*Electronic Notes in Theoretical Computer Science 57 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume57.html*](http://www.elsevier.nl/locate/entcs/volume57.html) *23 pages*

*Generic type-preserving traversal strategies Ralf Lammel* [*1*](#_bookmark0)

*CWI, Kruislaan 413, NL-1098 SJ Amsterdam*

*Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam*

*Abstract*

*A typed model of strategic rewriting with coverage of generic traversals is de- veloped. The corresponding calculus o ers, for example, a strategy operator 2( ), which applies the argument strategy to all immediate subterms. To provide a ty- peful model for generic strategies, one has to identify signature-independent, that is, generic types. In the present article, we restrict ourselves to TP|the generic type of all T ype-Preserving strategies. TP is easily integrated into a standard many-sorted type system for rewriting. To inhabit TP, we need to introduce a (left-biased) type-driven choice operator & . The operator applies its left argu- ment (corresponding to a many-sorted strategy) if the type of the given term ts, and the operator resorts to the right argument (corresponding to a generic default) otherwise. This approach dictates that the semantics of strategy application must be type-dependent to a certain extent.*

*1 Introduction Strategic rewriting*

*Several frameworks for rewriting o er means to describe strategies (as op-* posed to frameworks which assume a xed built-in strategy for normalisa- tion/evaluation). Strategies are supported, for example, by the speci cation formalisms Maude [[14](#_bookmark36)[,7]](#_bookmark29) and ELAN [[3,4].](#_bookmark26) Also, the -calculus [[5]](#_bookmark27) is very sui- table for the de nition of strategies. The programming language Stratego [[20](#_bookmark42)] based on system S [[21](#_bookmark43)] is entirely devoted to strategic programming. The idea of rewriting strategies goes back to Paulson's work on higher-order implemen- tation of rewriting strategies [[18](#_bookmark40)]. Strategies are useful to describe evaluation and normalisation strategies, e.g., to control rewriting for some rewrite rules which are not con uent or terminating when considered as a standard rewrite system. Strategies are particularly useful for the speci cation of traversals.

*1 Email:* [*Ralf.Laemmel@cwi.nl*](mailto:Ralf.Laemmel@cwi.nl)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

b

g g



f

h

5

3 c c

(II)



(I)



(III)

(IV)

b b **true** [3,5]

g g



f

h

6



f

g

h

g’

5

4 c c 3 c c

*Figure 1. Illustration of generic traversals*

*To describe traversals in standard rewriting (without extra support for traver-* sals), one has to resort to auxiliary function symbols, and rewrite rules have to be used to perform the actual traversal for the signature at hand (usually one rewrite rule per term constructor). Generic traversal primitives support an important dimension of genericity in programming. In [[19](#_bookmark41)], for example, generic traversal strategies are used for language implementation: Algorithms for free variable collection, substitution, uni cation and others are de ned in a generic, that is, language-independent manner by suitably parameterised generic traversals.

*Examples of generic traversals*

*In Figure* [*1,*](#_bookmark1) *four examples (I){(IV) of intentionally generic traversals are* illustrated. In (I), all naturals in the given term (say tree) are incremented as modelled by the rewrite rule N ! succ(N ). We need to turn this rule into a traversal strategy because the rule on its own is not terminating when considered as rewrite system. The strategy should be generic, that is, it should be applicable to any term. In (II), a particular pattern is rewritten according to the rewrite rule g(P ) ! g0 (P ). Assume that we want to control this replacement so that it is performed in bottom-up manner, and the rst matching term is rewritten only. The strategy to locate the desired node in the term is completely generic. The examples (III) and (IV) are examples of type-changing traversals, actually these are type-unifying traversals according

*to* [*[12*](#_bookmark34)[*,11].*](#_bookmark33) *In (III), we might test some property of the term, e.g., if naturals* occur at all. In (IV), we collect all the naturals in the term using a left-to-right traversal.

*Value of typing*

*The contribution of the article is a type system which covers generic tra-* versals as needed for the examples above. In the present article, we restrict ourselves to type-preserving strategies. A more exhaustive treatment including type-changing strategies can be found in [[10].](#_bookmark32) Let us understand why types are valuable. In general, typing should obviously prevent us from constructing ill-typed terms. Generic traversals typically employ many-sorted ingredients (say rewrite rules). A type system and the corresponding dynamic semantics should ensure that the speci c ingredients are applied in a type-safe manner. Consider, for example, the rewrite rule for incrementation N ! succ(N ) as assumed in example (I) above. This one-step rewrite rule should only be app- lied to naturals during a traversal. On the other hand, the complete traversal must be able to process any term. A type system should also prevent the pro- grammer from combining speci c and generic strategies in certain undesirable ways. Consider, for example, a left-biased choice ` + Id where the ordinary rewrite rule ` is applied if possible, and otherwise the \generic default" Id triggers. One might argue that this strategy is generic because Id is appli- cable to terms of all possible sorts. Actually, we favour two other possible interpretations. Either we refuse this choice altogether (because we would insist on the types of the argument strategies to be the same), or we take the non-generic argument type as the type of the compound strategy. In fact, strategies should not get generic too easily since we otherwise loose the valua- ble precision of a many-sorted type system. Even if ill-typed types cannot be constructed, accidentally generic strategies are likely to refuse terms (leading to failure), or they leave terms unchanged in an untraceable manner.

