ELSEVIER
Theoretical Computer Science 190 (1998) 61–85
Theoretical Computer Science

Computing in unpredictable environments: semantics, reduction strategies, and program transformations

Björn Lisper*,[1]

Department of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 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* with respect to deterministic rules, 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

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 receives 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

* E-mail: lisper@it.kth.se.

[1] This work was done while the author was with Swedish Institute of Computer Science, under the support of the ESPRIT BRA project CONFER, project no. 6454, and with travel support from the HCM network EXPRESS.

PII S0304-3975(97)00084-4

such as asynchrony, e.g., some communication events may take place in different order. The latter can give rise to *nondeterminism*, i.e., 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 [20, 35]

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. 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 for $\to_D$ there exists a semantically complete, nondeterministic (w.r.t. $\to_N$) reduction strategy, for any monotone interpretation.
2. If $\to_D$ and $\to_N$ commute, then certain transformations preserve the semantics, for any monotone interpretation.

"Semantically complete" 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 (possibly higher order) term rewriting rules. Program transformations described by such operations include fold–unfold transformations of recursive programs, but also "program simplification" like evaluation of constant subexpressions, and reverse β-reduction as used in lambda lifting [18].

Our choice to consider "semantical completeness" 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.

2. Related work

Our work is strongly related to the theory of recursive applicative program schemes [12, 27]. 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 [8] for first order term rewriting systems. It is also related to the theory of approximate normal forms in the λ-calculus [26, 37, 38]. What sets our work apart is that we formulate the semantics and prove some results in the more abstract setting of binary relations. The results on fold–unfold program transformations are obtained for an instance of the abstract semantics where computations take place on terms and the binary relation is given by a Combinatory Reduction System [22, 24].

Some of our results concern referential transparency. There is an interesting discussion of referential transparency and unfoldability in [33]; one can say that commutation of the computational and descriptive reduction systems *preserves referential transparency for the computational system also in a non-deterministic environment.*

Our framework is abstract and thus potentially applicable to a range of languages, but the ones we have in mind are, mainly recursive languages with concurrency constructs. Some existing, not referentially transparent languages in this class are Concurrent ML [30], Facile [34], 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 presence of nondeterminism arising from the concurrency. A language that seem to fit this charter is Concurrent Haskell [28], a lazy functional language extended with concurrency primitives.

A comparison can be made with Hughes' and Moran's work [17], where they give a natural semantics for normal order λ-calculus plus McCarty's *amb* operator for nondeterministic choice. It seems that this combination can be modeled in our framework as well. This would give a very simple semantics in comparison, and make our results about program transformations applicable. On the other hand, the rather detailed natural semantics of Hughes and Moran gives more information about how to actually implement a language with these operations.

Our results about folding and unfolding extend classical results [10–12, 25] for deterministic systems. They also hold for systems with higher order operations, which is true as well for the recent work on correctness for transformations of deterministic programs by Sands [32]. Our results provide some support for program transformation systems for program optimization [9, 16] and partial evaluation of nondeterministic languages. Semantics-preserving partial evaluation of such languages is listed as the "challenging problem no. 10.9" in [19]. The only partial evaluator for such languages that we are aware of is for the concurrent constraint programming language AKL [31]. 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 (abstract) *reduction systems* [15, 23] to give semantics of computations. We give the definitions of the most central concepts being used here. The notation is standard except possibly for the sets of finite, infinite and partial terms below.

Let $\rightarrow$, $\rightarrow_1$, $\rightarrow_2$ be binary relations over some set A. Then $\rightarrow_1$ and $\rightarrow_2$ *commute* if $\forall a,b,c\exists d[(a \rightarrow_1^* b \wedge a \rightarrow_2^* c) \Rightarrow (b \rightarrow_2^* d \wedge c \rightarrow_1^* d)]$. See Fig. 1. ($\rightarrow^*$ 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 $a \rightarrow a'$.

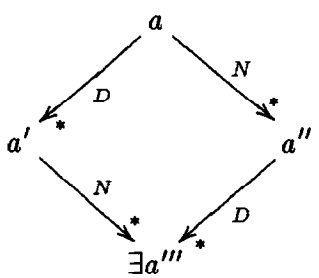

Fig. 1. $\rightarrow_D$ and $\rightarrow_N$ commute.

