Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

A model for I/O in equational languages with don't care non-determinism

H.R. Walters and J.F.Th. Kamperman

Computer Science/Department of Software Technology

CS-R9572 1995

Report CS-R9572
ISSN 0169-118X

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

A Model for I/O in Equational Languages with Don't Care Non-Determinism

H.R.Walters (pum@cwi.nl)
J.F.Th.Kamperman (jasper@cwi.nl)

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

Abstract

Existing models for I/O in side-effect free languages focus on functional languages, which are usually based on a largely deterministic reduction strategy, allowing for a strict sequentialization of I/O operations. In concurrent logic programming languages a model is used which allows for don't care non-determinism; the sequentialization of I/O is extensional rather than intensional. We apply this model to equational languages, which are closely related to functional languages, but exhibit don't care non-determinism. The semantics are formulated as constrained narrowing, a relation that contains the rewrite relation, and is contained in the narrowing relation.

We present constrained narrowing and some of its properties; a constructive method to transform conventional term rewriting systems to constrained narrowing systems; and a discussion on requirements for an implementation.

CR Subject Classification (1991): D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages, *Algebraic approaches to semantics*; F.4.1 [Mathematical Logic and Formal Languages]: Logic Programming.

AMS Subject Classification (1991): 68N17: Logic Programming, 68Q40: Symbolic computation, 68Q42: Rewriting Systems and 68Q65: Algebraic specification.

Keywords & Phrases: narrowing, I/O, declarative languages, term rewriting, (concurrent) logic programming, specification languages, formal semantics.

Note: Partial support received from the Foundation for Computer Science Research in the Netherlands (SION) under project 612-17-418, "Generic Tools for Program Analysis and Optimization".

1. Introduction

Several models have been formulated [Lan65, JW93, AvGP92] which aim to reconcile the side-effect free nature of functional languages with the inherently imperative nature of I/O. This is a hard nut to crack, since the very purpose of I/O is the effectuation of side effects. Side effects invalidate referential transparency, thereby inhibiting equational reasoning and complicating program transformation, and they imply significant sequentialization of operations, thereby opposing non-determinism and lazy program evaluation [JW93]. Most prominently, monadic I/O is found to be a model which addresses these issues, and which has all desired operational and formal properties. The basis of the models described in [Lan65, JW93, AvGP92], including that of monadic I/O, is a strict sequentialization of I/O operations.

Equational languages (we aren't aware of a generally accepted meaning of this phrase, so we use it loosely) such as OBJ ([GKK+88, KKM88]) or ASF+SDF ([BHK89]) are closely related to functional languages. One of the key differences, however, is the essential assumption of non-determinism: a functional reduction strategy is largely deterministic. When functional

I/O models are used in equational languages, the strict sequentialization of I/O operations imposes significant determinism.

Concurrent logic languages also embrace non-determinism, and there, a suitable model is defined, which is based on unification. We formulate that model in the context of equational languages with don't-care non-determinism.

1.1 Equational Languages and Non-Determinism

Implementations of equational languages differ in the way they approach non-determinism. The non-determinism occurs when more than one rule can be applied to a term, or when rules can be applied in different places in a term. There are two kinds of non-determinism: *don't care*, and *don't know* [MOI95].

If only some of the possible choices may lead to useful results (the others leading to failure or infinite computations), it is called don't-know non-determinism; if theoretical grounds exist which imply that any choice will lead to the desired result, it is don't-care non-detereminism.

Don't-know non-determinism is computationally much more expensive to implement, because the solution space needs to be searched exhaustively, either breadth-first (requiring much memory), or depth-first (backtracking). In the context of don't care non-determinism, only a single thread needs to be followed.

In this article we are solely interested in don't-care non-determinism.

The most notable example of a calculus requiring don't-care non-determinism is term rewriting over complete rewrite systems. We will now argue that TRSs are unsuitable to model output.

1.2 Term Rewriting and I/O

The term rewrite relation is a passive relation in the context of equational logic, whereas I/O is an expression of activity. Operationally this contradiction is less apparent because an implementation of term rewriting is an active procedure which computes one particular reduction sequence. In this article we strive to model I/O for such a procedure in a manner that is meaningful and consistent in the formal context, even though the notion of I/O is unimaginable there.

Input is relatively easily modeled by lazy introduction of the subject term. As long as the implementation has had no need to look at some sub-term, it is formally irrelevant whether it is already 'filled in' or not.

Output is less straightforward:

- In general, the end-result is not yet known in any detail, so parts of it can not yet be produced;
- Reports on individual rewrite steps and substitutions are unsatisfactory: due to non-determinism, reductions can be postponed arbitrary lengths of time (unless no other reduction is possible, of course), so output appears in arbitrary order.