*DiÆculties of typing*

*Some strategy combinators are easier to type than others. Combinators for* di erent kinds of choice, sequential composition, signature-speci c congruence operators and others are easy to type in a many-sorted setting. Some use of overloading and/or parametric types might be necessary. Indeed, ELAN is based on such a many-sorted type system. By contrast, generic traversal primitives (e.g., 2(s) to apply a strategy s to all immediate subterms of a given term as provided by Stratego or system S) are more challenging since standard many-sorted types are not applicable, and also other well-established concepts like polymorphism are insuÆcient to model the kind of genericity needed. Generic traversal strategies have to be applicable to terms of any sort (or at least to some class of types). Generic traversals are in a sense type- dependent (as opposed to polymorphism) since they are usually derived from speci c ingredients (say rewrite rules) to deal with some distinguished sorts

*in a speci c manner. We also refer the reader to* [*[19,11],*](#_bookmark33) *where it is argued* that typing generic traversals is diÆcult. Typing generic traversals is further complicated if type-changing strategies are covered [[10].](#_bookmark32)

*Contribution and structure of the article*

*In Section* [*2,*](#_bookmark2) *we shortly recall untyped strategies including primitives for* traversals. In Section [3,](#_bookmark13) we discuss standard many-sorted types for type- preserving strategies. [2](#_bookmark3) In Section [4,](#_bookmark17) we provide a type system which includes a generic strategy type TP for generic type-preserving strategies. To this end, we also need to introduce a type-dependent choice operator to mediate between many-sorted and generic strategies. In Section [5,](#_bookmark21) we discuss implementation issues. In the course of the article, we show that our type system for strategies is sensible from a strategic programmer's point of view. We envision that the presented type system disciplines strategic programs (employing generic traversals) in a useful and not too restrictive manner. We also show that generic type-preserving strategies can (more or less) easily be implemented. The article is concluded in Section [6](#_bookmark22).

*Acknowledgement*

*The work of the author was supported, in part, by NWO, in the project \Ge- neration of Program Transformation Systems". The ideas developed 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, Hora- tiu Cirstea, H el ene Kirchner, Claude Kirchner, Paul Klint, Pierre-E tienne Moreau, Christophe Ringeissen, Jurgen Vinju and Eelco Visser for discussions on the subject of the article. Finally, many thanks to the anonymous workshop referees for their constructive criticism.*

# *2 Untyped strategies*

*We set up a rewriting calculus very much inspired by ELAN, the -calculus,* and system S. We are very brief regarding explanations, examples, and details of the semantics. Some basic knowledge of strategic rewriting (as found in [[2](#_bookmark24)[,21,5])](#_bookmark27) is a helpful background for reading the present article.

*First, we give an overview on the strategy combinators we want to co-* ver, and we explain how to de ne new ones by means of strategy de nitions. Then, we explain the semantic model for strategy application. Finally, we

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

*devote a detailed explanation to the generic traversal primitives included in* our framework.

*2.1 Strategy combinators*

*We have the following grammar for strategy expressions:*

*s ::= t ! t j Id j Fail j s; s j s + s j s + s j f (s;::: ; s) j 2(s) j 3(s)*

*There is a form of strategy t ! t0 for one-step rules to be applied at the* root of the term. We adopt some common restrictions for rewrite rules. The left-hand side t determines the bound variables. (Free) variables on the right- hand t0 side also occur in t. If substitutions are applied, then we assume

*-conversion. Besides rule formation, there are standard primitives for the*

*identity strategy (Id), the failure strategy (Fail), sequential composition ( ; ),* symmetric choice ( + ), asymmetric left-biased choice ( + ). The strategy f (s1;::: ; sn) denotes the congruence strategy for the function symbol f . [3](#_bookmark5) The forms 2( ) and 3( ) are the traversal primitives. The strategy 2(s) applies the argument strategy s to all immediate subterms of the given term (say, children). The strategy 3(s) applies the argument strategy s to some child of the given term. We postpone discussing 2( ) and 3( ) in detail.

*Example 2.1 Let us consider the problem of ipping the top-level subtrees* in a binary tree with naturals at the leafs. We assume the following symbols to construct such trees:

*zero : Nat*

*succ : Nat ! Nat leaf : Nat ! Tree*

*fork : Tree Tree ! Tree*

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

*ip-top(leaf (N )) ! leaf (N )*

*ip-top(fork (T1; T2)) ! fork (T2; T1)*

*Alternatively, we can use a strategy ip-top without introducing additional* function symbols. We combine two one-step rewrite rules via + . Rule `1 models preservation of leafs, whereas rule `2 ips top-level subtrees:

*`1 = leaf (N ) ! leaf (N )*

*`2 = fork (T1; T2) ! fork (T2; T1)*

*ip-top = `1 + `2*

*3 I.e., the strategy f (s ;:::;s ) applies the argument strategies to the parameters of a term*

*with f as outermost symbol, otherwise the strategy fails unconditionally.*

*1*

*n*

*New strategy combinators can be de ned by means of an abstraction me-* chanism which we call strategy de nitions. Similar mechanisms are provided by ELAN, system S and Stratego. A de nition '( 1;::: ; n) = s introduces an n-ary strategy combinator '. An application '(s1;::: ; sn) of ' denotes the instantiation s[ 1 := s1;::: ; n := sn] of the body s of the de nition of '. Strategy de nitions can be recursive. [4](#_bookmark7)

*Example 2.2 Some sample de nitions are the following: try ( )= + Id*

*repeat( )= try ( ; repeat( ))*

*ip-top = try (`2)*

*The rst two strategies are generic. try (s) applies s, but behaves like Id if s* fails. repeat (s) repeatedly applies s as often as possible. The strategy ip-top is speci c in nature. It reconstructs the strategy of the same name introduced earlier in Example [2.1](#_bookmark4). The reconstruction illustrates the use of try.