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, it holds that $F^n(a)$ is a normal form 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 – "term reduction systems" (including first order term rewriting systems (TRS), as well as λ-calculus with β-reduction and more general higher order formalisms such as Klop's Combinatory Reduction Systems (CRS) [22]). 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, 14]. For $p \notin dom(t)$, we define $t(p)=\perp$.

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 $T_\perp^\infty$. We have $T \subset T^\infty \subset T_\perp^\infty$. Cf. the *Böhm trees* of λ-calculus [5, Ch. 10].

For the operator symbols, we define a flat partial order $\sqsubseteq$ by $\perp \sqsubseteq s$ and $s \sqsubseteq s$ for all symbols s. We lift this order to terms (i.e. functions from sequences) in $T_\perp^\infty$ by $t \sqsubseteq t'$ iff $t(p) \sqsubseteq t'(p)$ for all sequences p (cf. [5, Ch. 10.2]). It is straightforward to show that every ascending chain $t_0 \sqsubseteq t_1 \sqsubseteq \cdots$ in $T_\perp^\infty$ has a least upper bound in $T_\perp^\infty$. The maximal elements are exactly those in T^∞.

Two positions are *disjoint* if none is a prefix of the other. For any term $t \in T_\perp^\infty$ and position $p \in dom(t)$, we denote the subterm at p by t/p (not to be confused with the function symbol $t(p)$ at position 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. Combinatory reduction systems

Our results on the correctness of fold–unfold transformations in Section 9 are established for systems defined by Combinatory Reduction Systems. Therefore we give a short description of CRS here. For a full description, including formal definitions, see [22, 24].

Consider a set of terms T, constructed out of constant function symbols with fixed arity, nullary variables, and a binary *abstraction operator*, written $[\_]\_$ (i.e., if t is a term and x is a variable, then $[x]t$ is a term). A Combinatory Reduction System over T is a set of *reduction rules* $s \rightarrow t$, where s, t are *metaterms*, constructed as the terms in T, plus terms containing *metavariables*, written in upper case (Z, Z', etc.). Each metavariable has an arity (possibly 0): if Z has arity k, then $Z(t_1,\ldots,t_k)$, where $t_1,\ldots,t_k$ are metaterms, is a metaterm.

(Meta) terms are considered equivalent modulo renaming of bound variable (α-conversion). For metaterms s, t in reduction rules $s \to t$, we impose the following additional restrictions:

- s and t are *closed* (i.e., a variable x occurs only within the scope of a binding $[x]\_$).
- s has the form $F(t_1,\ldots,t_n)$, where F is a constant function symbol.
- Any metavariable occurring in t also occurs in s.
- A metavariable Z of arity k occurs only in the form $Z(x_1,\ldots,x_k)$, where $x_1,\ldots,x_k$ are pairwise distinct variables.

TRS rules [23] are exactly CRS rules with nullary metavariables only and no abstraction. But CRS rules also include reductions in systems with bound variables. For instance, β-reduction in λ-calculus is described by the CRS rule

$$@(\lambda([x]Z(x)),Z') \to Z(Z'). \tag{1}$$

(Here, @ is the binary function symbol for application.)

A CRS R generates a *reduction relation* $\to_R$ on the set of terms. Essentially, $t_1 \to_R t_2$ iff, for some rule $s \to t \in R$, position p in $dom(t_1)$ and *valuation* σ, holds that $t_1/p = \sigma(s)$ and $t_2 = t_1[p \leftarrow \sigma(t)]$. t_1/p is then a $s \to t$-*redex*. A valuation is a map from metavariables: if Z is n-ary, then $\sigma(Z)$ has the form $\lambda(x_1,\ldots,x_n).t$, where t is a term. Valuations are extended to homomorphisms on metaterms as follows:

- $\sigma(x) = x$, for variables x;
- $\sigma([x]t) = [x]\sigma(t)$;
- $\sigma(F(t_1,\ldots,t_n)) = F(\sigma(t_1),\ldots,\sigma(t_n))$, F constant function symbol;
- $\sigma(Z(t_1,\ldots,t_n)) = t[x_1 := \sigma(t_1),\ldots,x_n := \sigma(t_n)]$, where Z is a metavariable such that $\sigma(Z) = \lambda(x_1,\ldots,x_n).t$.

