the simple two-level genericity relation so far.

Type dependency The combinator $\cdot \lhd \cdot$ makes it explicit where we want to become generic. There is no hidden way how many-sorted ingredients may become generic (accidentally). A semantical consequence of this approach is that type dependency is very local and easy to realise. As the reduction rules for $\cdot \lhd \cdot$ clearly point out, reduction is truly type-dependent. That is, the types of the strategy s and the term t in $s \lhd \pi \mathbin{@} t$ are determined and compared to each other, and further reduction depends on the comparison. To this end, the typing context Γ is also part of the reduction judgement. We assume that all previous rules for reduction are lifted to the new form of judgement by propagating Γ. Otherwise, all rules stay intact, and hence we may claim that the incorporation of $\cdot \lhd \cdot$ corresponds to a conservative extension. The required type-dependent reduction for $\cdot \lhd \cdot$ might be regarded as a slight paradigm shift. However, conceptually, this semantics is sensible and essential for type-safe application. One should not confuse type dependency with lack of static type safety. Strategies are statically typed in S'_γ. Type dependency merely means that a generic strategy admits different behaviours for different sorts. As an aside, in Section 5.2, we will discuss an approach to eliminate the typing judgements in the reduction rules for $\cdot \lhd \cdot$ (although a certain type dependency remains). The basic idea is to employ a static elaboration judgement to include sufficient type information in the elaborated strategy.

[10] It is also the left unit of $\cdot \Leftarrow \cdot$, and both the left and the right unit of $\cdot + \cdot$.

Syntax

$$s \quad ::= \quad \cdots \mid s \lhd \pi$$

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

$$\frac{\Gamma \vdash s : \pi' \ \wedge \ \pi' \prec_\Gamma \pi}{\Gamma \vdash s \lhd \pi : \pi} \ [\mathsf{extend}]$$

Reduction $\boxed{\Gamma \vdash s \mathbin{@} t \rightsquigarrow r}$

Positive rule

$$\frac{\begin{array}{ll} & \Gamma \vdash s : \pi' \\ \wedge & \Gamma \vdash t : \tau \\ \wedge & \exists \tau'.\ \tau \to \tau' \preceq_\Gamma \pi' \\ \wedge & \Gamma \vdash s \mathbin{@} t \rightsquigarrow t' \end{array}}{\Gamma \vdash s \lhd \pi \mathbin{@} t \rightsquigarrow t'} \ [\mathsf{extend}^+]$$

Negative rules

$$\frac{\begin{array}{ll} & \Gamma \vdash s : \pi' \\ \wedge & \Gamma \vdash t : \tau \\ \wedge & \exists \tau'.\ \tau \to \tau' \preceq_\Gamma \pi' \\ \wedge & \Gamma \vdash s \mathbin{@} t \rightsquigarrow \uparrow \end{array}}{\Gamma \vdash s \lhd \pi \mathbin{@} t \rightsquigarrow \uparrow} \ [\mathsf{extend}^-.1]$$

$$\frac{\begin{array}{ll} & \Gamma \vdash s : \pi' \\ \wedge & \Gamma \vdash t : \tau \\ \wedge & \nexists \tau'.\ \tau \to \tau' \preceq_\Gamma \pi' \end{array}}{\Gamma \vdash s \lhd \pi \mathbin{@} t \rightsquigarrow \uparrow} \ [\mathsf{extend}^-.2]$$

Figure 14: Turning many-sorted strategies into generic ones

4.3 Restriction

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 specific contexts. Actually, there is not just one way to accommodate restriction. Compared to extension, restriction is conceptually much simpler since restriction is immediately type safe without further precautions. In general, we are used to the idea that a generic entity is used in a specific context, e.g., in the sense of parametric polymorphism.

Explicit restriction We consider a strategy combinator $s \rhd \pi$ which has no semantic effect, but at the level of typing it allows us to consider a (generic) strategy s to be of type π provided it holds $\pi \prec_\Gamma \pi'$ where π' is the actual type of s. Explicit restriction is defined in Figure 15 following very much the scheme of type annotations as covered in Figure 9. The combinator for restriction is immediately sufficient if we want to apply a generic strategy s to a term t of a certain sort τ. If we assume, for example, that s is of type TP, then the well-typed strategy application $s \rhd \tau \to \tau \mathbin{@} t$ can be employed. Thus, the rule [apply] for strategy applications from Figure 8 (or the updated rule from Figure 10) can be retained without modifications. Restriction is also used in the composition of strategies.

Syntax

$$s \quad ::= \quad \cdots \mid s \rhd \pi$$

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

$$\frac{\Gamma \vdash s : \pi' \ \wedge \ \pi \prec_\Gamma \pi'}{\Gamma \vdash s \rhd \pi : \pi} \ [\mathsf{restrict}]$$

Reduction $\boxed{\Gamma \vdash s \mathbin{@} t \rightsquigarrow r}$

$$\frac{\Gamma \vdash s \mathbin{@} t \rightsquigarrow r}{\Gamma \vdash s \rhd \pi \mathbin{@} t \rightsquigarrow r} \ [\mathsf{restrict}^{+/-}]$$

Figure 15: Explicit strategy restriction

In a sense, the three forms $s \rhd \pi$ (explicit restriction), $s : \pi$ (type annotation), and $s \lhd \pi$ (strategy extension) complement each other as they deal with the different ways how a strategy and a type can be related to each other via the partial order $\preceq_\Gamma$. There is another interesting twist between these

three forms. Type annotations can be removed without any effect on well-typedness and semantics. By contrast, a replacement of a restriction $s \rhd \pi$ by s will result in an ill-typed program, although s is semantically equivalent to $s \rhd \pi$. Finally, a replacement of an extension $s \lhd \pi$ by s will not just harm well-typedness, but an ultimate application of s is not even necessarily type-safe.

Inter-mezzo The concepts which we have explained so far are sufficient to assemble a calculus S'_{TP} which covers generic type-preserving traversals. Generic type-preserving strategies could be combined with both S'_{tp} and S'_{tc}. For simplicity, we have chosen the former as the starting point for S'_{TP}. For convenience, we summarise all ingredients of S'_{TP}:

- The basic calculus S'_0 (cf. Figure 6)
- Many-sorted type-preserving strategies (cf. Figure 8)
- Generic traversal primitives $\Box(\cdot)$ and $\Diamond(\cdot)$ (cf. Figure 12)
- The generic type TP (cf. Figure 13)
- Strategy extension (cf. Figure 14)
- Explicit restriction (cf. Figure 15)

In Theorem 1, and Theorem 2, we started to address properties of the many-sorted fragments S'_{tp} and S'_{tc} of S'_{γ}. Let us update the theorem for S'_{TP} accordingly.

Theorem 3 *The calculus S'_{TP} for (many-sorted and generic) type-preserving strategies obeys the following properties:*

1. *Strategies satisfy unicity of typing, and their types adhere to the scheme of type-preservation, i.e., for all well-formed contexts Γ, strategy types π, π' and strategies s:*
   (a) $\Gamma \vdash s : \pi \ \wedge \ \Gamma \vdash s : \pi'$ *implies* $\pi = \pi'$, *and*
   (b) $\Gamma \vdash s : \pi$ *implies* $\pi \preceq_\Gamma \mathsf{TP}$.
2. *Strategy applications satisfy unicity of typing, i.e., ... (cf. Theorem 1).*
3. *Reduction of strategy applications satisfies subject reduction, i.e., for all well-formed contexts Γ, strategies s, terms t, t', term types τ and strategy types π:*
   $\Gamma \vdash s : \pi \ \wedge \ \Gamma \vdash t : \tau \ \wedge \ \tau \to \tau \preceq_\Gamma \pi \ \wedge \ \Gamma \vdash s \mathbin{@} t \rightsquigarrow t'$ *implies* $\Gamma \vdash t' : \tau$.

$\diamond$

This theorem is an elaboration of Theorem 1 for many-sorted type-preserving strategies. The first property is strengthened since we now claim unicity of typing for strategies. At the same time, we also need to rephrase what it means that a strategy type adheres to the scheme of type-preservation. The definition of subject reduction is also affected by the introduction of generic types.

Proof 3

1. (a) *We prove this property by induction on s in $\Gamma \vdash s : \pi$.*
   Base cases*: Rewrite rules are uniquely typed by unicity of typing for terms. The other base cases trivially satisfy unicity of typing (by simple inspection of the type position in the conclusion of* [id], [fail], *and* [congr.1]*).*
   Induction step*: Unicity of typing for $\neg s$ is essentially implied by the IH since the auxiliary judgement for negatable encodes a function from the argument type to the type of the negated strategy. Unicity of typing for $s_1; s_2$ and $s_1 + s_2$ is immediately implied by the IH. In both cases, the type of the compound strategy coincides with the type of the arguments (cf.* [comp.1], [comp.2], [seq], [choice]*). Unicity of typing for $f(s_1, \ldots, s_n)$, $\Box(s)$, $\Diamond(s)$, $s \lhd \pi$ and $s \rhd \pi$ holds trivially (by simple inspection of the type position in the conclusion of the corresponding typing rules).*

(b) *We prove this property by induction on s in $\Gamma \vdash s:\pi$. We only need to cover the cases which were updated or newly introduced in the migration from S'_{tp} to S'_{TP}.*

Base cases*: The types of ϵ and δ are uniquely defined as* TP*, and hence, they trivially adhere to the required scheme.*

Induction step*: The type of $\Box(s)$ and $\Diamond(s)$ is defined as* TP*. The forms $\neg\, s$ and $s_1; s_2$ where type-preserving in S'_{tp}. There are still type-preserving in S'_{TP} since they added rules for negatable and composable types only admit* TP *as result type of negation and sequential composition (cf.* [negt.2] *and* [comp.2]*). The type π in $s \lhd \pi$ (and hence the type of $s \lhd \pi$ itself) must coincide with* TP *since this is the only possible strategy type admitted by $\prec_\Gamma$ in S'_{TP} (cf.* [extend] *and* [less.1]*). Dually, the type π in $s \rhd \pi$ (and hence the type of $s \rhd \pi$ itself) must coincide with a type of the form $\tau \to \tau$ because these are the only types of strategies in S'_{TP} admitted by $\prec_\Gamma$.*

2. *The property follows immediately from*

   - *unicity of typing for terms,*
   - *unicity of typing for strategies, and*
   - *the fact that* TP *is a proper generic type (cf. Definition 1).*

3. *Note that we only deal with type-preserving strategies which allows us to adopt Proof 1 to a large extent. Of course, the side condition for the employment of the IH has to be revised. That is, the IH can be employed for a premise $s_i @ t_i \rightsquigarrow t'_i$ and corresponding types τ_i and π_i if we can prove the following side condition:*

$$\Gamma \vdash s:\pi \ \wedge\ \Gamma \vdash t:\tau \ \wedge\ \tau \to \tau \preceq_\Gamma \pi \text{ implies } \Gamma \vdash s_i:\pi_i \ \wedge\ \Gamma \vdash t_i:\tau_i \ \wedge\ \tau_i \to \tau_i \preceq_\Gamma \pi_i$$

   Base cases*: Proof 1 is still applicable to rewrite rules and congruences for constants since these forms of strategies were completely preserved in S'_{TP} (compared to S'_{tp}). The strategy ϵ is said to be generically type-preserving according to* [id] *(while it was overloaded before). Reduction of $\epsilon @ t$ yields t. Hence, reduction of ϵ is type-preserving.*

   Induction step*: Negation is intentionally type-preserving since it only admits the input term as proper term reduct. There are two cases for sequential composition according to* [comp.1] *and* [comp.2]*. In both cases the type of s_1, s_2, and $s_1; s_2$ coincide. In the first case, the common type is of the form $\tau \to \tau$. In the second case, the common type is* TP*. Because of this coincidence of types, the IH is immediately enabled for the first premise $s_1 @ t \rightsquigarrow t'$, and then for the second premise $s_2 @ t' \rightsquigarrow t''$. The proof for choice and congruences was done in Proof 1. The interesting case is $s \lhd \pi @ t$ where we assume that t is of type τ. We want to employ the IH for the premise $\Gamma \vdash s @ t \rightsquigarrow t'$ in* [extend$^+$]*. Hence we need to show that the type π' of s covers τ. This is the case because of the premise $\exists \tau'.\ \tau \to \tau' \preceq_\Gamma \pi'$ in* [extend$^+$]*. Thus the IH is enabled, and subject reduction holds. As for $s \rhd \pi$, we directly resort to s. The IH for the reduction of s is trivially implied since $\tau \to \tau \preceq_\Gamma \pi$ implies $\tau \to \tau \preceq_\Gamma \pi'$ for $\pi \prec_\Gamma \pi'$ (cf.* [restrict]*).*

◇

Implicit restriction While extension required a dedicated combinator for the reasons we explained earlier, we do not need to insist on explicit restriction. Implicit restriction is desirable because otherwise a programmer needs to point out the many-sorted type in each location where generic strategy is applied. Implicit restriction is feasible because restriction has no impact on the reduction semantics of a strategy.

