ISRN SICS-R–95/07–SE

Computing in Unpredictable Environments: Semantics, Reduction Strategies, and Program Transformations

Björn Lisper

SICS research report
R95:07
ISSN 0283-3638

R95:07 0283-3638

Computing in Unpredictable Environments: Semantics, Reduction Strategies, and Program Transformations

by

Björn Lisper
Formal Design Techniques; October 4, 1995
lisper@it.kth.se

Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA
Sweden

Abstract

We study systems where deterministic computations take place in environments which may behave nondeterministically. We give a simple formalization by unions of abstract reduction systems, on which various semantics can be based in a straightforward manner. We then prove that under a simple condition on the reduction systems, the following holds: reduction strategies which are *cofinal* for the deterministic reduction system will implement the semantics for the combined system, provided the environment behaves in a "fair" manner, and certain program transformations, such as *folding* and *unfolding*, will preserve the semantics. An application is evaluation strategies and program transformations for concurrent languages.

Keywords: formal semantics, program transformations, nondeterminism, reduction systems, recursive program schemes.

Submitted to CAAP'96. This work was done under the support of the ESPRIT BRA project CONFER, project no. 6454.

1 Introduction

Computer programs can often be seen as having two parts: a *computational* component, describing what kind of computations will be carried out, given appropriate inputs, and a *descriptive* component, that models the environment in which the program will execute. A typical example is concurrent programs with primitives for asynchronous process communication. Here, the code for each process can often be seen as a more or less purely functional i/o-specification: given that the process recieves some values on some input channels, it will produce computed values on some output channels. This can be seen as the computational component of the program. The semantics of the communication primitives, on the other hand, describes environmental properties such as asynchrony, e.g., some communication events may take place in different order. The latter can give rise to *nondeterminism*, that is: the environment may have a certain freedom to behave in different ways (like writes to a communication channel occurring in different order), and this freedom may cause the system as a whole to behave nondeterministically.

Some programs have a more or less empty descriptive component. A prime example is a purely functional program. Purely functional programming is highly appropriate for describing purely computational tasks (as long as explicit resource handling for efficiency is not a concern, anyway). The simple semantics makes it particularly simple to analyze and transform functional programs: see, for instance, [15, 26].

There are many situations, however, where the descriptive component cannot be neglected. Systems for control of finite resources (such as various servers), operating systems, embedded systems – they all rely on the ability to model the environment where the specified computations are to take place.

It is well known that the presence of nondeterminism can make "evident" program transformations incorrect, in the sense that the set of possible outcomes can be changed (rather than just affecting the termination behaviour, as in the deterministic case). Also, the evaluation strategy can affect this set. Consider the choice operator "*or*", defined by $x\ or\ y \to x$, $x\ or\ y \to y$, and the function definition $f(x) = x - x$. If our language has call-by-value semantics, then $f(0\ or\ 1)$ can evaluate only to 0. But with call-by-name semantics, $f(0\ or\ 1)$ can evaluate also to 1 and -1. *Unfolding* the call $f(0\ or\ 1)$, i.e. replacing it with $(0\ or\ 1)-(0\ or\ 1)$, yields the same possible outcomes with call-by-name semantics but adds some if call-by-value applies. Consider, finally, the algebraic "simplification rule" $x - x \to 0$. If this rule is applied to the body of $f(x)$, then $f(0\ or\ 1)$ can evaluate only to 0 with both call-by-value and call-by-name, so with call-by-name semantics, some possible outcomes have disappeared.

The purpose of our work is to provide a framework in which to reason about languages with both computational and descriptive components. The setting is abstract: the computational part of a system is given by a confluent abstract reduction system $\to_D$, and the descriptive part by a possibly non-confluent (and thus non-deterministic) system $\to_N$. The system itself is described by the union of these reduction systems. A semantics is defined as the set of limits, under some monotone interpretation to a c.p.o., for the various (possibly infinite) reduction paths, in the vein of Boudol [6]. This can be thought of as a "normal form semantics", but extended to deal with infinite and divergent (but "fair") computations. In particular, we consider reduction systems over terms, with interpretation to c.p.o.'s of trees over the same signature. Within this framework, we prove the following:

1. If $\to_D$ and $\to_N$ commute, then for any cofinal reduction strategy F for $\to_D$ there exists a semantically correct, nondeterministic (w.r.t. $\to_N$) reduction strategy, for any monotone interpretation.

2. If $\rightarrow_D$ and $\rightarrow_N$ commute, then certain transformations preserve the semantics, for any monotone interpretation.

"Semantically correct" here means to have the same set of "normal forms" as $\rightarrow_D \cup \rightarrow_N$. Thus, there is always a deterministic "implementation" of the computational part that "preserves all possibilities" prescribed by the semantics, i.e., does not destroy the description of the environment.

The semantics-preserving transformations include symbolic folding and unfolding of recursive programs defined by term rewriting rules.

Our choice to consider "semantical correctness" or "preservation of semantics" as "preserving all normal forms" is because we use nondeterminism to model possible actions of an environment which can be described but not controlled. This should be contrasted with "don't care" nondeterminism which gives a looser, relational specification of the computational component, where a number of outcomes are allowed for a given input and the system is free to choose.

Due to space limitations, proofs are omitted or sketched in this paper.

