URL: http://www.elsevier.nl/locate/entcs/volume15.html 20 pages

A Core Language for Rewriting

Eelco Visser Zine-el-Abidine Benaissa

Pacific Software Research Center, Department of Computer Science and Engineering, Oregon Graduate Institute, P.O. Box 91000, Portland, Oregon 97291-1000, USA. visser@acm.org, benaissa@cse.ogi.edu

Abstract

System S is a calculus providing the basic abstractions of term rewriting: matching and building terms, term traversal, combining computations and handling failure. The calculus forms a core language for implementation of a wide variety of rewriting languages, or more generally, languages for specifying tree transformations. In this paper we show how a conventional rewriting language based on conditional term rewriting can be implemented straightforwardly in System S. Subsequently we show how this implementation can be extended with features such as matching conditions, negative conditions, default rules, non-strictness annotations and alternative evaluation strategies.

1 Introduction

Term rewriting is a theoretically well-defined paradigm that consists of reducing a term to normal form with respect to a set of rewrite rules [12,5,1]. However, in practical instantiations of this paradigm a wide variety of features are added to this basic paradigm. This has resulted in the design and implementation of a number of rewriting languages and systems (e.g., OBJ [7,10], ASF+SDF [6], Larch [9], ELAN [2], Maude [4]). Although sharing the same principles, each adds its own special features that are useful in the particular application setting of the system. These languages are usually implemented in an ad-hoc fashion, making extension with new features difficult.

In this paper we present System S and argue that it is a core language for implementing a wide variety of rewriting languages. System S is a hierarchy of calculi representing the basic ingredients of rewriting engines. System S_0 defines operators for non-deterministic sequential programming. System S_1

[1] This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

©1998 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.

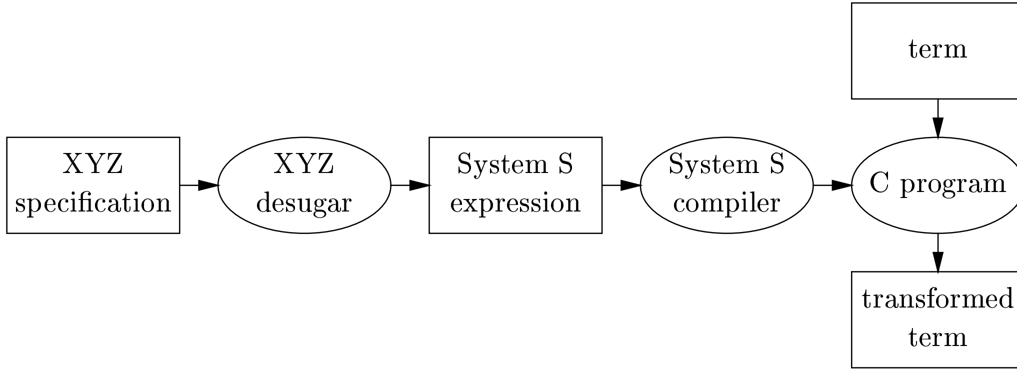

Fig. 1. Architecture of implementation of a rewriting language XYZ using System S.

adds operators for term-traversal. System S_2 introduces variable binding environments and operators for matching and building terms. This system is adequate to express the rule selection strategy, the evaluation strategy and the rule features of rewriting languages. Implementation of a language by encoding into System S makes it easy to experiment with alternative evaluation strategies or to add new features to rules.

Figure 1 shows the typical architecture of an implementation of a rewriting language using System S. A desugaring phase encodes specifications into System S expressions. The generic System S compiler maps this to a C program that transforms terms to terms according to the original specification. In [14] this approach is used to develop a language for the specification of high-level rewriting rules and strategies targeted at program optimization. In this paper we illustrate the approach by presenting the encoding of a conventional conditional term rewriting language with several non-standard features.

In the next section we define System S. In Section 3 we implement a basic scheme for conditional term rewrite systems with join conditions and an innermost evaluation strategy. In Section 4 we extend this scheme with several extensions including matching conditions, negative conditions, default rules and alternative evaluation strategies. We discuss related work in Section 5.

2 System S

In this section we define System S, a hierarchy of operators that forms a core language for rewriting. Expressions in the language are non-deterministic sequential programs, also called rewriting strategies, that define transformations on terms. The language presented in this section is based on our earlier work on rewriting strategies [13,14].

$$\frac{}{t \xrightarrow{\epsilon} t} \qquad \frac{}{t \xrightarrow{\delta} \uparrow}$$

$$\frac{t \xrightarrow{s} t'}{t \xrightarrow{\mathsf{test}\ s} t} \qquad \frac{t \xrightarrow{s} \uparrow}{t \xrightarrow{\mathsf{test}\ s} \uparrow}$$

$$\frac{t \xrightarrow{s} \uparrow}{t \xrightarrow{\neg s} t} \qquad \frac{t \xrightarrow{s} t'}{t \xrightarrow{\neg s} \uparrow}$$

$$\frac{t \xrightarrow{s_1} t' \quad t' \xrightarrow{s_2} t''}{t \xrightarrow{s_1;s_2} t''} \qquad \frac{t \xrightarrow{s_1} \uparrow}{t \xrightarrow{s_1;s_2} \uparrow} \qquad \frac{t \xrightarrow{s_1} t' \quad t' \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1;s_2} \uparrow}$$

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1+s_2} t'} \qquad \frac{t \xrightarrow{s_2} t'}{t \xrightarrow{s_1+s_2} t'} \qquad \frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1+s_2} \uparrow}$$

$$\frac{t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 \leftarrow\!\!\!+ s_2} t'} \qquad \frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} t'}{t \xrightarrow{s_1 \leftarrow\!\!\!+ s_2} t'} \qquad \frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1 \leftarrow\!\!\!+ s_2} \uparrow}$$

$$\frac{t \xrightarrow{s[x:=\mu x(s)]} t'}{t \xrightarrow{\mu x(s)} t'} \qquad \frac{t \xrightarrow{s[x:=\mu x(s)]} \uparrow}{t \xrightarrow{\mu x(s)} \uparrow}$$

Table 1
Operational semantics for System S_0.