Example 15 *Consider the strategy* NAT *which was defined in Figure 5. It involves a congruence $succ(\epsilon)$, where ϵ is supposed to be applied to a natural. In S'_{TP}, we have to rephrase the congruence as $succ(\epsilon \rhd Nat)$. In a calculus with implicit restriction, $succ(\epsilon)$ can be retained.* ◇

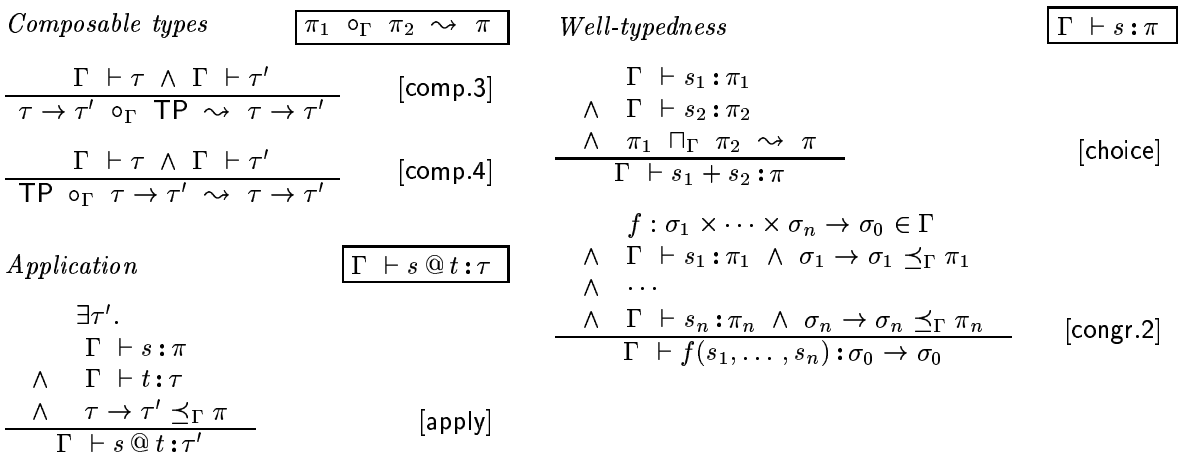

Figure 16: Refinement of the typing rules for implicit restriction

Let us first consider one form of implicit restriction where we update all typing rules which have to do with potentially many-sorted contexts. The corresponding refinement is shown in Figure 16. We will soon describe a much simpler way to achieve the same effect. The value of the refinement in Figure 16 is that we are very precise about where restriction might be needed. Moreover, we can maintain unicity of typing for this system. 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 many-sorted argument (if any). As for congruences, generic strategies can be used as argument strategies. We also want to relax strategy application so that we can apply a generic strategy without further precautions to any term.

The revised rules in Figure 16 can be read as follows. Consider the updated rule [apply] for well-typedness of strategy applications. A strategy s of type π can be applied to a term t of type τ, if τ is admitted as domain by π. As an aside, note that the definition is sufficiently general to cope with type-changing strategies. As for $\cdot;\cdot$, we relax the definition of composable types to cover composition of a many-sorted type and the generic type TP in both possible orders (cf. [comp.3] and [comp.4]). As for $\cdot + \cdot$, we do not insist on equal argument types anymore, but we employ an auxiliary judgement $\pi_1 \sqcap_\Gamma \pi_2 \rightsquigarrow \pi$ for the greatest lower bound w.r.t. $\preceq_\Gamma$ (cf. [choice]). Finally, we relax the argument types for congruences via the $\preceq_\Gamma$ relation. The refinement in Figure 16, in a sense, automates the cast from TP to many-sorted strategy types. Let us stress that implicit restriction does not harm type safety in any way. In the worst case, implicit restriction might lead to accidentally many-sorted strategies. However, such accidents will not go unnoticed. If the intentionally generic strategy is applied in a generic context or annotated with a generic type, a type error will be reported.

There is a somewhat simpler approach to implicit restriction. We can include a typing rule which models that a generic strategy can also be regarded as a many-sorted strategy. The rule is shown in Figure 17. In fact, the two approaches in Figure 16 and Figure 17 are semantically equivalent. The former approach requires that many typing rules need to be aware of $\prec_\Gamma$ whereas the latter approach suffers from another slight drawback, namely unicity of typing does not hold anymore for strategies. However, one can easily see that the multiple types are closed under $\preceq_\Gamma$. Thus, one can safely replace unicity of typing by the existence of a principal type.

According to Figure 17, a generic strategy can also be typed with all instances underlying the generic type. We simply regard the generic type of a strategy (if any) as its principal type. Note that unicity of typing still holds for strategy application as a consequence of the definition of a proper generic type.

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

$$\frac{\begin{array}{ll}\Gamma & \vdash s : \pi \\ \wedge & \pi' \prec_\Gamma \pi\end{array}}{\Gamma \vdash s : \pi'} \quad [\text{implicit}]$$

Figure 17: Implicit restriction relying on principal types

4.4 Strategies of type $\mathsf{TU}(\cdot)$

Generic type-preserving traversals correspond certainly to the most obvious class of generic traversals. The second generic type covered by S'_γ is $\mathsf{TU}(\tau)$ modelling strategies which map any term to a term of the distinguished type τ.

Syntax

$$s \quad ::= \quad \cdots \mid \bigcirc^s(s) \mid \sharp(s) \mid \ \underline{\perp} \ \mid s \parallel s$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow t'}$

Positive rules

$$\frac{s \mathbin{@} t_1 \rightsquigarrow t'_1}{\bigcirc^{s_\circ}(s) \mathbin{@} f(t_1) \rightsquigarrow t'_1} \quad [\text{reduce}^+.1]$$

$$\frac{\begin{array}{ll} & s \mathbin{@} t_1 \rightsquigarrow t'_1 \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow t'_n \\ \wedge & s_\circ \mathbin{@} \langle t'_1, t'_2 \rangle \rightsquigarrow t'_{n+1} \\ \wedge & s_\circ \mathbin{@} \langle t'_{n+1}, t'_3 \rangle \rightsquigarrow t'_{n+2} \\ \wedge & \cdots \\ \wedge & s_\circ \mathbin{@} \langle t'_{2n-2}, t'_n \rangle \rightsquigarrow t'_{2n-1} \end{array}}{\bigcirc^{s_\circ}(s) \mathbin{@} f(t_1, t_2, \ldots, t_n) \rightsquigarrow t'_{2n-1}} \quad [\text{reduce}^+.2]$$

$$\frac{\exists i \in \{1, \ldots, n\}. s \mathbin{@} t_i \rightsquigarrow t'_i}{\sharp(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow t'_i} \quad [\text{select}^+]$$

$$\underline{\perp} \mathbin{@} t \rightsquigarrow \langle\rangle \quad [\text{void}^+]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow t_1 \ \wedge \ s_2 \mathbin{@} t \rightsquigarrow t_2}{s_1 \parallel s_2 \mathbin{@} t \rightsquigarrow \langle t_1, t_2 \rangle} \quad [\text{spawn}^+]$$

Negative rules

$$\bigcirc^{s_\circ}(s) \mathbin{@} c \rightsquigarrow \uparrow \quad [\text{reduce}^-.1]$$

$$\frac{\exists i \in \{1, \ldots, n\}. s \mathbin{@} t_i \rightsquigarrow \uparrow}{\bigcirc^{s_\circ}(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\text{reduce}^-.2]$$

$$\frac{\begin{array}{ll} & s \mathbin{@} t_1 \rightsquigarrow t'_1 \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow t'_n \\ \wedge & s_\circ \mathbin{@} \langle t'_1, t'_2 \rangle \rightsquigarrow \uparrow \end{array}}{\bigcirc^{s_\circ}(s) \mathbin{@} f(t_1, t_2, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\text{reduce}^-.3]$$

$$\frac{\begin{array}{ll} & s \mathbin{@} t_1 \rightsquigarrow t'_1 \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow t'_n \\ \wedge & s_\circ \mathbin{@} \langle t'_1, t'_2 \rangle \rightsquigarrow t'_{n+1} \\ \wedge & \cdots \\ \wedge & s_\circ \mathbin{@} \langle t'_{n+i-2}, t'_i \rangle \rightsquigarrow \uparrow \end{array}}{\bigcirc^{s_\circ}(s) \mathbin{@} f(t_1, t_2, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\text{reduce}^-.4]$$

$$\sharp(s) \mathbin{@} c \rightsquigarrow \uparrow \quad [\text{select}^-.1]$$

$$\frac{\begin{array}{ll} & s \mathbin{@} t_1 \rightsquigarrow \uparrow \\ \wedge & \cdots \\ \wedge & s \mathbin{@} t_n \rightsquigarrow \uparrow \end{array}}{\sharp(s) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\text{select}^-.2]$$

$$\frac{s_1 \mathbin{@} t \rightsquigarrow \uparrow \ \vee \ s_2 \mathbin{@} t \rightsquigarrow \uparrow}{s_1 \parallel s_2 \mathbin{@} t \rightsquigarrow \uparrow} \quad [\text{spawn}^-]$$

Figure 18: Generic traversal combinators (intended for type-unification)

Combinators In Figure 18, the reduction semantics of the combinators for intentionally type-unifying traversals is defined.[11] Thereby, we complete the combinator suite of S'_γ. The combinator $\bigcirc^{\cdot}(\cdot)$ for reducing all children deserves two positive rules. In [reduce$^+$.1], the case of a term with exactly one child is covered. In [reduce$^+$.2], reduction is shown for $n \geq 2$ children. Every child t_i is pre-processed to make it compatible for reduction. Pairwise composition is employed to compute a final term t'_{2n-1} from all the intermediate results. Note that pairwise composition is performed from left to right. This is a kind of arbitrary choice at this point. We will come back to that issue in Section 5.5. Note also that the reduction semantics for $\bigcirc^{\cdot}(\cdot)$ does actually not enforce a total temporal order on how pairwise composition is intertwined with pre-processing the children. There are at least two sensible readings of [reduce$^+$.2]. Either we first pre-process all children, and then we perform pairwise composition, or we immediately perform pairwise composition whenever a new child has been pre-processed. The negative rules for $\bigcirc^{\cdot}(\cdot)$ expose similarities to the negative rules for $\Box(\cdot)$ and $\Diamond(\cdot)$. A constant cannot be reduced as in the case of $\Diamond(\cdot)$ (cf. [reduce$^-$.1]). Reduction fails if there is one child which cannot be processed as in the case of $\Box(\cdot)$ (cf. [reduce$^-$.2]). The remaining two negative rules deal with failure of pairwise composition (cf. [reduce$^-$.3] and [reduce$^-$.4]).

Selection of a child is considerably simpler. The overall scheme regarding both the positive rule and the two negative rules for $\sharp(\cdot)$ is very similar to the combinator $\Diamond(\cdot)$. The combinator $\sharp(\cdot)$ differs from $\Diamond(\cdot)$ only in that the shape of the input term is not preserved. Recall that in the reduct of an application of $\Diamond(\cdot)$, the outermost function symbol and all non-processed children were preserved. Instead, selection simply yields the processed child. As in the case of $\Diamond(\cdot)$, we cannot process constants ([select$^-$.1]), and we also need to fail if none of the children admits selection (cf. [select$^-$.2]).

Let us finally consider the auxiliary combinators $\bot$ and $s_1 \parallel s_2$. The combinator $\bot$ simply accepts any kind of term, and reduction yields the empty tuple $\langle\rangle$. The combinator is in a sense similar to ϵ as it succeeds for every term. However, the term reduct is of a trivial type, namely $\langle\rangle$ regardless of the type of the input term. The strategy $s_1 \parallel s_2$ applies both strategies to the input term, and the intermediate results are paired (cf. [spawn$^+$]). If either of the strategy applications fails, $s_1 \parallel s_2$ fails, too (cf. [spawn$^-$]). Note that one could attempt to describe the behaviour underlying $\bot$ and $\cdot \parallel \cdot$ by the following strategies involving rewrite rules:

$$\begin{aligned} \textsc{Void} &= X \to \langle\rangle \\ \textsc{Spawn}(\nu_1, \nu_2) &= X \to \langle Y_1, Y_2\rangle \text{ where } Y_1 = \nu_1 \mathbin{@} X \text{ where } Y_2 = \nu_2 \mathbin{@} X \end{aligned}$$

However, the involved rewrite rules (and the variables) had to be generically typed. This is in conflict with the design decisions which we postulated for S'_γ. Rewrite rules in S'_γ are many-sorted. All genericity should arise from distinguished primitive combinators. Recall that these requirements are meant to support a clean separation of genericity and specificity, and to minimise the employment of machinery going beyond basic many-sorted rewriting.

The generic type $\mathsf{TU}(\cdot)$ The formalisation of the generic type (constructor) $\mathsf{TU}(\cdot)$ is presented in Figure 19. We basically have to perform the same steps as we discussed for TP. Firstly, well-formedness of $\mathsf{TU}(\cdot)$ is defined (cf. [pi.3]). Secondly, the type scheme underlying $\mathsf{TU}(\tau)$ is defined (cf. [less.2]). Thirdly, the auxiliary judgement for sequential composition is updated (cf. [comp.5]) to describe how $\mathsf{TU}(\cdot)$ is promoted. Consider a sequential composition $s_1; s_2$ where s_1 is of type $\mathsf{TU}(\tau)$, and s_2 is of type $\tau \to \tau'$. The result is of type $\mathsf{TU}(\tau')$.[12] Finally, typing rules for the relevant strategy combinators are provided. The situation is rather simple. Reduction of all children to a type τ via $\bigcirc^{s_\circ}(s)$ requires $s_\circ$ to be able to map a pair of type $\langle\tau, \tau\rangle$ to a value of type τ (in the sense of pairwise composition), and the strategy s for pre-processing the children has to be type-unifying w.r.t. the same τ (cf. [reduce]). The typing rule for the combinator $\sharp(\cdot)$ directly states that the combinator

[11] Note that we ignore the typing context which might be needed for the reduction of $\cdot \lhd \cdot$. It is always trivial to update reduction rules to perform propagation of the typing context.