The meaning of the metasubstitution above is the simultaneous replacement of all occurrences of the respective variables x_i with t_i. Thus, the last rule could be written $\sigma(Z(t_1,\ldots,t_n)) = \sigma(Z)(\sigma(t_1),\ldots,\sigma(t_n))$. We do not incorporate automatic renaming of bound variables in metasubstitutions to avoid the capturing of free variables: instead, the following is required. A CRS rule $s \to t$ being matched to a subterm by a valuation σ must be *safe* for σ, which means that there is no metavariable Z in s or t such that $\sigma(Z)$ has a free variable x occurring in an abstraction $[x]$ in s or t. Also, σ is required to be *safe with respect to itself*, which means that there are no Z, Z' such that $\sigma(Z)$ contains a free variable appearing bound in $\sigma(Z')$.

One can always rename bound variables in a valuation so that it becomes safe with respect to itself, and in a rewrite rule so it becomes safe for a certain valuation. We allow this renaming in the matching process, i.e.: $t_1[p \leftarrow \sigma(s')] \to t_1[p \leftarrow \sigma(t')]$ iff $s' \to t'$ and σ fulfil the safety conditions and if $s' \to t'$ is obtained from the CRS rule $s \to t$ by renaming bound variables.

If all metavariables in s (and thus t) are nullary, then valuations are equivalent to substitutions for metavariables. If there are no abstractions, we then obtain exactly the reduction relation for the corresponding TRS.

An important concept for CRS is *orthogonality*. Two CRS rules $s \to t$, $s' \to t'$ are orthogonal if:

- The rules are *left-linear*, i.e., none of s and s' contains multiple occurrences of any metavariable.
- They are *non-overlapping*, that is: whenever a $s \to t$-redex contains a $s' \to t'$-redex (or vice versa), then the contained $s' \to t'$-redex must be contained in an instantiation of a metavariable of $s \to t$.

A CRS is orthogonal if all its rules are orthogonal (even to themselves). Two CRSs R, R' are mutually orthogonal if any rule in R is orthogonal to any rule in R'.

The following result for mutually orthogonal CRSs was first stated for first order TRSs [29, Proposition 10], but the proof carries over to mutually orthogonal CRSs [36]. Due to this theorem, our results in Sections 7–9 are directly applicable to mutually orthogonal CRSs.

Theorem 1 (Raoult and Vuillemin [29]). *If the CRSs R and R' are mutually orthogonal, then $\to_R$ and $\to_{R'}$ commute.*

Also many results on reduction strategies carry over from orthogonal TRSs to orthogonal CRSs. For instance, the leftmost-outermost reduction strategy (normal order reduction) is normalizing for any *left-normal* CRS (all constants and functions symbols precede all metavariables in the LHS of every rule) [22]. Of particular interest are results about *cofinal* reduction strategies, since our results in Section 7 concern such strategies. For orthogonal CRSs, the following result by Klop [22] 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 2 (Klop [22]). *For orthogonal CRS, any fair reduction strategy is cofinal.*

We find CRSs interesting since it seems like a large number of programming primitives can be modeled by orthogonal CRS rules. For instance, any first order TRS is a CRS which means that essentially all first order primitives commonly occuring in programming languages can be modeled, as well as recursive definitions of first order functions. But CRS rules can also model the execution of higher order constructs such as λ-terms, local recursive function definitions ("letrec"), pattern matching etc. The results on normalizing and cofinal reduction strategies give direct support for deterministic implementations of languages modeled by orthogonal CRS rules. Furthermore nondeterministic operations, such as nondeterministic choice and merge are readily modeled, and even some process communication primitives which involve the binding of variables (see Section 10). It thus seems like CRS can provide a suitable framework for reduction-oriented reasoning about languages with such primitives.

5. Semantics for abstract computations

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

Definition 3. A $\rightarrow$-*computation* in A is an infinite sequence of elements $\boldsymbol{a} = \{\boldsymbol{a}_i\}_0^\infty$ in A such that
- If $\boldsymbol{a}_i$ is a $\rightarrow$-nf, then $\boldsymbol{a}_i = \boldsymbol{a}_{i+1}$.
- Otherwise, $\boldsymbol{a}_i \rightarrow \boldsymbol{a}_{i+1}$.

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

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

Consider an ARS $\langle A, \rightarrow \rangle$, a c.p.o. $\langle C, \sqsubseteq \rangle$ (possibly without bottom), and a mapping $f\colon A \rightarrow C$ which is monotone w.r.t. $\rightarrow$ and $\sqsubseteq$. Cf. [2, 27, 37], and also [5, Lemma 14.3.7]. 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 $\boldsymbol{a}$ yields a l.u.b. $\bigsqcup_{i=0}^\infty f(\boldsymbol{a}_i)$ in C, denoted $\bigsqcup f(\boldsymbol{a})$, which can be seen as the result of the computation.