2.1 Non-deterministic Sequential Programs

System S_0 defines non-deterministic sequential state transforming programs with failure. A program s specifies a relation $\xrightarrow{s} \subseteq \mathcal{S} \times (\mathcal{S} \cup \{\uparrow\})$, between states and states extended with the special element $\uparrow$ that represents failure. We write $t \xrightarrow{s} t'$ if $t, t' \in \mathcal{S}$ and $(t, t') \in \xrightarrow{s}$; we say that s succeeds for t. We write $t \xrightarrow{s} \uparrow$ if $t \in \mathcal{S}$ and $(t, \uparrow) \in \xrightarrow{s}$; we say that s fails for t. If there is no pair $(t, x) \in \xrightarrow{s}$ we say that s is undefined on t. Undefinedness corresponds to non-terminating computations.

Atomic programs are elements from a set $\mathcal{A}$ of state transforming actions such that $a \in \mathcal{A}$ implies that for each state t, either $t \xrightarrow{s} t'$ for exactly one t' or $t \xrightarrow{s} \uparrow$. Programs can be composed from atomic programs with the following operators: success (ϵ), failure (δ), negation ($\neg$), test, sequential composition (;), non-deterministic choice ($+$), deterministic or left choice ($\leftarrow\!\!\!+$) and recursion ($\mu_(_)$). The operational semantics of these operators are defined in Table 1.

We briefly describe the operators. The *identity* program ϵ always succeeds without affecting the state. The *failure* program δ fails on all states. The program $\mathsf{test}(s)$ fails if s fails and succeeds if s succeeds, but reverts to the original state, i.e., only the success/failure aspect of s is captured. The *negation* $\neg s$ of s is similar to $\mathsf{test}(s)$, but succeeds (with the identity transformation) when s fails and fails if s succeeds. The *sequential composition* $s_1 ; s_2$ of s_1 and s_2 first applies s_1 and, if that succeeds, applies s_2 to the result of s_1. It fails if either s_1 or s_2 fails. The *non-deterministic choice* $s_1 + s_2$ chooses one of s_1

and s_2, provided that the chosen alternative succeeds. If both are succesful, either one can be chosen. The *deterministic* or *left choice* $s_1 \mathrel{<\!\!+} s_2$ chooses one of s_1 and s_2, but tries s_1 first, ie., it will consider s_2 only if s_1 fails. The *recursion* operator $\mu x(s)$ is a least fixed point operator that defines recursive programs. An invocation of the variable x in the body s of the program $\mu x(s)$ corresponds to a recursive call to the program $\mu x(s)$.

2.1.1 Deterministic Programs

A program s is *deterministic* if for all states t, $t \xrightarrow{s} x$ and $t \xrightarrow{s} y$ implies $x = y$, for $x, y \in \mathcal{S} \times \{\uparrow\}$. In general, programs are not deterministic. For example, consider the program $(s_1 + s_2); s_3$. If $t \xrightarrow{s_1} t'$ and $t \xrightarrow{s_2} t''$, but $t' \xrightarrow{s_3} \uparrow$ and $t'' \xrightarrow{s_3} t'''$, then both $t \xrightarrow{(s_1+s_2);s_3} t'''$ and $t \xrightarrow{(s_1+s_2);s_3} \uparrow$. In other words, depending on the order in which s_1 and s_2 are tried this program will succeed or fail. This does not hold for left choice. Once the left branch has succeeded the right branch can never be chosen. Therefore, programs without choice are deterministic.

Lemma 2.1 *Programs without choice are deterministic.*

Proof. We prove by induction on derivations Φ for programs without choice that $\Phi \Vdash t \xrightarrow{s} x$ and $t \xrightarrow{s} y$ implies $x = y$ for $x, y \in \mathcal{S} \times \{\uparrow\}$. (Here $\Phi \Vdash t \xrightarrow{s} x$ asserts that Φ is a derivation for $t \xrightarrow{s} x$.) Let Φ be a derivation for a program without choice. Assume that for all subderivations of Φ the property holds. Suppose $\Phi \Vdash t \xrightarrow{s} x$ and $t \xrightarrow{s} y$, then there exists some Φ' such that $\Phi' \Vdash t \xrightarrow{s} y$. Now we show for all possible last steps of Φ that $x = y$. (Note that by assumption this does not include choice.)

- Actions in $\mathcal{A}$ and ϵ and δ admit only one derivation, therefore $\Phi = \Phi'$ and hence $x = y$.
- Sequential composition ($s \equiv s_1 ; s_2$) Any derivations for $s_1 ; s_2$ must have the form

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s_1} x' \quad \cdots}{t \xrightarrow{s_1;s_2} x} \qquad \text{and} \qquad \Phi' = \frac{\Vdash t \xrightarrow{s_1} y' \quad \cdots}{t \xrightarrow{s_1;s_2} y}$$

By induction (for Φ_1) we have $x' = y'$. Then we consider the following cases (a) $x' = \uparrow = x$, then $y' = x' = x = \uparrow = y$; (b) $x' = t'$, then

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s_1} t' \quad \Phi_2 \Vdash t' \xrightarrow{s_2} x}{t \xrightarrow{s_1;s_2} x} \qquad \text{and} \qquad \Phi' = \frac{\Vdash t \xrightarrow{s_1} t' \quad \Vdash t' \xrightarrow{s_2} y}{t \xrightarrow{s_1;s_2} y}$$

By induction (for Φ_2), we have $x = y$.

- Deterministic choice ($s \equiv s_1 \mathrel{<\!\!+} s_2$) Any derivations for $s_1 \mathrel{<\!\!+} s_2$ must have the form

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s_1} x' \quad \cdots}{t \xrightarrow{s_1 \mathrel{<\!\!+} s_2} x} \qquad \text{and} \qquad \Phi' = \frac{\Vdash t \xrightarrow{s_1} y' \quad \cdots}{t \xrightarrow{s_1 \mathrel{<\!\!+} s_2} y}$$

By induction (for Φ_1), we have $x' = y'$. Then we consider the following cases (a) $x = x' = t'$ and hence $y = y' = t' = x$; (b) $x' = \uparrow$, then

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s_1} \uparrow \quad \Phi_2 \Vdash t \xrightarrow{s_2} x}{t \xrightarrow{s_1 \mathbin{<\!\!+} s_2} x} \qquad \text{and} \qquad \Phi' = \frac{\Vdash t \xrightarrow{s_1} \uparrow \quad \Vdash t \xrightarrow{s_2} y}{t \xrightarrow{s_1 \mathbin{<\!\!+} s_2} y}$$

By induction (for Φ_2), we have $x = y$.

