On Rewriting Terms with Strict Functions and Error Propagation

Olaf Owe
Department of Informatics
University of Oslo

September 1990

Abstract

Partial functions play an essential role in the semantics of programs as a means to formalize non-terminating computations and computations terminating in an exception or error situation. However, when properly formalized, partial functions and errors often complicate underlying formalisms and reasoning systems. In this paper we focus on term rewriting systems, and discuss how partial functions and error propagation may be treated. In particular, we will be interested in strict functions because they arise from various implementation techniques. Reasoning with rewrite systems about such functions is non-trivial in the presence of errors due to the non-strict parameter passing semantics inherent in rewrite systems. In particular, rules formalizing strictness easily destroy rewrite rule confluence. We show how to mechanically extend rewrite rule systems so that strict parameter passing is imposed and so that convergent reasoning about terms which may turn out to be erroneous, becomes possible. Integration of non-strict functions will also be possible.

Contents

1 Introduction **2**

2 Limitations of order sorted rewriting **4**
2.1 Definite Errors 5
2.2 Strong Convergence 6

3 Guarded Rewriting **7**

4 Extensions **10**
4.1 Semantically Constrained Functions 10
4.2 Non-strict and Strict Functions Combined 10

5 Conclusions **11**

1 Introduction

Partial functions play an essential role in the semantics of programs as a means to formalize non-terminating computations and computations terminating in an exception or error situation. However, when properly formalized, partial functions and errors often complicate underlying formalisms and reasoning systems. In this paper we focus on term rewriting systems, and discuss how strict partial functions and error propagation may be imposed.

Examples of partial functions are division, subtraction on natural numbers restricted to natural results, pop- and top-operations on a stack, and push on a bounded stack. Consider the terms

$x \neq 0 \Rightarrow x/x = 1$
$x > y \Rightarrow x - y > 0$
$top(pop(push(push(s, x), y)))$ where s is a bounded stack

How can we rewrite such terms? In the two first terms the conditions should enable the result true. But what should the result be if the conditions are omitted?

A rewrite system [**?**] consists of a set of rules of form $a = b$ where each variable occurring in the right hand side b must also occur in the the left hand side a. A term t reduces to t' by a rule $a = b$ if t' is obtained from t by replacing an occurrence of an instance of a by the corresponding instance of b. The transitive reflexive closure of the "reduces-to" relation is denoted $\longrightarrow$. If each term t reduces to a unique irreducible term t' in finite steps, we say that the system is convergent, and we say that t' is *the reduction* of t.

Earlier approaches have dealt with partial functions in different ways. A simple approach [**?**] is to leave error situations unspecified. Rules may then freely be instantiated with terms that may contain errors. However, obtaining convergence is often difficult in the presence of error situations. For instance, the rules $x*0 = 0$ and $(x/y)*y = x$ are not confluent, since $(x/0)*0$ can be reduced to both 0 and x. One may avoid this kind of non-confluence using a conditional rewrite system, where each rule has a condition ensuring that all its applications of partial functions are welldefined; for instance the latter rule may be conditioned by $y \neq 0$. An irreducible term may then (to some extent) give an indication of possible error situations, but could just as well indicate unknown semantics. Therefore reasoning about presence of error is limited.

One may achieve a certain degree of reasoning about presence of error situations and error propagation by adding a special symbol **error** (or sev-

eral) indicating an error situation. (For the purpose of rewriting it could be considered a constant.) Rules with explicit occurrences of **error** in the right hand sides, but not in the left hand sides, identify error situations. Rules with occurrences of **error** in the left hand sides and with **error** as the right hand side identify error propagation. Again confluence can easily be destroyed with such rules. For instance, the rules $x/0 = \mathbf{error}$ and $(x/y) * y = x$ are not confluent. Strict error propagation rules of the form $f(.., \mathbf{error}, ..) = \mathbf{error}$ cause confluence problems even with a conditional system, because rewrite systems allow non-strict (and even non-monotonic) parameter passing semantics. For example, consider the rules

$$x * 0 = 0$$
$$x/0 = \mathbf{error}$$

The term $(x/0)*0$ reduces uniquely to 0. But confluence is lost when adding the propagation rule

$$\mathbf{error} * x = \mathbf{error}$$

Error propagation even affects the boolean operators. Consider the rules (for conjunction and negation)

$$\begin{aligned} x \wedge x &= x \\ \neg\mathbf{error} &= \mathbf{error} \\ x \wedge \neg x &= \text{false} \end{aligned}$$