[12] If we want to make sequential composition fit for implicit restriction, we could also consider a generic type for s_2. We still could lookup τ' due to the definition of a proper generic type.

Syntax

$$\gamma \quad ::= \quad \cdots \mid \mathsf{TU}(\tau)$$

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

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{TU}(\tau)} \quad [\mathsf{pi.3}]$$

Genericity $\boxed{\pi \prec_\Gamma \pi'}$

$$\frac{\Gamma \vdash \tau \ \wedge \ \Gamma \vdash \tau'}{\tau' \to \tau \prec_\Gamma \mathsf{TU}(\tau)} \quad [\mathsf{less.2}]$$

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

$$\frac{\Gamma \vdash \tau \ \wedge \ \Gamma \vdash \tau'}{\mathsf{TU}(\tau) \ \circ_\Gamma \ \tau \to \tau' \ \rightsquigarrow \ \mathsf{TU}(\tau')} \quad [\mathsf{comp.5}]$$

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

$$\frac{\begin{array}{c}\Gamma \vdash s_\circ : \langle \tau, \tau \rangle \to \tau \\ \wedge \quad \Gamma \vdash s : \mathsf{TU}(\tau)\end{array}}{\Gamma \vdash \bigcirc^{s_\circ}(s) : \mathsf{TU}(\tau)} \quad [\mathsf{reduce}]$$

$$\frac{\Gamma \vdash s : \mathsf{TU}(\tau)}{\Gamma \vdash \sharp(s) : \mathsf{TU}(\tau)} \quad [\mathsf{select}]$$

$$\Gamma \vdash \underline{\bot} : \mathsf{TU}(\langle\rangle) \quad [\mathsf{void}]$$

$$\frac{\begin{array}{c}\Gamma \vdash s_1 : \mathsf{TU}(\tau_1) \\ \wedge \quad \Gamma \vdash s_2 : \mathsf{TU}(\tau_2)\end{array}}{\Gamma \vdash s_1 \parallel s_2 : \mathsf{TU}(\langle \tau_1, \tau_2 \rangle)} \quad [\mathsf{spawn}]$$

Figure 19: The generic type $\mathsf{TU}(\cdot)$

is a transformer on type-unifying strategies (cf. [select]). The typing rule for $\underline{\bot}$ states that every type is mapped to the most trivial type (cf. [void]). Finally, $\cdot \parallel \cdot$ takes two type-unifying strategies, and produces another type-unifying strategy. If $\mathsf{TU}(\tau_1)$ and $\mathsf{TU}(\tau_2)$ are the types of the argument strategies in $s_1 \parallel s_2$, then $\mathsf{TU}(\langle \tau_1, \tau_2 \rangle)$ is the type of the resulting strategy (cf. [spawn]).

Assembly of S'_γ Let us compose the calculus S'_γ. It accomplishes both type-preserving and type-changing strategies. Furthermore, tuples are supported. Ultimately, the generic types TP and $\mathsf{TU}(\cdot)$ are enabled. We favour implicit restriction for S'_γ. For convenience, we summarise all ingredients for S'_γ:

- The basic calculus S'_0 (cf. Figure 6)
- Many-sorted type-preserving strategies (cf. Figure 8)
- Many-sorted type-changing strategies (cf. Figure 10)
- Polyadic strategies (cf. Figure 11)
- The combinators $\Box(\cdot)$ and $\Diamond(\cdot)$ (cf. Figure 12)
- The generic type TP (cf. Figure 13)
- The combinators $\bigcirc^{\cdot}(\cdot)$, $\sharp(\cdot)$, $\underline{\bot}$, and $\cdot \parallel \cdot$ (cf. Figure 18)
- The generic type $\mathsf{TU}(\cdot)$ (cf. Figure 19)
- Strategy extension (cf. Figure 14)
- Implicit restriction (cf. Figure 16 or Figure 17)

Theorem 4 *The calculus S'_γ obeys the following properties:*

1. *Strategies satisfy unicity of typing, i.e., ... (cf. Theorem 3).*
2. *Strategy applications satisfy unicity of typing, i.e., ... (cf. Theorem 1).*

3. *Reduction of strategy applications satisfies subject reduction, i.e., for all well-formed contexts* Γ*, strategies* s*, terms* t*,* t'*, term types* τ*,* τ' *and strategy types* π*:*
$\Gamma \vdash s : \pi \ \wedge \ \Gamma \vdash t : \tau \ \wedge \ \tau \to \tau' \preceq_\Gamma \pi \ \wedge \ \Gamma \vdash s @ t \rightsquigarrow t'$ *implies* $\Gamma \vdash t' : \tau'$.

$\diamond$

We omit the proof because it is a simple combination of the ideas from Proof 2 and Proof 3. In the former proof, we generalised the scheme for many-sorted type-preserving strategies from Proof 1 to cope with type-changing strategies (as type-unifying strategies are, too). In the latter, we generalised Proof 1 in a different dimension, namely to cope with generic strategies (as type-unifying strategies are, too). It is trivial to cope with implicit restriction (instead of explicit restriction in S'_{TP}), neither does the introduction of tuples pose any challenge.

5. Sophistication

In the previous two sections we studied the reduction semantics and the type system for all the S'_γ primitives. In this section, we want to complement this development with a few supplementary concepts. Firstly, we will consider strategy definitions corresponding to an indispensable tool for strategic programming. They are needed to introduce reusable combinators (in terms of primitive combinators), and to express recursive strategies at all. Secondly, we will refine the model underlying the formalisation of S'_γ to obtain a reduction semantics which does not employ typing judgements in the reduction rules anymore. To this end, we work out a static elaboration scheme for S'_γ. Thirdly, we introduce some syntactic sugar for what we call biased type-dependent choice. This concept complements the combinator $\cdot \lhd \cdot$ for strategy extension in a convenient manner. The new form of choice allows one to update generic strategies for a specific sort. Thereby, the extension of a many-sorted strategy is not necessarily based on failure as default anymore. Fourthly, we describe a form of overloaded strategies, that is, strategies which are applicable to terms of several types. Interestingly, the types of overloaded strategies are located between many-sorted and generic strategy types regarding genericity (as formalised by $\preceq_\Gamma$). Finally, we will discuss the potential for further combinators for processing children in generic traversals.

5.1 Strategic programs

A strategic program is of the following form:

$$\Gamma \ \Delta \ s$$

Here Γ corresponds to type declarations for the program, Δ is a list of strategy definitions, and s is the main expression of the program. A strategy definition is of the form $\varphi(\nu_1, \ldots, \nu_n) = s$ where $\nu_1, \ldots, \nu_n$ are the formal parameters. We omit the parentheses if φ has no parameters. We assume that s does not contain other strategy variables than $\nu_1, \ldots, \nu_n$. The formal definition of strategic programs is shown in Figure 20.[13]

To consider well-formedness and well-typedness of strategic programs we need to extend the grammar for contexts Γ as described in Figure 20. Contexts may contain type declarations for strategy combinators and types of strategy variables. A strategic program is well-formed if the strategy definitions and the main expression of a program are well-typed (cf. [prog]). A strategy definition is well-typed if the body can be shown to have the declared result type of the combinator while assuming the formal parameters to be of appropriate type (cf. [def.3]). When a strategy variable is encountered by the well-typedness judgement, its type is determined via the context (cf. [arg]). An application of a combinator is well-typed if the types of the actual parameters are equal to the types of the formal parameters. As for the reduction semantics, we propagate the strategy definitions as context parameter Δ in the reduction judgement. Reduction of programs and applications of strategy combinators is defined in a straightforward manner (cf. [prog$^{+/-}$] and [comb$^{+/-}$]). The reduction

[13] We assume α-conversion for the substitution of strategy variables in Figure 20.

Syntax

$$
\begin{array}{lcll}
p & ::= & \Gamma\ \Delta\ s & \text{(Programs)} \\
\nu & & & \text{(Strategy variables)} \\
\Gamma & ::= & \cdots \mid \varphi : \pi \times \cdots \times \pi \to \pi \mid \nu : \pi & \text{(Contexts)} \\
\Delta & ::= & \emptyset \mid \Delta, \Delta \mid \varphi(\nu, \cdots, \nu) = s & \text{(Definitions)} \\
s & ::= & \cdots \mid \nu \mid \varphi(s, \ldots, s) & \text{(Strategies)}
\end{array}
$$

Well-formedness / Well-typedness

Programs $\boxed{\Gamma \vdash p : \pi}$

$$\frac{\Gamma \vdash \Delta \ \wedge\ \Gamma \vdash s : \pi}{\vdash \Gamma\ \Delta\ s : \pi}\ [\mathsf{prog}]$$

Definitions $\boxed{\Gamma \vdash \Delta}$

$$\Gamma \vdash \emptyset\ [\mathsf{def.1}]$$

$$\frac{\Gamma \vdash \Delta_1 \ \wedge\ \Gamma \vdash \Delta_2}{\Gamma \vdash \Delta_1, \Delta_2}\ [\mathsf{def.2}]$$

$$\frac{\begin{array}{l}\varphi : \pi_1 \times \cdots \times \pi_n \to \pi_0 \in \Gamma \\ \wedge\ \ \Gamma, \nu_1 : \pi_1, \ldots, \nu_n : \pi_n \vdash s : \pi_0\end{array}}{\Gamma \vdash \varphi(\nu_1, \ldots, \nu_n) = s}\ [\mathsf{def.3}]$$

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

$$\frac{\nu : \pi \in \Gamma}{\Gamma \vdash \nu : \pi}\ [\mathsf{arg}]$$

$$\frac{\begin{array}{l}\varphi : \pi_1 \times \cdots \times \pi_n \to \pi_0 \in \Gamma \\ \wedge\ \ \Gamma \vdash s_1 : \pi_1 \\ \wedge\ \ \cdots \\ \wedge\ \ \Gamma \vdash s_n : \pi_n\end{array}}{\Gamma \vdash \varphi(s_1, \ldots, s_n) : \pi_0}\ [\mathsf{comb}]$$

Reduction

Programs $\boxed{\Delta \vdash p \mathbin{@} t \rightsquigarrow r}$

$$\frac{\Gamma, \Delta \vdash s \mathbin{@} t \rightsquigarrow t'}{\Gamma\ \Delta\ s \mathbin{@} t \rightsquigarrow t'}\ [\mathsf{prog}^{+/-}]$$

Strategies $\boxed{\Gamma, \Delta \vdash s \mathbin{@} t \rightsquigarrow r}$

$$\frac{\begin{array}{l}\varphi(\nu_1, \cdots, \nu_n) = s \in \Delta \\ \wedge\ \ s' = s[\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n] \\ \wedge\ \ \Gamma, \Delta \vdash s' \mathbin{@} t \rightsquigarrow r\end{array}}{\Gamma, \Delta \vdash \varphi(s_1, \ldots, s_n) \mathbin{@} t \rightsquigarrow r}\ [\mathsf{comb}^{+/-}]$$

Figure 20: Strategic programs

judgement involves Γ in the context in order to enable strategy extension. We could easily relax the typing rules for strategic programs to enable implicit restriction. Note that implicit restriction is indeed relevant for the application of strategy combinators if we wanted to use generic strategies as actual parameters on many-sorted parameter positions.

Type-parameterised strategy definitions Let us also enable type-parameterised strategy definitions. In Figure 21, we define an elaboration of the strategy definitions in Figure 20 to cope with type parameters in strategy definitions. The formalisation is straightforward.[14] Term type variables are regarded as another form of a term type. The extension of the grammar rule for Γ details that types of strategy combinators can contain a number of universally quantified type variables. Type variables are scoped by the corresponding strategy definition (cf. [def.4]). If the well-formedness judgements

[14] We assume α-conversion for the substitution of type variables in Figure 21.

Syntax

$$
\begin{array}{lcll}
\alpha & & & \text{(Term type variables)} \\
\tau & ::= & \cdots \mid \alpha & \text{(Term types)} \\
\Gamma & ::= & \cdots \mid \varphi : \forall \alpha, \ldots, \alpha.\ \pi \times \cdots \times \pi \to \pi & \text{(Contexts)} \\
\Delta & ::= & \cdots \mid \varphi[\alpha, \ldots, \alpha](\nu, \ldots, \nu) = s & \text{(Definitions)} \\
s & ::= & \cdots \mid \varphi[\tau, \ldots, \tau](s, \ldots, s) & \text{(Strategies)}
\end{array}
$$

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

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad [\mathsf{tau.4}]$$

Definitions $\boxed{\Gamma \vdash \Delta}$

$$\frac{\begin{array}{ll} & \varphi : \forall \alpha_1, \ldots, \alpha_m.\ \pi_1 \times \cdots \times \pi_n \to \pi_0 \in \Gamma \\ \wedge & \Gamma, \nu_1 : \pi_1, \ldots, \nu_n : \pi_n, \alpha_1, \ldots, \alpha_m \vdash s : \pi_0 \end{array}}{\Gamma \vdash \varphi[\alpha_1, \ldots, \alpha_m](\nu_1, \ldots, \nu_n) = s} \qquad [\mathsf{def.4}]$$

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

$$\frac{\begin{array}{ll} & \varphi : \forall \alpha_1, \ldots, \alpha_m.\ \pi_1 \times \cdots \times \pi_n \to \pi_0 \in \Gamma \\ \wedge & \Gamma \vdash s_1 : \pi_1[\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m] \\ \wedge & \cdots \\ \wedge & \Gamma \vdash s_n : \pi_n[\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m] \end{array}}{\Gamma \vdash \varphi[\tau_1, \ldots, \tau_m](s_1, \ldots, s_n) : \pi_0[\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m]} \qquad [\mathsf{comb{-}forall}]$$

Figure 21: Type-parametrised strategy definitions

for types encounter a term type variable, it has to be in the context (cf. [tau.4]). The application of a combinator φ involves type application, namely substitution of the type variables by the actual types (cf. [comb−forall]). For brevity, we do not refine the reduction semantics from Figure 20.