2 Related Work

Our work is most directly applicable to recursively defined programs (which often can be modeled as rewriting systems) and is thus strongly related to the theory of recursive applicative program schemes [10, 19]. Our results can be seen as generalizations of some classical results for deterministic such schemes [9] to a class of nondeterministic ones, although our basic results are formulated in the more abstract setting of abstract reduction systems. The semantics we use is related to the one developed by Arnold, Naudin and Nivat for nondeterministic recursive schemes [4] and in particular to the one of Boudol [6] for first order term rewriting systems. See also the interesting discussion of referential transparency and unfoldability in [24]; one can say that our commutation condition on the computational and descriptive reduction systems *preserves referential transparency for the computational system also in the presence of a non-deterministic environment.*

Our framework is abstract and thus applicable to a range of languages, but the ones we have mainly in mind are recursive languages with concurrency constructs. Some existing, not referentially transparent languages in this class are Concurrent ML [22], Facile [25], and Erlang [3]. A goal of our work is to support the design of recursive languages with concurrency, where the "serial part" is referentially transparent also in the precence of nondeterminism arising from the concurrency.

Our results about folding and unfolding extend classical results [8, 10, 18] for deterministic systems. They are applicable to program transformation systems for program optimization [7, 13], and partial evaluation of nondeterministic languages. Semantics-preserving partial evaluation of such languages is listed as the "challenging problem no. 10.9" in [14]. The only partial evaluator for such languages that we are aware of is for the concurrent constraint programming language AKL [23]. But this partial evaluator gives correct results only under a number of restrictions on the program and assumptions about the intended semantics. We believe that our results can support the design of nondeterministic languages where it is more evident what "semantically correct" partial evaluation is.

3 Preliminaries

In this paper, we will use *reduction systems* [12, 17] (sometimes called *abstract reduction systems*) to give semantics of computations. We give the definitions of the most central concepts being used here.

Let $\rightarrow$, $\rightarrow_1$, $\rightarrow_2$ be binary relations over some set A. Then $\rightarrow_1$ and $\rightarrow_2$ *commute* if $\forall a, b, c \exists d[(a \rightarrow_1^* b \wedge a \rightarrow_2^* c) \implies (b \rightarrow_2^* d \wedge c \rightarrow_1^* d)]$. ($\rightarrow^*$ denotes the transitive-reflexive closure of $\rightarrow$.) $\rightarrow$ is *confluent* if it self-commutes. A $\rightarrow$-*nf* (normal form of $\rightarrow$) is an element a where there is no a' such that $a \rightarrow a'$.

A (many-step) *abstract reduction strategy* w.r.t. $\rightarrow$ is a function F such that $F(a) = a$ if a is a normal form, and $a \rightarrow^+ F(a)$ otherwise. ($\rightarrow^+$ denotes the transitive closure of $\rightarrow$.) F is *normalizing* if, whenever a has a normal form a', it holds that $F^n(a) = a'$ for some n. F is *cofinal* if for all a, a' where $a \rightarrow^* a'$ there exists an n such that $a' \rightarrow^* F^n(a)$.

We are especially interested in reduction systems over terms. We will call such systems "term reduction systems" (of which term rewriting systems is a special case, and also systems like λ-calculus with β-reduction). We denote the set of the finite terms under discourse with T, and the set of finite and infinite terms with T^∞. Formally, a term t can be defined as a partial function from the set of sequences of natural numbers ("positions") to operator symbols, such that $dom(t)$ is prefix-closed and respects arities; see, e.g. [4, 11]. For $p \notin dom(t)$, we define $t(p) = \bot$.

We can relax the conditions that arities are respected and allow also terms which are undefined at leaf positions, and we denote the set of these terms by $T_\bot^\infty$. We have $T \subset T^\infty \subset T_\bot^\infty$. If T is the set of λ-terms, then $T_\bot^\infty$ are the *Böhm trees* [5, Ch. 10].

For the operator symbols, we define a flat partial order $\sqsubseteq$ by $\bot \sqsubseteq s$ and $s \sqsubseteq s$ for all symbols s. We lift this order to terms (i.e. functions from sequences) in $T_\bot^\infty$ by $t \sqsubseteq t'$ iff $t(p) \sqsubseteq t'(p)$ for all sequences p. It is straightforward to show that every ascending chain $t_0 \sqsubseteq t_1 \sqsubseteq \cdots$ in $T_\bot^\infty$ has a least upper bound in $T_\bot^\infty$. The maximal elements are exactly those in T^∞. Thus, if S is a set of operator symbols with arities, $\bot$ is nullary, and $T_\bot^\infty$ are the terms formed according to the arity constraints, then $\langle T_\bot^\infty, \sqsubseteq, \bot, S\rangle$ is a free continuous S-algebra [11].

Two positions are *disjoint* if none is a prefix of the other. For any term $t \in T_\bot^\infty$, and position $p \in dom(t)$, we denote the subterm at p by t/p and the term obtained by replacing t/p by t' in t by $t[p \leftarrow t']$. A position p such that t/p matches a rule in a term rewriting system is a *redex* for that rule. If t is rewritten into t' (at some other position), then the positions in t' to where p is sent are called the *residuals* of p.