The term $\mathbf{error} \wedge \neg\mathbf{error}$ can be reduced to both **error** and false.

The problems demonstrated above may be summarized as follows: The parameter passing semantics inherent in rewrite systems may easily give rise to operators which are non-strict and even non-monotonic (with respect to the flat ordering with error as the least element); for instance, the rule $x - x = 0 \quad [x : \mathit{Int}]$ makes integer subtraction non-monotonic; and the term $1/0 - 0/0$ reduces to $\mathbf{error} - \mathbf{error}$ (given $x/0 = \mathbf{error}$), and then to 0. Rewrite rules are often intended for instantiation of variables by defined values only. Strange results and loss of confluence then arise when such rules are instantiated with terms containing errors. One avoids these problems if instantiation is restricted to "defined values"; this gives strict parameter passing semantics. Under certain restrictions one may enforce strict parameter passing if rewriting is restricted to bottom-up (inside-out); but the reasoning power over terms with variables is drastically reduced.

With the approach of Ordered Sorted Algebra (OSA) [**?**, **?**, **?**] one may define a partial order between sorts; partial functions may (in many cases)

then be treated as total functions on the appropriate subdomains — and therefore conditions on the rules are not needed. By introducing functions (and constants) with subsorts as codomains, one may by sort analysis sometimes conclude that a term is syntactically wellformed, which implies that it is semantically welldefined (has a welldefined value) in a certain sense (in the initial algebra). The reverse is not true in general, i.e. there are non-wellformed terms who are welldefined (for a given set of rules). However, all such terms (and more) can be made wellformed after insertion of so-called retracts (which corresponds to coercion between subsorts of a common sort). Sometimes such applications of retracts may be removed by repeated type analysis after some rewriting steps (when no longer needed to ensure well-formedness). We say that a wellformed term is *strongly wellformed* if it contains no retracts, otherwise *weakly wellformed*.

A weakness is that reasoning about presence of errors is limited, since it is usually required that all rules are strongly wellformed. Strictness can then be ensured by insisting that only strongly wellformed terms of sort S may substitute an S-variable in a rule instantiation. One may then reduce only strongly wellformed (sub)terms. Even though retracts may sometimes be removed (as before), little reasoning can be done in general about terms which do not always have a welldefined value. (See section 2.)

Alternatively one may avoid retracts by introducing so-called error supersorts, which ranges over error values as well [**?**, **?**]. By so-called stratification one may in a systematic manner introduce the supersorts needed. Supersort rules may be applied to weakly wellformed terms, but since (most) user defined rules are restricted to the subsorts of welldefined values, the problems discussed above reappear. As suggested in [**?**] one may give the user the responsibility of providing explicit (sub)sort information of certain subterms, so that the relevant subsort rules apply. However, this approach has some serious draw-backs: It is non-trivial to decide which subterms to explicitly sort, and by which subsorts. Since the subsort claims will not be checked, the approach opens for the possibility of writing meaningless and logically inconsistent specifications. Finally, the approach assumes strictness of all functions. In practice, one may want at least some predefined functions to be non-strict (for instance an if-construct).