5.2 Standard reduction

In the context of strategy extension, we encountered a complication regarding the judgement for reduction. In order to define the type-safe application of a many-sorted strategy s in a generic context, we had to compare the type of the term at hand, and the type of s. To this end, the typing context Γ was required as context of the reduction judgement, and typing judgements were placed as premises in the rule for $\cdot \lhd \cdot$ (cf. Figure 14). Conceptually, this is fine because we point out in the most direct way that $\cdot \lhd \cdot$ is about type-dependent application of a strategy. We would like to obtain a form of reduction which is more standard, in that typing and reduction judgements are strictly separated. We will employ an intermediary static elaboration judgement to annotate strategies accordingly. Furthermore, we need some simple machinery to be able to determine types of terms during reduction. The resulting reduction semantics is better geared towards implementation.

Static elaboration So far, we only considered well-typedness judgements and the reduction judgement for strategy application. We want to refine the model for the formalisation of S'_γ to include a static elaboration judgement of the following form:

$$\Gamma \vdash s \rightsquigarrow s'$$

Static elaboration $\boxed{\Gamma \vdash s \rightsquigarrow s'}$

$$\Gamma \vdash t_l \to t_r \rightsquigarrow t_l \to t_r \quad [\mathsf{rule}^{\rightsquigarrow}]$$

$$\Gamma \vdash \epsilon \rightsquigarrow \epsilon \quad [\mathsf{id}^{\rightsquigarrow}]$$

$$\Gamma \vdash \delta \rightsquigarrow \delta \quad [\mathsf{fail}^{\rightsquigarrow}]$$

$$\frac{\Gamma \vdash s \rightsquigarrow s'}{\Gamma \vdash \neg\, s \rightsquigarrow \neg\, s'} \quad [\mathsf{neg}^{\rightsquigarrow}]$$

$$\frac{\Gamma \vdash s_1 \rightsquigarrow s_1' \ \wedge\ \Gamma \vdash s_2 \rightsquigarrow s_2'}{\Gamma \vdash s_1; s_2 \rightsquigarrow s_1'; s_2'} \quad [\mathsf{seq}^{\rightsquigarrow}]$$

$$\frac{\Gamma \vdash s_1 \rightsquigarrow s_1' \ \wedge\ \Gamma \vdash s_2 \rightsquigarrow s_2'}{\Gamma \vdash s_1 + s_2 \rightsquigarrow s_1' + s_2'} \quad [\mathsf{choice}^{\rightsquigarrow}]$$

$$\Gamma \vdash c \rightsquigarrow c \quad [\mathsf{congr}^{\rightsquigarrow}.1]$$

$$\frac{\begin{array}{rl} & \Gamma \vdash s_1 \rightsquigarrow s_1' \\ \wedge & \cdots \\ \wedge & \Gamma \vdash s_n \rightsquigarrow s_n' \end{array}}{\Gamma \vdash f(s_1, \ldots, s_n) \rightsquigarrow f(s_1', \ldots, s_n')} \quad [\mathsf{congr}^{\rightsquigarrow}.2]$$

Figure 22: General scheme of static elaboration

The general idea of static elaboration is that the input strategy s can be transformed in a semantics-preserving manner. There are several potential applications of static elaboration. We will emphasise its application to the problem of eliminating typing judgements in the reduction rules for $\cdot \lhd \cdot$. One could employ static elaboration for the definition of syntactic sugar or for program specialisation (say partial evaluation) aiming at optimisation (cf. [JGS93, JV01]). The semantic model for typeful strategies needs to be updated to consist of three phases:

1. The given strategy s is checked to be well-typed.
2. s is (statically) elaborated resulting in a strategy s'.
3. Given a suitable term t, the strategy s' is applied to t to derive a reduct.

These phases obviously maps nicely to an implementation model where type-checking and elaboration is done once and for all statically (without insisting on an input term), and the reduction semantics is used in its interpreter-like reading. In general, static elaboration might be type-dependent, that is, the typing context Γ is part of the judgement (as in the case of the well-formedness and well-typedness judgements). In Figure 22, we illustrate the general scheme of static elaboration restricting ourselves to the combinators of the basic calculus S_0'.

Annotated extensions In order to eliminate the typing premises in the reduction rules for $\cdot \lhd \cdot$ we replace strategy expressions of the form $s \lhd \pi$ by $s : \pi' \lhd \pi$ where π' denotes the actual type of s. Thereby, the type of s does not need to be determined during reduction. Furthermore, we assume that terms are tagged (at least) at the top-level. Thus, the original type dependency reduces to a simple comparison of type tag of the term at hand and the annotated type π' from $s : \pi' \lhd \pi$. The rule for static elaboration and the updated reduction rules are shown in Figure 23. We harmonise the grammar of terms to include terms tagged by types.

Tagged terms The assumption that terms are tagged by a type is actually not trivial to realise in the reduction semantics. For several forms of strategies, we need to be able to determine the types of subterms along the steps of a reduction, namely for congruences, and for applications of traversal primitives like $\Box(\cdot)$. Furthermore, type-changing strategies (e.g., type-changing rewrite rules) imply that we cannot simply preserve the tag of the input term as the tag of the reduct, even if we act at the top-level of a term. Static elaboration can be employed to solve part of the problem. For the generic traversals primitives we need an additional technique discussed later. To cope with strategies which process subterms or which change the type of the input term, strategies can be annotated so

Syntax

$$t \quad ::= \quad \cdots \mid t : \tau \quad \text{(Terms)}$$

Static elaboration $\boxed{\Gamma \vdash s \rightsquigarrow s'}$

$$\frac{\Gamma \vdash s : \pi'}{\Gamma \vdash s \lhd \pi \rightsquigarrow s : \pi' \lhd \pi} \quad [\mathsf{extend}^{\rightsquigarrow}]$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow r}$

Positive rule

$$\frac{s \mathbin{@} t : \tau \rightsquigarrow t'}{s : \tau \to \tau' \lhd \mathbin{@} t : \tau \rightsquigarrow t'} \quad [\mathsf{extend'}^{+}]$$

Negative rules

$$\frac{\tau \neq \tau''}{s : \tau \to \tau' \lhd \tau \mathbin{@} t : \tau'' \rightsquigarrow \uparrow} \quad [\mathsf{extend'}^{-}.1]$$

$$\frac{s \mathbin{@} t : \tau \rightsquigarrow \uparrow}{s : \tau \to \tau' \lhd \mathbin{@} t : \tau \rightsquigarrow \uparrow} \quad [\mathsf{extend'}^{-}.2]$$

Figure 23: Γ-free reduction

Static elaboration $\boxed{\Gamma \vdash s \rightsquigarrow s'}$

$$\frac{\Gamma \vdash t_l : \tau \ \wedge \ \Gamma \vdash t_r : \tau'}{\Gamma \vdash t_l \to t_r \rightsquigarrow t_l \to t_r : \tau \to \tau'} \quad [\mathsf{rule}^{\rightsquigarrow}]$$

$$\frac{\begin{array}{l}\Gamma \vdash s_1 \rightsquigarrow \pi_1 \\ \wedge \quad \cdots \\ \wedge \quad \Gamma \vdash s_n \rightsquigarrow \pi_n\end{array}}{\begin{array}{c}\Gamma \vdash f(s_1, \ldots, s_n) \\ \rightsquigarrow f(s_1 : \pi_1, \ldots, s_n : \pi_n)\end{array}} \quad [\mathsf{congr}^{\rightsquigarrow}.2]$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow r}$

Positive rules

$$\frac{\exists \theta.(\theta(t_l) = t \wedge \theta(t_r) = t')}{t_l \to t_r : \tau \to \tau' \mathbin{@} t : \tau \rightsquigarrow t' : \tau'} \quad [\mathsf{rule}^{+}]$$

$$\frac{\begin{array}{l}s_1 \mathbin{@} t_1 : \sigma_1 \rightsquigarrow t_1' : \sigma_1 \\ \wedge \quad \cdots \\ \wedge \quad s_n \mathbin{@} t_n : \sigma_n \rightsquigarrow t_n' : \sigma_n\end{array}}{\begin{array}{c}f(s_1 : \sigma_1 \to \sigma_1, \ldots, s_n : \sigma_n \to \sigma_n) \\ \mathbin{@} f(t_1, \ldots, t_n) : \tau \rightsquigarrow \\ f(t_1', \ldots, t_n') : \tau\end{array}} \quad [\mathsf{congr}^{+}.2]$$

Figure 24: Propagation of tags

that reduction can rely on the annotations. This idea is illustrated in Figure 24 for rewrite rules and congruences. We only show the updated positive rules.[15] Static elaboration adds type annotations (cf. [$\mathsf{rule}^{\rightsquigarrow}$] and [$\mathsf{congr}^{\rightsquigarrow}.2$]). The new reduction rules insists on type-annotated strategies (cf. [rule^{+}] and [$\mathsf{congr}^{+}.2$]). In this manner, the proper type tags can be propagated. Note that sequential composition and choice do not need to be elaborated because they do not directly operate on terms. They rather resort to the argument strategies. In principle, we could simply annotate all strategy expressions and ignore those tags during reduction which are not needed.

Generic traversal primitives As for the traversal combinators, it is more difficult to consistently deal with tagged terms. Since the argument strategies are inherently generic we cannot derive the types of the subterms from the type of the argument strategy (as we managed to do in the case of congruences). There are the following options to deliver types of subterms:

[15] Where-clauses also tend to process subterms. Static elaboration could also be employed in this context.

Syntax

$$\begin{array}{rcll} \Sigma & ::= & \emptyset \mid \Sigma, \Sigma & (\Gamma\text{-like sets}) \\ & \mid & f : \sigma \times \cdots \times \sigma \to \sigma & (\text{Types of subterms}) \end{array}$$

Reduction $\boxed{\Sigma \vdash s \mathbin{@} t \rightsquigarrow r}$

Positive rules

$$\Sigma \vdash \Box(s) \mathbin{@} c : \sigma \rightsquigarrow c : \sigma \qquad [\mathsf{all}^+.1]$$

$$\frac{\begin{array}{rl} & f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in \Sigma \\ \wedge & \Sigma \vdash s \mathbin{@} t_1 : \sigma_1 \rightsquigarrow t_1' : \sigma_1 \\ \wedge & \cdots \\ \wedge & \Sigma \vdash s \mathbin{@} t_n : \sigma_n \rightsquigarrow t_n' : \sigma_n \end{array}}{\Sigma \vdash \Box(s) \mathbin{@} f(t_1, \ldots, t_n) : \sigma_0 \rightsquigarrow f(t_1', \ldots, t_n') : \sigma_0} \qquad [\mathsf{all}^+.2]$$

$$\frac{\begin{array}{rl} & f : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0 \in \Sigma \\ \wedge & \exists i \in \{1, \ldots, n\}. \Sigma \vdash s \mathbin{@} t_i : \sigma_i \rightsquigarrow t_i' : \sigma_i \end{array}}{\Sigma \vdash \Diamond(s) \mathbin{@} f(t_1, \ldots, t_n) : \sigma_0 \rightsquigarrow f(t_1, \ldots, t_i', \ldots, t_n) : \sigma_0} \qquad [\mathsf{one}^+]$$

Figure 25: Determining term types by a simple lookup

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 (as opposed to a proper well-typedness judgement). 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 the other traversal primitives. That is, we had to instantiate the scheme for $\Box(\cdot)$ etc. for all function symbols.

All of these formulations lead—more or less directly—to an efficient implementation. Option (1.) has an impact on the representation of terms. It also implies that rewrite rules had to be elaborated so that the terms for left- and right-hand side are pervasively tagged. The option can be easily used in an actual implementation of a rewriting engine. Option (2.) relies on an extra context lookup per application of a traversal primitive. Option (3.) requires program generation. The specialisation of program schemes is often used in the context of the implementation of generic algorithms. In Figure 25, we formalise option (2.). To this end, we update the reduction semantics for $\Box(\cdot)$ and $\Diamond(\cdot)$ to rely on a trivial context parameter Σ for types of function symbols. The rule for congruences for constants actually is not changed (cf. $[\mathsf{all}^+.1]$) since no subterms are processed. In the case of compound terms, the types of the children are simply looked up from Σ and propagated when we process the children (cf. $[\mathsf{all}^+.2]$ and $[\mathsf{one}^+]$).

Untagged terms In fact, the availability of a context parameter like Σ can also be employed in a way that we do not tag terms at all, but we only determine the type of a term when needed, that is, in the reduction rule for $\cdot \lhd \cdot$. We will illustrate this option when we discuss a prototype implementation of S'_γ in Section 6.