4 Semantics for Abstract Computations

Computations can be modeled by reduction sequences in some abstract reduction system $\langle A, \rightarrow\rangle$:

Definition 1 *A* $\rightarrow$-computation *in A is an infinite sequence of elements $\vec{a} = \{\vec{a}_i\}_0^\infty$ in A such that:*

- *If $\vec{a}_i$ is a $\rightarrow$-nf, then $\vec{a}_i = \vec{a}_{i+1}$.*
- *Otherwise, $\vec{a}_i \rightarrow \vec{a}_{i+1}$.*

We say that $\vec{a}$ is a-rooted *if $\vec{a}_0 = a$.*

Sometimes we will abuse notation and write $\vec{a}$ even when considered as a relation, i.e. $\bigcup_{i \in \mathbf{N}}\{(\vec{a}_i, \vec{a}_{i+1})\}$.

Consider an ARS $\langle A, \rightarrow\rangle$, a c.p.o. $\langle C, \sqsubseteq\rangle$, and a mapping $f\colon A \rightarrow C$ which is monotone w.r.t. $\rightarrow$ and $\sqsubseteq$. Cf. [19]. f is then a *monotone interpretation* of $\langle A, \rightarrow\rangle$ into $\langle C, \sqsubseteq\rangle$, and $\sqsubseteq$ models the increase of information as a computation proceeds along $\rightarrow$. Any $\rightarrow$-computation $\vec{a}$ yields a l.u.b. $\bigsqcup_{i=0}^\infty f(\vec{a}_i)$ in C, denoted $\bigsqcup f(\vec{a})$, which can be seen as the result of the computation.

Definition 2 *$\vec{a}$ is* majorated *by $\vec{a'}$ iff $\forall i \exists j : \vec{a}_i \to^* \vec{a'}_j$. It is* strictly majorated *by $\vec{a'}$ iff it is majorated by $\vec{a'}$ and $\bigsqcup f(\vec{a}) \sqsubset \bigsqcup f(\vec{a'})$.*

Strict majoration can be seen as an abstract "unfairness" condition for the strictly majorated computation: essentially, such a computation always has some information-increasing path which is left unexplored.

Proposition 1 *If $\vec{a}$ is majorated by $\vec{a'}$, then $\bigsqcup f(\vec{a}) \sqsubseteq \bigsqcup f(\vec{a'})$.*

Proposition 2 *(Strict) majoration is transitive.*

Proposition 3 *The a-rooted $\vec{a}$ is (strictly) majorated by some $\vec{a'}$ iff is (strictly) majorated by some a-rooted $\vec{a''}$.*

We can now define the semantics for an element $a \in A$, given a monotone interpretation f of $\langle A, \to \rangle$ into $\langle C, \sqsubseteq \rangle$ (cf. [6, Ch. 4.2]):

Definition 3 *The* semantics *of $a \in A$ w.r.t. f and $\to$, $S(a, f, \to)$, is the set $\{ \bigsqcup f(\vec{a}) \mid \vec{a}$ is a-rooted and not strictly majorated by any computation$\}$.*

Definition 3 yields a semantics that takes only "fair" computations into account. Computations that "diverge" (i.e. their limits are not maximal w.r.t. $\sqsubseteq$) may contribute, but only if there is no "better" majorating computation starting in a. The limits of the computations that do contribute can be thought of as "infinite normal forms" since they represent cases where no more information can be gained.

5 Semantics for Computations on Terms

Consider a term reduction system $\langle T, \to \rangle$. The following monotone interpretation of $\langle T, \to \rangle$ into $\langle T^\infty_\perp, \sqsubseteq \rangle$ is a natural generalization of classical interpretations:

Definition 4 *$f_\to : T \to T^\infty_\perp$ is defined by: for all positions p, $f_\to(t)(p) = t(p)$ if, for all t' such that $t \to^* t'$, holds that $t'(p) = t(p)$, and if $p = uv$ then $t'(u) = t(u)$. Otherwise $p \notin dom(f_\to(t))$.*

It is easy to verify that indeed $f_\to(t) \in T^\infty_\perp$ for all $t \in T$. $f_\to$ extends the interpretation of Nivat [19] for recursive applicative program schemes to general term reduction systems.

Proposition 4 *$f_\to$ is monotone w.r.t. $\to$ and $\sqsubseteq$.*

Thus, we can define a semantics $S(t, f_\to, \to)$ w.r.t. $f_\to$ for terms t in T. If $S(t, f_\to, \to) \subseteq T$, then all possible computations starting from t are terminating. If $S(t, f_\to, \to) \subseteq T^\infty$, then we may call them convergent. If, finally, there is some s in $S(t, f_\to, \to)$ not in T^∞, then we can say that there is some divergent computation from t.

If $T(F, X)$ is the set of pure λ-terms and if $\to_\beta$ is the reduction relation given by β-reduction to head normal form, then the single element of $S(t, f_{\to_\beta}, \to_\beta)$ is essentially the Böhm tree for t [5]. If $\to$ is given by a first order term rewriting system, then $S(t, f_\to, \to)$ coincides with the semantics for such systems by Boudol [6, Ch. 4.2].

$$\begin{tikzcd} & a \arrow[dl, "D^*"'] \arrow[dr, "N^*"] & \\ a' \arrow[dr, "N^*"'] & & a'' \arrow[dl, "D^*"] \\ & \exists a''' & \end{tikzcd}$$

Figure 1: $\rightarrow_D$ and $\rightarrow_N$ commute.