We shall show a different way of handling welldefinedness, by introducing guards. This allows all reductions possible with weakly wellformed rule instantiation, without any help from the user, and therefore the reasoning power is significantly increased. As in OSA the variables of the user-defined rules are understood to range over welldefined values only, and propagation

rules are implicitly given. Thus rules need not be carefully conditioned. The guards ensure convergence, and a strict semantics, and do also provide a separated analysis of welldefinedness aspects. It is not required that all functions are strict, one may introduce non-strict functions such as an if-construct.

For convenience, the presentation below will start from order sorted algebra with retracts rather than error supersorts.

2 Limitations of order sorted rewriting

In order sorted algebra it is usually required that the rules are sort decreasing, i.e. that the sort of a right hand side is the same as, or a subtype of, the sort of the corresponding left hand side. And an order sorted substitution must respect sorts, in the sense that a variable of sort S may only be replaced by a term of sort S (or smaller). In addition the term should be strongly wellformed, in order to avoid undesired semantics, as shown by the following example:

> **Example 1.** Given that division of *Int* (Integer) and *NZ* (NonZero) is *Int*, f of *NZ* is *NZ*, 0 is *Zero*, *Zero* and *NZ* are subtypes of *Int*, and given the rule $x/x = 1 \quad [x : NZ]$. The non-wellformed term
>
> $$f(0)/f(0)$$
>
> becomes after sort analysis
>
> $$f(r_{NZ}(0))/f(r_{NZ}(0))$$
>
> where r_{NZ} denotes a retract into *NZ*. The term would reduce to 1 if x of the rule is instantiated to the weakly wellformed *NZ*-term $f(r_{NZ}(0))$. (The same result would occur for the term $f(y)/f(y) \quad [y : Int]$.) This reduction is clearly undesirable and illustrates why strong wellformedness is required.

The next example shows how this requirement greatly limits the reductions possible on weakly wellformed terms.

> **Example 2.** Bounded stacks (*BS*) of elements (*EL*) with the usual operations may be defined as follows:

$$
\begin{array}{l}
empty : \rightarrow ES \\
push : NF * EL \rightarrow NE \\
pop : NE \rightarrow NF \\
top : NE \rightarrow EL
\end{array}
$$

$$
\begin{array}{ll}
pop(push(s, x)) = s & [s : NF, x : EL] \\
top(push(s, x)) = x &
\end{array}
$$

where NE (non-empty, bounded stacks) and NF (non-full, bounded stacks) are subsorts of BS, and ES (empty stacks) is subsort of NF.

The term $top(pop(push(push(s, x), y)))$ $[s : NF;\ x, y : EL]$ becomes, after sort analysis,

$$top(r_{NE}(pop(push(r_{NF}(push(s, x)), y))))$$

which is irreducible. And so are all instances of this term. It would be desirable to remove the outermost retract, which obviously cannot cause an error; and then be able to show that the term equals

$$top(r_{NF}(push(s, x)))$$

However, this is not possible in OSA since both terms are irreducible.

We write $S < S'$ if S is a proper subsort of S', and the subsort relation is extended to products of sorts in the obvious way.

A profile $f : D \rightarrow S$ is said to be a *subprofile* of another, if the latter has the form $f : D' \rightarrow S'$ and $D < D'$ and $S < S'$. A profile is $f : D \rightarrow S$ is *redundant* by $f : D' \rightarrow S'$ if $D \leq D'$ and $S' \leq S$. For instance the profile $push : ES * EL \rightarrow NE$ is redundant by, but not a subprofile of, $push : NF * EL \rightarrow NE$ (and given $ES < NF$).

2.1 Definite Errors

One may not reason about presence of errors in OSA. Even terms like $pop(empty)$ and $x/0$ are treated as weakly wellformed ones. In order to detect definite errors, we add the assumption that the least greatest lower bound of two types in the subtype graph corresponds to intersection of the

value sets, as suggested in [**?**]. Throughout the paper we assume that this assumption is satisfied, and we use the term *strong* order sorted algebra to denote order sorted algebra where the satisfaction concept is strengthened to accommodate our assumption. The intersection of sorts $\cap$ is then syntactically computable and is defined as the greatest lower bound, adding $\emptyset$ as the bottom type, corresponding to the empty set. Two sorts T, TT are said to be disjoint if $T \cap TT$ is $\emptyset$. Notice that the subsort graph need not be closed with respect to union of value sets.