For this reason we leave the realm of pure term rewriting, as we will see in the direction of narrowing. Unfortunately narrowing involves don't-know non-determinism, and is computationaly less attractive [MOI95]. We develop a calculus, called constrained narrowing, that

allows output to be modelled, but has the computational advantages of term rewriting. As a relation, constrained narrowing subsumes term rewriting and is subsumed by narrowing.

1.3 Simple Examples

Consider a rewrite system consisting of a single rule: $f(a) \to b$ (here, a and b are constants), and consider the term $f(x)$ (where x is a variable). Under ordinary term rewriting the term $f(x)$ is a normal form, but it can be narrowed (to b) by applying the substitution $x \mapsto a$. Assuming x is an *output variable*, this reduction is also valid under constrained narrowing. Observe that a variable is used to indicate the place where output will be produced, and that the actual output is defined in the substitution that instantiates that variable. That is, when $f(x)$ is narrowed to b, the output $x \mapsto a$ can be effectuated in the outside world.

As a second example, consider the two rules $f(g(x)) \to h(x)$ and $h(a) \to b$, and consider the term $f(y)$. This term can be narrowed, first to $h(z)$ (by applying the substitution $y \mapsto g(z)$), and then to b, by applying $z \mapsto a$. The output defined by this constrained narrowing sequence is contained in the two substitutions $y \mapsto g(z)$ and $z \mapsto a$, or equivalently, $y \mapsto g(a)$. Note that the output is produced partially ordered top-down. Otherwise, non-determinism is maintained entirely.

Earlier, we mentioned that input can be modeled by lazy, incremental introduction of the subject term. For uniformity's sake we present this in the same framework of constrained narrowing. A not-yet-filled-in term is represented as a variable. Unlike an output variable, an input variable is constrained by the actual input, and a narrowing step is only a *constrained* narrowing step if that variable is instantiated in accordance with the input.

Consider the two rules $g(a) \to c$ and $g(b) \to d$, and consider the term $g(x)$, where x is an *input variable*. Suppose that input is provided stating that x should only be narrowed to a. In this context the narrowing step $g(x) \to d$ is not a constrained narrowing step, whereas $g(x) \to c$ is.

To summarize: a constrained narrowing step is either a term rewriting step, or a narrowing step in which only certain variables can be instantiated and under certain conditions.

1.4 Overview

We do not limit ourselves to text-oriented I/O, but rather assume there to be an interpretation from (arbitrary) terms to the contents of files. Thus, for instance, a list of characters (for example built using the function symbols *cons*, *nil* and the characters), can be interpreted as a text file, and an association table as an indexed file with variant records.

In this paper we take 'file' to refer to proper files as well as I/O streams.

The remainder of this paper is organized as follows. In Section 2, we present an illustrative example of the model, in Section 3, we define *constrained narrowing*, which combines well-known concepts from term rewriting ([Klo92, Red85, Hul80, MOI95]) and concurrent logic programming ([Sha89]), and we show how this models I/O. In Section 4 we discuss a few properties of constrained narrowing. In Section 5 we discuss how a CNS (constrained narrowing system) can be obtained from an ordinary TRS. In Section 6 we briefly discuss the implementation of the model. In Section 7, we discuss related work. Finally, we present our conclusions.

2. Example

In this section we consider a practical example of a constrained narrowing system involved in I/O. We define a function *inverse*, which produces on output a bit-wise inverted copy of its input, and which counts the number of bits that were inverted. Input and output are strings of bits, represented using the constants 0 and 1, and the constructors *cons* and *eof*. The normal form of application of *inverse* on proper input shall be a number in *successor-zero* notation. Finally, we use the auxiliary function *not*, the rules for which we will not show. In this section, b, f and g, possibly with subscripts, indicate variables.

$$inverse(cons(b, f), cons(not(b), g)) \rightarrow succ(inverse(f, g)) \quad (2.1)$$

$$inverse(eof, eof) \rightarrow zero \quad (2.2)$$

Consider the term $inverse(f_1, f_2)$.

In the context of constrained narrowing, we must indicate whether variables are input, output, or neither. For this example we will assume any variable with an odd index to be input related, and any variable with an even index to be output related.

Also, the input – say, the string "0" – is provided by expressing the requirement that f_1 should only be instantiated with a term that matches $cons(0, eof)$.

There are are infinitely many narrowing sequences applicable to the subject: rule 2.2 can be applied, producing the normal form *zero*, or rule 2.1 can be applied, which results in the successor of a term that is identical to the original term up to renaming of variables.