6 Modeling Systems with Computational and Descriptive Components

In the setting of abstract reduction systems, we can model computational and descriptive components simply as reduction systems over a set A, and a system comprised of such components as the union of the respective reduction systems. We denote the computational reduction relation by $\rightarrow_D$ and the descriptive relation by $\rightarrow_N$. Thus, for any $a \in A$, there are a number of "enabled computations" $a \rightarrow_D a'$ and "enabled actions of the environment" $a \rightarrow_N a'$. The former are under control and may be implemented by a reduction strategy, see Sect. 7.

We consider only the case where the computational component is deterministic (that is: we rule out dont-care nondeterminism). Thus, we will always assume that $\rightarrow_D$ is confluent. $\rightarrow_N$, on the other hand, can model nondeterministic behaviour and is then nonconfluent.

Our results below all hold under the fundamental condition that *$\rightarrow_D$ and $\rightarrow_N$ commute*, i.e. the diagram in figure 1 is valid. Some hold under the weaker condition that $\rightarrow_D$ and $\rightarrow_D \cup \rightarrow_N$ commute. We have:

Proposition 5 *If $\rightarrow_D$ and $\rightarrow_N$ commute and if $\rightarrow_D$ is confluent, then $\rightarrow_D$ and $\rightarrow_D \cup \rightarrow_N$ commute.*

Proof. Simple diagram chase. ■

For term reduction systems, commutation can often be established by applying purely syntactical conditions. Two first order term rewriting systems are *mutually orthogonal* if all their rules are left-linear and no left-hand side in one system overlaps any left-hand side in the other system. The following result is due to Raoult and Vuillemin [21, Proposition 10]:

Theorem 1 (Raoult and Vuillemin) *If the term rewriting systems R and R' are mutually orthogonal, then $\rightarrow_R$ and $\rightarrow_{R'}$ commute.*

The concept of orthogonality can be extended to higher-order rewriting formalisms, in particular Klop's *Combinatory Reduction Systems* (CRS) [16]. CRS'es include first order rewriting, systems with bound variables such as various λ-calculi and certain process communication primitives (see Sect. 10), and combinations thereof. They therefore provide a suitable formalism for recursive languages with higher-order constructs. In practice, recursive languages can very often be described by combined CRS'es where the computational component is orthogonal (to itself) and also mutually orthogonal to the descriptive component. For instance, the computational component can be PCF-like [20], as the sum of λ-calculus, first order term rewriting rules for "base operations" such as arithmetical operations, conditionals etc., and recursive definitions of the form $f(x_1, \ldots, x_n) \to t$ seen as rewrite rules. Interestingly, the proof of Theorem 1 carries over verbatim to mutually orthogonal CRS'es.

7 Reduction Strategies

Definition 5 *A* nondeterministic reduction strategy w.r.t. $\to$ *is a set of computations w.r.t.* $\to$*, such that there is at least one a-rooted computation for each element* a*.*

Definition 6 *The* semantics *of the nondeterministic reduction strategy* $\vec{A}$ *for* a *(under the monotone interpretation* f*) is*
$\overline{S}(a, f, \vec{A}) = \{ \bigsqcup f(\vec{a}) \mid \vec{a} \in \vec{A}$ *and is* a*-rooted*$\}$.

In contrast to $S(a, f, \to)$, $\overline{S}(a, f, \vec{A})$ contains also the results of possible nonfair computations. It is more appropriate to define the semantics of a reduction strategy in this way since a reduction strategy indeed can be nonfair (in the abstract sense here).

Definition 7 $\vec{A}$ *is* semantically correct *w.r.t.* $\to$ *iff* $\overline{S}(a, f, \vec{A}) = S(a, f, \to)$ *for all* a*.*

Proposition 6 *For any* $\to$ *and monotone interpretation* f *there is a semantically correct nondeterministic reduction strategy.*

Proof. For any a and $s \in S(a, f, \to)$, pick an a-rooted $\to$-computation $\vec{a}_s$ such that $\bigsqcup f(\vec{a}_s) = s$. This yields $\bigcup(\vec{a}_s \mid a \in A, s \in S(a, f, \to))$. ∎

For systems $\to_D \cup \to_N$, where $\to_D$ describes a deterministic computational component, we can define:

Definition 8 $\vec{A}$ *is* $\to_D$-deterministic *if, for all* a*, there exists at most one* $\vec{a} \in \vec{A}$ *such that, for some* i*,* $a = \vec{a}_i$ *and* $\vec{a}_i \to_D \vec{a}_{i+1}$*.*

We now have the following central result:

Theorem 2 *If* $\to_D$ *and* $\to_N$ *commute and if* $\to_D$ *is confluent, then, for any cofinal reduction strategy* F *for* $\to_D$*, there exists a* $\to_D$*-deterministic, semantically correct nondeterministic reduction strategy* $\vec{F}$*, where for each* $\vec{a} \in \vec{F}$ *and* $i \in \mathbf{N}$ *holds that* $\vec{a}_{i+1} = F(\vec{a}_i)$ *or* $\vec{a}_i \to_N \vec{a}_{i+1}$*.*

Proof. Pick any semantically correct strategy $\vec{A}$. For any $\to_D \cup \to_N$-computation in $\vec{A}$, a majorating $F \cup \to_N$-computation can be constructed (induction proof with diagram chase according to figure 2). The set of these computations yields the desired nondeterministic reduction strategy. ∎

When is a deterministic reduction strategy cofinal? For orthogonal CRS'es, the following result by Klop [16] applies. A reduction strategy F is *fair* (or *secured*) if, for any term t, there exists an n such that $F^n(t)$ does not contain any residual of any redex in t.