For instance, the sorts *NZ* and *Zero* of example 1 are now interpreted as disjoint. Example 2 should be enriched with another sort, say *NEF*, less than both *NE* and *NF*, otherwise *NE* and *NF* would be interpreted as disjoint sorts. We believe that disjointness of subsorts is usually known in practical program specification; then our assumption does not cause other complications than identifying the existence of certain subsorts.

We add the rule:

$$r_T(t) = r_{T \cap TT}(t) \qquad [t : TT]$$

which shows that all retracts between sorts not directly related can be removed (by rewriting). A retract of form $r_\emptyset(..)$ represents a definite error. It follows that retracts between disjoint sorts must fail and represent definite errors.

Intersection of sort-products is defined in the obvious way; and we define the intersection of two f-profiles, say $f : D \to S$ and $f : D' \to S'$, as $f : D \cap D' \to S \cap S'$. For any set of profiles Σ we may derive a regular and monotonic set of profiles Σ' by taking the closure with respect to this intersection, and removing redundant profiles. If the set Σ' contains a profile $f : D \to \emptyset$ with D non-empty, we say that the set is *syntactically inconsistent* (since it cannot be modeled), as in [**?**].

The following rules for reasoning about definite errors, are obviously satisfied:

$$\begin{array}{ll} r_\emptyset(t) & = \mathbf{error} \\ f(.., \mathbf{error}, ..) & = \mathbf{error} \qquad \text{for each (strict) function argument} \end{array}$$

where the explicit **error** may be regarded as a (failing) retract into the empty sort. A system R extended with these rules is denoted $R_{\mathbf{error}}$. The extension preserves convergence, since no other rules mention **error** and since strongly wellformed terms never reduce to **error**.

In $R_{\mathbf{error}}$ one may reason with terms that contain definite errors, but reasoning with weakly wellformed terms which do not reduce to error, is as

limited as in R; i.e. if t reduces to t' in $R_{\mathbf{error}}$, and t' is not error, then t also reduces to t' in R. For instance, the term $x/0$ can be reduced to **error** (given the syntax of example 1), since $r_{NZ}(0)$ reduces to $r_{\emptyset}(0)$ and then to **error**. On the other hand, the term $x/x \quad [x : Int]$ becomes $x/r_{NZ}(x)$ as before.

2.2 Strong Convergence

A function specified by the profile $f : D \to S$ normally has function values in any specified subsort S' of S for some arguments in D (otherwise the first profile could be improved), i.e. the function satisfies $f : D' \to S'$ for some subproduct D' of D. However, in order to express such a D' one may need to introduce subsorts. Introduction of subprofiles (and subtypes) is quite natural and important when refining specifications. One may expect that addition of such subtypes and subprofiles (in a conservative manner) do not destroy convergence. However, a weakness of the OSA-concept of convergence is that one may loose convergence, both termination and confluence, by adding such subprofiles:

Example 3. (Loss of confluence)
Given $f : Int \to Int$, $+ : Int * Int \to Int$ and $Nat < Int$, and the rules

$$\begin{array}{ll} f(x+y) = x+y & [x, y : Int] \\ f(z) \quad\; = z+z & [z : Nat] \end{array}$$

This system is convergent, but confluence is lost by adding the profile $+ : Nat * Nat \to Nat$.

Example 4. (Loss of termination)
Given $g, h, f : Int \to Int$ and $Nat < Int$, and the rules

$$\begin{array}{ll} h(f(x)) = g(f(x)) & [x, y : Int] \\ g(z) \quad\; = h(z) & [z : Nat] \end{array}$$

This system is convergent, but termination is lost by adding the profile $f : Nat \to Nat$.