Given the constraints, only one *constrained* narrowing sequence exists, up to renaming of variables. It is shown below. We use the symbol $\succ\!\Longrightarrow$ for constrained narrowing steps, and above that symbol, the most general unifier underlying the step is shown. The rules applied are 2.1, and 2.2, respectively.

$$inverse(f_1, f_2) \underbrace{\begin{array}{l} f_1 \equiv cons(b_1, f_3) \\ f_2 \equiv cons(not(b_1), f_4) \end{array}}_{\succ\!\Longrightarrow} succ(inverse(f_3, f_4)) \underbrace{\begin{array}{l} f_3 \equiv eof \\ f_4 \equiv eof \end{array}}_{\succ\!\Longrightarrow} succ(zero)$$

As in our earlier examples, the output defined by this narrowing sequence is defined as whatever f_2 is (eventually) narrowed to, which is $cons(not(b_1), eof)$. The input variable b_1 is associated with a part of the input which has not yet been inspected. It is consistent with our model to produce output containing such variables, but in practice it may be desirable to instantiate that variable in accordance with the input.

Secondly, the output term is not a normal form. Our model does not require it to be one, but in practice this may again be desirable. In that case the term should be normalized, at which time the entire term is inspected after all. The produced output is $cons(1, eof)$.

To summarize:

- Input is provided by defining a constraint on the value that input variables can be instantiated with. This constraint makes certain narrowing steps invalid in constrained

narrowing. A rule is only applicable if a unifyer exists which only affects input and output variables, and which concurs with the input constraint.

- The input is inspected in discreet stages (because each narrowing step only inspects the part defined by the most general unifier underlying the narrowing step). Variables introduced in the most general unifier correspond to uninspected parts of the input, and they are implicitly constrained by the original input requirement.
- Output variables in the subject term may be instantiated as a result of narrowing steps. The grand total of these instantiations constitutes the output.
- Not all input needs to be inspected eventually, and uninspected input (i.e., unbound variables) may occur in the output (however, normalizing the output necessitates inspection of those variables).
- The output is not automatically put in normal form. A practical operationalization is the requirement that all output is normalized, in which case unused parts are inspected after all.

3. Constrained Narrowing

In this section we define *constrained narrowing*. First we introduce notation and terminology, which are essentially consistent with [Klo92] and [DJ90].

3.1 Basics

A *signature* Σ consists of:

- A countably infinite set $\mathcal{V}$ of *variables*: $x, y, f_1, c_2, \ldots$;
- A non-empty set $\mathcal{F}$ of *function symbols*: $f, g, cons, \ldots$, each with an *arity* (≥ 0), which is the number of arguments the function requires. In this paper, variables and functions can be distinguished from context.

The set $T(\Sigma)$ of terms over Σ is the smallest set satisfying

- $\mathcal{V} \subset T(\Sigma)$;
- for all $f \in \mathcal{F}$ with arity n, and $t_1, \ldots t_n \in T(\Sigma)$, we have $f(t_1, \ldots, t_n) \in T(\Sigma)$.

We write $x \in t$ if x occurs in t, and $var(t)$ for $\{x \in \mathcal{V} | x \in t\}$.

A path in a term is represented as a sequence of positive integers. By $t|_p$, we denote the *sub-term* of t at path p. For example, if $t = f(g, h(f(y, z)))$, then $t|_{2.1}$ is the first sub-term of t's second sub-term, which is $f(y, z)$. We write $p \in s$ if p is a valid path in s (i.e., indicates a sub-term of s), and $p_1 \leq p_2$ if p_1 is a prefix of p_2 (i.e., $\exists p_3 : p_2 = p_1.p_3$). The empty path (referring to root) is written as ε. We write $t[s]_p$ for the term resulting from the replacement of $t|_p$ in t by s.

A *substitution* is a (total) map $\sigma : T(\Sigma) \mapsto T(\Sigma)$ which satisfies $\forall f \in \mathcal{F} \ : \ \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. By convention, we often write t^σ for $\sigma(t)$.

Let the *carrier* of a substitution σ (denoted as $Car(\sigma)$) be the set $\{x \in \mathcal{V} | \sigma(x) \neq x\}$, and the *variable range* ($Ran(\sigma)$) the set $\{x \in \mathcal{V} | \exists y \in Car(\sigma) : x \in \sigma(y)\}$. A substitution σ_1 is more general than a substitution σ_2 ($\sigma_1 \preceq \sigma_2$) if a substitution σ_3 exists such that $\sigma_2 = \sigma_3 \circ \sigma_1$. We shall write $\bigcirc_{i \leq n} \tau_i$ for $\tau_n \circ \ldots \circ \tau_1$, or just $\bigcirc \tau_i$ if the range is clear from context.

A *unifier* of a set of terms $t_1, \ldots, t_n$ is a substitution σ which satisfies $t_1^\sigma = \ldots = t_n^\sigma$. The *most general unifier* (abbreviated *mgu*) of a set of terms is the smallest unifier, w.r.t. $\preceq$, and it is unique up to renamings of variables.