- Recursion ($s \equiv \mu x(s')$) We have

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s[x:=\mu x(s)]} x}{t \xrightarrow{\mu x(s)} x} \qquad \text{and} \qquad \Phi' = \frac{\Vdash t \xrightarrow{s[x:=\mu x(s)]} y}{t \xrightarrow{\mu x(s)} y}$$

by induction, we have $x = y$.

- The cases for negation and test are similar. □

Although programs without choice are deterministic, this does not entail that a program with choices is not deterministic. Two programs s_1 and s_2 are *orthogonal* (with respect to each other) if for all states t, $t \xrightarrow{s_1} t'$ implies $t \xrightarrow{s_2} \uparrow$, and $t \xrightarrow{s_2} t'$ implies $t \xrightarrow{s_1} \uparrow$. A sum $s_1 + s_2$ is orthogonal if s_1 and s_2 are orthogonal. A single program s is orthogonal if all sums in all its finite unfoldings are orthogonal.

Lemma 2.2 *Orthogonal programs are deterministic.*

Proof. This property is proved by extending the proof of the previous lemma by adding the following case:

- Choice ($s \equiv s_1 + s_2$) There are three possible cases for Φ. We handle one, the others are similar. Suppose Φ has the form

$$\Phi = \frac{\Phi_1 \Vdash t \xrightarrow{s_1} t'}{t \xrightarrow{s_1 + s_2} t'}$$

There are the following three cases to consider for Φ':

$$\Phi' = \frac{t \xrightarrow{s_1} t''}{t \xrightarrow{s_1 + s_2} t''} \qquad \Phi' = \frac{t \xrightarrow{s_2} t''}{t \xrightarrow{s_1 + s_2} t''} \qquad \Phi' = \frac{t \xrightarrow{s_1} \uparrow \quad t \xrightarrow{s_2} \uparrow}{t \xrightarrow{s_1 + s_2} \uparrow}$$

In the first case, by induction $t' = t''$. In the second case, by orthogonality, we have $t \xrightarrow{s_2} \uparrow$, and thus, $t'' = \uparrow$, which is a contradiction, i.e., this case cannot occur. Finally, in the third case, by induction $t' = \uparrow$, which again is a contradiction. □

2.1.2 Definitions

Common program patterns can be named by means of definitions. A definition $\varphi(x_1, \ldots, x_n) = s$ introduces a new n-ary program operator φ. An application $\varphi(s_1, \ldots, s_n)$ of φ to n programs denotes the instantiation $s[x_1 := s_1 \ldots x_n := s_n]$ of the body s of the definition. Program definitions are not recursive and

not higher-order, i.e., it is not possible to give a program operator as argument to a program operator. Example definitions are:

$$\begin{aligned}\texttt{try}(s) &= s \leftarrow\!\!\!+\, \epsilon\\ \texttt{repeat}(s) &= \mu x(\texttt{try}(s\,;x))\end{aligned}$$

2.2 Terms

Up to this point we have considered abstract states and abstract state transforming actions. In term rewriting and program transformation we are interested in transforming terms or abstract syntax trees. From now on we will use terms as states. Atomic actions are transformations on terms that apply to the root of a term and that may fail for some terms.

A first-order term is either a variable x or an application $f(t_1,\ldots,t_n)$ of a constructor to one or more terms. We denote the set of all variables by X, the set of terms with variables by $T(X)$ and the set of ground terms (terms without variables) by T. Terms can be typed by means of signatures. For simplicity of presentation, we will consider only untyped terms, i.e., a signature Σ is a function mapping operators to their arities. Constants are constructors with zero-arity.

2.3 Term Traversal

The operators we introduced above form programs that apply transformation rules at the root of a term. This is not adequate for describing general transformations. In order to apply transformations to subterms of a term we need operators to traverse the term structure. For this purpose we introduce the extension System S_1 with the operators: $i(s)$ (path), $f(s_1,\ldots,s_n)$ (congruence), $\Diamond(s)$ (one), $\Box(s)$ (all) and $\Box\!\!\!\!\!\Diamond(s)$ (some). The operational semantics of these operators are defined in Table 2.

The program $i(s)$ applies program s to the i-th child. This fails if applied to a constructor with arity less than i. The *congruence* operator $f(s_1,\ldots,s_n)$ is a program that specifies a program to be applied to each child of a term with constructor f. It fails if either of those applications fails. It also fails if applied to a term with another outermost function symbol than f. An example of the use of congruence operators is the program $\texttt{map}(s)$ that applies a program s to each element of a list constructed with `Cons` and `Nil`. It is defined as:

$$\texttt{map}(s) = \mu x(\texttt{Nil} + \texttt{Cons}(s,x))$$

The path and congruence operators are useful for constructing programs for a specific data structure. To construct more general programs that can abstract from a concrete representation we introduce the operators $\Box$, $\Diamond$ and $\Box\!\!\!\!\!\Diamond$.

The program $\Box(s)$ applies s to each child of the root. This only succeeds if s succeeds for each direct subterm. In case of constants, i.e., constructors

$$\frac{t_i \xrightarrow{s} t_i'}{f(\ldots, t_i, \ldots) \xrightarrow{i(s)} f(\ldots, t_i', \ldots)} \qquad \frac{t_i \xrightarrow{s} \uparrow}{f(\ldots, t_i, \ldots) \xrightarrow{i(s)} \uparrow}$$

$$\frac{}{f(t_1, \ldots, t_n) \xrightarrow{i(s)} \uparrow} \text{ if } i > n$$

$$\frac{t_1 \xrightarrow{s_1} t_1' \quad \ldots \quad t_n \xrightarrow{s_n} t_n'}{f(t_1, \ldots, t_n) \xrightarrow{f(s_1,\ldots,s_n)} f(t_1', \ldots, t_n')} \qquad \frac{t_i \xrightarrow{s_i} \uparrow}{f(t_1, \ldots, t_n) \xrightarrow{f(s_1,\ldots,s_n)} \uparrow}$$

$$g(t_1, \ldots, t_m) \xrightarrow{f(s_1,\ldots,s_n)} \uparrow \quad \text{if } f \neq g$$

$$\frac{t_i \xrightarrow{s} t_i'}{f(\ldots, t_i, \ldots) \xrightarrow{\Diamond(s)} f(\ldots, t_i', \ldots)} \qquad \frac{t_1 \xrightarrow{s} \uparrow \quad \ldots \quad t_n \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) \xrightarrow{\Diamond(s)} \uparrow}$$