*2.2 The semantic model*

*The application of a strategy s to a term t is denoted by hsit. As for the* dynamic semantics, we employ a judgement for strategy application ` hsit ; r where r is the reduct which is either a term t0 or failure denoted by \"". This model has been adopted from system S. Note that the given judgement is not suÆcient to de ne the semantics of applications of strategy de nitions. For that purpose, we had to propagate the de nitions via a context parameter. Note also that we assume that strategies are only applied to ground terms, and then also yield ground terms. [5](#_bookmark8) We employ a certain style for the speci cation of the deduction rules. We give positive rules for cases when the reduct is a term, and we give negative rules for the remaining cases with failure as the reduct.

*We show an excerpt of the evaluation judgement in Figure* [*2.*](#_bookmark9) *It covers the* positive and negative rules for asymmetric left-biased choice. These rules also illustrate why we need to include failure as reduct. Otherwise, the semantics could not query whether a certain application did not succeed (cf. [lchoice+:2]).

*2.3 Generic traversal primitives*

*Let us take a closer look at the generic traversal primitives to apply a strategy* s to all children (2(s)), or to some child (3(s)). The operators 2( ) and 3( ) are de ned like in system S. For brevity, we do not consider the hybrid operator 32 from system S which applies a strategy to one or more children.

*4 Recursive de nitions are common in ELAN, whereas system S and oÆcial Stratego em- ploy a special recursion operator : .*

*5 The assumption is well in line with standard rewriting, especially for ordinary rst-order rewrite rules.*

*Semantics of strategy application*

*` hsit ; r*

*Positive rules Negative rule*

*` hsit ; t0*

*` hs + s0 it ; t0*

*[lchoice+:1]*

*` hsit ; "*

*` hsit ; "*

*^ ` hs0 it ; t0*

*` hs + s0 it ; t0*

*[lchoice+:2]*

*^ ` hs0 it ; "*

*` hs + s0 it ; "*

*[lchoice ]*

*Figure 2. Positive and negative rules for application of left-biased choice*

*Similar operators can also be de ned in the -calculus (cf. (s) and (s) in*

[*[5*](#_bookmark27)*] corresponding to 2(s) and 3(s)).*

*In Figure* [*3,*](#_bookmark10) *we show some useful derivable generic traversal strategies* de ned in terms of 2( ) and 3( ). The de nitions are adopted from [[21](#_bookmark43)] except the last one. Note that is a meta-variable for strategies in all these de nitions.

|  |  |  |
| --- | --- | --- |
| *topdown ( )* | *= ; 2(topdown ( ))* | *(Apply in top-down manner)* |
| *bottomup( )* | *= 2(bottomup( ));* | *(Apply in bottom-up manner)* |
| *oncetd ( )* | *= + 3(oncetd ( ))* | *(Apply once in top-down manner)* |
| *oncebu( )* | *= 3(oncebu( )) +* | *(Apply once in bottom-up manner)* |
| *innermost ( )* | *= repeat (oncebu( ))* | *(Innermost evaluation strategy for )* |
| *stoptd ( )* | *= + 2(stoptd ( ))* | *(Apply in top-down manner with \cut")* |

*Figure 3. Some derived generic traversal strategies*

*Example 2.3 Let us solve the rst two problems (I) and (II) illustrated in* Figure [1](#_bookmark1) in the introduction of the article:

*nat = zero + succ(Id)*

*traverse(I) = stoptd (nat; N ! succ(N )) traverse(II) = oncebu(g(P ) ! g0 (P ))*

*nat is an auxiliary strategy testing for naturals in terms of the congruence* strategies for zero and succ. We use nat as type check to enable the applica- bility of the rewrite rule N ! succ(N ) in the de nition of traverse(I). Recall, traverse(I) is meant to increment all naturals. The corresponding strategy is de ned in terms of the generic strategy stoptd (cf. Figure [3)](#_bookmark10) which descends

*into the given term as long as the argument strategy does not succeed. This* is exactly the traversal scheme we need to increment all naturals in a tree. Encountering naturals in a top-down manner we apply the incrementation rule, but then we do not further descend into the term. If we used topdown instead of stoptd , we describe a non-terminating strategy. The de nition of traverse(II) is also very easy to read. traverse(II) nds the rst pattern of form g(x) in bottom-up manner (as required), and replaces it by g0 (x) (as expres- sed by the rewrite rule). Note the genericity of the traversals traverse(I) and traverse(II). They can be applied to any term. Of course, the strategies are somewhat speci c because they rely on some constant or function symbols, namely zero, succ, g, and g0 .

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

*The positive semantics rules for 2( ) and 3( ) are shown in Figure* [*4.*](#_bookmark12) *The* rule [all+:1] says that 2(s), when applied to a constant, immediately succeeds because there are no children which s has to be applied to. The rule [all+:2] directly encodes what it means to apply s to all children of a term f (t1;::: ; tn). Note that the function symbol f is preserved in the result. The de nition of 3(s) is similar. The rule [one+:1] says that s is applied to some subterm ti of f (t1;::: ; tn).

*Semantics of strategy application*

*` hsit ; r*

*` h2(s)ic ; c [all+:1]*

*` hsit1 ;* *1*

*t*

*0*

*^*

*^ ` hsitn ; n*

*t*

*0*

*[all+:2]*

*` h2(s)if (t ;::: ;t ) ; f (t0 ;::: ; t0 )*

*1 n 1 n*

*9i 2 f1;::: ; ng: ` hsiti ;* *i*

*t*

*0*

*[one+:1]*

*` h3(s)if (t ;::: ;t ) ; f (t ;::: ; t0 ;::: ;t )*

*1 n 1 i n*

*Figure 4. Evaluation of 2(s) and 3(s)*

# *3 Many-sorted strategies*

*As a warm-up, we provide a type system for (non-generic, say many-sorted)* type-preserving strategies. First, we will explain the model for the type sy- stem. Then, we discuss the actual deduction rules in some detail. Finally, we

*discuss some design properties of the type system, mainly to prepare it for an* extension to cover generic strategies (as developed in main part of the paper, that is, Section [4](#_bookmark17)).