It seems useful to have a stronger notion of convergence which allows refinement of syntactic information (through adding subprofiles and subsorts in a conservative manner), without loss of convergence.

Definition. A *syntactic refinement* of a strongly order sorted rewrite system R is an extension obtained by adding new subsorts and subprofiles such that the rules remain sort decreasing, and no inconsistent profile can be derived, and (naturally) such that the extended subsort graph is closed with respect to intersection of value sets, without modifying $\cap$ on R-sorts.

It is natural to require that added subsorts do not change the greatest lower bound of R-sorts (it is sufficient that each new subsort have only one direct supertype among the old ones), since otherwise the closure of the extension of the closure of R may differ from the closure of the extension of R, and thus the extension would add non-trivial semantic information. [1]

Definition. We say that R is *strongly convergent* if every syntactic refinement of R is convergent.

One may prove strong convergence without looking at possible refinements: R is strongly convergent if and only if R is convergent when retracts are allowed in unification of a term with a variable (which gives more superpositions to consider). And the concept of strong convergence causes no additional concern regarding the termination issue provided the termination proof is based on a simplification ordering which does not depend on subsort information. [2]

Notice that if R is strongly convergent then $R_{\mathbf{error}}$ is also. Examples 1 and 2 above are strongly convergent, but obviously not 3 and 4 (the indicated refinements are syntactic).

> **Example 5.** Given that Nat is a subsort of Int and $+ : Int * Int \to Int$, the two rules
>
> $$0 \leq x = true \qquad [x : Nat]$$
> $$0 \leq (x + y) = -x \leq y \qquad [x, y : Int]$$
>
> form a convergent system, but not a strongly convergent one, since convergence is lost when adding the subprofile $+ : Nat * Nat \to Nat$.

[1] One could also require that no original R-profile becomes redundant (in a certain sense) in order to disregard extensions of more serious nature. However, this would not influence the results to come.

[2] One could allow a precedence order of the function symbols by ordering overloaded versions of a function according to the subsort order on the codomains, provided there is no overlap with other functions.

3 Guarded Rewriting

In the following we show how to extend a strongly order sorted rewrite system R into a stronger one, denoted R^+, which is capable of non-trivial reasoning with non-welldefined terms.

We consider a strongly convergent system R with wellformed and sort decreasing rules. We add the rule

$$r_S(t) = t \in S \mid t$$

(t is of any sort connected with S), introducing two new operators: $\mid$ (for guarded terms) and $\in S$ (for membership in sort S). The operator $\mid$ binds weaker than any other (except $=$), and associates to the right. With this rule one may remove all retracts, and the resulting term may be considered strongly wellformed! The guarded term $d|t$ may be understood as "if d then t else **error**". This motivates the following rules (schemas):

$f(..,d|t,..) = d \mid f(..,t,..)$ for all (strict) R-functions f, including retracts
$d1|d2|t = d1 \wedge d2 \mid t$

(The first rule schema should be extended to handle all combinations of one or more guarded arguments.) The two first rules ensure that inner occurrences of $\mid$ can be removed. Notice that the OSA requirement of strongly wellformed instantiation implies that inner occurrences of retracts must be removed first. In the resulting conjunction of guards, guards corresponding to inner retracts will appear before those corresponding to outer retracts. These rules could be performed efficiently during the sort analysis in an inside-out manner. (The two occurrences of t introduced by the first rule could be represented once if terms are represented by a graph structure rather than a tree structure.)

Clearly we may let a guard $t \in S$ allow t to be used as a strongly wellformed S-term inside the guarded (sub)term, since truth of the guard implies success of the retract. This means that t may safely be used for instantiation when applying a rule inside the subterm. In fact, after all retracts are turned into guards and all guards are moved outermost, all matching of rules can safely be done without the requirement of strong wellformedness! Therefore all reductions on the original term with weakly wellformed instantiation, can be done to the the guarded term, and the results would be the same, except for the guard, which gives a result in accordance with strong instantiation. For instance the term $f(y)/f(y) \quad [y :$

Int] from the example above now results in $y \in NZ \mid 1$, expressing that the term is 1 when y is NZ and otherwise error.