5.3 Type-dependent choice

So far, the only way to turn a many-sorted strategy s into a generic one is based on the form $s \lhd \pi$. This kind of casting implies that the lifted strategy will fail at least for all term types different from the domain of s. This is often not desirable, e.g., in applications of $\Box(\cdot)$. To this end, an extension

usually entails a complementary choice. In the present section, we want to argue that the separation of lifting (by $\cdot \lhd \cdot$) and completion (by $\cdot + \cdot$ and friends) is problematic, and it should be eliminated by a combinator for asymmetric type-dependent choice. Fortunately, the additional combinator can be regarded as syntactic sugar.

Example 16 *Consider again the definition of* StopTD *in Figure 4 which we repeat here for convenience:*

$$\begin{array}{rcl} \textsc{StopTD} & : & \mathsf{TP} \to \mathsf{TP} \\ \textsc{StopTD}(\nu) & = & \nu \Leftarrow \Box(\textsc{StopTD}(\nu)) \end{array}$$

Left-biased choice is used to first try the generic argument s of StopTD(s) *but to descend into the children if s fails. Note that s could fail for two reasons. To this end, let us assume that s was obtained by strategy extension from a many-sorted strategy s'. Then, the first reason for s to fail could be that s is faced with a term different from the domain of s'. However, even if s' is applicable in the sense of typing, s' (and hence s) could fail because applicability constraints of s' (other than typing) are not met. In the present formulation,* StopTD *will always recover from failure of s since* StopTD *will at least succeed at the leafs. To see that, one can check that even $\Box(\delta)$ succeeds when it is applied to a constant. More generally, we cannot encode the argument s (or the underlying s') of* StopTD(s) *in a way that we point out applicability constraints. For certain traversal problems, we might favour a variant of* StopTD *which fails globally if the applicability constraints for s are not met for a typeful application of s to some node during traversal.* ◇

We improve the situation as follows. We introduce asymmetric type-dependent choice. In the left-biased notation $s_1 \mathbin{\&\!\!\!\leftarrow} s_2$, s_1 is regarded as an update for the default s_2. Hence, we call this form left-biased type-dependent choice. The many-sorted strategy s_1 should be applied if the type of the term at hand fits, and the generic default s_2 should be applied only of the type of s_1 does not fit. One essential ingredient of the definition is a type guard. For convenience, we also define a dedicated combinator to place type guards. A type guard for the term type τ is simply written as τ. It denotes a generic strategy which is supposed to succeed if and only if the given term is of type τ. Here is the syntactic sugar for type guards, and asymmetric type-dependent choice:

$$\begin{array}{rcl} \tau & = & \epsilon \rhd \tau \to \tau \lhd \mathsf{TP} \\ s_1 \mathbin{\&\!\!\!\leftarrow} s_2 & = & s_1 \lhd \pi + (\neg\, \tau; s_2) \text{ where } s_1 : \tau \to \tau', s_2 : \pi \\ s_1 \mathbin{\&\!\!\!\rightarrow} s_2 & = & s_2 \mathbin{\&\!\!\!\leftarrow} s_1 \end{array}$$

A type guard τ is derived from ϵ where restriction is used to test for τ, and the subsequent extension recovers genericity. The resulting strategy indeed fails for all term types except τ. The definition of $s_1 \mathbin{\&\!\!\!\leftarrow} s_2$ employs a negated type guard to block the application of the generic default s_2 in case s_1 is applicable (in the sense of typing).[16] It is very instructive to compare the syntactic sugar constituting the two different forms of asymmetric choice encountered in the present article. In the case of $s_1 \Leftarrow s_2$ (i.e., left-biased choice controlled by success and failure), the success of s_1 rules out the application of s_2. In the case of $s_1 \mathbin{\&\!\!\!\leftarrow} s_2$, the applicability of s_1 solely in terms of its type rules out the application of s_2. In both cases, the employment of negation is essential. To understand this twist, consider the following compound strategy (approximating $s_1 \mathbin{\&\!\!\!\leftarrow} s_2$):

$$s_1 \lhd \pi_2 \Leftarrow s_2 \text{ where } s_2 \text{ is of type } \pi_2$$

This formulation attempts to compensate for the type-guard in the definition of type-dependent choice by resorting to a left-biased choice controlled by success and failure. That is, we attempt to simulate left-biased type-dependent choice by left-biased choice controlled by success and failure. This attempt is not faithful since s_2 might be applied to a term t even if the types of s_1 and t fit, namely if s_1 fails on t.

[16] Type-dependent choice and type guards can also be defined via static elaboration or strategy definitions.

Example 17 *Let us define a variant of* StopTD *which interprets failure of the argument strategy as global failure. Thereby, a programmer can model applicability constraints via the argument strategy:*

$$\begin{array}{rcl} \textsc{StopTD}' & : & \forall\alpha.\ (\alpha \to \alpha) \to \mathsf{TP} \\ \textsc{StopTD}'[\alpha](\nu) & = & \nu \mathbin{\not\&} \Box(\textsc{StopTD}'[\alpha](\nu)) \end{array}$$

StopTD′ *is different from* StopTD *in that the argument of* StopTD *is a generic strategy whereas we use a type parameter in* StopTD′ *for the type of the argument strategy. The type-dependent choice to derive a generic strategy from the argument is part of the definition of* StopTD′. ◇

5.4 Overloaded strategies

We want to consider an intermediate form of genericity, namely overloaded strategies. Let us recall the difference between many-sorted and generic strategies. A truly generic strategy is applicable to terms of all sorts (although it might fail for some terms, or even for all terms of certain sorts). A many-sorted strategy such as a rewrite rule is applicable to terms of a distinguished sort. Overloading means that we can cope with strategies which are applicable to terms of a number of sorts. We will describe overloading in a way that overloaded strategies are inhabited by a form of choice. Intuitively, if we can choose between two rewrite rules with different domains, we might say that the combined strategy defined in terms of a choice is applicable to terms of two different types. The choice which is needed here must be type-dependent to be type-safe. Since the domains of the operands are different, it is actually a disjoint type-dependent choice. We refer to the discussion of type dependency in Section 4.2. The extension of S'_γ to support overloaded strategies relies on two instances of the binary operator $\cdot \& \cdot$, one instance at the strategy type level, and another instance at the strategy level. The former instance has to be regarded as a type constructor for overloaded strategy types. The latter instance is the combinator for disjoint (say symmetric) type-dependent choice. It complements the asymmetric type-dependent choice discussed in Section 5.3.

Example 18 *Consider the following constructors for naturals and integers:*

$$\begin{array}{rcl} \mathit{one} & : & \mathit{NatOne} \\ \mathit{succ} & : & \mathit{NatOne} \to \mathit{NatOne} \\ \mathit{zero} & : & \mathit{NatZero} \\ \mathit{notzero} & : & \mathit{NatOne} \to \mathit{NatZero} \\ \mathit{positive} & : & \mathit{NatZero} \to \mathit{Int} \\ \mathit{negative} & : & \mathit{NatOne} \to \mathit{Int} \end{array}$$

NatZero includes 0*, whereas NatOne excludes* 0*. Integers are constructed via two branches, one for positive integers (including zero), and another for truly negative integers. Let us define an (overloaded) strategy which is capable of incrementing terms of three above sorts:*

$$\begin{array}{rcl} \textsc{Inc} & = & \mathit{NatOne} \to \mathit{NatOne} \,\&\, \mathit{NatZero} \to \mathit{NatZero} \,\&\, \mathit{Int} \to \mathit{Int} \\ \textsc{Inc} & = & \mathit{one} \to \mathit{succ}(\mathit{one}) + \mathit{succ}(\textsc{Inc}) \\ & \& & \mathit{zero} \to \mathit{notzero}(\mathit{one}) + \mathit{notzero}(\textsc{Inc}) \\ & \& & \mathit{positive}(\textsc{Inc}) + \mathit{negative}(\textsc{Inc}) \end{array}$$

The strategy is defined via disjoint type-dependent choice with three cases, one for each sort. Otherwise, the functionality to increment is defined by primitive recursion. To this end, the case discrimination for the term constructors of each sort is encoded in terms of symmetric choice controlled by success and failure. As an aside, we assume implicit restriction (for overloaded strategies) since the recursive occurrences Inc *are used for specific sorts covered by the overloaded type of* Inc. ◇

Syntax

$$\begin{array}{rcl} \pi & = & \cdots \mid \pi \,\&\, \pi \\ s & = & \cdots \mid s \,\&\, s \end{array}$$

Well-formedness $\boxed{\Gamma \vdash \pi}$

$$\frac{\begin{array}{ll} & \Gamma \vdash \pi_1 \ \wedge\ \textsc{Dom}(\pi_1) \Rightarrow \tau s_1 \\ \wedge & \Gamma \vdash \pi_2 \ \wedge\ \textsc{Dom}(\pi_2) \Rightarrow \tau s_2 \\ \wedge & \tau s_1 \cap \tau s_2 = \emptyset \end{array}}{\Gamma \vdash \pi_1 \,\&\, \pi_2} \quad [\mathsf{pi.4}]$$

Domain $\boxed{\textsc{Dom}(\pi) \rightsquigarrow \mathcal{P}_{\mathit{fin}}(\tau)}$

$$\textsc{Dom}(\tau \to \tau') \rightsquigarrow \{\tau\} \quad [\mathsf{dom.1}]$$

$$\frac{\begin{array}{ll} & \textsc{Dom}(\pi_1) \rightsquigarrow \tau s_1 \\ \wedge & \textsc{Dom}(\pi_2) \rightsquigarrow \tau s_2 \end{array}}{\textsc{Dom}(\pi_1 \,\&\, \pi_2) \rightsquigarrow \tau s_1 \cup \tau s_2} \quad [\mathsf{dom.2}]$$

Genericity $\boxed{\pi \prec_\Gamma \pi'}$

$$\frac{\Gamma \vdash \pi \ \wedge\ \Gamma \vdash \pi' \ \wedge\ \exists \pi''.\ \pi' \approx \pi \,\&\, \pi''}{\pi \prec_\Gamma \pi'} \quad [\mathsf{less.3}]$$

$$\frac{\pi_1 \prec_\Gamma \pi \ \wedge\ \pi_2 \prec_\Gamma \pi}{\pi_1 \,\&\, \pi_2 \prec_\Gamma \pi} \quad [\mathsf{less.4}]$$

Composable types $\boxed{\pi \ \circ_\Gamma \ \pi' \rightsquigarrow \pi''}$

$$\frac{\begin{array}{ll} & \pi_1 \approx \pi_1' \,\&\, \pi_1'' \\ \wedge & \pi_2 \approx \pi_2' \,\&\, \pi_2'' \\ \wedge & \pi_1' \ \circ_\Gamma \ \pi_2' \rightsquigarrow \pi_3' \\ \wedge & \pi_1'' \ \circ_\Gamma \ \pi_2'' \rightsquigarrow \pi_3'' \end{array}}{\pi_1 \ \circ_\Gamma \ \pi_2 \rightsquigarrow \pi_3' \,\&\, \pi_3''} \quad [\mathsf{comp.6}]$$

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

$$\frac{\begin{array}{ll} & \Gamma \vdash s_1 : \pi_1 \\ \wedge & \Gamma \vdash s_2 : \pi_2 \\ \wedge & \Gamma \vdash \pi_1 \,\&\, \pi_2 \end{array}}{\Gamma \vdash s_1 \,\&\, s_2 : \pi_1 \,\&\, \pi_2} \quad [\mathsf{amp}]$$

Reduction $\boxed{\Gamma \vdash s \,@\, t \rightsquigarrow r}$

$$\frac{\begin{array}{ll} & \exists i \in \{1,2\}. \\ & \Gamma \vdash t : \tau \\ \wedge & \Gamma \vdash s : \pi_i \\ \wedge & \textsc{Dom}(\pi_i) \Rightarrow \tau s_i \\ \wedge & \tau \in \tau s_i \\ \wedge & \Gamma \vdash s_i \,@\, t \rightsquigarrow r \end{array}}{\Gamma \vdash s_1 \,\&\, s_2 \,@\, t \rightsquigarrow r} \quad [\mathsf{amp}^{+/-}]$$

Figure 26: Overloaded strategies

In Figure 26, overloaded strategies and their types are defined formally. The type of an overloaded strategy is of the form $\tau_1 \to \tau_1' \,\&\, \cdots \,\&\, \tau_n \to \tau_n'$. The type models strategies which are applicable to terms of types $\tau_1, \ldots, \tau_n$. If such a strategy is actually applied to a term of type τ_i, the result will be of type τ_i. We use an auxiliary judgement $\textsc{Dom}(\pi)$ to denote the finite set of term types admitted as domains by a strategy type π. This judgement is not applicable to generic types for reasons which will become clear in a minute. We require that the domains of two types involved in overloading must be disjoint (cf. [pi.4]). This requirement is indispensable for unicity of typing of strategy applications. In [less.3]–[less.4], we update the relation $\prec_\Gamma$ on strategy types. To this end, we employ an equivalence $\approx$ on strategy types modulo associativity and commutativity of $\cdot \,\&\, \cdot$. Rule [less.3] models that π is less generic than any type π' which is of the form $\pi \,\&\, \pi''$. Clearly, this rule is needed to relate simple many-sorted and overloaded strategy types to each other, The rule also relates overloaded strategy types among each other. Rule [less.4] models that the type of an overloaded strategy is less generic than another type π, if both components π_1 and π_2 of the overloaded type are also less generic than π. Again, one can easily see that this rule is needed to relate overloaded strategy types and generic types to each other. The elaboration of $\prec_\Gamma$ creates a type hierarchy with the simple many-sorted strategies as the least elements, and the generic types as the greatest elements. An overloaded strategy is constructed by type-dependent choice $s_1 \,\&\, s_2$ where the types of the arguments s_1 and s_2 have to admit the construction of an overloaded strategy type (cf. [amp]). As for the reduction semantics of

type-dependent choice, reduction resorts to either the strategy s_1 or s_2 depending on the type of the input term (cf. [amp$^+$]). Note that s_1 and s_2 are applicable in a mutually exclusive manner since their domains are required to be disjoint. It is easy to check that the described form of type-dependent choice is symmetric. However, the reduction semantics also details that the order of the operands in $\cdot \,\&\, \cdot$ is relevant for the formal semantics in a trivial sense. That is, we must be able to associate the components of an overloaded strategy type with the corresponding component in a type-dependent choice.[17]

We should note that some details of our formalisation of generic strategies are outdated if we consider overloaded strategies. The typing rules and the original reduction rules for strategy extension are still sufficient if overloaded strategies are taken into account. However, the refined reduction semantics from Section 5.2 is geared towards the extension of simple many-sorted strategies. Also, the formalisation of asymmetric type-dependent choice insists on simple many-sorted strategies for non-generic argument. These limitations could easily be eliminated.