$$\frac{t_1 \xrightarrow{s} t_1' \quad \ldots \quad t_n \xrightarrow{s} t_n'}{f(t_1, \ldots, t_n) \xrightarrow{\Box(s)} f(t_1', \ldots, t_n')} \qquad \frac{t_i \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) \xrightarrow{\Box(s)} \uparrow}$$

$$\frac{\exists j \forall i : i, j \in \{1..n\} \wedge P(t_i) = \begin{cases} t_i' & \text{if } t_i \xrightarrow{s} t_i' \\ t_i & \text{if } t_i \xrightarrow{s} \uparrow \wedge i \neq j \end{cases}}{f(t_1, \ldots, t_n) \xrightarrow{\Box\kern-0.72em\Diamond(s)} f(P(t_1), \ldots, P(t_n))} \qquad \frac{t_1 \xrightarrow{s} \uparrow \quad \ldots \quad t_n \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) \xrightarrow{\Box\kern-0.72em\Diamond(s)} \uparrow}$$

Table 2
Operational semantics for System S_1.These rules are schemata that define a rule for each $f \in \Sigma$. In all rules $n \equiv \Sigma(f)$, $m \equiv \Sigma(g)$ and $1 \leq i \leq n$, unless otherwise stated.

without arguments, the program always succeeds, since there are no children. This allows us to define very general traversal programs. For example, the following programs apply a program s to each node in a term, in preorder (topdown), postorder (bottomup) and a combination of pre- and postorder (downup):

$$\begin{aligned} \texttt{topdown}(s) &= \mu x(s \,;\, \Box(x)) \\ \texttt{bottomup}(s) &= \mu x(\Box(x) \,;\, s) \\ \texttt{downup}(s) &= \mu x(s \,;\, \Box(x) \,;\, s) \end{aligned}$$

The program $\Diamond(s)$ applies s non-deterministically to one child. It fails if there is no child for which s succeeds. In particular, it fails for constants, since they have no child for which s can succeed. Example applications of this operator are the following traversal operators:

$$\begin{aligned} \texttt{oncetd}(s) &= \mu x(s \leftarrow\!\!\!+ \Diamond(x)) \\ \texttt{oncebu}(s) &= \mu x(\Diamond(x) \leftarrow\!\!\!+ s) \end{aligned}$$

These programs find a single subterm for which the application of s succeeds.

The program $\texttt{oncetd}(s)$ searches in a topdown fashion; it first attempts to apply s at the root of the term; if that fails it tries to find an s application recursively at one of the children. The program $\texttt{oncebu}(s)$ similarly searches in a bottomup fashion. Observe that these programs fail if no s application is found.

Finally, $\boxdiamond(s)$ is a hybrid of $\Box(s)$ and $\Diamond(s)$ that applies s to some children. It is like $\Diamond$ because it has to succeed for at least one child and it is like $\Box$ because it applies to all children. The difference from $\Box$ is that it does not have to succeed for all children. The analogue of `oncebu` with $\boxdiamond$ is the strategy `somebu`, defined as:

$$\texttt{somebu}(s) = \mu x(\boxdiamond(x) \mathrel{\leftarrow\!\!+} s)$$

Where `oncebu` finds a single subterm for which s succeeds, `somebu` finds as many subterms as possible to which s applies, but at least one. The analogous pre-order strategy is:

$$\texttt{sometd}(s) = \mu x(s \mathrel{\leftarrow\!\!+} \boxdiamond(x))$$

2.4 Rewrite Rules

One possible instantiation System S_1^r of System S_1 is the use of labeled unconditional rewrite rules as basic programs. Rewrite rules have the form $\ell : l \to r$, where ℓ is a label, l and r are first-order terms. A rewrite rule specifies a single step transformation of a term as defined by the operational semantics in Table 3.

Rule	Condition
$\dfrac{}{t \xrightarrow{\ell:l\to r} t'}$	if $\exists\sigma : \sigma(l) = t \wedge \sigma(r) = t'$
$\dfrac{}{t \xrightarrow{\ell:l\to r} \uparrow}$	if $\neg\exists\sigma : \sigma(l) = t$

Table 3
Operational semantics for System S_1^r.

The first rule defines that a rule ℓ transforms a term t into a term t' if there exists a substitution σ mapping variables to terms such that t is a σ-instance of the left-hand side l and t' is a σ-instance of the right-hand side r. The second rule states that an attempt to transform a term t with rule ℓ fails, if there is no substitution σ such that t is a σ-instance of l. Recall that a rewrite rule (an action in S_1) applies at the *root* of a term.

A rewrite specification consisting of unconditional rules $\ell_1, \ldots, \ell_n$ can be implemented in System S_1^r by defining a program that reduces terms to normal form. In the next section we will explore this further.

$$\frac{}{t : \mathcal{E} \xrightarrow{\mathsf{match}(x)} t : \mathcal{E} \cup \{x \mapsto t\}} \text{ if } x \notin \mathsf{dom}(\mathcal{E})$$

$$\frac{}{t : \mathcal{E} \xrightarrow{\mathsf{match}(x)} t : \mathcal{E}} \text{ if } \mathcal{E}(x) = t \qquad \frac{}{t : \mathcal{E} \xrightarrow{\mathsf{match}(x)} \uparrow} \text{ if } \mathcal{E}(x) \neq t$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{\mathsf{match}(t'_1)} t_1 : \mathcal{E}_1 \quad \ldots \quad t_n : \mathcal{E}_{n-1} \xrightarrow{\mathsf{match}(t'_n)} t_n : \mathcal{E}_n}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\mathsf{match}(f(t'_1, \ldots, t'_n))} f(t_1, \ldots, t_n) : \mathcal{E}_n}$$

$$\frac{}{g(t_1, \ldots, t_m) : \mathcal{E} \xrightarrow{\mathsf{match}(f(t'_1, \ldots, t'_n))} \uparrow} \text{ if } f \neq g$$

$$\frac{}{t : \mathcal{E} \xrightarrow{\mathsf{build}(t')} \mathcal{E}(t') : \mathcal{E}} \text{ if } \mathrm{vars}(t') \subseteq \mathsf{dom}(\mathcal{E})$$

$$\frac{}{t : \mathcal{E} \xrightarrow{\mathsf{build}(t')} \uparrow} \text{ if } \mathrm{vars}(t') \not\subseteq \mathsf{dom}(\mathcal{E})$$

$$\frac{t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'}{t : \mathcal{E} \xrightarrow{\mathsf{where}\ s} t : \mathcal{E}'} \qquad \frac{t : \mathcal{E} \xrightarrow{s} \uparrow}{t : \mathcal{E} \xrightarrow{\mathsf{where}\ s} \uparrow}$$

$$\frac{t : \mathcal{E} \backslash \boldsymbol{x} \xrightarrow{s} t' : \mathcal{E}'}{t : \mathcal{E} \xrightarrow{\{\boldsymbol{x} : s\}} t' : (\mathcal{E}' \backslash \boldsymbol{x}) \cup (\mathcal{E} | \boldsymbol{x})} \qquad \frac{t : \mathcal{E} \backslash \boldsymbol{x} \xrightarrow{s} \uparrow}{t : \mathcal{E} \xrightarrow{\{\boldsymbol{x} : s\}} \uparrow}$$