In many texts, this latter property is used implicitly to obtain renamings of an *mgu* in which no unintended name-clashes occur. In this paper we are more explicit. We introduce the notation mgu_S for a renaming of the most general unifier in which no variable occurring in any of the terms in the control set $S = \{s_1, \ldots, s_k\}$ is used. That is, if $\sigma = mgu_{\{s_1,\ldots,s_k\}}(t_1, \ldots, t_n)$, then $x \in s_i \Rightarrow x \notin Ran(\sigma)$.

For example, three renamings of $mgu(f(x), f(y))$ are $\{x \mapsto y, y \mapsto y\}$, $\{x \mapsto x, y \mapsto x\}$ and $\{x \mapsto z, y \mapsto z\}$, but only the third is (an instance of) $mgu_{\{x,y\}}(f(x), f(y))$.

A *rewrite rule* is a pair of terms written as $s \to t$ with $s, t \in T(\Sigma)$. It is assumed that the left-hand side s of a rule $s \to t$ is not a sole variable, and, in the case of ordinary term rewriting, that $var(t) \subset var(s)$.

A *term rewriting system* $\mathcal{R}$ consists of a signature Σ and a collection of rewrite rules R.

A term rewriting system defines a *rewrite relation* $\to_{\mathcal{R}}$. Since the subscript $\mathcal{R}$ is usually clear from the context, it is omitted. The overloading of $\to$ (relation and notation of rules) is by convention.

$$s \to t \stackrel{\text{def}}{\iff} \exists \sigma, p, u \to v \in R : s|_p = u^\sigma \wedge t = s[v^\sigma]_p.$$

The sub-term $s|_p$ is referred to as *redex* (for reducible expression); the sub-term $t|_p$ as *reduct*, as is t itself, on occasion. A series of terms $s_1, s_2, \ldots$ such that $s_1 \to s_2 \to \ldots$ is called a *rewrite sequence*. A term s is said to be in *normal form* if there is no t such that $s \to t$.

A term rewriting system also defines the (one step) *narrowing relation*[1] $\rightarrowtail$.

$$s \rightarrowtail t \stackrel{\text{def}}{\iff} \exists p \in s,\ u \to v \in R, \sigma = mgu(s|_p, u) : s|_p \notin \mathcal{V} \wedge t = s^\sigma[v^\sigma]_p.$$

Various properties of narrowing are discussed in [Hul80, Han94, MOI95, Klo92]. The definition of narrowing allows for pathetic renamings of the unifier, which we would like to avoid. We introduce the *most general narrowing relation* to provide 'global' generality: the substitution used in a rewrite step does not introduce variables occurring in the rewrite relation R; the initial subject term; or which have already been introduced by earlier substitutions. Note that this is not a new relation, but rather a constraint on 'legal' renamings.

[1] In most texts, this relation is trivially extended to equations, and it it is used for solving unification problems in equational theories that are presented by confluent TRSs, most notably the determination of solutions to equations in equational theories. These applications of narrowing are so common, that they are are identified with the relation itself, but in this paper we use the relation for another purpose.

$$s_n \overset{\sigma_n}{\rightarrowtail}_{(s_1,\sigma_1,s_2,\ldots,\sigma_{n-1})} s_{n+1} \overset{\text{def}}{\Longleftrightarrow}$$
$$s_1 \overset{\sigma_1}{\rightarrowtail} s_2 \overset{\sigma_2}{\rightarrowtail} \ldots \overset{\sigma_{n-1}}{\rightarrowtail} s_n \wedge$$
$$\exists p_n \in s_n, u_n \rightarrow v_n \in R, \sigma_n = mgu_{\cup_{i<n} Ran(\sigma_i) \bigcup \{s_1\} \bigcup var(R)}(s_n|_{p_n}, u_n) :$$
$$s_n|_{p_n} \notin \mathcal{V} \wedge s_{n+1} = s_n^{\sigma_n}[v_n^{\sigma_n}]_{p_n}.$$

In this definition some subscripts of $\rightarrowtail$ have been left out, which are clear from the context.

3.2 Cascades

In order to express the incremental nature by which input is consumed, or output is produced we introduce the notion of a *cascade* of substitutions which gradually instantiates a set of variables.

A series of substitutions $\sigma_1, \sigma_2, \ldots$ is called an *cascade* if it exhibits the following properties:

1. $\forall i \neq j : Car(\sigma_i) \cap Car(\sigma_j) = \emptyset$.
   Variables are instantiated only once; further instantiation is done by refining nested variables.
2. $\forall i \neq j : Ran(\sigma_i) \cap Ran(\sigma_j) = \emptyset$.
   Variables are introduced only once.
3. $\forall i \leq j : Car(\sigma_i) \cap Ran(\sigma_j) = \emptyset$.
   A variable must be introduced before it is instantiated.