Theorem 3 (Klop) *For orthogonal CRS'es, any fair reduction strategy is cofinal.*

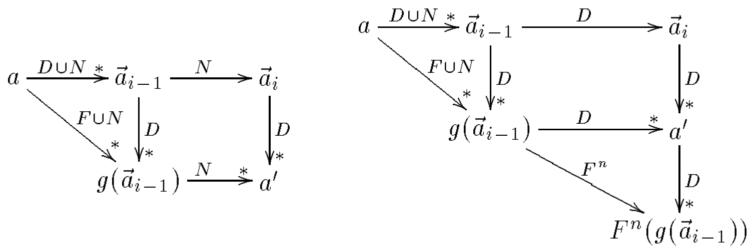

Figure 2: Diagram chase to prove Theorem 2.

8 Referential Transparency

We will now prove a result about "referential transparency", i.e. under which circumstances one can "replace equals for equals" and still have the same meaning, in a possibly nondeterministic context. First, some technical lemmata:

Lemma 1 *If $a \to^* a'$, and if for any $s \in S(a, f, \to)$ there is an a-rooted computation $\vec{a}$ such that $\bigsqcup f(\vec{a}) = s$ and an a'-rooted computation majorating $\vec{a}$, then $S(a', f, \to) = S(a, f, \to)$.*

Lemma 2 *If $\to_D \subseteq \to^*$, if $\to_D$ commutes with $\to$, and if $a \to_D^* a'$, then any a-rooted computation is majorated by some a'-rooted computation.*

Proof. A simple diagram chase, using the commutation of $\to_D$ and $\to$. ■

The following corollary to Lemma 1 and 2 is a major stepping stone:

Corollary 1 *If $\to_D \subseteq \to^*$, if $\to_D$ commutes with $\to$, and if $a \to_D^* a'$, then $S(a', f, \to) = S(a, f, \to)$.*

By Proposition 5, Corollary 1 holds for $\to_D \cup \to_N$-computations when $\to_D$ and $\to_N$ commute. We now develop a class of equivalence relations.

Definition 9 *Let Θ be a set of functions $A \to A$. Then $a \equiv_\Theta a'$ iff $S(\theta(a), f, \to) = S(\theta(a'), f, \to)$ for all $\theta \in \Theta$.*

Typically, A will be a set of terms and Θ will be some set of substitutions that instantiates all free variables. Now, let $\leftrightarrow_D^*$ denote the transitive-reflexive-symmetric closure of $\to_D$. Define $a \equiv_{D\Theta} a'$ iff $\theta(a) \leftrightarrow_D^* \theta(a')$ for all $\theta \in \Theta$.

Theorem 4 *If $\to_D \subseteq \to^*$ and if $\to_D$ commutes with $\to$, then, for any Θ, $a \equiv_{D\Theta} a' \implies a \equiv_\Theta a'$.*

Proof. $x \leftrightarrow_D^* x' \implies S(x, f, \to) = S(x', f, \to)$ is proved by simple induction over $\leftrightarrow_D^n$ using Corollary 1. Instantiating $x = \theta a$, $x' = \theta a'$ gives the result. ■

We can relate Theorem 4 to algebraic formulations of equivalences in the following way. Let $\to_D$, $\to_N$ be term reduction relations. Assume that for a subset T_A of the terms, each element $t \in T_A$ has an interpretation $i(t)$ as an element in some algebra with carrier A, and that $\to_D$ rewrites each $t \in T_A$ into $i(t)$. Then $t \leftrightarrow_D^* t' \iff i(t) = i(t')$. For terms t, t' with free variables in X, where $\theta t, \theta t' \in T'$ for all possible substitutions $\theta: X \to A$, an algebraic equivalence $t =_E t'$ holds if

$i(\theta t) = i(\theta t')$ for all these θ. Thus, if Θ denotes the set of such substitutions, then $t =_E t' \implies t \equiv_{D\Theta} t'$.