Table 4

Operational semantics for System S_2—environment operators.

2.5 Matching and Building Terms

For many purposes we need more complex rules, for instance rules with conditions or contexts. Implementing each new feature in an ad-hoc fashion is possible, but leads to a non-uniform and complicated implementation. Therefore, we reexamine the domain of rewriting in order to achieve a uniform set of primitives that can be used to express a wide variety of rules.

A rewrite rule $\ell : l \to r$ first matches the term against the left-hand side l producing a binding of subterms to the variables in l. Subsequently it builds a new term by instantiating the right-hand side r with those variable bindings. By introducing the primitives match and build we can break down ℓ into a program $\mathsf{match}(l) ; \mathsf{build}(r)$. However, this requires that we carry the bindings obtained by match over the sequential composition to build. Bindings are represented by means of an environment, a mapping from variables to ground terms. We denote the instantiation of a term t by an environment $\mathcal{E}$ by $\mathcal{E}(t)$. We extend the reduction relation $\xrightarrow{s}$ from a relation between terms and reducts to a relation on pairs of terms and environments, i.e., a program s transforms a term t and an environment $\mathcal{E}$ into a transformed term t' and an extended environment $\mathcal{E}'$, denoted by $t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'$, or fails, denoted by $t : \mathcal{E} \xrightarrow{s} \uparrow$. The operational semantics of the environment operators are defined in Table 4.

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s_1} t'_1 : \mathcal{E}_1 \quad \ldots \quad t_n : \mathcal{E}_{n-1} \xrightarrow{s_n} t'_n : \mathcal{E}_n}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{f(s_1,\ldots,s_n)} f(t'_1, \ldots, t'_n) : \mathcal{E}_n}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s_1} t'_1 : \mathcal{E}_1 \quad \ldots \quad t_i : \mathcal{E}_{i-1} \xrightarrow{s_i} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{f(s_1,\ldots,s_n)} \uparrow}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s} t'_1 : \mathcal{E}_1 \quad \ldots \quad t_n : \mathcal{E}_{n-1} \xrightarrow{s} t'_n : \mathcal{E}_n}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\Box(s)} f(t'_1, \ldots, t'_n) : \mathcal{E}_n}$$

$$\frac{t_1 : \mathcal{E}_0 \xrightarrow{s} t'_1 : \mathcal{E}_1 \quad \ldots \quad t_i : \mathcal{E}_{i-1} \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\Box(s)} \uparrow}$$

$$\frac{\exists j \forall i : i, j \in \{1..n\} \wedge P(t_i) = \begin{cases} t'_i & \text{if } t_i : \mathcal{E}_{i-1} \xrightarrow{s} t'_i : \mathcal{E}_i \\ t_i & \text{if } t_i : \mathcal{E}_{i-1} \xrightarrow{s} \uparrow \wedge \\ & \quad \mathcal{E}_i = \mathcal{E}_{i-1} \wedge i \neq j \end{cases}}{f(t_1, \ldots, t_n) : \mathcal{E}_0 \xrightarrow{\boxtimes(s)} f(P(t_1), \ldots, P(t_n)) : \mathcal{E}_n}$$

$$\frac{t_1 : \mathcal{E} \xrightarrow{s} \uparrow \quad \ldots \quad t_n : \mathcal{E} \xrightarrow{s} \uparrow}{f(t_1, \ldots, t_n) : \mathcal{E} \xrightarrow{\boxtimes(s)} \uparrow}$$

Table 5
Operational semantics for System S_2 (modification of rules for traversal operators).

The change in the format of the operational semantics should be reflected in the semantics of the operators introduced earlier. In the remainder of the paper the rules in Tables 1 and 2 should be read as follows: a transition $t \xrightarrow{s} t'$ denotes a transition $t : \mathcal{E} \xrightarrow{s} t' : \mathcal{E}'$. The only exceptions are the rules for congruence, $\Box$ and $\boxtimes$. See Table 5 for their definitions in the extended semantics.

Once a variable is bound it cannot be rebound to a different term. To use a rule more than once we introduce variable scopes. A scope $\{\boldsymbol{x} : s\}$ locally undefines the variables $\boldsymbol{x}$ and then applies s. The notation $\mathcal{E} \setminus \boldsymbol{x}$ denotes $\mathcal{E}$ without bindings for variables in $\boldsymbol{x}$. $\mathcal{E}|\boldsymbol{x}$ denotes $\mathcal{E}$ restricted to $\boldsymbol{x}$. The operator where is similar to the test operator of Section 2 in that it tries a strategy and returns the original term if it succeeds. However, it keeps the transformation on the environment. This operator can be used to encode a local computation that binds the answer to a variable to be used outside it, without actually transforming the term.

Note that this definition supports matching with non-linear patterns. If a variable x occurs more than once in a pattern t, then match(t) succeeds only if all occurrences of x in t are bound to the same term. Moreover, if a variable x in t was already bound by a previous match, it should match to the exact same term that it was bound to before.

3 Conditional Term Rewrite Systems

In this section we present the encoding of conditional term rewrite systems in System S. We first define a basic scheme for an evaluation strategy for unconditional rules. Then we extend the basic scheme to include rules with join conditions.