Definition 4. Let f be a monotone interpretation. $\boldsymbol{a}$ is *dominated* by $\boldsymbol{a}'$ iff $\forall i \exists j$: $\boldsymbol{a}_i \rightarrow^* \boldsymbol{a}'_j$. It is *strictly dominated by* $\boldsymbol{a}'$ *under* f iff it is dominated by $\boldsymbol{a}'$ and $\bigsqcup f(\boldsymbol{a}) \sqsubset \bigsqcup f(\boldsymbol{a}')$.

Strict domination can be seen as an abstract "unfairness" condition for the strictly dominated computation: essentially, such a computation always has some information-increasing path which is left unexplored. A classical example from the $\lambda\beta$-calculus (considered as a CRS β) is the set of $\rightarrow_\beta$-computations rooted in the term $\lambda x.y((\lambda x.xx)(\lambda x.xx))$ under the monotone interpretation $[\beta]$ of Section 6. The non-terminating computation $\lambda x.y((\lambda x.xx)ax.xx)) \rightarrow_\beta \lambda x.y((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \cdots$, where the outermost redex is never reduced, has the limit $\perp$ under $[\beta]$, whereas any computation reducing this redex ends up in y with limit y. These computations all dominate the infinite computation strictly, thus, it is unfair on our sense.

The following propositions are easily established:

Proposition 5. *If* $\boldsymbol{a}$ *is dominated by* $\boldsymbol{a}'$, *then, for any monotone interpretation* f, *it holds that* $\bigsqcup f(\boldsymbol{a}) \sqsubseteq \bigsqcup f(\boldsymbol{a}')$.

Proposition 6 (Strict). *Domination is transitive.*

We can now define the semantics for an element $a \in A$, given a monotone interpretation f of $\langle A, \rightarrow \rangle$ into $\langle C, \sqsubseteq \rangle$ (cf. [8, Ch. 4.2]):

Definition 7. The *semantics* of $a \in A$ w.r.t. the monotone interpretation f and $\rightarrow$, $S(a, f, \rightarrow)$, is the set

$$\{\bigsqcup f(\boldsymbol{a}) \mid \boldsymbol{a} \text{ is } a\text{-rooted and not strictly dominated by any computation}\}.$$

Definition 7 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" dominating 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.

It is easy to see that if $\rightarrow$ has the *cofinality property* (CP) [23], then $S(a, f, \rightarrow)$ has a single element. Since CP is equivalent to confluence for countable reduction systems [23], confluence thus implies "uniqueness of infinite normal forms" for such systems. This is in contrast to the situation in *transfinite term rewriting* [13, 21]. There, the transfinite reduction relation may be nonconfluent even when the finite relation is, which then makes it possible to have several infinite normal forms. The reason is, roughly speaking, that the definitions of convergence for sequences of terms allow some "nonfair" infinite sequences to converge. Our definition of "infinite normal form" disallows this.

6. Semantics for computations on terms

We consider semantics for computations on terms defined by Combinatory Reduction Systems. This semantics will be used in Section 9, where we consider fold/unfold transformations of such systems. Semantics for computations on terms can be defined in many different ways; the one developed here reflects the view that the result of a computation is a value built out of *constructors*, which are function symbols without any further interpretation. This view is the usual one for functional languages, where the constructors typically are constants, cons operations for lists, pairing constructors, etc.

Definition 8. A function symbol F is a *constructor w.r.t. the CRS R* if there is no rule in R with a left-hand side of the form $F(t_1, \ldots, t_n)$.

Definition 9. For any CRS R over a set of terms T, $[R] : T \rightarrow T_{\perp}^{\infty}$ is defined for any $t \in T$ by

- $[R](t)(p) = t(p)$, if $t(p)$ is a constructor w.r.t. R, an abstraction, or a variable, and for all prefixes p' of p holds that $t(p')$ is a constructor w.r.t. R, an abstraction, or a variable.
- Otherwise, $[R](t)(p) = \perp$.