Bounded quantification Consider a combinator with an argument which is only constrained to be type-preserving no matter if it is a simple many-sorted, an overloaded or a generic strategy. To denote the type of such an argument, we should employ bounded quantification on strategy type parameters. So far, we only consider universally quantified type parameters for term types. We do not formalise bounded quantification but we illustrate the concept by examples. We use ω to range over variables for strategy types. We can place constraints on such variables for bounded quantification. The two relational symbols $\prec$ and $\preceq$ are sensible.

Example 19 *In Example 17, we discussed a variant of* StopTD *which takes a many-sorted strategy argument to perform the left-biased type-dependent choice inside the traversal scheme. If we reconsider the definition, it is easy for confirm that the type of the argument was declared unnecessarily restrictive. The following type is more general. We also repeat the definition of the variant for convenience.*

$$\begin{array}{rcl} \textsc{StopTD}' & : & \forall\omega \prec \mathsf{TP}.\ \omega \to \mathsf{TP} \\ \textsc{StopTD}'[\omega](\nu) & = & \nu \mathbin{\&\!\!\!\leftarrow} \Box(\textsc{StopTD}'[\omega](\nu)) \end{array}$$

Note that only the option $\prec$ is sensible but not $\preceq$ since the type constraints for left-biased type-dependent choice cannot be met otherwise. ◇

Example 20 *Recall the discussion in Example 12 where we argued that we need to separate whether a many-sorted strategy is completed for a certain term type (cf.* Try′ *in Example 12), or a generic strategy is completed (cf.* Try*). This separation is useful but bounded quantification enables us to maintain this separation via parameterisation (as opposed to two separate combinators). Here is a more flexible combinator for completion which subsumes* Try *and* Try′*.*

$$\begin{array}{lcl} \textsc{Complete} & : & \forall\omega \preceq \mathsf{TP}.\ \omega \to \omega \\ \textsc{Complete}[\omega] & = & \nu \mathbin{\leftarrow\!\!\!\!\leftarrow} \epsilon \\ \textsc{Try}(s) & = & \textsc{Complete}[\mathsf{TP}] \\ \textsc{Try}'[\alpha](s) & = & \textsc{Complete}[\alpha \to \alpha](s) \end{array}$$

◇

Expressiveness The described form of overloading is simple but expressive. One can encode families of functions (say strategies) for a specific set of sorts in a direct manner. Based on the notion of asymmetric type-dependent choice, one can even dynamically update an overloaded strategy. One can also extend an already overloaded strategy to cover another sort via $\cdot \,\&\, \cdot$. Overloading is also useful to

[17] This is also the reason that associativity and commutativity of $\cdot \,\&\, \cdot$ on types is only enforced if solely types are compared (cf. [less.3] and [comp.6]). Conceptually, types are truly equivalent modulo $\approx$.

describe ingredients of generic traversals which are specific w.r.t. several term types. As an aside, in the latter scenario of overloading, the overloaded strategy will usually be non-recursive since the actual (recursive) scheme of traversal is separated out in the traversal strategy. Overloading is convenient in strategic programming but not essential. In fact, overloading can be usually circumvented with some additional coding effort. As for Example 18, we had to define three separate strategies for the three different sorts involved. As for generic traversals which are specific w.r.t. several term types τ_1, $\ldots$, τ_n, we had to iterate asymmetric type-dependent choice for each τ_i. The drawbacks of such an iteration are that the many-sorted ingredients can only be grouped as a generic strategy, and the type system cannot be employed to enforce that the τ_1, $\ldots$, τ_n are distinct.

Let us also consider the question if generic strategies can be encoded as overloaded strategies. To this end, we also had to setup a correspondence between the corresponding forms of types. Since overloaded strategy types are signature-dependent, we cannot hope to eliminate generic types altogether by overloading. On the other hand, overloaded strategy types are also more precise than generic types. If we consider a fixed signature, then, we can represent the signature-specific instantiations of generic types as overloaded types. Consider, for example, the generic type TP. The corresponding overloaded type is obtained by overloading all $\tau \to \tau$ for all well-formed τ w.r.t. to the given signature. Note that this construction becomes infinite if we enable tuple types. We could also attempt to represent the generic traversal combinators as signature-specific overloaded combinators defined in terms of congruences.

Finally note that generic strategies do not make sense as arguments of $\cdot \,\&\, \cdot$ since a generic type already covers all term types, and hence a generic strategy cannot form part of disjoint type-dependent choice. Recall that we indeed did not provide a definition for $\textsc{Dom}(\gamma)$ in Figure 26.

Complementary forms of choice Let us summarise on the various forms of choice which we have seen in the present article. Choice between strategies of the same type is solely modelled by the combinators $\cdot + \cdot$ and friends. The choice is controlled by success and failure. Non-deterministic and biased choice differ in the sense if there is a preferred order on the arguments of the choice. We might accept different types for the argument strategies. Then, we implicitly restrict the applicability of the choice to the greatest lower bound. Choice between strategies of different types where the applicability of the choice extends to the smallest upper bound of the domains of the arguments is modelled by type-dependent choice. The corresponding combinators are not at all controlled by success and failure. Instead, the type of the term at hand determines the branch to be taken. The arguments in an asymmetric type-dependent choice are related via $\prec_\Gamma$, whereas the arguments in a disjoint type-dependent choice are characterised by mutual exclusion regarding the applicability in the typing sense. In conclusion, choice by success and failure and type-dependent choice complement each other. The division of labour between the two kinds of choice was also nicely illustrated by Example 18.

5.5 Variations on traversals

The selection of the traversal primitives of S'_γ has been driven by the requirement that we did not want to employ any universal representation type. For that reason the children are never directly exposed to the strategic program. Instead, one has to select the appropriate combinator to process the children. In this section, we want to indicate the potential for generalised or additional traversal primitives while keeping in mind the aforementioned requirement.

Order of processing children The reduction rules for the traversal primitives left the order of processing children largely unspecified. As for $\Box(\cdot)$, the order does not seem to be an issue since all the children are processed anyway, and the children can be processed in a completely independent manner. Actually, one could still argue that the order might be significant in the case that processing one of the children fails. In this case, the other children do not need to be considered anymore, and hence, a certain order might have an impact on performance or even on termination. As for $\Diamond(\cdot)$, a flexible

order is desirable for yet another reason. One might argue that in a certain application, the left-most or the right-most child should be preferred. To cope with such variations, one can consider a refined combinator $\Diamond_o(s)$ where we assume that the child to be processed by s is found according to the order constraint o. The options for o could be like search from left-to-right (say "$\rightarrow$"), or vice versa (say "$\leftarrow$"), or without any preferred order (say an empty o). Such an order constraint might be useful to employ or to enforce invariants of the traversed term representation. Thereby, order constraints are an issue of functional correctness. As for the type-unifying combinators for reduction and selection, order constraints make sense as well. In $\bigcirc_o^{s_\circ}(s)$, the constraint o could directly be used to control how the pairwise composition $s_\circ$ is applied to the processed children. A simple investigation of the original formalisation of $\bigcirc^{s_\circ}(s)$ in Figure 18 makes clear that a left-to-right reduction was specified (although it was not constrained if pairwise composition is intertwined with pre-processing the children). A certain order o for reduction might be relevant to cope with combinators $s_\circ$ which do not admit associativity and/or commutativity. As for selection, basically the same arguments apply to $\sharp_o(s)$ as to $\Diamond_o(s)$.

Pairwise composition vs. list processing One might argue that reduction as defined for $\bigcirc^{\cdot}(\cdot)$ could be defined in a simpler way without performing the pairwise composition as part of the reduction. Instead one could process all children and turn the results into a list. As these intermediate results will all be of the same type in the type-unifying case, basic list-processing functions could be employed to further reduce such lists, e.g., a list-like *foldl* or *foldr* (cf. [BW88, MFP91]). The main reason that we did not formalise reduction in this manner is the following. We do not want to cement a separation of processing the children and the pairwise composition of the intermediate results within the reduction semantics. If we postponed combining intermediate results of reduction until all children have been processed, we obtain an unnecessarily eager style of reduction.

Syntax

$$s \quad ::= \quad \cdots \mid (\!|s, s|\!)$$

Reduction $\boxed{s \mathbin{@} t \rightsquigarrow t'}$

Positive rules

$$\frac{s_{nil} \mathbin{@} () \rightsquigarrow t'_0}{(\!|s_{nil}, s_{cons}|\!) \mathbin{@} c \rightsquigarrow t'_0} \quad [\mathsf{fold}^+.1]$$

$$\frac{\begin{array}{ll} & s_{nil} \mathbin{@} () \rightsquigarrow t'_n \\ \wedge & s_{cons} \mathbin{@} (t_n, t'_n) \rightsquigarrow t'_{n-1} \\ \wedge & \cdots \\ \wedge & s_{cons} \mathbin{@} (t_1, t'_1) \rightsquigarrow t'_0 \end{array}}{(\!|s_{nil}, s_{cons}|\!) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow t'_0} \quad [\mathsf{fold}^+.2]$$

Negative rules

$$\frac{s_{nil} \mathbin{@} () \rightsquigarrow \uparrow}{(\!|s_{nil}, s_{cons}|\!) \mathbin{@} c \rightsquigarrow \uparrow} \quad [\mathsf{fold}^-.1]$$

$$\frac{s_{nil} \mathbin{@} () \rightsquigarrow \uparrow}{(\!|s_{nil}, s_{cons}|\!) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\mathsf{fold}^-.2]$$

$$\frac{\begin{array}{ll} & \exists i \in \{1, \ldots, n\}. \\ & s_{nil} \mathbin{@} () \rightsquigarrow t'_n \\ \wedge & s_{cons} \mathbin{@} (t_n, t'_n) \rightsquigarrow t'_{n-1} \\ \wedge & \cdots \\ \wedge & s_{cons} \mathbin{@} (t_i, t'_i) \rightsquigarrow \uparrow \end{array}}{(\!|s_{nil}, s_{cons}|\!) \mathbin{@} f(t_1, \ldots, t_n) \rightsquigarrow \uparrow} \quad [\mathsf{fold}^-.3]$$

Figure 27: A generic (intentionally type-unifying) traversal combinator for folding the children

Reduction vs. folding It turns out that reduction as modelled by the combinator $\bigcirc^{\cdot}(\cdot)$ can be further generalised to more strongly intertwine processing of children and composition (as opposed to a mere order constraint). This is useful if the way how a child should be processed depends on

previously processed children. In fact, one can define a combinator $(\!|\cdot,\cdot|\!)$ which folds directly over the children of a term very much in the sense of folding a list according to the pattern *foldr* for lists. The first fundamental difference to a list-fold is that we have to cope with an intentionally heterogeneous list corresponding to the children of a given term. For that reason, the list cannot be exposed to the strategic program, but folding must be necessarily a primitive operator. The second difference between a list-fold and a generic fold is that in the latter case, we need a generic ingredient to fold a child and an intermediate value of reduction. The reduction semantics of the (right-associative) combinator $(\!|\cdot,\cdot|\!)$ is defined in Figure 27. Informally, the strategy $(\!|s_{nil}, s_{cons}|\!)$ folds the children of the term at hand. The first strategy parameter s_{nil} encodes the initial value for folding. In the case of a constant symbol, this strategy defines the result of folding (cf. [fold$^+$.1]). The second strategy parameter s_{cons} is used to compose a child with an intermediate value of folding (cf. [fold$^+$.2]).