3.1 Unconditional Rules

With the primitives for matching and building terms, an unconditional rule $\ell : l \rightarrow r$ translates to the definition

$$\bar{\ell} = \{\text{vars}(l, r) : \mathsf{match}(l) \mathbin{;} \mathsf{build}(r)\}$$

that first matches the subject term to the left-hand side l and if that succeeds replaces it with the instantiation of the right-hand side r. This expression is embedded in a scope that makes the variables of l and r local to the rule. Given a set of rewrite rules $\ell_1, \ldots, \ell_n$ we can now construct the corresponding programs $\bar{\ell}_1, \ldots, \bar{\ell}_n$.

3.2 Rule Selection and Evaluation Strategy

Next we need to combine these programs into an evaluation strategy that applies the rules to a term until it is in normal form. An evaluation strategy consists of two components: (1) a strategy for selecting a rule to apply and (2) a strategy for selecting a subterm (the redex) to apply a rule to.

In rewriting with unconditional rules the order of rules is not important, i.e., any rule can be selected non-deterministically from the set. This is expressed by the program $\bar{\ell}_1 + \cdots + \bar{\ell}_n$, which attempts to apply one of the rules ℓ_i. It fails if none of the rules apply.

The most general evaluation strategy repeatedly applies a rule to an arbitrary subterm. This strategy is formalized by the operator `reduce`, which is defined as:

$$\texttt{reduce}(s) = \texttt{repeat}(\mu x(s + \diamond(x)))$$

The inner expression either applies s at the root or recursively attempts to apply s at one of the children. This process is repeated until no more redices are found. An evaluation strategy for a set of rules $\ell_1, \ldots, \ell_n$ is then defined by $\texttt{reduce}(\bar{\ell}_1 + \cdots + \bar{\ell}_n)$.

Languages such as OBJ [7] and ASF+SDF [6] use an innermost reduction strategy for evaluation. Innermost reduction repeatedly reduces innermost redices; a redex is innermost if none of its proper subterms is a redex. This is formalized by the following definition:

$$\texttt{innermost}(s) = \texttt{repeat}(\texttt{oncebu}(s))$$

The program $\texttt{oncebu}(s)$ (see Section 2.3) searches for a redex starting at the leaves. This entails that it will only attempt to apply s to a term after all its proper subterms have been tried. This guarantees that the redex is indeed an innermost redex. Searching for an innermost redex is repeated until no more are found. The following program defines a strategy that reduces all innermost redices at once:

$$\texttt{innermost-par}(s) = \texttt{repeat}(\texttt{somebu}(s))$$

These definitions of innermost reduction are not very efficient because they repeatedly start searching for a redex at the root. A more efficient definition of innermost reduction is the following:

$$\texttt{innermost'}(s) = \mu x(\Box(x)\,;\,\texttt{try}(s\,;\,x))$$

This strategy first normalizes all subterms of a term and then tries to apply s. If that fails this means that the term is in normal form. Otherwise, the reduct resulting from applying s is further normalized by recursively applying the normalization strategy through the recursion variable x.

Now we have a rule selection strategy and an evaluation strategy. Combining these gives us the program $\texttt{innermost'}(\bar{\ell}_1 + \cdots + \bar{\ell}_n)$ that normalizes terms with respect to a set of rewrite rules $\ell_1, \ldots, \ell_n$.

3.3 Conditional Rules

A conditional rule $\ell : C_1 \wedge \ldots \wedge C_n \Rightarrow l \rightarrow r$ specifies that a term matching l can be rewritten to r if the conditions C_i hold. This is expressed by the program

$$\bar{\ell}(s) = \{\text{vars}(l, r, C_1, \ldots, C_n) : \mathsf{match}(l)\,;\,\mathsf{where}(\bar{C}_1(s)\,;\ldots;\,\bar{C}_n(s))\,;\,\mathsf{build}(r)\}$$

that first matches the left-hand side l against the subject term, then tests the conditions and finally builds the right-hand side r to replace the subject term; provided of course that each of these steps is succesful. A program $\bar{C}_i(s)$ denotes the translation of condition C_i. It is parameterized by a program s that is used to normalize terms as part of testing the condition. Again the program is embedded in a scope that makes the variables used in left-hand side, right-hand side and conditions local to the rule.

3.4 Join Conditions

A basic form of conditions is to test for equality of two terms. For example, the following rules use equality conditions to define insertion of an element in

a sorted list:

$$\texttt{Ins1} : \texttt{Insert}(x, \texttt{Nil}) \to \texttt{Cons}(x, \texttt{Nil})$$
$$\texttt{Ins2} : \texttt{Greater}(x, y) = \texttt{False} \Rightarrow \texttt{Insert}(x, \texttt{Cons}(y, l)) \to \texttt{Cons}(x, \texttt{Cons}(y, l))$$
$$\texttt{Ins3} : \texttt{Greater}(x, y) = \texttt{True} \Rightarrow \texttt{Insert}(x, \texttt{Cons}(y, l)) \to \texttt{Cons}(y, \texttt{Insert}(x, l))$$

In specification formalisms, equality is usually interpreted as membership of the symmetric transitive closure $\leftrightarrow^*$ of the rewrite relation $\to$. An approximation of this relation is a test for equality of the normal forms of the two sides of the condition; join conditions of the form $t_i = t'_i$ are satisfied if the terms t_i and t'_i rewrite to the same normal form.

Join conditions are implemented as follows: a condition $C_i \equiv (t_i = t'_i)$ induces the program definition

$$\bar{C}_i(s) = \{\texttt{x} : \mathsf{build}(t_i)\ ;\ s\ ;\ \mathsf{match}(x)\ ;\ \mathsf{build}(t'_i)\ ;\ s\ ;\ \mathsf{match}(x)\}$$

The parameter s is the evaluation strategy used to evaluate the terms in the condition. The program first builds the left-hand side of the condition, evaluates it using s and binds the result to the variable x. Then it does the same for the right-hand side of the condition. The final match only succeeds if the results are the same.

A set of conditional rewrite rules $\ell_1, \ldots, \ell_n$ can now be implemented by the strategy

$$\mu x(\texttt{innermost'}(\bar{\ell}_1(x) + \ldots + \bar{\ell}_n(x)))$$

where the strategy passed to the individual rules for evaluation of conditions is the evaluation strategy itself.

4 Extensions

In the previous section we have presented the implementation of rewriting with conditional rewrite rules with join conditions. In this section we consider several extensions of this scheme.

4.1 Matching Conditions

Join conditions test whether two terms have the same normal form, but cannot introduce new variables in the conditions or the right-hand side. A matching condition $t := t'$ specifies that the normal form of t' should match against the pattern t. Matching conditions can be used to bind an intermediate result to a variable that is used more than once in the right-hand side; to test that the result has a certain form; or to deconstruct the result of a computation and bind subterms to variables.