The following rules restrict the $\in S$ predicate for each sort S:

$$\begin{array}{lll} t \in S & = t \in S \cap T & [t:T] \\ t \in \emptyset & = false & \end{array}$$

The first follows from the assumption of intersection-closure. [3] In order to apply the first rule as often as possible, we let the sort analysis take advantage of other guards, say $t' \in T'$, where t' is a subterm of t: the rule applies if it can be found that t is of sort T when using the fact that t' is of sort T'.

> **Example 6.** Consider again integers *Int* with the subtypes *NZ* (non-Zero), *Nat* (non-negative naturals), *Neg* (non-positive negatives), where *Zero* is a subtype of *Nat* and *Neg*, and where *Nat1* is a subtype of *NZ* and *Nat*, and where *Neg1* is a subtype of *NZ* and *Neg*. Negation has the profiles $- : Int \to Int$, $- : Nat \to Neg$, $- : Neg \to Nat$, $- : NZ \to NZ$, the (truncated) square-root function has the profile $sqrt : Nat \to Nat$, multiplication $* : Int * Int \to Int$, and division as before. Consider the term $sqrt(x) * sqrt(-x)/x$, which becomes $x \in Nat \wedge -x \in Nat \wedge x \in NZ \mid sqrt(x) * sqrt(-x)/x$ in which the second guard may be simplified to $-x \in Zero$ and then to $-x \in \emptyset$ and finally to $false$. Thus the whole term is recognized as a definite error.

Occurrences of true and false in guards are simplified as follows:

$$\begin{array}{ll} true|t & = t \\ false|t & = false \mid \Box \end{array}$$

where $\Box$ is a new constant, considered strongly wellformed of any sort (i.e. of sort $\emptyset$). Intuitively, $\Box$ may be understood as the unknown (or as "don't care"); and $false|\Box$ represents an error. The $\Box$ ensures uniqueness in the case of false guards.

4

[3] Without this assumption, the rule would be $t \in S = true \quad [t:S]$ which corresponds to the OSA removal of retracts. Notice that this rule reduces $t \in S \wedge t \in T$ to $t \in S$ if S is a subsort of T.

[4] When non-strict functions are considered: It may be unified with any term ???

Let R^+ denote the rewriting system consisting of the original R-rules, the explicitly added rules, and the following rules for $\wedge$.

$$\begin{aligned} true \wedge t &= t \\ false \wedge t &= false \\ t \wedge t &= t \end{aligned}$$

Rather than rewriting modulo commutative, associative rules for $\wedge$, a simple sorting of its arguments suffices, for instance with respect to term length, and secondly lexicographical ordering (provided this does not conflict with rules for $\wedge$ in R).

Lemma 1. Assume R is strongly convergent for strongly wellformed terms. Then R^+ is also strongly convergent.

Lemma 2. The R^+-reduction of a weakly wellformed term is strongly wellformed, and it may not contain **error**.

R^+ is an extension of R in the sense that the two systems have the same strongly wellformed irreducible reductions:

Lemma 3. Assume R is strongly convergent, and let t' be the R-reduction of t. If t' is strongly wellformed, it is also the R^+-reduction of t. Otherwise, the R^+-reduction of t has the form $d|t''$ where t'' may be obtained from t by R ignoring the requirement of strongly wellformed instantiation, and then removing any remaining retracts by the rules above. The resulting guard will be implied by d.

The guard d expresses the condition that t is welldefined in the sense that d can be reduced to true for exactly those instantiations for which t has a strongly wellformed R-reduction. And d expresses strict parameter passing semantics in the following sense:

Lemma 4. Reduction to **error** in $R_{\mathbf{error}}$ implies reduction to $false|\Box$ in R^+. Furthermore, R^+ and $R^+_{\mathbf{error}}$ give the same reductions when the rule $\mathbf{error} = false \mid \Box$ is added to the latter system.