Example 21 *Let us attempt a reconstruction of the strategy* Flatten *from Figure 4. For convenience, we first show the original definition in terms of* $\bigcirc^{\cdot}(\cdot)$*. Then, we show a reconstruction which employs the combinator* $(\!|\cdot,\cdot|\!)$*.*

$$\begin{aligned}\textsc{Flatten}(\nu,\nu_u,\nu_\circ) &= (\textsc{Con};\underline{\perp};\nu_u)+(\textsc{Fun};\textstyle\bigcirc^{\nu_\circ}(\nu))\\ &= (\!|\nu_u,\langle\nu,\epsilon\rangle;\nu_\circ|\!)\end{aligned}$$

This reconstruction immediately illustrates why the combinator $(\!|\cdot,\cdot|\!)$ *is more powerful than the combinator* $\bigcirc^{\cdot}(\cdot)$*. As the second argument in the above application of* $(\!|\cdot,\cdot|\!)$ *points out, a child is processed independent of the intermediate value of reduction (cf. the congruence* $\langle\nu,\epsilon\rangle$*), and both values are composed in a subsequent step by* $\nu_\circ$*. This is precisely the scheme underlying* $\bigcirc^{\cdot}(\cdot)$*. In principle,* $(\!|\cdot,\cdot|\!)$ *could also be used in a mode where the second argument observes both the child at hand and the intermediate result at the same time.* ◇

We cannot type the combinator $(\!|\cdot,\cdot|\!)$ in a simple way in our present type system. Consider the intended type of s_{cons} in $(\!|s_{nil}, s_{cons}|\!)$. The strategy should process a pair consisting of a term of any type (corresponding to some child), and a term of the distinguished type for type-unification. One could introduce another distinguished generic type for products for that purpose. Unfortunately, this is not sufficient because to compose strategies of this new type, a more generic treatment of tuples is due. Due to these complications we do not attempt to work out typing rules for $(\!|s_{nil}, s_{cons}|\!)$.

Environments and states There are further type schemes than those underlying type-preserving and type-unifying traversals. In the following table, we list three further schemes:

TP	$\equiv$	$\forall\alpha.\ \alpha\to\alpha$	(Type preservation)
TU(τ)	$\equiv$	$\forall\alpha.\ \alpha\to\tau$	(Type unification)
TE(τ)	$\equiv$	$\forall\alpha.\ \langle\alpha,\tau\rangle\to\alpha$	(Environment passing)
TA(τ)	$\equiv$	$\forall\alpha.\ \langle\alpha,\tau\rangle\to\tau$	(Accumulation)
TS(τ)	$\equiv$	$\forall\alpha.\langle\alpha,\tau\rangle\to\langle\alpha,\tau\rangle$	(State passing)

Note that most generic type schemes are actually type constructors. The type (constructor) TE(τ) denotes all strategies which map a term of any type to a term of the same type in the presence of an additional argument e of type τ. Consequently, the domain of TE(τ) is a tuple type. Assuming that e will passed down during traversal we might think of e as an environment. In a similar way, the type scheme for accumulation can be explained. A strategy of type TA(τ) takes a pair $\langle x,a\rangle$ where x can be of any term type and a is of type τ, and it returns the resulting value a' of the accumulator. Roughly, state passing according to the type TS(τ) corresponds to a combination of TP and TA(τ).

The ultimate question is of course how to inhabit the above type schemes. As for TP and TU($\cdot$), we provided dedicated traversal primitives in S'_γ. We cannot derive sensible combinators for the other type schemes solely in terms of the S'_γ primitives. However, corresponding variants of the existing

TP

$$\frac{\exists i \in \{1,\ldots,n\}.s \mathbin{@} t_i \rightsquigarrow t_i'}{\Diamond(s) \mathbin{@} f(t_1,\ldots,t_n) \rightsquigarrow f(t_1,\ldots,t_i',\ldots,t_n)} \quad [\mathsf{one}^+]$$

$\mathsf{TE}(\cdot)$

$$\frac{\exists i \in \{1,\ldots,n\}.s \mathbin{@} \langle t_i, e\rangle \rightsquigarrow t_i'}{\Diamond(s) \mathbin{@} \langle f(t_1,\ldots,t_n), e\rangle \rightsquigarrow f(t_1,\ldots,t_i',\ldots,t_n)} \quad [\mathsf{one}^+]$$

$\mathsf{TS}(\cdot)$

$$\frac{\exists i \in \{1,\ldots,n\}.s \mathbin{@} \langle t_i, a\rangle \rightsquigarrow \langle t_i', a'\rangle}{\Diamond(s) \mathbin{@} \langle f(t_1,\ldots,t_n), a\rangle \rightsquigarrow \langle f(t_1,\ldots,t_i',\ldots,t_n), a'\rangle} \quad [\mathsf{one}^+]$$

Figure 28: Variants of $\Diamond(\cdot)$

traversal primitives are easily defined. Consider, for example, $\mathsf{TE}(\cdot)$ which is intended for environment passing. Dedicated traversal combinators should not simply apply a given strategy to some children, but the environment has also to be pushed through the term. Let us characterise a corresponding variant for $\Diamond(\cdot)$. If $\Diamond(s)$ is applied to $\langle f(t_1,\ldots,t_n), e\rangle$, then the strategy application to rewrite a suitable child t_i is of the form $s \mathbin{@} \langle t_i, e\rangle$. The positive rules for the variants of $\Diamond(\cdot)$ for TP, $\mathsf{TE}(\cdot)$ are $\mathsf{TS}(\cdot)$ shown in Figure 28. All other traversal primitives admit similar variations. As an aside, in a (higher-order) functional programming context, monads [Mog89, Spi90, Wad92] can be employed to merge effects like environment or state passing with the basic scheme of type-preserving or type-unifying traversals. This observation also reminds us of the possibility of different kinds of reducts, e.g., lists or sets of terms instead of an optional term. In the former case, the list monad could be employed for non-determinism. In S_γ', we actually hardwired the latter choice (corresponding essentially to the maybe monad).

6. Implementation

On the one hand, we have implemented a prototype implementation of S_γ' completely independent of any existing rewriting framework. We have chosen Prolog for such a self-contained implementation due to Prolog's suitability for prototyping language syntax, static and dynamic semantics (cf. [LR01]). On the other hand, we have also analysed the impact of an integration of the S_γ' concepts into an existing rewriting framework. This investigation was driven by our primary goal to provide a simple and useful model of generic programming on the grounds of basically first-order and many-sorted rewriting. We discuss these two facets of implementation in the sequel.

Prolog prototype It is well-known that deduction rules map nicely to Prolog clauses (cf. [Des88]). Prolog's unification (say logical variables) and backtracking enable the straight execution of a large class of deduction systems. In fact, this is the case for the formalisation from the present article. The formalisation was mapped to Prolog in the following manner. Well-formedness, well-typedness, static elaboration and reduction judgements constitute corresponding predicate definitions. Terms are represented as (ground and basically untyped) Prolog terms. Strategic programs are represented as files of period-terminated Prolog terms encoding type declarations, and strategy definitions. In this manner Prolog I/O can be used instead of parsing. Prolog variables are used to encode term variables (in rewrite rules), strategy variables (in strategy definitions), and term type variables (in type declarations). From a meta-programming perspective (cf. [BA82, Bow98]), the implementation of the S_γ' judgements corresponds to a meta-program, and we use a non-ground representation of object-programs (cf. [HL88]), i.e., strategic programs.

The encoding of strategies is illustrated in Figure 29. On the left side, the Prolog encoding for

Reusable strategy definitions

```
try : tp -> tp.
try(S) = S <+ id.

repeat : tp -> tp.
repeat(S) = try(S; repeat(S)).

oncebu : tp -> tp.
oncebu(S) = one(oncebu(S)) <+ S.

stoptd : tp -> tp.
stoptd(S) = S <+ all(stoptd(S)).
```

Traversals (I) and (II)

```
data nat   = zero | succ(nat).
data gsort = c | g(gsort) | gprime(gsort).
data other = b(other,gsort) | f(other,nat) | h(nat,gsort).

inc : nat -> nat.
inc = N -> succ(N).

traverseI : tp.
traverseI = stoptd(inc < tp).

traverseII : tp.
traverseII = oncebu((g(X) -> gprime(X)) < tp).
```

Figure 29: Strategic programs in Prolog

some reusable strategies which were originally defined in Figure 3 and Figure 4 are shown. On the right side, the strategies for the traversal problems (I) and (II) from the introduction are shown. The rewrite rule to increment a natural is for example represented as `N -> succ(N)`. One can see that the mapping is very straight (modulo notational conventions posed by Prolog such as the period to terminate a term to be read from a file). `tp` denotes the type TP. The `data` directive is used to declare algebraic data types (contributing to the signature part of the context Γ of a strategic program). We do not declare types of term variables since it is very easy to infer their types using the non-ground representation for rewrite rules.

Reduction of $\Box(\cdot)$

```
apply(G,all(S),T0,T1)
 :-
    T0 =.. [F|Ts0],
    map(apply(G,S),Ts0,Ts1),
    T1 =.. [F|Ts1].
```

Elaboration of $\cdot \lhd \cdot$

```
elaborate(G,S<T,(S:Pi)<T)
 :-
     wtStrategy(G,S,Pi).
```

Reduction of $\cdot \lhd \cdot$

```
apply(G,(S:Tau->_)<_,T0,T1)
 :-
    T0 =.. [F|_],
    fInGamma(F,G,Tau,_),
    apply(G,S,T0,T1).
```

Figure 30: Implementation of S'_γ in Prolog

The implementation of S'_γ is illustrated with a few excerpts in Figure 30. We show some clauses for the predicates encoding reduction of strategy application, and static elaboration. The left-most excerpt shows the very simple implementation of the combinator $\Box(\cdot)$ in Prolog. Here we resort to the Prolog operator "`=..`" to access the children as a list, and we employ a higher-order predicate `map/3` to map the argument strategy over the children. In the middle, we show the encoding of the static elaboration rule from Figure 23. The right-most excerpt implements a reduction rule for an annotated application of $\cdot \lhd \cdot$. It uses the outermost function symbol to lookup the type of the current term. This option was mentioned in Section 5.2 (although the original formalisation favoured explicitly tagged terms).

As an aside, strategic programming can be integrated into Prolog in a way that it is immediately useful for the Prolog programmer. That is, Prolog programs can contain portions of strategic rewriting functionality. Such an approach is sketched in [LR01]. Essentially, strategy combinators can be represented as higher-order predicates. There is one exception, namely the concept of rewrite rules can be replaced by the concept of predicates. This is convenient for (logic) programmers, and it also simplifies the implementation since the machinery for unification, substitution, and α-conversion carries over from Prolog.

Integration into a rewriting framework Due the conceptual simplicity of S'_γ, we consider S'_γ as a lightweight proposal for typed generic traversals. This claim is supported by the simplicity of its integration into existing rewriting frameworks. Let us consider, for example, how S'_γ can be integrated into the many-sorted strategic rewriting framework ELAN which does not (yet) support generic traversal combinators. For brevity, we restrict ourselves to S'_{TP} (cf. Section 4.3). In ELAN, there is a module for strategy combinators parameterised by a sort. One needs to instantiate this module for each relevant sort in the given signature. Consequently, the strategy combinators are overloaded for all possible sorts. Thus, one can say that typing for strategy expressions is realised in a sense by parsing. Generic type-preserving traversals are particularly simple to implement in such a setting. First, we add the distinguished sort TP and the combinators specifically defined on it, namely ϵ, δ, $\Box(\cdot)$, and $\Diamond(\cdot)$. The sort and the symbols can be defined in a module dedicated to TP. Then, we need to support the notions of extension and restriction. The corresponding symbols have to be declared in the parameterised module for many-sorted strategies (since we need them for each sort). As for restriction, we can easily realise the appealing implicit restriction approach via ELAN's support for anonymous injections. As for extension, we simply include $\cdot \lhd \cdot$. In fact, we can omit the generic type in $\cdot \lhd \cdot$, and we simply use the notation $s\Uparrow$ to lift a many-sorted strategy s to TP. Note that the lifting combinator is overloaded in the same sense as the other basic strategy combinators. Hence, each application of the lifting combinator in a compound strategy refers to a specific sort, and static elaboration is not needed to determine the domain of the strategy to be lifted. The rewrite rules for $\Box(\cdot)$ and $\Diamond(\cdot)$ are either generated by a pre-processor (in similarity to the dynamic typing and implosion + explosion approach in [BKKR01], or the the rewriting engine must be aware of these (and the symbols for extension and restriction too). If the latter road is taken, we have to assume that the type of the term at hand is always known to the rewriting engine. To summarise, the described simple implementation is enabled by some fundamental concepts of ELAN, namely parameterised modules (needed for sort-indexed overloading of strategy combinators), and a general parsing method (to cope with highly overloaded signatures and local ambiguities). Thereby, we get typing and type-dependent reduction for free. A simple implementation is also conceivable for other frameworks for rewriting or algebraic specification, e.g., ASF+SDF [BHK89, Kli93, BHJ$^+$01], Maude [MCLM96, CDE$^+$99], and CASL [ABK$^+$01].

7. Related work

A comparison of the S'_γ combinator suite and other strategic frameworks was already given in Section 2.5. Here, we want to comment on efforts in the rewriting community related to traversals, genericity, and typing. Afterwards, some proposals for genericity in (functional) programming are related to the rewriting-based calculus S'_γ. Finally, a number of further related concepts for the design of type systems are briefly discussed.