*3.1 The type model*

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

*to range over strategy types. The forms of type expressions are initially* de ned by the following grammar:

*::= (Term types)*

*::= ! (Strategy types)*

*In the typing judgements, we use a context parameter to keep track of sorts*

*, to map constant symbols c, function symbols f , term variables x, strategy* variables , and combinators ' to types. We use the following grammar for contexts:

*::= ; j [ (Contexts as sets)*

*j j c : j f : ! (Signature part) j x : j : (Term and strategy variables) j ' : ! (Strategy combinators)*

*Thus, contains a many-sorted term signature, variable declarations (for term* variables and strategy variables), and combinator type declarations origina- ting from strategy de nitions. Let us state a few well-formedness require- ments. We assume that the various kinds of symbols and variables are not confused (i.e., there are di erent name spaces), and the symbols and variables are not associated with di erent types in (in particular, we do not consider overloading). All sorts used in some declaration have also to be introduced in . All declarations have to be well-formed (w.r.t. the well-formedness jud- gements de ned below). Note that is assumed to be static (say given) in all upcoming judgements. Thus, we assume explicit type declarations for the various kinds of variables and symbols. [6](#_bookmark14) It is easy to infer instead.

*The principal judgement of the type system is the type judgement for* strategies. It is of the form ` s : , and it holds if the strategy s is of strategy type in the context of .

*6 Declarations for variables, rewriting functions and strategies are common in several fra- meworks for rewriting, e.g., in ASF+SDF and ELAN.*

*3.2 Deduction rules*

*The deduction rules for the various judgements are shown in Figure* [*5.*](#_bookmark16) *For* brevity, we omit the typing rules for strategy de nitions.

*Type-preservation is prescribed by the well-formedness judgement for stra-* tegy types (cf. rule [pi:1]). Some other rules also explicitly enforce type- preservation (cf. rules [comp:1], [apply], [rule], [id], [fail], and [congr]). The type system for many-sorted strategies should not be regarded as a contribu- tion of the present article. It is rather straightforward, and it corresponds very much to the kind of type system assumed for ELAN. Let us read some infe- rence rules for convenience. Rule [apply] says that a strategy application hsit is well-typed if the strategy s is of type ! , and the term t is of type . The strategies Id and Fail have many types, namely any type ! where ` holds (cf. rules [id] and [fail]). The strategy types for compound strategies are regulated by the rules [seq], [choice], [lchoice], and [congr]. The compound strategy s1; s2 refers to an auxiliary judgement for composable types. As for many-sorted type-preserving strategies, composable types are trivially de ned according to rule [comp:1]. The compound strategies s1 + s2 and s1 + s2 are well-typed if both strategies s1 and s2 are of a common type which also determines the type of the compound strategy.

*3.3 Discussion*

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

*Example 3.1 The strategy ip-top from Example* [*2.1*](#_bookmark4) *is type-preserving. Thus, the type ip-top : Tree ! Tree can be approved for its de nition.*

*On the negative side, there is no way to assign types to certain other* strategies which we have seen so far. Certainly, we cannot assign types to generic traversals as they are not restricted to a speci c sort. But we cannot even assign types to some strategies which do not involve traversals. What are, for example, the types of try and repeat de ned in Example [2.2?](#_bookmark6) In a sense, these combinators take a type-preserving strategy, and return a type- preserving strategy. Like Id and Fail, the combinators try and repeat could be associated with many types. However, this view interferes with the ideal of unicity of typing. Actually, the typing rules for Id and Fail also violate unicity of typing, but as these are primitives, this violation can be regarded as an acceptable formulation of overloading, or as an encoding of an parametric type. [7](#_bookmark15)

*7 We will later discuss the possible employment of parametric types.*

*Term types*

*Strategies*

*2*

*`*

*`*

*Strategy types*

*`*

*` !*

*Terms*

*c : 2*

*[tau:1]*

*[pi:1]*

*[const]*

*` t :*

*^ ` t0 :*

*` s :*

*` t ! t0 : !*

*`*

*`*

*` Id : !*

*`*

*` t :*

*` Fail : !*

*` s1 : 1*

*^ ` s2 : 2*

*[rule]*

*[id]*

*[fail]*

*` c :*

*^ 1 Æ*

*2 ;*

*[seq]*

*f : 1 n ! 2*

*^ ` t1 : 1*

*^*

*^ ` tn : n*

*[fun]*

*` s1; s2 :*

*` s1 :*

*^ ` s2 :*

*` s1 + s2 :*

*[choice]*

*` f (t1;:::; tn ) :*

*x : 2 X*

*` x :*

*Composable types*

*1 Æ 2 ;*

*`*

*! Æ ! ; !*

*[var]*

*[comp:1]*

*` s1 + s2 :*

*` s1 + s2 :*

*f : 1 n ! 2*

*^ ` s1 : 1 ! 1*

*^*

*^ ` sn : n ! n*

*` f (s ;:::;s ) : !*

*1*

*n*

*[lchoice]*

*[congr]*

*Strategy application*

*` hsit :*

*` s : !*

*^ ` t :*

*` hsit :*

*[apply]*

*Figure 5. Many-sorted type-preserving strategies*

# *4 Generic strategies*

*An important property of 2(s) and 3(s) is that they are supposed to be* applicable to terms of any sort, i.e., they are generic. Clearly, this is also the case for Id and Fail. Contrast that with a rewrite rule. It is only applicable to a term of a speci c sort because of the way it is constructed from speci cally- typed terms. Note that the parameters of the traversal primitives have to be generically typed, too. Consider, for example, 2(s). The argument s must be potentially applicable to subterms of any sort. Thus, we need to add a form of generic strategy type to our type model. Then, we are able to assign types to strategies involving 2( ) and 3( ) (and, at the same time, we also resolve the unicity problems with Id and Fail, and we can assign types to strategies like try and repeat).