Theorem (1): Let $s_1 \overset{\sigma_1}{\rightarrowtail} s_2 \overset{\sigma_2}{\rightarrowtail} \ldots$ be a most general rewrite sequence, and let τ_i be the substitutions derived from σ_i by disregarding variables in rewrite rules (i.e., $\tau_i(x) = \sigma_i(x)$ for $x \notin R$, and $\tau_i(x) = x$ otherwise). Then $\tau_1, \tau_2, \ldots$ is a cascade. We call this cascade the *trace* of the rewrite sequence.

Proof: Observe that 2 is trivial, because the sequence is most general.

Now we will prove 3. If $x \in Car(\tau_i)$ (and hence $x \in Car(\sigma_i)$), we have $x \in s_i$, because σ_i is *mgu*. Hence, $x \in s_1$ or $x \in Ran(\sigma_k)$ for some $k < i$. But $x \in Ran(\tau_j)$ implies $x \in Ran(\sigma_j)$, and $\sigma_j = mgu_{\cup_{l<j} Ran(\sigma_l) \cup \{s_1\}}(\ldots)$. This is a contradiction.

Finally we will prove 1. Suppose $x \in Car(\tau_i) \cap Car(\tau_j)$ and suppose $i < j$. By (3) we have $x \notin Ran(\sigma_i)$. Hence, $x \notin s_{i+1}$. Also, $x \in Car(\tau_j)$ implies $x \in s_j$. But then it must have been introduced by, say, σ_k for $i < k \leq j$. That is, $x \in Ran(\sigma_k)$. Since $x \in Car(\tau_i)$, and therefore $x \in s_i$ we have $x \in s_1$ or $x \in Ran(\sigma_l)$ for some $l < i$. This is a contradiction.

3.3 Constrained Narrowing

Constrained narrowing differs from ordinary narrowing in its distinction of three types of variables: input, output and ordinary. Ordinary variables may *never* be instantiated during rewriting; output variables may be instantiated when this is required for the application of a rule, and input variables may be instantiated if it is required and if it is in accordance with the input that is provided. In addition, whenever an input or output variable x is instantiated with a term s, the variables that are introduced in s must be of the same type as x.

Firstly, we assume the existence of a function *type* on $\mathcal{V}$ with range $\{o, i, p\}$, for output, input and ordinary (plain) variables, and we say that a unifier σ is *I/O-conformant* (written as $conf(\sigma)$) when

$$conf(\sigma) \stackrel{\text{def}}{\Longleftrightarrow} \forall x \in Car(\sigma) : type(x) \neq p \wedge \forall y \in var(\sigma(x)) : type(x) = type(y).$$

Secondly, actual input acts as a constraint on potential unifiers. We model input as a *cascade* of substitutions (rather than a single substitution), since input may be defined incrementally (e.g., user input). This is a separate aspect from the fact that input is *used* incrementally during rewriting. We require that the variables occurring in carriers of substitutions in this cascade are input variables.

$$\tau_1, \tau_2, \ldots \models \sigma_1, \sigma_2, \ldots \stackrel{\text{def}}{\Longleftrightarrow} \forall n \forall x \in Car(\tau_n) : type(x) = i \wedge \forall n \exists m \forall x \in Car(\bigcirc_{i \leq m} \tau_i) : \bigcirc_{i \leq n} \sigma_i(x) \preceq \bigcirc_{i \leq m} \tau_i(x).$$

We now present the definition of **constrained narrowing**:

$$\begin{array}{l} \tau_1, \tau_2, \ldots \models s_n \overset{\sigma_n}{\succ\!\!\Longrightarrow}_{(s_1, \sigma_1, \ldots, s_{n-1}, \sigma_{n-1})} s_{n+1} \stackrel{\text{def}}{\Longleftrightarrow} \\ \quad conf(\sigma_n) \wedge \\ \quad \tau_1, \tau_2, \ldots \models s_1 \overset{\sigma_1}{\succ\!\!\Longrightarrow} \ldots \overset{\sigma_{n-1}}{\succ\!\!\Longrightarrow} s_n \wedge \\ \quad \exists p_n \in s_n,\ u_n \to v_n \in R, \sigma_n = mgu_{\cup_{i<n} Ran(\sigma_i) \bigcup \{s_1\} \bigcup var(R)}(s_n|_{p_n}, u_n)\ : \\ \qquad s_n|_{p_n} \notin \mathcal{V} \wedge \\ \qquad \tau_1, \tau_2, \ldots \models \sigma_1, \ldots, \sigma_n \wedge \\ \qquad s_{n+1} = s_n^{\sigma_n}[v_n^{\sigma_n}]_{p_n} \end{array}$$

The ordinary term rewriting relation is subsumed by the constrained narrowing relation, which is again subsumed by narrowing. These facts are easily verified.