Now assume that $\rightarrow_D$ is closed under replacement. Then $t \equiv_{D\Theta} t' \implies t \equiv_{D\Theta\cup C} t'$, where C is the set of all replacements in any possible context, even nondeterministic ones. By Theorem 4 $t \equiv_{D\Theta\cup C} t'$ yields $t \equiv_{\Theta\cup C} t'$. This means the following: *a subterm, that only can be instantiated to a deterministic term, can be transformed according to any algebraic equivalence without altering the nondeterministic semantics for the whole expression.* For instance, if the equation $x - x = 0$ is valid, then a subterm $x - x$ can be transformed to 0 at compile time *provided that x will always be instantiated to something that can be interpreted in an algebra where this equation is valid.*

What about situations when we do not have that knowledge? Theorem 4 has the following corollary:

Corollary 2 *If $\rightarrow_D \subseteq \rightarrow^*$, if $\rightarrow_D$ commutes with $\rightarrow$, if $\rightarrow_D$ is closed under θ, and if $a \leftrightarrow_D^* a'$, then $S(\theta(a), f, \rightarrow) = S(\theta(a'), f, \rightarrow)$.*

If $\rightarrow_D$ is closed under substitution and replacement, then $\leftrightarrow_D^*$ is an equality relation on the whole set of terms. Corollary 2 then says that we can replace "equals for equals" without changing the semantics, even for nondeterministic terms. This is our notion of "referential transparency". Note, though, that $\leftrightarrow_D^*$ might not contain all equivalences valid for the purely deterministic part: in particular, the commutation with $\rightarrow_N$ must not be violated. As an example, consider a system where $\rightarrow_N$ is given by the term rewriting rules $x \ or\ y \rightarrow x$, $x \ or\ y \rightarrow y$. Consider again the term $x - x$. If this term is rewritten to 0, and we instantiate $x = 0 \ or\ 1$, then the normal forms 1 and -1 of $(0 \ or\ 1) - (0 \ or\ 1)$ are lost. This is due to the non-commutation of the rule $x - x \rightarrow 0$ with the rules for or.

9 Fold/unfold Transformations

So far we have considered transformations of terms, given a term reduction system. But what if the term reduction system itself is transformed? If recursive definitions are expressed as, say, first order term rewriting rules within a CRS, then program transformations are really transformations of the rules rather than the terms they compute on, and a transformation of the CRS R into R' will in general yield a reduction relation $\rightarrow_{R'} \neq \rightarrow_R$. For deterministic recursive applicative program schemes there is a classical theory for fold/unfold-transformations, see, e.g., [10, 18]. The results in this section can be seen as generalizations of some classical results to the nondeterministic case.

For simplicity we restrict the discussion below to pure term rewriting systems (TRS) without bound variables. Such a system R has rules of the form $t \rightarrow s$ where t and s are formed from (nullary) variables and constant function symbols of fixed arity. If $t \rightarrow s \in R$ and if $t'/p = \sigma t$ for some position p and substitution σ, then $t' \rightarrow_R t'[p \leftarrow \sigma s]$.

Definition 10 *The reduction relations $\rightarrow$ and $\rightarrow'$ are* equivalent *iff $S(t, f_\rightarrow, \rightarrow) = S(t, f_{\rightarrow'}, \rightarrow')$ for all terms t.*

We say that the TRS'es R and R' are equivalent if their reduction relations are.

Unfolding means to apply a rule $t' \rightarrow s'$ in a redex p in the RHS of the first-order term rewriting rule $t \rightarrow s$, where $s/p = \theta t'$ for some substitution θ, such that a new rule $t \rightarrow s[p \leftarrow \theta s']$ is formed. A TRS R containing these rules can now be transformed into a new TRS R' by replacing $t \rightarrow s$ with the new rule.

$$
\begin{tikzcd}
u \arrow[rr, "D'\cup N"] \arrow[d, leftrightarrow, "="] & & u[p' \leftarrow \phi(s[p \leftarrow \theta s'])] \arrow[d, leftrightarrow, "="] \\
u \arrow[r, "D\cup N"] & u[p' \leftarrow \phi s] \arrow[r, "D\cup N"] & u[p' \leftarrow (\phi s)[p \leftarrow \phi\theta s']]
\end{tikzcd}
$$

Figure 3: Emulation of a t-reduction in $D' \cup N$ by a t- and t'-reduction in $D \cup N$.

$$
\begin{tikzcd}
u \arrow[r, "D"] & u[p' \leftarrow \phi s] \arrow[rr, "X"] \arrow[dr, "D"] & & t' \arrow[dr, "D", "*" near end] & \\
& & u[p' \leftarrow (\phi s)[p \leftarrow \phi\theta s']] \arrow[rr, "X"] & & t''
\end{tikzcd}
$$

Figure 4: Majoration of a $\rightarrow_{D\cup N}$-computation by a computation that emulates a $\rightarrow_{D'\cup N}$-computation. $X = D$ or N.

Folding simply means matching a rule $t' \rightarrow s'$ "backwards" in another term rewriting rule $t \rightarrow s$, such that $s/p = \theta s'$, and rewriting this rule into $t \rightarrow s[p \leftarrow \theta t']$.

Now, let D and N be left-linear TRS'es, orthogonal to each other, where D is orthogonal to itself (so D is confluent, and $\rightarrow_D$ and $\rightarrow_N$ commute). Our results below then say that fold/unfold-transformations of D into D' yield a TRS $D' \cup N$ which is equivalent to $D \cup N$ under essentially the same conditions as for classical, deterministic recursive schemes. For simplicity, we state the results for the case where a single rule is folded/unfolded by a single rule.