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

*4.1 The type of all type-preserving strategies*

*We extend our syntax for strategy types , namely we add one case for ge-* neric types . In this article, we only consider one particular generic type, namely TP representing the type of all T ype-P reserving strategies. In [[10],](#_bookmark32) we also consider type-changing strategies. Our grammar of types is extended as follows:

*::= j*

*::= TP*

*Example 4.1 All the strategies in Figure* [*3*](#_bookmark10) *are generic type-preserving stra-* tegies. Also, the argument strategy for all the de nitions is of type TP. The same holds for the strategies try and repeat de ned in Example [2.2](#_bookmark6). Thus, we assume the type TP ! TP for all these strategy de nitions.

*In Figure* [*6,*](#_bookmark18) *we extend the typing judgements. We use a partial order* on types to measure genericity of types. A many-sorted type is `'less" generic or general than a generic type. Rule [typeless:1] axiomatises TP. The rule says that ! TP for all well-formed . This directly encodes the idea of type-preserving strategies. A strategy of type TP can be applied to a term of any sort. Id and Fail are de ned to be (generic) type-preserving strategies in rules [id] and [fail]. 2(s) and 3(s) and their argument strategy s are also de ned to be type-preserving in rules [all] and [one].

*The type system strictly separates many-sorted strategies (such as rewrite* rules), and generic strategies (such as applications of 2( )). As for the moment being, we cannot turn many-sorted strategies into generic ones, neither the other way around. We will provide a corresponding re nement of the strategy

*Strategy types Application*

*` hsit :*

*`*

*` TP [pi:2]*

*Genericity*

*0*

*`*

*` s :*

*^ ` t :*

*^ !*

*` hsit :*

*[apply]*

*! TP*

*[typeless:1]*

*Strategies*

*` s :*

*Composable types*

*TP Æ TP ; TP [comp:2]*

*` Id : TP [id]*

*` Fail : TP [fail]*

*1 Æ 2 ;*

*` s : TP*

*` 2(s) : TP*

*[all]*

*` s : TP*

*` 3(s) : TP*

*[one]*

*Figure 6. Generic type-preserving strategies*

*calculus soon. The well-typedness rule for strategy application (cf. [apply])* certainly clari es how a generic strategy can be applied to a term of a speci c sort.

*4.2 Mediation between speci city and genericity*

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

*One approach to the inhabitation of TP is to use a generic default (initially* Id and Fail, but not just these) if the many-sorted strategy is not applicable for typing reasons. We might attempt to turn, for example, a rewrite rule ` into a generic strategy using the forms ` + Id or ` + Fail. This is not a good idea since the operator + is concerned with choice controlled by success and failure. What we are looking for in the context of quali cation of speci c strategies to become generic, is a di erent form of choice. We need a type- dependent form of choice where the speci c strategy is chosen if the actual term is covered by its domain. Otherwise the generic strategy (serving as a

*kind of default) should be chosen. We introduce a corresponding operator:* s ::= j s & s

*The left argument is the many-sorted strategy whereas the right argument is*

*the generic default. The static and dynamic semantics of the operator are* de ned in Figure [7.](#_bookmark19)

*Example 4.2 We recall the solution to the rst problem from the introduc-* tion as given in Example [2.3](#_bookmark11). The original (Stratego-like) solution is not typeable because a many-sorted strategy, namely nat; N ! succ(N ), is pas- sed to stoptd which expects a generic argument. We recover typeability by the following rede nition of traverse(I):

*traverse(I) = stoptd ((nat; N ! succ(N )) & Fail)*

*= stoptd ((N ! succ(N )) & Fail)*

*This solution illustrates that we can qualify speci c rewrite rules to become* generic by & . As an aside, the simpli cation to eliminate the test for naturals is enabled by the typed model. The type of the rewrite rule suÆciently restricts its applicability. The other strategy traverse(II) from Example [2.3](#_bookmark11) can be made t in the same manner.

*The operator & serves for asymmetric left-biased choice of strategies* based on the type of the term in the application. As the deduction rules detail, if s1 & s2 is applied to a term t of type , and the type of s1 coincides with

*! , then s1 is chosen (cf. [lplus+:1]). Otherwise, we resort to the generic*

*strategy s2 (cf. [lplus+:2]). Thus, & is di erent from + in the sense* that & is left-biased w.r.t. type-sensitivity, whereas + is left-biased

*w.r.t. success and failure. This separation makes it explicit where we want* to become generic. There is no hidden way how speci c ingredients can get generic accidentally. Without separating the two kinds of choice, strategies get too easily (say accidentally) generic and typeable.

*Note that the semantics judgement needs to be able to determine the type* of the speci c strategy, and the type of the given term. For that reason, we add the typing context to the judgement for strategy application. The negative semantics rules for & are also shown in Figure [7](#_bookmark19) since they are instructive. One can clearly see how the type of the term is used to determine the appli- cability of the left operand s1 in s1 & s2 during strategy application. This type-sensitivity might be regarded as a paradigm shift. We postpone discus- sing a way to eliminate the premises to determine the types of the ingredients of hs1 & s2it in the semantics rules.