Input is expressed in the constraint τ. This cascade limits the narrowing relation, but doesn't depend on it. Output occurs when the unifier σ instantiates output variables in the subject sub-term. To be precise, the input consumed, and the output produced by a rewrite sequence are defined as follows:

Output: For every output file ϕ a unique output variable x^ϕ is selected which must occur at least once in the initial query s_1. The rewrite sequence $s_1 \succ\!\!\Longrightarrow s_2 \succ\!\!\Longrightarrow \ldots$ defines a trace $v_1, v_2, \ldots$, which is a cascade. The contents of the output file is defined as $\bigcirc v_i(x^\phi)$.

Observe that this output can be effectuated incrementaly, while narrowing is still in progress, as each substitution in the trace becomes available.

Input: For every input file ψ an input variable x^ψ is chosen which must occur at least once in the initial query s_1. The contents of the file are represented in the constraint $\tau_1, \tau_2, \ldots$ governing the rewrite sequence, with $\bigcirc_{i \leq m} \tau_i(x^\psi)$ approximating the eventual contents for increasing m.

4. Some Properties of Constrained Narrowing

Constrained narrowing is a new calculus in the area of term rewriting and narrowing, which we propose as a suitable alternative for term rewriting as an operational semantics for equational

languages. In this section we discuss a few of its properties, relating to non-determinism and computational suitability. We will not provide extensive proofs.

In this section we abbreviate term rewriting, constrained narrowing and narrowing to TR, CN and NA, respectively.

Property 1: Let $t_1 \rightarrow \ldots \rightarrow t_n$ be a TR sequence, and let t_1 contain no input or output variables. Then $t_1 \succ\!\!\Rightarrow \ldots \succ\!\!\Rightarrow t_n$ is a CN sequence.

No I/O variables can be introduced, so none occur altogether. Then, CN coincides with TR.

Property 2: Let $s_1 \succ\!\!\Rightarrow \ldots \succ\!\!\Rightarrow s_n$ be a CN sequence for empty input $\bigcirc\tau$ (i.e., $var(t) \cap Car(\bigcirc\tau) = \emptyset$), and suppose s_1 contains no output variables. Then $s_1 \rightarrow \ldots \rightarrow s_n$ is a TR sequence.

The only steps possible are proper rewrite steps.

Property 3: Let $s_1 \succ\!\!\Rightarrow \ldots \succ\!\!\Rightarrow s_n$ be a CN sequence for input $\bigcirc\tau$, and suppose s_1 contains no output variables. Let $t_i = \bigcirc\tau(s_i)$. Then $t_1 \rightarrow \ldots \rightarrow t_n$ is a TR sequence.

Narrowing step i implies that a sub-term in s_i can be unified with a rule in accordance with the constraint. In t_i that constraint has been 'filled in', so a match must occur.

Properties 1—3 mean that CN without output is equivalent to TR. Note however, that the practical aspect of interactive I/O is lost in this formal equivalence.

Property 4: Let $s_1 \succ\!\!\Rightarrow \ldots \succ\!\!\Rightarrow s_n$ be a CN sequence for constraint $\bigcirc\tau$ and trace $\bigcirc\sigma$, and let $t_i = \bigcirc\sigma(\bigcirc\tau(s_i))$. Then $t_1 \rightarrow \ldots \rightarrow t_n$ is a TR sequence.

This property is derived in a similar vein as above, and it suggests that CN does not add essentially new reductions. It is not reasonable to say that CN is equivalent in this case, because we use the output that is to be computed; unlike input, it is not available beforehand.

Corrollary 5: These properties characterize infinite CN sequences: they coincide with infinite TR sequences; they process infinite input; or they produce infinite output.

Property 6: In Section 5 we present a transformation of a TRS defining output as a proper function of input to a CNS which produces the intended output under CN. The transformed system is confluent and terminating under CN if the TRS is confluent and terminating, respectively, under TR. In addition, the transformed system has the property of output-confluence, to be introduced now.

4.1 Output Confluence

A comment must be made regarding the desirable property of confluence, which is closely related to non-determinism. Two kinds can be distinguished: confluence and output-confluence. A TRS is *confluent* if, for any s, t_1 and t_2 where (possibly empty) reduction sequences $s \rightarrow \ldots \rightarrow t_1$ and $s \rightarrow \ldots \rightarrow t_2$ exist, there is a u such that (possibly empty) reduction sequences $t_1 \rightarrow \ldots \rightarrow u$ and $t_2 \rightarrow \ldots \rightarrow u$ exist. Confluence guarantees that reachable reducts

remain reachable in light of non-determinism, and, given termination, that normal forms are unique.