Theorem 5 *If $t \rightarrow s$ in D is unfolded into $t \rightarrow s[p \leftarrow \theta s']$ by $t' \rightarrow s'$, then the resulting TRS $D' \cup N$ is equivalent to the original TRS $D \cup N$.*

Proof. $S(u, f_{\rightarrow_{D'\cup N}}, \rightarrow_{D'\cup N}) \subseteq S(u, f_{\rightarrow_{D\cup N}}, \rightarrow_{D\cup N})$: For any $\rightarrow_{D'\cup N}$-computation $\vec{t'}$ there exists a $\rightarrow_{D\cup N}$-computation $\vec{t}$, and an injection $n: \mathbf{N} \rightarrow \mathbf{N}$ such that $n(i) \rightarrow \infty$ when $i \rightarrow \infty$ and vice versa, where $\vec{t}_{n(i)} = \vec{t'}_i$ for all i. See Fig. 3 for the crucial step. $\bigsqcup f_{\rightarrow_{D\cup N}}(\vec{t}) = \bigsqcup f_{\rightarrow_{D'\cup N}}(\vec{t'})$ follows.

Now assume that $\bigsqcup f_{\rightarrow_{D'\cup N}}(\vec{t'}) \in S(u, f_{\rightarrow_{D'\cup N}}, \rightarrow_{D'\cup N})$. Will $\bigsqcup f_{\rightarrow_{D\cup N}}(\vec{t}) \in S(u, f_{\rightarrow_{D\cup N}}, \rightarrow_{D\cup N})$ follow? Yes, since any $\rightarrow_{D\cup N}$-redex in $\vec{t}$ but not in $\vec{t'}$ eventually will disappear, due to the infinite emulation ($t \rightarrow s$-redexes in particular). Thus, there can be no $\rightarrow_{D\cup N}$-computation strictly majorating $\vec{t}$.

$S(u, f_{\rightarrow_{D\cup N}}, \rightarrow_{D\cup N}) \subseteq S(u, f_{\rightarrow_{D'\cup N}}, \rightarrow_{D'\cup N})$: This is proved by showing that for any $\rightarrow_{D\cup N}$-computation there is a majorating $\rightarrow_{D\cup N}$-computation that emulates a $\rightarrow_{D'\cup N}$-computation. See Fig. 4 (the crucial steps are $t \rightarrow s$-reductions, all others are directly emulated in $D' \cup N$). ■

We prove correctness of folding under the simplifying condition that *a rule is not folded with itself.* This corresponds to "restricted folding-unfolding" in [10].

Theorem 6 *If $t \rightarrow s$ in D is folded into $t \rightarrow s[p \leftarrow \theta t']$ by $t' \rightarrow s'$, where $t' \rightarrow s' \neq t \rightarrow s$, then the resulting TRS $D' \cup N$ is equivalent to the original TRS $D \cup N$.*

Proof. The proof is dual to the proof of Theorem 5. For any $\rightarrow_{D\cup N}$-computation, we can construct an emulating $\rightarrow_{D'\cup N}$-computation, which proves
$S(u, f_{\rightarrow_{D\cup N}}, \rightarrow_{D\cup N}) \subseteq S(u, f_{\rightarrow_{D'\cup N}}, \rightarrow_{D'\cup N})$. In order to prove

$S(u, f_{\rightarrow_{D \cup N}}, \rightarrow_{D \cup N}) \subseteq S(u, f_{\rightarrow_{D' \cup N}}, \rightarrow_{D' \cup N})$ we construct, for any $\rightarrow_{D' \cup N}$-comput/-ation, a majorating $\rightarrow_{D' \cup N}$-computation that emulates a $\rightarrow_{D \cup N}$-computation. ■

10 A Simple Process Language Example

Consider a simple language fragment that describes a class of CSP-like communicating processes. This language fragment has types *Proc*, *Event*, *Chan*, a parallel composition "|" of type $Proc \times Proc \rightarrow Proc$ which is associative and commutative, an empty process $\mathbf{0}$, and a prefix operation "." of type $Event \times Proc \rightarrow Proc$. Furthermore, for all possible value types *Val* in the language there is a channel write operation "!" of type $Chan \times Val \rightarrow Event$ and a channel read operation "?" of the same type, but with the restriction that the argument of type *Val* must be a variable. Finally, we assume a number of constants of type *Event*, and of type *Chan*.

We have the following computation rules. For each constant a of type *Event* and c of type *Chan*, we have rules

$$a.P|x.Q \;\rightarrow\; a.(P|x.Q) \tag{1}$$

$$c!y.P|c?x.Q \;\rightarrow\; P|Q\{x/y\} \tag{2}$$

The first group of rules (1) express that "basic" events are interleaved as parallel processes execute. Thus, the result of a computation is essentially a trace of events. The second group of rules (2) are communication rules. P, Q, x, y are variables above: thus, the rules (1) are pure term rewriting rules whereas the rules (2) are akin to β-reduction. Furthermore, we introduce auxiliary term rewriting rules, viz.:

$$(P|Q)|R \;\rightarrow\; P|(Q|R) \tag{3}$$

$$P|(Q|R) \;\rightarrow\; (P|Q)|R \tag{4}$$

$$P|Q \;\rightarrow\; Q|P \tag{5}$$

They express equivalence modulo AC for parallel composition. Their permutative nature gives rise to cyclic, infinite computations which are in some sense artificial. However, these computations will be strictly majorated whenever progress can be made through some of the first rules. Thus, they will not contribute to the semantics.