Strategic rewriting There is no previous work on statically typed generic traversal strategies. In particular, the type-dependent combinators which we proposed in the article constitute a unique contribution. These combinators are crucial for the dynamic assembly of overloaded and generic strategies. Generic traversal combinators have been defined in the ρ-calculus (cf. $\Psi(s)$ and $\Phi(s)$ in [CK99] corresponding to $\Box(s)$ and $\Diamond(s)$) but these definitions cannot be typed in the available typed fragments of the ρ-calculus [CKL01]. In the ongoing research on the ρ-cube (in the sense of the λ-cube [Bar92]), it is conceivable that the genericity underlying S'_γ can contribute to an instance of the ρ-calculus with coverage of generic traversals. In [MCMO01], generic (namely polytypic) entities are defined in terms of the reflection and meta-programming capabilities of Maude. However, the polytypic definitions are merely typed against the representation types of the object programs. Also, a mixture of many-sorted and generic functionality is not considered. In [BKV01], a fixed set of traversal strategies is supported by so-called traversal functions extending the algebraic specification formalism ASF [BHK89]. The programmer cannot define new traversal schemes. The central idea is to declare function symbols which serve implicitly as traversal functions based on predefined strategies

for top-down, bottom-up and accumulating traversals. The programmer refines traversal functions by rewrite rules with the traversal symbols as outermost symbol. However, dynamic update of traversals (in the sense of type-dependent choice) is not enabled. In [BKKR01], dynamic types [ACPR92] are employed to cope with some generic (traversal) strategies in ELAN. A universal data type *any* is used to represent terms of any sort. For that purpose, a parameterised module $any[X]$ is offered which can be instantiated for any sort which is subject to generic programming via the any data type. 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. Thereby, traversals can access the children of a term. The actual implementation employs a pre-processing approach to obtain an instantiation of the interface of $any[X]$ for a given X. At a first glance, specifications relying on *any* are type-safe. However, if the manipulated terms of sort *any* do not represent terms of the "intended sorts" an ultimate term implosion will simply fail. Thus, type safety comes at the expense of potential for failure. This problem is irrelevant for S'_γ since types are statically enforced, and there is no universal (and hence imprecise) sort like *any*. There is no representation type at all.

Generic programming In programming language research, especially in functional programming, several forms of genericity have been studied. We need to consider forms which go beyond parametric polymorphism [Mil78, Gir72, Rey74]. Despite the complexity and expressiveness of such systems (usually based on some kind of extended typed λ-calculus) it is not immediately clear how to define S'_γ-like generic traversal combinators in those settings. We discuss a few limitations of some proposals for generic programming. Some actual attempts to typefully encode generic traversals in functional languages can be found in [LV00, LV01]. Let us start with dynamic typing [ACPR92]. The use of dynamic typing to perform generic traversals is not fully typeful because a dynamic type is easily spread into too many parts of a program. Also, the ability to destruct and construct terms generically goes beyond the basic idea of dynamic typing. One can of course choose a representation type (possibly supported by a language extension) which allow us to access subterms but then it is not trivial to avoid the implosion problems which we mentioned for ELAN above. In [LV01], we show how to use S'_γ-like traversal combinators to effectively hide dynamic typing or a representation type resp. from the programmer of both generic traversals and reusable schemes. Let us consider other approaches to generic programming. The S'_γ model poses a challenge regarding the kind of second-order type schemes underlying the traversal primitives. Let us reconsider the types of $\diamond(\cdot)$ and $\sharp(\cdot)$ where we expand TP and $\mathsf{TU}(\cdot)$ according to the underlying type schemes:

$$\begin{aligned} \diamond(\cdot) &: (\forall\alpha.\ \alpha \to \alpha) \to (\forall\beta.\ \beta \to \beta) \\ \sharp(\cdot) &: \forall\alpha.\ (\forall\beta.\ \beta \to \alpha) \to (\forall\gamma.\ \gamma \to \alpha) \end{aligned}$$

Just looking at the shape of these types, they remind us of rank-2 type systems. If we attempt to employ, for example, polytypism [JJ97, Mee96] to implement $\diamond(\cdot)$ or other traversal primitives, we fail because even the more recent language proposals [Hin99, Hin00] are restricted to top-level quantification of polytypic parameters. Let us consider another complication. Any kind of traversal combinator needs to perform traversal of terms (of algebraic data types). With intensional and extension polymorphism [DRW95, CWM99] one can also encode functions which are parametric in a type and where the behaviour is defined via a type-case w.r.t. to the run-time type (as opposed to parametric polymorphism). However, the corresponding systems do not cover algebraic data types in the first place (but only products, function space, and basic data types). Hence, one cannot encode traversal combinators. In polytypic languages such as PolyP [JJ97] or Generic Haskell [Hin99], one can traverse elements of algebraic data types since they are treated as sums of products in the polytypic scheme. One can also provide specific cases for distinguished data types statically (cf. ad-hoc definitions in [Hin99]). However (in addition to the aforementioned rank-2 issue), polytypic language proposals do not involve the concept of dynamic composition and update of generic definitions.

More on type systems Let us briefly discuss some more relevant ideas and concepts from the field of type systems. Type systems in the tradition of the λ-cube usually admit type-erasure (cf. [Bar92, BLRU97]), i.e., reduction leads to the same result even if type annotations are removed. By contrast, in S'_γ, the types which are involved in strategy extension are semantically relevant. A similar kind of type-dependent reduction has been considered for an extended λ-calculus $\lambda\&$ in [CGL95] for overloading functions. Type-dependent reduction is used in $\lambda\&$ to resort to the most appropriate "branch" of a function based on the run-time type of the argument. Other models of overloading are usually based on static type information only, e.g., the popular approach for type classes in functional programming [Jon95]. Type-dependency in the sense of $\lambda\&$ is essential to model late binding (in the object-oriented reading). Generic traversals are not considered in $\lambda\&$. In the strategic setting of S'_γ, type-dependent reduction is basically employed to decide if a strategy can be applied at all (and reduction fails otherwise).

An interesting question is what the deeper semantic interpretation of generic and overloaded strategies should be. Let us first look at overloaded types because then the meaning of generic strategies can derived in a simple way. One might suggest intersection types as a model (cf. [CDCV81, BDCd95]). If a function f (say strategy) is of type $a \cap b$ then we assume that f can play the role of both an element of type a and of type b. In [CGL95], this interpretation is challenged for the case that overloaded functions rely on type-dependent reduction, and thereby the selection of the role is crucial for the computation. In fact, in [Rey91], a coherence property is required (for intersection types and others) such that the selection of the role should not have an impact on the computation. Consequently, we prefer a different interpretation of overloaded strategies. That is, they correspond to sort-indexed families of (first-order) functions. This interpretation also leads to a useful model for generic strategies. That is, a generic strategy is a family which is parametric in the set of sorts to be covered.

8. Conclusion

In the present article, we developed a typed calculus S'_γ for term rewriting strategies. The main contribution of the article is that generic traversals are covered, namely generic type-preserving and type-unifying ones. The typed model for generic traversals is based on a few simple traversal primitives and two distinguished generic types. The emphasis in the design of S'_γ was to obtain a simple, self-contained model of typed generic traversals on the grounds of basically many-sorted, first-order term rewriting. This leads to a conceptually simple type system which is straightforward to implement. To explain what we mean by "simple type system" and "straightforward implementation", we mention that the development of the Prolog prototype which we discussed earlier took two days. Contrast that with other approaches to generic programming such as [DRW95, JJ97, CWM99, Hin99] which usually require(d) several man years of design and (prototype) implementation. Hence, the conclusion is that basically many-sorted and first-order rewriting can easily be made fit to allow for generic programming (in the sense of term traversals).

Applications of strategic programming (in program transformation) are motivated in abundance elsewhere (cf. [VBT98, LV00, Vis01, BKV01, LV01]). In the present article, we have rather focused on the motivation and the definition of typeful strategic programming. We have shown that types make strategic programming safer and more structured. At a language design level, our main contribution is the S'_γ-like style of strategy extension. Thereby, one can combine specific and generic functionality in a very flexible manner. Actually, one can compose and update generic and overloaded strategies dynamically as opposed to a static scheme to derive generic entities. Still, strategies in S'_γ are statically typed.

References

[ABK+01] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 2001. To appear.

[ACPR92] 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 Applications*, pages 92–103, San Francisco, U.S.A., June 1992. Association for Computing Machinery.

[BA82] K.A. Bowen and R. A.Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic programming*, volume 16 of *APIC studies in data processing*, pages 153–172. Academic Press, 1982.

[Bar92] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[BDCd95] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119(2):202–230, June 1995.

[BHJ+01] M.G.J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Oliver, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. The ACM Press in cooperation with Addison-Wesley, 1989.

[BKK96] P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In Meseguer [Mes96].

[BKK+98] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In Kirchner and Kirchner [KK98].

[BKKR01] 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.

[BKV01] M.G.J. van den Brand, P. Klint, and J. Vinju. Term Rewriting with Traversal Functions. Technical report, CWI, Amsterdam, 2001. In preparation.

[BLRU97] S. van Bakel, L. Liquori, S.R. della Rocca, and P. Urzyczyn. Comparing cubes of typed and type assignment systems. *Annals of Pure and Applied Logic*, 86(3):267–303, July 1997.

[Bow98] A. Bowers. *Effective Meta-programming in Declarative Languages.* PhD thesis, Department of Computer Science, University of Bristol, January 1998.

[BSV00] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of Components for Software Renovation Factories from Context-free Grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.

[BW88] R. Bird and P. Wadler. *An Introduction to Functional Programming.* Prentice-Hall, 1988.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. The Maude System—System Description. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *LNCS*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[CK99] H. Cirstea and C. Kirchner. Introduction to the Rewriting Calculus. Rapport de recherche 3818, INRIA, December 1999.

[CKL01] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In Furio Honsell, editor, *Foundations of Software Science and Computation Structures*, volume 2030 of *LNCS*, pages 168–183, Genova, Italy, April 2001.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[CWM99] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 301–312. ACM, June 1999.

[Des88] T. Despeyroux. TYPOL: A formalism to implement natural semantics. Technical report 94, INRIA, March 1988.

[DRW95] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[Fow99] Martin Fowler. *Refactoring—Improving the Design of Existing Code.* Addison Wesley, 1999.

[Geu93] H. Geuvers. *Logics and Type Systems.* PhD thesis, Computer Science Institute, Katholieke Universiteit Nijmegen, 1993.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GL01] B. Gramlich and S. Lucas, editors. *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume SPUPV 2359, Utrecht, The Netherlands, May 2001. Servicio de Publicaciones - Universidad Politécnica de Valencia.

[Hin99] R. Hinze. A generic programming extension for Haskell. In E. Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28.

[Hin00] R. Hinze. A New Approach to Generic Functional Programming. In T.W. Reps, editor,

Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21, pages 119–132, January 2000.

[HL88] P.M. Hill and J.W. Lloyd. Analysis of Meta-Programs. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–51. MIT-Press, 1988.

[Jeu00] J. Jeuring, editor. *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[JJ97] 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.

[Jon95] M.P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 97–136. Springer-Verlag, 1995.

[JV01] P. Johann and E. Visser. Fusing Logic and Control with Local Transformations: An Example Optimization. Technical report, Institute of Information and Computing Sciences, Universiteit Utrecht, 2001.

[Kah87] G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, 19–21 February 1987. Springer-Verlag.

[KK98] C. Kirchner and H. Kirchner, editors. *Proceedings of the International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *ENTCS*, Pont-à-Mousson, France, September 1998. Elsevier Science.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology, 2(2)*, pages 176–201, 1993.

[Läm01] R. Lämmel. Generic Type-preserving Traversal Strategies. In Gramlich and Lucas [GL01].

[LMS93] G. Longo, K. Milsted, and S. Soloviev. The Genericity Theorem and the Notion of Parametricity in the Polymorphic λ-Calculus. *Theoretical Computer Science*, 121(1–2):323–349, 1993.

[LR01] R. Lämmel and G. Riedewald. Prological Language Processing. In M.G.J. van den Brand and D. Parigot, editors, *Proc. LDTA'01*, volume 44 of *ENTCS*. Elsevier Science, April 2001.

[LV00] R. Lämmel and J. Visser. Type-safe Functional Strategies. In *Draft proc. of SFP'00, St Andrews*, July 2000.

[LV01] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. Technical report, CWI, Amsterdam, aug 2001.

[LVK00] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [Jeu00], pages 46–59.

[MCLM96] S. Eker M. Clavel, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [Mes96].

[MCMO01] F. Duran M. Clavel and N. Marti-Oliet. Polytypic Programming in Maude. In K. Futatsugi, editor, *ENTCS*, volume 36. Elsevier Science, 2001.

[Mee96] L. Meertens. Calculate Polytypically! In H. Kuchen and S.D. Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume

1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.

[Mes96] 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 *ENTCS*, September 1996.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[MR92] Q. Ma and J.C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 7th International Conference, PA, USA, March 1991, Proceedings*, volume 598 of *LNCS*, pages 1–40. Springer-Verlag, 1992.

[Opd92] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. thesis, University of Illinois, 1992.

[Pau83] L.C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, August 1983.

[Rey74] J.C. Reynolds. Towards a Theory of Type Structures. In *Programming Symposium (Colloque sur la Programmation, Paris)*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

[Rey91] J.C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 657–700. Springer-Verlag, September 1991.

[Sch94] D.A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series. MIT Press, 1994.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[VB98] E. Visser and Z. Benaissa. A Core Language for Rewriting. In Kirchner and Kirchner [KK98].

[VBT98] 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, September 1998.

[Vis00] E. Visser. Language Independent Traversals for Program Transformation. In Jeuring [Jeu00], pages 86–104.

[Vis01] E. Visser. A Survey of Strategies in Program Transformation Systems. In Gramlich and Lucas [GL01].

[Wad89] P. Wadler. Theorems for Free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, 1989.

[Wad92] Philip Wadler. The essence of functional programming. In ACM, editor, *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–14. ACM Press, 1992.

Table of Contents