The same definition is applicable to CNSs, but it provides no guarantees w.r.t. the output that is produced.

A CNS is *output-confluent* if, for any s, t and distinct narrowing sequences $s \succ\!\!\Longrightarrow_1 \ldots \succ\!\!\Longrightarrow_1 t$ and $s \succ\!\!\Longrightarrow_2 \ldots \succ\!\!\Longrightarrow_2 t$ with traces $\sigma_1^1, \sigma_2^1, \ldots$ and $\sigma_1^2, \sigma_2^2, \ldots$, respectively, we have, for any $x \in s$, that $\bigcirc\sigma^1(x)$ and $\bigcirc\sigma^2(x)$ have a common reduct (i.e., a u exists such that $\bigcirc\sigma^1(x) \succ\!\!\Longrightarrow \ldots \succ\!\!\Longrightarrow u$ and $\bigcirc\sigma^2(x) \succ\!\!\Longrightarrow \ldots \succ\!\!\Longrightarrow u$).

Output confluence guarantees uniquely defined output for CNSs, which is the normal form of the generated output.

Sufficient conditions can be formulated under which confluence and output-confluence are guaranteed, but we will not go into that in this paper. In Section 5 a method is presented to transform conventional TRSs defining output, to CNSs that produce that output under constrained narrowing. The transformation is such that a confluent TRS is transformed to a confluent, output-confluent CNS.

5. Transforming Cnventional TRSs

This paper would be incomplete without a constructive description of how a CNS with a required output behavior is obtained. Note, however, that we will not go into the proofs, merely sketching the method. Also note that we do not suggest all CNSs can or should be constructed in this manner; we merely acknowledge that writing TRSs and writing CNSs are different skills.

We assume that the output can be specified by an ordinary TRS as a function of the input, and without loss of generality we consider a function f, taking a single (input) argument, and yielding the intended output. That is, given an input s, the intended output is the normal form of $f(s)$.

We will describe how an ordinary TRS defining f can be transformed into a CNS which produces that output under constrained narrowing. This method is very similar to the definition of functions as relations in logic programming languages.

Let $\tilde{f}$ be a function with arity 2, and let $f(u) \to v$ be one of the rules for f. If v does not have a (recursive) occurrence of f, then define the following rule: $\tilde{f}(u, v) \to \mathit{dummy}$ (where *dummy* is an auxiliary free constant). If v has one recursive occurrence of f, say $v|_p = f(u')$, then define the rule $\tilde{f}(u, v[x]_p) \to \tilde{f}(u', x)$, where x is a fresh variable.

If more than one recursive occurrence of f exists, we use an auxiliary function *also*, which is governed by the rule $\mathit{also}(\mathit{dummy}, \mathit{dummy}) \to \mathit{dummy}$. Suppose v has two occurrences of f (say $v|_{p_i} = f(u_i)$ for $i \in \{1, 2\}$). If the two occurrences are independent (i.e., $p_1 \not\leq p_2 \wedge p_2 \not\leq p_1$), we define $\tilde{f}(u, v[x_1]_{p_1}[x_2]_{p_2}) \to \mathit{also}(\tilde{f}(u_1, x_1), \tilde{f}(u_2, x_2))$. Otherwise, say if $p_1 \leq p_2$, and let p_3 be such that $p_1 \cdot 1 \cdot p_3 = p_2$ (1 is the index of f's sole argument), we define $\tilde{f}(u, v[x_1]_{p_1}) \to \mathit{also}(\tilde{f}(u_2, x_2), \tilde{f}((v|_{p_1 \cdot 1})[x_2]_{p_3}, x_1))$

This can be extended for more recursive occurrences.

This transformation results in the definition of a function $\tilde{f}$, such that the normal form of $\tilde{f}(s, y)$ (where y is an output variable) under constrained narrowing is *dummy*, whilst the trace reflects that $y = f(s)$. This is easily verified.

Theorem (2): If the TRS defining f is confluent then the CNS obtained from this transformation is confluent and output-confluent.

We will not discuss the proof of this theorem.

6. Implementation

The implementation of constrained narrowing is straightforward. It is based on a special action that is triggered when a variable is encountered in a subject term.

Consider the recursive algorithm which matches a subject term s against the left-hand side of a rule u, and suppose a variable is encountered in the subject term (i.e., $s \in \mathcal{V}$ for some recursive instance). If $type(s) = i$, u is matched against the currently available input. If this match fails (i.e., the constraint differs, or is not yet available) then the entire match fails. If it succeeds, the variables in the current instance of u are replaced by fresh input variables and the result is bound to s. Binding means global replacement, which can efficiently be implemented by graph rewriting, such that variables in subject-terms are represented as (exhaustively) shared sub-terms. Then, binding involves a single *in-situ* replacement.