It is easy to verify that $[R]$ is indeed a monotone interpretation of $\langle T, \rightarrow_R \rangle$ into $\langle T_\perp^\infty, \sqsubseteq \rangle$. A similar construct is the interpretation of Nivat [27] for recursive applicative program schemes. For graph rewriting systems, the *instant semantics* of Ariola and Arvind [2] corresponds to $[R]$. A third example is Wadsworth's *best direct approximants* for λ-terms under β-reduction [37] (see also [5, Definition 14.3.6]).

Since $[R]$ is a monotone interpretation, R defines a semantics $S(t, [R], \rightarrow_R)$, denoted $S_R(t)$, w.r.t. R for terms t in T. This semantics consists of all trees, possibly partial and/or infinite, which are limits of "fair" t-rooted computations under $[R]$. We call it the *constructor tree semantics* for R. For instance, if R consists of the rules

$$f(x, y) \rightarrow x : y : f(x, y)$$
$$g \rightarrow g$$

then the single element of $S_R(f(g, c))$ is the infinite partial tree $\perp : c : \perp : c : \cdots$. If T is the set of pure λ-terms and if β is the CRS defining β-reduction (with reduction relation $\rightarrow_\beta$) to head normal form, then the single element of $S_\beta(t)$ is very similar to the Böhm tree for t [5].

7. Reduction strategies for 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.

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*. Some hold under the weaker condition that $\rightarrow_D$ and $\rightarrow_D \cup \rightarrow_N$ commute. We have:

Proposition 10. *If* $\rightarrow_D$ *and* $\rightarrow_N$ *commute and if* $\rightarrow_D$ *is confluent, then* $\rightarrow_D$ *and* $\rightarrow_D \cup \rightarrow_N$ *commute.*

Proof. A simple diagram chase according to Fig. 2. □

Definition 11. A is a *nondeterministic reduction strategy w.r.t.* $\rightarrow$ if it is a subrelation of $\rightarrow$ such that any A-nf is a $\rightarrow$-nf.

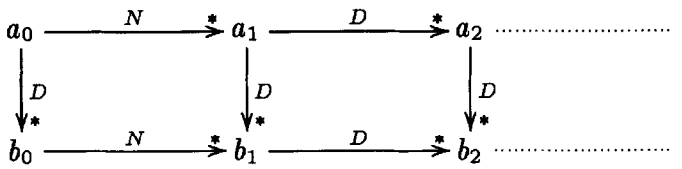

Fig. 2. Diagram chase to prove commutation of $\rightarrow_D$ and $\rightarrow = \rightarrow_D \cup \rightarrow_N$.

The requirement that any A-nf is a $\rightarrow$-nf can be seen as a progress condition for A, ensuring that it will not "terminate" prematurely; cf. the usual definition of (deterministic) reduction strategies in Section 3.

Definition 12. The *nonfair semantics* of $a \in A$ w.r.t. the monotone interpretation f and $\rightarrow$, $\overline{S}(a, f, \rightarrow)$, is the set $\{\bigsqcup f(\boldsymbol{a}) \mid \boldsymbol{a}$ is an a-rooted $\rightarrow$-computation$\}$.

In contrast to $S(a, f, \rightarrow)$, $\overline{S}(a, f, \rightarrow)$ contains also the results of possible nonfair computations. We will use the nonfair semantics to define the meaning of reduction strategies, which is appropriate since a reduction strategy can indeed be nonfair (in the abstract sense here).

Definition 13. A is *semantically complete* w.r.t. $\rightarrow$ and f iff $\overline{S}(a, f, A) = S(a, f, \rightarrow)$ for all a.

Proposition 14. *For any $\rightarrow$ and monotone interpretation f there is a semantically complete nondeterministic reduction strategy.*

Proof. For any a and $s \in S(a, f, \rightarrow)$, pick an a-rooted $\rightarrow$-computation $\boldsymbol{a}_s$ such that $\bigsqcup f(\boldsymbol{a}_s) = s$. This yields $A = \bigcup(\boldsymbol{a}_s \mid a \in A,\ s \in S(a, f, \rightarrow))$ (where $\boldsymbol{a}_s$ is considered a relation). Since A is formed from one a-rooted $\rightarrow$-computation for each $a \in A$, and since any $\rightarrow$-computation must pick a successor w.r.t. $\rightarrow$ whenever one exists, it follows that any A-nf is a $\rightarrow$-nf. □

Definition 15. Let A be a nondeterministic reduction strategy w.r.t. $\rightarrow_D \cup \rightarrow_N$. A is *$\rightarrow_D$-deterministic* if, for any a, there is at most one a' such that $a\, A\, a'$ and $a \rightarrow_D a'$.