Notice that error propagation rules may be included even for $\in$ and $|$ (in both arguments), showing that these function are strict as well. However, the rules $\mathbf{error} \mid t = \mathbf{error}$ and $\mathbf{error} \in S = \mathbf{error}$ can never be applied since errors may not occur in guards. And the rule $(d \mid \mathbf{error}) = \mathbf{error}$ in not needed because of $\mathbf{error} = false \mid \Box$.

Example 7. Consider the bounded stack system of example 2 (with *NEF*). The (previously irreducible) term

$$top(r_{NE}(pop(push(r_{NF}(push(s, x)), y)))) \quad [s : BS]$$

now reduces to

$$push(s, x) \in NF \mid x$$

which may be read as: if $push(s, x)$ is inside NF then x else error.
This reduction result is also obtained for the term $top(r_{NF}(push(s, x)))$, which shows that the two *top*-terms are equal.

Consider example 1. The term $x/x \quad [x : Int]$ now reduces to $x \in NZ \mid 1$. (And the term $x/0 \quad [x : Int]$ reduces to $false \mid \Box$.)

The system R^+ offers reasoning about (the definedness and the value of) weakly wellformed terms by means of only strict functions and strict operators (including $|$, $\in$ and $\wedge$). Even though $|$ may be interpreted by a non-strict if-construct, R^+ ensures that both arguments of $|$ always are strongly wellformed, and hence $|$ may be defined as strict. In the next sections we extend our system to cater for semantically constrained functions and non-strict functions.

4 Extensions

4.1 Semantically Constrained Functions

We may extend the system above to handle (strict) functions defined on semantically constrained domains, such as minus on natural numbers (letting $x - y$ be defined for naturals x, y if and only if $y \leq x$). The sort analysis may handle such functions like retracts, inserting the constraint as a guard. For instance the (sub)term $a - b$ becomes after parsing $b \leq a \mid a - b$ where $a - b$ is considered strongly wellformed; after this parsing all subterms may be considered strongly wellformed. As before further reductions are possible even if the constraint cannot be reduced to true or false.

Constrained functions may occur in a rule if the guards (constraints) obtained by sort analysis of the right hand side are contained in those of the left hand side. Such a rule may only be instantiated when the constraints of the instantiated left hand side are satisfied, or occur as guards to the (sub)term being reduced. In our system this is always the case, since the sort analysis has produced the necessary guards!

A similar discussion applies to the use of retracts in right hand sides.

The results above generalizes to the case of constrained functions, if we require constraints to be strongly wellformed terms. Convergence of guards requires that constraints converges, which is implied by convergence of R.

4.2 Non-strict and Strict Functions Combined

The results above may be extended to systems with both strict and non-strict functions. For instance it may be desirable to allow an if-then-else construct with only the first argument strict.

For non-strict argument positions, we must of course remove the corresponding rules prescribed above defining strict propagation of errors and guards; and instead provide propagation rules of form

$$f(..,d|t,..) = g \mid f(..,t,..)$$

where d and t are variables and where the guard g is a strongly wellformed term containing strict arguments and d (and guarded non-strict arguments). This implies that all guards can be collected outmost and that the resulting guard is properly protected against (internal) errors. For instance, for the non-strict arguments of the if-construct we need

$$\begin{array}{ll} if(a,d|b,c) & = a \Rightarrow d \mid if(a,b,c) \\ if(a,b,d|c) & = \neg a \Rightarrow d \mid if(a,b,c) \end{array}$$

(which requires rules for $\neg$ and $\Rightarrow$). For convergent analysis of guards, we now need rules reducing all propositional tautologies to true. However, it suffices with two valued propositional logic since any error occurrence in a guard is properly protected. For instance, the guard of $if(y \leq x, sqrt(x - y), sqrt(y - x)) \quad [x, y : Nat]$ is $(y \leq x \Rightarrow y \leq x) \wedge ((\neg y \leq x) \Rightarrow x \leq y)$ which reduces to true with appropriate rules for (two-valued) $\Rightarrow$ and $\leq$.