A matching condition $C \equiv (t := t')$ induces a program $\bar{C}(s)$ defined as:

$$\bar{C}(s) = \mathsf{build}(t')\ ;\ s\ ;\ \mathsf{match}(t)$$

where the parameter strategy s is used to evaluate the right-hand side of the condition.

4.2 Negative Conditions

Given the encodings of conditions above it is straightforward to define negative join conditions and negative matching conditions. A negative join condition $t \neq t'$ translates to $\neg(\bar{C}(s))$, where $\bar{C}(s)$ is the translation of the join condition $t = t'$. Such a negative condition tests whether the normal forms of two terms are different. Similarly, a negative matching condition $t :\neq t'$ translates to $\neg(\bar{C}(s))$, where $\bar{C}(s)$ is the translation of the matching condition $t := t'$. Such a negative condition tests whether the normal form of t' does *not* match the pattern t. Note that the variables in t are not bound in case the condition succeeds.

4.3 Rule Selection

In the innermost evaluation strategy of the previous section rules are selected non-deterministically, i.e., the order in which rules are tried is unspecified. This means that the implementation is free to choose any suitable ordering of the rules. This is the way most rewriting languages are defined. In case of non-confluent unconditional rules this means that the result of normalization can be any of the normal forms of a term, as would be expected. In the case of non-confluent *conditional* rules this leads to a more unpleasant situation. For example, consider the following set of rules:

$$\mathtt{R1} : \mathtt{A} \to \mathtt{B} \qquad \mathtt{R2} : \mathtt{A} \to \mathtt{C} \qquad \mathtt{R3} : \mathtt{A} = \mathtt{C} \Rightarrow \mathtt{D} \to \mathtt{E}$$

Depending on the choice made between `R1` and `R2`, the condition of rule `R3` could fail or succeed leading to the normal forms `D` and `E`, respectively. There are three approaches to this problem.

First, we could use *full backtracking* instead of local backtracking to find alternative ways to make a strategy succeed. In the case of our implementation of conditions, this entails that all choices made during the evaluation of either side of the condition are reconsidered if the condition fails. This approach guarantees completeness of condition checking. The major disadvantage of this approach is that choicepoints have to be stored until the end of evaluation.

Another solution is to abandon non-deterministic choice altogether and express all choices by means of left choice. This means that one has to specify a rule order; if $\ell_1, \ldots, \ell_n$ is an ordering of the rules of a specification, then

$$\mu x(\mathtt{innermost'}(\bar{\ell}_1(x) \leftarrow\!\!+ \cdots \leftarrow\!\!+ \bar{\ell}_n(x))$$

is the deterministic evaluation strategy for these rules. This is the solution chosen in functional languages where the ordering is usually the textual ordering of the rules in a program. The advantages of this approach are that the semantics of a specification is deterministic and that it admits an efficient implementation, i.e., local backtracking. The disadvantage of this approach is that the meaning of a rule becomes to depend on its context, i.e., its position in the rule order.

Finally, we can keep non-deterministic choice with local backtracking and be aware of the problems it causes. This seems to be the position of most conventional rewriting engines. It has the advantage of the efficient implementation of local choice and moreover gives freedom to the implementation to find an ordering on rules. The disadvantage of non-deterministic semantics can be turned into a methodological advantage by requiring rules to be mutually exclusive or orthogonal in the sense of Section 2, i.e., not to depend on an ordering. This means that rules can be considered as transformation rules indepent of the rest of the specification. Conventional specification languages summarize this requirement by means of phrases like "specifications are assumed to be confluent and terminating". Since orthogonality is an undecidable property, this requirement could be strengthend by choosing a decidable subset of orthogonal programs, e.g., non-overlapping match programs as guards of the branches of a choice. However, this is usually too severe a restriction for practical specifications. For example, the insert rules of the previous section do not have orthogonal left-hand sides (even identical), but the conditions make the rules orthogonal. This approach and the previous one can be realised in System S.

4.4 Default Rules

ASF+SDF [6] provides *default rules* that apply only if no other rules apply. This is very useful to reduce the size of specifications. For instance, an equality predicate $\texttt{Eq}(x, y)$ can be defined by

$$
\begin{aligned}
&\texttt{Eq1} : \texttt{Eq}(x, x) \to \texttt{True} \\
&\texttt{Eq2} : \texttt{Eq}(x, y) \to \texttt{False} \quad \textbf{otherwise}
\end{aligned}
$$

Whereas without default rules a rule has to be defined for each pair of constructors in the signature. The default rule is an example of a rule that does not make sense if its position in the rule order is not considered.

Default rules can be implemented in our scheme by separating the rules in two sequences of regular rules $\ell_1, \ldots, \ell_n$ and default rules $d_1, \ldots, d_m$. The evaluation strategy is then defined by

$$\mu x(\texttt{innermost'}((\bar{\ell}_1(x) + \ldots + \bar{\ell}_n(x)) \nleftarrow (\bar{d}_1(x) + \ldots + \bar{d}_m(x)))$$

where the rule selection is such that a default rule is only chosen in case all regular rules fail.

Default rules are a mechanism to break through the order of the rules in the specification. To take this one step further we can introduce a system with aribitrary many levels of priority. This would in effect give the user a mechanism to define the rule selection strategy.

4.5 Alternative Evaluation Strategies

Default rules provide an alternative rule selection mechanism. We can also vary the evaluation strategy.

Innermost evaluation sometimes does too much work because some subtrees are never used. For instance, an if-then-else operator only needs to evaluate one branch depending on the outcome of the condition. This can be improved by using an outermost strategy such as

$$\texttt{outermost}(s) = \texttt{repeat}(\texttt{oncetd}(s))$$

that looks repeatedly for a redex starting at the root. A more efficient version of this strategy is the parrallel outermost strategy defined as

$$\texttt{outermost-par}(s) = \texttt{repeat}(\texttt{sometd}(s))$$

that looks repeatedly for all outermost redices in parallell.