Thus, for a $\rightarrow_D$-deterministic strategy, the nondeterminism is entirely due to $\rightarrow_N$ since, for any a, it is completely determined from which element $\rightarrow_D$ will compute no matter which computation path is chosen. We now have the following central result:

Theorem 16. *If $\rightarrow_D$ and $\rightarrow_N$ commute and if $\rightarrow_D$ is confluent, then, for any cofinal reduction strategy F for $\rightarrow_D$, there exists a $\rightarrow_D$-deterministic, semantically complete nondeterministic reduction strategy $\boldsymbol{F}$, where for each $\boldsymbol{F}$-computation $\boldsymbol{a}$ and $i \in \boldsymbol{N}$ it holds that $\boldsymbol{a}_{i+1} = F(\boldsymbol{a}_i)$ or $\boldsymbol{a}_i \rightarrow_N \boldsymbol{a}_{i+1}$.*

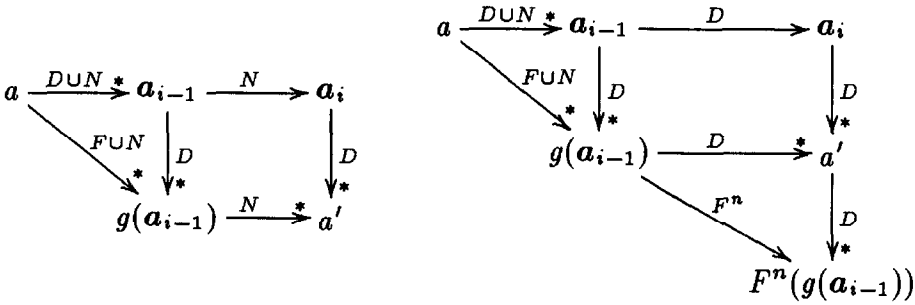

Fig. 3. Diagram chase to prove Theorem 16.

Proof. Choose a semantically complete nondeterministic reduction strategy A w.r.t. $\rightarrow_D \cup \rightarrow_N$. Now, we show inductively that for any a and a-rooted $\rightarrow_D \cup \rightarrow_N$-computation $\boldsymbol{a} \in A$ there exists a dominating a-rooted $F \cup \rightarrow_N$-computation; these computations must then have the same limit since A is semantically complete. For each $\boldsymbol{a}_i$ we find an element $g(\boldsymbol{a}_i)$ such that: $a\ (F \cup \rightarrow_N)^*\ g(\boldsymbol{a}_i)$ and $\boldsymbol{a}_i \rightarrow_D^* g(\boldsymbol{a}_i)$. For $i=0$ we can pick $g(\boldsymbol{a}_0)=\boldsymbol{a}_0=a$. For $i>0$, assume true for $i-1$. We then have $\boldsymbol{a}_{i-1} \rightarrow_D^* g(\boldsymbol{a}_{i-1})$. There are three cases:

- $\boldsymbol{a}_{i-1}$ is an A-nf. Then $\boldsymbol{a}_{i-1}=\boldsymbol{a}_i$ and we must have $g(\boldsymbol{a}_{i-1})=g(\boldsymbol{a}_i)$. However, $\boldsymbol{a}_{i-1}$ must also be a $\rightarrow_D \cup \rightarrow_N$-nf, and thus a $\rightarrow_D$-nf. Therefore $g(\boldsymbol{a}_{i-1})$ equals $\boldsymbol{a}_{i-1}$ and so is a $\rightarrow_D$-nf. Therefore, $F(g(\boldsymbol{a}_{i-1}))=g(\boldsymbol{a}_{i-1})=g(\boldsymbol{a}_i)$.
- $\boldsymbol{a}_{i-1} \rightarrow_N \boldsymbol{a}_i$. Then, by commutativity, there is an element a' such that $\boldsymbol{a}_i \rightarrow_D^* a'$ and $g(\boldsymbol{a}_{i-1}) \rightarrow_N^* a'$, so we can choose $g(\boldsymbol{a}_i)=a'$.
- $\boldsymbol{a}_{i-1} \rightarrow_D \boldsymbol{a}_i$. Then, by confluence, there is again an a' s.t. $\boldsymbol{a}_i \rightarrow_D^* a'$ and, this time, $g(\boldsymbol{a}_{i-1}) \rightarrow_D^* a'$. By cofinality of F there now exists an n such that $a' \rightarrow_D^* F^n(g(\boldsymbol{a}_{i-1}))$, and we can, since $\boldsymbol{a}_i \rightarrow_D^* F^n(g(\boldsymbol{a}_{i-1}))$, pick $g(\boldsymbol{a}_i)=F^n(g(\boldsymbol{a}_{i-1}))$.