If $type(s) = o$, the match succeeds implicitly, and that variable is simply bound to a freshly renamed version of u.

If $type(s) = p$ the match succeeds iff u is identical to s.

Observe that the rule isn't applied if input is not yet available. A possible mechanism is that execution is suspended until input becomes available, but in the context of complete, output confluent rewrite systems, it may be preferable to continue normalization. This produces the same result in another manner, or leads to an open term which can be regarded as a weak normal form representing the partially evaluated subject term. As soon as input becomes available the input variables can be instantiated, and execution can continue.

Note that we do not require output to be in normal form. If this is needed, then the output routines must normalize terms bound to output variables, before the output is actualized.

7. Related Work

We will first discuss the origins of our notion of constrained narrowing, and indicate the few details in which we differ from other definitions of narrowing and rewriting. Then, we will discuss (concurrent) logic programming languages and functional programming languages. Finally, we will discuss existing techniques for reconciling *pure* functional programming languages with I/O.

In most texts and supporting software environments term rewriting without narrowing is considered. That is, variables occurring in the subject term are left unchanged. In [Klo92] the notion of narrowing is discussed, although not specifically as an extension of term rewriting, and certainly not to handle I/O.

The notion of constrained narrowing has to our knowledge not been defined elsewhere.

Implementations of conventional logic programming languages such as Prolog, perform I/O by 'calling' so-called *extra-logical* predicates which have side-effects that cannot be undone during backtracking.

In *concurrent logic programming* [Sha89], logical variables are used in a similar way as in

our model. The semantics of these languages requires that results of failing (failing in the sense of Prolog) computations are observable, and I/O is meaningful, regardless whether the computation is successful or not. Our work can be seen as an investigation of these ideas in the context of *equational* logic, instead of *predicate* logic which is fundamental in (concurrent) Prolog.

Functional programming is very much related to equational programming. There are several combinations of logical and functional programming [Han94, Red85]. A good account of logical variables in the context of functional programming was given by Reddy [Red85]. It is interesting to note that most of the combinations studied in these references assume backtracking (don't know non-determinism) in their implementations, instead of don't care non-determinism.

In pure functional languages, several proposals have been made for clean I/O models. We mention Landin's streams [Lan65], Monads [Mog89, JW93], and Uniqueness Types [AvGP92]. Compared to our approach, these models are more deterministic, especially with respect to output. More generally, these models sequentialize I/O actions by imposing static constraints (usually expressed as types) on the program.

8. Conclusions

We have provided a model for I/O in equational languages with don't care non-determinism. The operational semantics are intuitively appealing; are independent of any reduction strategy; and allow for don't-care non-determinism. An implementation based on conventional rewriting is straightforward, and can be highly efficient [HF+96]. The model concerns I/O of arbitrary terms, and does not focus on a particular file structure.

The model is formulated as a calculus combining the good computational properties of term rewriting with the needed additional expressiveness of narrowing. Sufficient conditions for confluence and termination exist (but have not been discussed). We have sketched how constrained narrowing systems can be easily obtained from ordinary rewrite systems defining output as a function of input, and how confluence of such a TRS implies confluence and output confluence of the result.

References

[AvGP92] P.M. Achten, J.H.G. van Groningen, and M.J. Plasmeijer. High-level specification of i/o in functional languages. In John Launchbury, editor, *Proceedings Glasgow Workshop on Functional Programming*. Springer-Verlag, 1992.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[DJ90] N. Dershowitz and J.-P Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol B.*, pages 243–320. Elsevier Science Publishers, 1990.

[GKK+88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.

[Han94] Michael Hanus. Combining lazy narrowing and simplification. In *Proceedings of*

the International Conference on Programming Language Implementation and Logic Programming, PLILP '94, 1994.

[HF+96] Pieter H. Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 1996. Accepted for publication.

[Hul80] J.M. Hullot. Canonical forms and unification. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, 1980.

[JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles Of Programming Languages (POPL)*, pages 71–84, 1993.

[KKM88] C. Kirchner, H. Kirchner, and J. Meseguer. Operational Semantics of OBJ-3. In T. Lepistö and A. Salomaa, editors, *Proceedings of the Fifteenth International Conference on Automata, Languages and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 287–301, 1988.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[Lan65] P.J. Landin. A correspondence between algol 60 and church's lambda-notation: Parts i and ii. *Communications of the ACM*, 8(2,3):89–101,158–165, February and March 1965.

[Mog89] E. Moggi. Computational lambda calculus and monads. In *Logic in Computer Science*. IEEE, 1989.

[MOI95] Aart Middeldorp, Satoshi Okui, and Tesuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. In *Proceedings of the 20th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[Red85] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 1387–151, 1985.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. Technical Report CS89-08, Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, May 1989.