*So far, we only considered one direction of mediation. We should also re ne* our type system so that generic strategies can be easily applied in many-sorted contexts. This requirement amounts to a simple relaxation of the typing for argument strategies of the strategy combinators. Basically, we want to state that the type of a compound strategy like s1; s2 and s1 + s2 is dictated by a many-sorted argument (if any). As for congruence strategies, we simply

*Well-typedness*

*` s1 : !*

*^ ` s2 :*

*^ !*

*` s : 0 ! 0*

*^ ` t :*

*` s :*

*1*

*^ 6= 0*

*^ ` hs it ; t0*

*2*

*[lplus+:2]*

*` s1 & s2 :*

*` hsit ; r*

*[lplus]*

*0*

*` hs1 & s it ; t*

*2*

*Semantics Positive rules*

*` s1 : !*

*Negative rules*

*` s1 : !*

*^ ` t :*

*^ ` hs1it ; "*

*` hs1 & s2it ; "*

*[lplus :1]*

*^ ` t :*

*^ ` hs it ; t0*

*1*

*[lplus+:1]*

*0 0*

*` s1 : !*

*0*

*` hs1 & s it ; t*

*2*

*^ ` t :*

*^ 6= 0*

*^ ` hs2it ; "*

*` hs1 & s2it ; "*

*[lplus :2]*

*Figure 7. Turning speci c strategies into generic ones*

*employ to relate formal and actual parameter types. There are no fur-* ther non-generic contexts for the given combinator suite. A corresponding re nement of our type system is de ned in Figure [8.](#_bookmark20)

*As for ; , we relax the de nition of composable types to cover composition* of a speci c and a generic type in both possible orders (cf. rules [comp:3] and [comp:4]). As for + (and hence for + as well), we do not insist on equal argument types anymore, but we assume that we can determine the greatest lower bound for types w.r.t. (cf. rule [choice]). Finally, we relax the argument types for congruence strategies via the relation. The re nement in a sense, automates the type specialisation for generic strategies. This is not considered as problem (as opposed to hidden ways for a many-sorted strategy to become generic) since an accidentally many-sorted strategy would be easily realised by the programmer when he or she attempts to apply the strategy in a generic context, that is, the type system will catch such accidents.

*4.3 Properties of the calculus*

*Our ultimate typed strategy calculus is obtained by starting from many-sorted* strategies (cf. Figure [5),](#_bookmark16) and updating it with the & operator (cf. Figure [7](#_bookmark19))

*Composable types Well-typedness*

*1 Æ 2 ;*

*` s :*

*`*

*! Æ TP ; !*

*`*

*TP Æ ! ; !*

*[comp:3]*

*[comp:4]*

*` s1 : 1*

*^ ` s2 : 2*

*^ 1 u 2 ;*

*` s1 + s2 :*

*f : 1 n ! 2*

*^ ` s1 : 1 ^ 1 ! 1 1*

*^*

*^ ` sn : n ^ n ! n n*

*` f (s1;:::; sn ) : !*

*[choice]*

*[congr]*

*Figure 8. Application of generic strategies in a potentially speci c context*

*and the relaxations from above (cf. Figure* [*8*](#_bookmark20)*). We call this calculus S0 (to*

*TP*

*point out its close relation to system S). The following theorem summarises*

*desirable properties of S0 .*

*TP*

*Theorem 4.3 The type system of S0*

*TP*

*(i) Strategies satisfy unicity of typing.*

*obeys the following:*

*(ii) Strategy application satis es unicity of typing.*

*(iii) The semantics for strategy application satis es subject reduction.*

*Proof*

*1. By induction on the strategy in the well-typedness judgement: Base cases:* The type of a rewrite rule (cf. rule [rule]) is uniquely de ned by the involved terms. Unicity of typing holds, of course, for the type judgement for terms. The types for the constant strategies Id and Fail are uniquely de ned (cf. rules [id] and [fail]). Induction step: The type of all argument strategies of all combinators are unique by the induction hypothesis. As for the binary operator ; , the result type is de ned as the type composed from the argument types. As for + , and + , the result type is de ned as the greatest lower bound of the argument types. The corresponding judgements Æ ; and

*u ; obviously encode functions. Hence, unicity of typing holds. The*

*type of a congruence strategy (cf. rule [congr]) is dictated by the well-formed* context which is used to lookup the sort of the function symbol at hand. The types of 2(s) and 3(s) are uniquely de ned as TP. The type of s1 & s2 is the type of s2.

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

*3. (Sketch) By induction on the strategy in the semantics judgement. Note* that we only deal with type-preserving strategies which simpli es matters be- cause we basically have to show that strategy application preserves term types as well. Base cases: The treatment of rewrite rules is standard. We can ignore Fail since we are only interested in proper term reducts. Subject reduction holds for Id since the resulting term coincides with the input term, and hence, type-preservation holds. Induction step: As for s1 + s2 and s1 + s2, reduc- tion simply resorts in all cases directly to one of the arguments. This is also enabled by the typing rules. Hence, the induction hypothesis is applicable for the arguments, and subject reduction holds; similarly for s1; s2. Subject reduction holds for congruence strategies and for the generic traversal combi- nators because the outermost function symbol is preserved and the types of all children are preserved by the induction hypothesis. The interesting case is s1 & s2. Rule [lplus+:2] (where we resort to the generic default s2 dictating the type of the compound strategy) can be covered using the same arguments used for s1 + s2. As for rule [lplus+1], we resort to the many-sorted s1 while the compound strategy can be applied to terms of any sort. Still, subject reduction holds because reduction according to rule [lplus+:1] is guarded by the typing premises to ensure that s1 is applicable. *2*