See Fig. 3. We can now form a nondeterministic reduction strategy fulfilling the conditions in the theorem as the set of all dominating computations constructed as above, for all a. □

When $\rightarrow_D$ is given by an orthogonal CRS Theorems 2 and 16 link Klop's notion of fairness to our more abstract notion of fairness, since a fair reduction strategy (in the sense of Klop) is cofinal for such CRSs.

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:

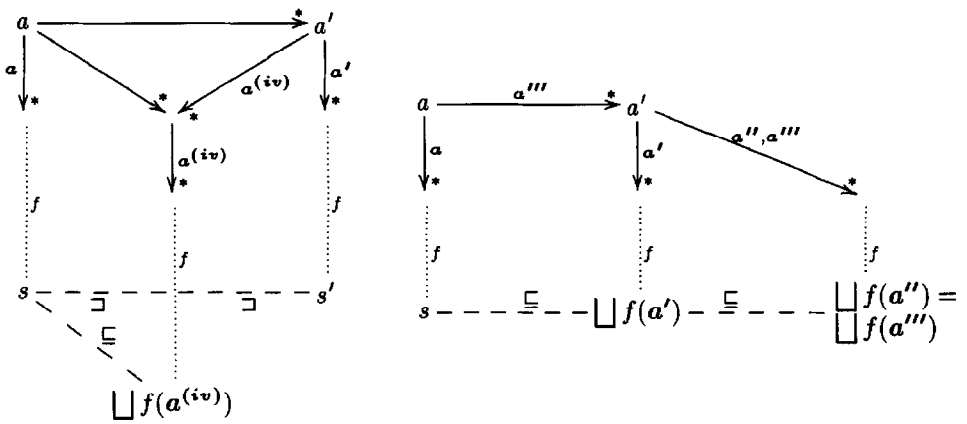

Fig. 4. $S(a',f,\rightarrow)\subseteq S(a,f,\rightarrow)$ and $S(a,f,\rightarrow)\subseteq S(a',f,\rightarrow)$: illustrations of the respective proofs. Dotted lines indicate limits under the monotone interpretation f, dashed lines indicate that elements are related w.r.t. $\sqsubseteq$ ($\sqsubset$).

Lemma 17. *If $a\rightarrow^* a'$, and if for any $s\in S(a,f,\rightarrow)$ there is an a-rooted computation $\boldsymbol{a}$ such that $\bigsqcup f(\boldsymbol{a})=s$ and an a'-rooted computation dominating $\boldsymbol{a}$, then $S(a',f,\rightarrow)=S(a,f,\rightarrow)$.*

Proof. $S(a',f,\rightarrow)\subseteq S(a,f,\rightarrow)$: consider any $s'\in S(a',f,\rightarrow)$. There is an a'-rooted $\boldsymbol{a}'$ where $\bigsqcup f(\boldsymbol{a}')=s'$. Furthermore, there is an a-rooted $\boldsymbol{a}''$ formed by prefixing $\boldsymbol{a}'$ with the elements on some path $a\rightarrow^* a'$. We have $\bigsqcup f(\boldsymbol{a}'')=s'$. Thus, $s'\in S(a,f,\rightarrow)$ unless there is some other a-rooted $\boldsymbol{a}'''$ strictly dominating $\boldsymbol{a}''$. The existence of such an $\boldsymbol{a}'''$ implies the existence of an $s\in S(a,f,\rightarrow)$ such that $s'\sqsubset s$. Then, by assumption, there is an a-rooted $\boldsymbol{a}$ with $\bigsqcup f(\boldsymbol{a})=s$ and an a'-rooted computation $\boldsymbol{a}^{(iv)}$ dominating $\boldsymbol{a}$. But we then have $s'\sqsubset\bigsqcup f(\boldsymbol{a}^{(iv)})$ which contradicts $s'\in S(a',f,\rightarrow)$.