The semantics of non-strict operators may be defined by ordinary rules (without guards), which may be applied as usual. These rules have no critical pair with the propagation rules, and therefore the two sets of rules can be developed independently, which makes the convergence issue easier. For many practical purposes a fixed set of non-strict functions suffices. Then a convergent set of predefined rules could be developed, such that they would not cause superpositions with later rules provided the latter ones do not contain non-strict functions in their left hand sides.

5 Conclusions

We have focused on rewriting of weakly wellformed terms. A non-welldefined term t may be rewritten to a guarded term of form $d|t'$ where d is a boolean term expressing the welldefinedness of t, and where t' equals t when d is

satisfied. Neither d nor t contain errors. This form of guarded rewriting may be used to reason about partial functions, both syntactically and semantically constrained, and which allows rules for strict error propagation. In particular, we have developed restrictions that ensure strong convergence. Integration of non-strict functions is also possible.

Our approach gives significantly stronger reasoning about non-welldefined terms than the OSA approach with retracts and the one with error supersorts. In contrast to the use of error supersorts our method allows unrestricted rewriting of partly erroneous terms.

We have shown our ideas using OSA, but the results are not limited to OSA. In sorted algebra all partial functions must be handled as semantically constrained functions, letting the guard introduce the constraint.

In addition we have found a notion of strong confluence, which has proved to be useful.

References

[1] R.J. Cunningham, A.J.J. Dick: "Rewrite Systems on a Lattice of Types" *Acta Informatica* 22, 149-169, 1985.

[2] K. Futasugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: "Principles of OBJ2." In *Proceedings, 1985 Symposium on Principles of Programming Languages and Programming*, Association for Computing Machinery, 1985, pp. 52-66. W. Brauer, Ed., Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 194.

[3] M. Gogolla: "Algebraic specifications with partially ordered sorts and declarations." Forchungsbericht 169, Universität Dortmund, Abteilung Informatik, 1983. Revised version: Partially ordered sorts in algebraic specifications. In: B. Courcelle, editor, *Proceedings of the Ninth Colloquium on Trees in Algebra and Programming*, pages 139-153, Cambridge University Press, 1984.

[4] J.A. Goguen, J. Meseguer: "Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations." SRI International, 1988.

[5] J.A. Goguen, T. Winkler: "Introducing OBJ3." SRI International, SRI-CSL-88-9, 1988.

[6] J.V. Guttag, J.J. Horning, J.M. Wing: "Larch in Five Easy Pieces." Digital Systems Research Center, Palo Alto, California, July 1985.

[7] Knuth & Bendix: "Simple Word Problems in Universal Algebras." In *Computational Problems in Abstract Algebra*, Pergamon Press, New York, 1970.

[8] O. Owe, O.-J. Dahl: "Generator Induction in Order Sorted Algebras." *Formal Aspects of Computing*, vol. 3, pp. 2-20, 1991.

[9] G. Smolka, W. Nutt, J.A. Goguen, J. Meseguer: "Order-Sorted Equational Computation." SEKI-Report SR-87-14, Universität Kaiserlautern, Fachbereich Informatik, 1987. Revised version in: H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures,* Volume 2, pages 297-367, Academic Press, 1989.

[10] H. Comon: "Equational Formulas in Order Sorted Algebras." In: M.S. Paterson, editor, *automata, Languages and Programming, 17th International Colloquium, Warwick, England, Proceedings*, LNCS 443, pages 674-688, Springer Verlag, 1990.

[11] C. Kirchner, H. Kirchner: "Constrained Equational Reasoning." Centre de Recherche en Informatique de Nancy, 1989.