*4.4 Beyond TP*

*A note on generality is maybe in place. The presented type system (especially* Figure [7](#_bookmark19) and Figure [8)](#_bookmark20) is really geared towards generic type-preserving stra- tegies, and we assume that we have only two levels: many-sorted and generic strategies. Type-changing strategies (especially rewrite rules) are also sensi- ble since strategies can control that type changes are performed consistently. This is di erent in conservative rewriting where type-changing rewrite rules are incompatible with the idea of a xed strategy (like innermost). Especially, if we talk about generic strategies, one important subclass of type-changing strategies follows the scheme of type-uni cation [[12],](#_bookmark34) that is, the result type of the generic strategy is of a xed type (such as Boolean values or lists of naturals for the problems (III) and (IV) in the introduction) regardless of the type of the input term. In addition to many-sorted and generic strategies, one might also consider strategies with a nite set of term types covered by them. Such strategies could be called overloaded strategies. In this context, we might think of a more general form of s1 & s2 where the types of s1 and s2 are solely constrained by *One can also think of a symmetric combina-*

*tor & to perform a kind of disjoint union. We refer to* [*[10]*](#_bookmark32) *for a thorough*

*treatment of all the aforementioned classes of strategies.*

# *5 Implementation*

*The calculus S0 (and generalisations of it) can be implemented without major*

*TP*

*problems. We have done simple experiments based on Prolog which allowed*

*us to execute the judgements for typing and reduction almost as is.*

*There is one concern which needs to be addressed in order to obtain a* practical implementation, namely the separation of typing and reduction. The reduction rules for hs1 & s2it involve premises to determine the type of the term t, and the type of the strategy s1 (cf. Figure [7](#_bookmark19)). Conceptually, this is

*ne because we point out in the most direct way that & is about type-*

*dependent choice. Still this type-dependent reduction might be regarded as* a debatable paradigm shift, and, in particular, as an obstacle for eÆcient

*implementation of S0 . Fortunately, there is a simple way to eliminate the*

*TP*

*typing premises. The elimination is considered in this section in some detail.*

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

*5.1 Static elaboration*

*To eliminate the typing premise determining the type of the many-sorted* strategy s1 in s1 & s2, the following approach is appropriate. We statically perform an elaboration step which follows the very scheme of the type jud- gement for strategies, but transforms strategies. We want to turn strategy expressions of the form s1 & s2 into s1 : & s2 where we propagate the strategy type of s1 explicitly as type annotation. During strategy applica- tion, the annotation can be used to organise the choice. The updated rules for strategy application will be shown in a second. The relevant elaboration rule takes the following form:

*` s1 : 1*

*` s1 & s2 ; s1 : 1 & s2*

*[lplus]*

*5.2 Tagged terms*

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

*` hs it : ; t0 :*

*1*

*` hs : ! & s it : ; t0 :*

*1 2*

*[lplus+:1]*

*6=*

*0*

*^ ` hs it : ; t0 :*

*2*

*` hs : 0 ! 0 & s it : ; t0 :*

*1 2*

*[lplus+:2]*

*As we can see, the static typing context is not needed anymore. Instead* the reduction of hs1 : & s2it : relies on the annotations and . To be precise, the context is de nitely not needed for the semantics of & anymore, but the combinators 2( ) and 3( ) deserve an additional comment. As these combinators descend into terms, the types of the subterms of a term also need to be known. Some options to accomplish this knowledge are the following:

*(i) We assume that all subterms are tagged by their types at any level of* nesting.

*(ii) We can determine the type of any (well-typed) term via its outermost* function symbol. The declarations of function symbols (as being part of the typing context parameter ) are suÆcient for that purpose.

*(iii) We use specialised (signature-aware) variants of 2( ) and 3( ), that is, we* had to instantiate the scheme for 2( ) and 3( ) for all function symbols.

*All these formulations lead|more or less directly|to an eÆcient implemen-* tation. Option (i) has an impact on the representation of terms. Option

*(ii) relies on an extra lookup per application of & . Option (iii) requires* program generation.

*5.3 Integration into rewriting environments*

*The calculus S0 ts very well into the setting of a many-sorted strategic*

*TP*

*rewriting framework as ELAN. In ELAN, there is a module for many-sorted* strategy combinators parameterised by a sort. One needs to instantiate this module for each relevant sort in the given signature. Consequently, the strat- egy 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 speci cally de-

*ned on it, namely Id, Fail, 2( ), and 3( ). The sort and the symbols can*

*be de ned in a module dedicated to TP. Then, we need to overload & for all sorts in the signature at hand in the same way as the ordinary many-* sorted strategy combinators. Each application of the combinator & in a compound strategy refers to a speci c sort, and hence static elaboration is not needed to determine the type of the many-sorted strategy in a type- dependent choice. The rewrite rules for 2( ) and 3( ) could be generated by a pre-processor in similarity to the dynamic typing and implosion + explosion approach in [[4].](#_bookmark26) One can also leave it to the rewrite engine to implement 2( ) and 3( ). As for type-dependent choice, the rewrite engine is in fact the more obvious choice. Here we assume that the rewrite engine has access to the type of the given term. To summarise, the described simple implementation is ena- bled 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 overloaded signatures and local ambiguities). A* simple implementation is also conceivable for other frameworks for rewriting or algebraic speci cation.