The rules (1)-(5) can easily be put into CRS format (in (2), x will be abstracted over Q). It is now immediate, from the CRS version of Theorem 1, that this process language can be enriched with a computational component described by CRS rules, orthogonal to the rules (1)-(5). For instance, any "PCF-like" language, as discussed in Sect. 6, can be added. Then, all the results in Sections 7, 8 and 9 apply and they can be used to implement and transform programs in the combined language.

Our process primitives above compute traces. Thus, semantical equivalence is essentially trace equivalence. Trace equivalence is known to be the finest of a number of process algebra equivalences, such as bisimulation [27]. Therefore, program transformations which are correct in our sense will also preserve these equivalences.

11 Conclusion and Further Research

We have presented results regarding semantically correct evaluation strategies and program transformations for programs with a computational, deterministic part and a descriptive, possibly non-deterministic part. For a computation-based semantics, it was shown that a simple schematical property (commutation of relations) makes the computational part "referentially transparent" even in the presence of non-determinism in the environment. Certain program transformations on the computational part, such as fold/unfold transformations, were shown correct, and conditions were given for when algebraic equalities could be used to transform program parts. Cofinal reduction strategies for the computational part were shown to induce semantically correct nondeterministic reduction strategies for the combined system.

The abstract schematic properties were applied to computations on terms. Classical results for CRS'es about commutation and cofinal reduction strategies made it possible to apply the developed theory to recursive higher order languages with λ-abstraction and process communication. A simple example language with CSP style process communication was defined and it was shown to have the desired properties.

A possible application of the theory developed here is to support the design and implementation of lazy recursive languages with process primitives. In order to do this, three things must be observed: first, lazy languages use a *normalizing* reduction strategy rather than a cofinal one. We believe that the correctness can be proved for combined reduction strategies, employing a normalizing (rather than cofinal) strategy for certain terms, provided that a coarser semantic tree c.p.o. is used where the strictness of certain base operations is taken into account. Second, laziness also implies sharing of already computed results. This is not expressible in the simple term rewriting systems considered here, and must be modeled in a formalism such as explicit substitutions [1]. Furthermore, it seems that sharing must not be hidden in languages with nondeterminism, since reuse of a nondeterministic result is not the same as computing it anew. Third, lazy languages reduce to weak head normal form rather than normal form. The semantic tree c.p.o. must then be modified accordingly, since the pure Böhm tree model is not appropriate for the lazy λ-calculus [2].

The correctness of fold/unfold-transformations was proved for TRS'es rather than CRS'es, but we believe the results carry over more or less verbatim. Furthermore, the correctness of folding was proved under the simplifying condition that a rule would not be used to fold itself. In the classical theory for fold/unfold-tranformations, conditions relaxing this restriction have been given by Courcelle [8] and Kott [18]. We conjecture that these results carry over to our framework.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Proc. Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 31–46, San Francisco, Jan. 1990.

[2] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, chapter 4, pages 65–116. Addison-Wesley, Reading, MA, 1989.

[3] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[4] A. Arnold, P. Naudin, and M. Nivat. On semantics of nondeterministic recursive program schemes. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 1, pages 1–33. Cambridge University Press, 1985.

[5] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1981.

[6] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 5, pages 170–236. Cambridge University Press, 1985.

[7] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. Assoc. Comput. Mach.*, 24(1):44–67, Jan. 1977.

[8] B. Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Math. System. Theory*, 13:131–180, 1979.

[9] B. Courcelle. Equivalence and transformations of regular systems. *Theoret. Comput. Sci.*, 42:1–122, 1986.

[10] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 9, pages 459–492. Elsevier Science Publishers B. V., 1990.

[11] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics ans continuous algebras. *J. Assoc. Comput. Mach.*, 24(1):68–95, Jan. 1977.

[12] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[13] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Inform.*, 11(1):31–55, Jan. 1978.

[14] N. D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, Oct. 1987*, pages 1–14. North-Holland, 1988.

[15] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, Hertfordshire, UK, 1993.

[16] J. W. Klop. *Combinatory Reduction Systems.* PhD thesis, CWI, Amsterdam, 1980. Mathematical Centre Tracts Nr. 127.

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

[18] L. Kott. Unfold/fold program transformations. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 411–434. Cambridge University Press, 1985.

[19] M. Nivat. On the interpretation of recursive polyadic program schemes. *Symposia Mathematica*, 15:255–281, 1975.

[20] G. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5(3):223–256, Dec. 1977.

[21] J.-C. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *J. Assoc. Comput. Mach.*, 27(4):772–796, Oct. 1980.

[22] J. H. Reppy. CML: A higher order concurrent language. In *Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[23] D. Sahlin. Partial evaluation of AKL. In *Proc. First International Workshop on Concurrent Constraint Programming*, Cà Dolfin, Venice, May 1995.

[24] H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[25] B. Thomsen, L. Leth, and A. Giacalone. Some Facile chemistry. Tech. Rep. ECRC-92-14, European Computer-Industry Research Centre, 1992.

[26] D. A. Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley, Reading, MA, 1989.

[27] R. J. van Glabbeek. The linear time – branching time spectrum II: The semantics of sequential systems with silent moves. In E. Best, editor, *Proc. CONCUR'93*, Volume 715 of *Lecture Notes in Comput. Sci.*, pages 66–81, Hildesheim, Germany, Aug. 1993. Springer-Verlag.