The outermost strategies try to do as little as possible. However, for many applications most operators are usually strict in all their arguments (i.e., all arguments should be normalized), except for a few operators. For instance, consider a language with a conditional construct $\texttt{If}(c, x, y)$ defined by the rules

$$\begin{aligned} \texttt{If1} &: \texttt{If}(\texttt{True}, x, y) \to x \\ \texttt{If2} &: \texttt{If}(\texttt{False}, x, y) \to y \end{aligned}$$

that perform the selection of the appropriate branch. In such situations it is attractive to evaluate all operators eagerly, i.e., innermost, but for the exceptions. The following strategy implements this idea. It is the same as $\texttt{innermost}'$, but it first tries to apply the strategy $\texttt{nonstrict}(x)$ instead of normalizing all direct subterms with $\Box(x)$.

$$\texttt{innermost-ns}(s) = \mu x((\texttt{nonstrict}(x) \nleftarrow \Box(x)) \,;\, \texttt{try}(s \,;\, x))$$

The strategy $\texttt{nonstrict}(x)$ tests whether the outermost function symbol of the subject term is an operator with non-strict arguments and if so evaluates only those arguments that are strict. For instance, in case of the `If` operator above, `nonstrict` is defined as:

$$\texttt{nonstrict}(x) = \texttt{If}(x, \epsilon, \epsilon)$$

Only the condition is evaluated with x, the branches of the conditional are not normalized.

5 Related Work

Algebraic specification formalisms such as ASF+SDF [6] and OBJ [7] provide ad-hoc extensions with non-standard features of the basic paradigm of rewriting. In this paper we have shown how such languages can be implemented in System S; we have presented all rewriting features of ASF+SDF except for list matching; furthermore, we have presented some new features, e.g., negative matching conditions and non-strictness annotations. System S provides a general framework for reasoning about issues such as non-determinism in rule selection and implementation of rule features.

User-definable strategies arose in theorem proving since proof steps are naturally non-confluent and non-terminating. The theorem proving framework LCF [8] introduced tactics for proving theorems. A tactic transforms a goal to be proved into a list of subgoals and a proof. By repeatedly applying tactics, a list of goals is reduced to the empty list, which indicates that the original goal is proven.

In the specification formalism ELAN [11] the notion of transformation of goals to a list of subgoals is generalized to arbitrary term rewrite rules. Strategies are regular expressions over the set of rule labels. In [11] this approach is used to define constraint solvers that consist of rules that rewrite a list of constraints into a new list of constraints. A strategy repeatedly applies such rules until a solution is found. In later versions of the language, e.g., [2,3], the set of strategy operators is extended with congruence operators to support term traversal.

System S generalizes the strategy language of ELAN and differs with it on the following points: (1) ELAN does not provide generic traversal operators analogous to our $i(s)$, $\Box$, $\Diamond$ and $\boxdiamond$ that enable very concise specification of term traversal. Instead traversals have to be defined explicitly for each datatype using congruences. (2) ELAN has two kinds of non-determinism: dc (don't care) and dk (don't know). The strategy $dc(s_1, s_2)$ chooses one of s_1 or s_2, which corresponds to local backtracking, i.e., our choice operator. The strategy $dk(s_1, s_2)$ applies both s_1 and s_2 and returns the set of all possible results. This corresponds to global backtracking. (3) ELAN has a fixed syntax for rewrite rules. Using System S rewrite rules are defined in terms of match, build and scope such that rules can easily be extended with expressive features that are implemented in terms of the core language. (4) System S has test and negation of strategies and has the explicit recursion operator $\mu x(s)$ where ELAN has recursive strategy definitions.

Maude [4] is a specification formalism based on rewriting logic. It provides equations that are interpreted with innermost rewriting and labeled rules that are used with an outermost strategy. Strategies for applying labeled rules can

be specified in Maude as a rewrite system. A reflection mechanism is used to reify strategies to evaluate terms. In this approach rewrite rules with a fixed strategy become the core language making strategies extensible. In System S strategy operators form the core language in terms of which rewrite rules are defined. In this approach rewrite rules are extensible, but the primitives of the strategy language are frozen.

The language described in this paper was inspired by the strategy language of ELAN. The first version was described in [13], which presents a strategy language with identity, sequential composition, choice, recursion, and a generic 'push-down' operator that is used to define $\Box$ and $\boxtimes$. An interpreter for strategy expressions is specified in the algebraic specification formalism ASF+SDF [6]. Basic strategies are unconditional ASF+SDF rewrite rules. In [14] System S is used as the core language for a language that is used for the specification of transformation rules and strategies. In particular, it introduces rules with contextual matching replacing. The language is illustrated by means of a specification of a program optimizer.

Technical contributions of our work in the setting of strategy languages include the modal operators $\Box$, $\Diamond$ and $\boxtimes$ that enable very concise specification of term traversal; a set of general purpose traversal strategies; the explicit recursion operator $\mu x(s)$; the refinement of rewrite rules into match and build; and the encoding of complex transformation rules into System S programs.

6 Conclusions

We have presented the implementation of conditional rewriting in System S and explored several extensions of the basic scheme, in particular rewriting with default rules, matching conditions and non-strictness annotations. This use of System S as a core language for rewriting is not limited to traditional languages based on rewrite rules but can also be applied to other kinds of languages that provide operations on trees. The added advantage of using System S is a systematization of the implementation of extensions of rewriting languages by means of primitives for matching and building terms and flexible definition of term traversal. Furthermore, it supports reasoning about non-determinism in specifications. Although the current implementation of System S cannot yet compete with dedicated implementations of innermost term rewriting, there is no reason that this could not be the case in the future.

Acknowledgements

We thank Bas Luttik and Andrew Tolmach for many discussions on rewriting strategies and their applications. Jan Bergstra pointed out the problems of non-determinism and conditional rules in languages such as ASF+SDF. Eugenio Moggi made clear that our semantics does not model programs with global backtracking, but rather local backtracking, which was what we were looking for.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[3] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In Masahiko Sato and Yoshihito Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific.

[4] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier, 1990.

[6] A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, September 1996.

[7] J. A. Goguen and T. Winkler et. al. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International Computer Science Laboratory, March 1992.

[8] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF. A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1979.

[9] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[10] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational semantics of OBJ-3. In *Proceedings of 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1988.

[11] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Implementing computational systems with constraints. In P. Kanellakis, J-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming*, pages 166–175, Providence R.I., USA, 1993. Brown University.

[12] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, Oxford, England, 1992.

[13] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[14] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In P. Hudak and Ch. Queinnec, editors, *International Conference on Functional Programming (ICFP'98)*, Baltimore, Maryland, September 1998. ACM.