*6 Concluding remarks Polymorphism*

*Let us consider the type scheme underlying TP:*

*TP 8 : !*

*In the scheme, we point out that is a universally quanti ed type variable. It* is easy to see that the scheme is appropriate. Generic type-preserving traver- sals process terms of any sort (i.e., ), and they return terms of the same sort (i.e., ). If we read the type scheme in the sense of parametric polymorphism, we can only inhabit it in a trivial way. The scheme can only be inhabited by the identity function. Hence, the kind of polymorphism underlying gene- ric traversals goes beyond parametric polymorphism. Parametricity [[22,15,13](#_bookmark35)] does not holds since generic traversals usually employ many-sorted ingredients (say rewrite rules) to deal with some distinguished sorts in a speci c manner. This form of genericity implies that the reduction semantics involves type dependencies although the type of strategies and strategy applications is sta- tically known. It is not clear how to inhabit somewhat arbitrary type schemes. This is also the reason that we do not favour type schemes to represent types of generic strategies in the rst place but we rather employ the distinguished constant TP.

*Related work*

*In the present article, we developed a type system for term rewriting stra-* tegies. The contribution of the article is that generic traversals are covered, namely generic type-preserving ones. We have designed another model for typed strategies in the context of functional programming [[11].](#_bookmark33) The latter approach originated in turn from our research on (dynamically updatable) generalised monadic folds for systems of datatypes [[12](#_bookmark34)].

*There is no previous work on statically typed generic strategies in the* narrow context of rewriting. In [[4],](#_bookmark26) dynamic types [[1]](#_bookmark23) are employed to cope with some generic (traversal) strategies in ELAN. A universal datatype any is used to represent terms of \any" sort. For that purpose, a parameterised module any[X] is o ered which can be imported for any sort which is subject to generic programming via the any datatype. The module o ers an injec- tion and a projection to mediate between any and the terms of sort X. As for generic traversals, there are explode and implode functions to destruct and construct terms (say to access the children of a term). The actual implementa- tion employs a (transparent) pre-processing approach to obtain a many-sorted instantiation of the interface of any[X] according to X, and program schemes

*for explode and implode. Speci cations relying on any are type-safe. However,* if during rewriting the manipulated terms of sort any do not represent terms of the \intended sorts", at some point projection and term implosion is going

*to fail. This problem is irrelevant for S0 since types are statically enforced,*

*TP*

*and there is no universal (and hence imprecise) sort like any.*

*The presented concepts were inspired by polytypic programming* [*[8,16*](#_bookmark38)*]. A* polytypic function is de ned by induction on its argument type (with cases

*for sums, products, and others). Generic traversals in S0 are performed in a*

*TP*

*somewhat similar manner. Generic traversals are de ned in terms of 2( ) and* 3( ) (corresponding to the polytypic cases for sums and products), usually by recursive strategy de nitions (roughly corresponding to induction). While polytypic programming is placed in the context of higher-order functional pro- gramming, our approach contributes to the eld of strategic, (not necessarily)

*rst-order rewriting. An idea which is central to our approach is that we*

*want to have simple but exible means to mix genericity and speci city in the* context of many-sorted signatures (e.g., language syntaxes), while the bulk of polytypic programming focuses on statically de ned, polytypic values for all (parameterised) datatypes.

*Perspective*

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

*Another topic for future work is the integration of our results into existing* rewriting calculi. The -calculus [[5]](#_bookmark27) provides an ambitious rewriting calculus. Part of the -cube is typed (but not generic traversals expressiveness) [[6](#_bookmark28)]. One of the challenging properties of the -calculus is that rewrite rules are higher-

*order. The developed typed calculus S0 could be easily rephrased to cover*

*TP*

*some rst-order fragment of the calculus (however with generic traversals!).* The question is how generic traversals can be enabled for richer fragments in the -cube, e.g., fragments o ering higher-orderness, full polymorphism, and dependent types.

# *References*

*[1] M. Abadi, L. Cardelli, B. Pierce, and D. R emy. 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.*

*[2] P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by* *Rewriting. In Meseguer* [*[17*](#_bookmark39)*].*

*[3] P. Borovansky , C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, Proceedings of the International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science, Pont-a-Mousson, France,* *Sept. 1998. Elsevier Science.*

*[4] P. Borovansky , C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. International Journal of* *Foundations of Computer Science, 2001.*

*[5] H. Cirstea and C. Kirchner. Introduction to the rewriting calculus. Rapport* *de recherche 3818, INRIA, Dec. 1999.*

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

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

*[8] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on* *Principles of Programming Languages, pages 470{482. ACM Press, 1997.*

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

*[10] R. Lammel. Typed Generic Traversals in S0 . Technical Report SEN-R0122,*

*CWI, Aug. 2001.*

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

*[12] R. Lammel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring* [*[9*](#_bookmark31)*], pages 46{59. available at* [*http://www.cwi.nl/~ralf/.*](http://www.cwi.nl/~ralf/)

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

*[14] S. E. M. Clavel, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer* [*[17*](#_bookmark39)*].*

*[15] Q. Ma and J. 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.*

*[16] 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.*

*[17] J. Meseguer, editor. Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Paci c Grove, CA, USA),* *volume 4 of Electronic Notes in Theoretical Computer Science, Sept. 1996.*

*[18] L. C. Paulson. A higher-order implementation of rewriting. Science of* *Computer Programming, 3(2):119{149 (or 119{150??), Aug. 1983.*

*[19] E. Visser. Language independent traversals for program transformation. In* *Jeuring* [*[9],*](#_bookmark31) *pages 86{104.*

*[20] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In International Conference on Functional Programming* *(ICFP'98), Baltimore, Maryland. ACM SIGPLAN, pages 13{26, Sept. 1998.*

*[21] E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. Electronic Notes in Theoretical Computer Science, 15, September 1998. In C. Kirchner and H. Kirchner, editors, Proceedings of the Second International Workshop on* *Rewriting Logic and its Applications (WRLA'98), Pont-a-Mousson, France.*

*[22] 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.*