$S(a,f,\rightarrow)\subseteq S(a',f,\rightarrow)$: Let $s\in S(a,f,\rightarrow)$. Then there is an a-rooted $\boldsymbol{a}$ with $\bigsqcup f(\boldsymbol{a})=s$ and an a'-rooted $\boldsymbol{a}'$ dominating $\boldsymbol{a}$. Thus, $s\sqsubseteq\bigsqcup f(\boldsymbol{a}')$. The existence of an a'-rooted $\boldsymbol{a}''$ dominating $\boldsymbol{a}$ and where $\bigsqcup f(\boldsymbol{a}'')\in S(a',f,\rightarrow)$ follows, by transitivity of domination. We can form an a-rooted $\boldsymbol{a}'''$ with the same property by prefixing $\boldsymbol{a}''$ with the elements on some path $a\rightarrow^* a'$. But then we must have $s=\bigsqcup f(\boldsymbol{a}''')=\bigsqcup f(\boldsymbol{a}'')\in S(a',f,\rightarrow)$, since otherwise $\boldsymbol{a}$ would be strictly dominated by $\boldsymbol{a}'''$ which would contradict $s\in S(a,f,\rightarrow)$. □

See Fig. 4 for an illustration of the crucial parts of the proof of Lemma 17.

Lemma 18. *If $\rightarrow_D\subseteq\rightarrow^*$, if $\rightarrow_D$ commutes with $\rightarrow$, and if $a\rightarrow_D^* a'$, then any a-rooted computation is dominated by some a'-rooted computation.*

Proof. A simple diagram chase, using the commutation of $\rightarrow_D$ and $\rightarrow$. See Fig. 5. □

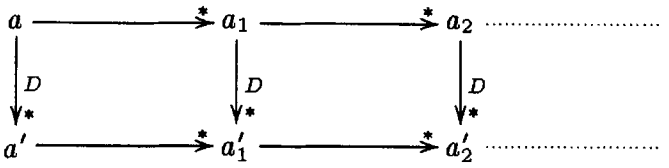

Fig. 5. Diagram chase to prove Lemma 18.

The following corollary to Lemmas 17 and 18 is a major stepping stone:

Corollary 19. *If* $\rightarrow_D \subseteq \rightarrow^*$, *if* $\rightarrow_D$ *commutes with* $\rightarrow$, *and if* $a \rightarrow_D^* a'$, *then* $S(a', f, \rightarrow) = S(a, f, \rightarrow)$.

By Proposition 10, Corollary 19 holds for $\rightarrow_D \cup \rightarrow_N$-computations when $\rightarrow_D$ and $\rightarrow_N$ commute. We now develop a class of equivalence relations.

Definition 20. Let Θ be a set of functions $A \rightarrow A$. Then $a \equiv_\Theta a'$ iff $S(\theta(a), f, \rightarrow) = S(\theta(a'), f, \rightarrow)$ for all $\theta \in \Theta$.

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

Theorem 21. *If* $\rightarrow_D \subseteq \rightarrow^*$ *and if* $\rightarrow_D$ *commutes with* $\rightarrow$, *then, for any* Θ, $a \equiv_{D\Theta} a' \Rightarrow a \equiv_\Theta a'$.

Proof. $x \leftrightarrow_D^* x' \Rightarrow S(x, f, \rightarrow) = S(x', f, \rightarrow)$ is proved by simple induction over $\leftrightarrow_D^n$ using Corollary 19. Instantiating $x = \theta a$, $x' = \theta a'$ for each $\theta \in \Theta$ then gives the result. □

We can relate Theorem 21 to algebraic formulations of equivalences in the following way. Let $\rightarrow_D$, $\rightarrow_N$ be term reduction relations. Assume that for a subset T_A of the terms, each element $t \in T_A$ is closed and has an interpretation $i(t)$ as an element in some algebra with carrier A. Assume furthermore that $\rightarrow_D$ is confluent and terminating on T_A, and that $i(t) = i(t')$ implies that $S(t, f, \rightarrow_D) = S(t', f, \rightarrow_D) = \{t''\}$ with $i(t'') = i(t) = i(t')$. That is, if t and t' have the same interpretation then they should be rewritten to the same normal form by $\rightarrow_D$ (in some sense, $\rightarrow_D$ then implements the interpretation).

Consider now an extended set of terms $T_A(X)$, where X is a set of variables, such that for any substitution $\theta : X \rightarrow A$ of elements in A for variables the resulting terms belong to T_A. Let us call the set of these assignments Θ. Under the assumptions on $\rightarrow_D$ above it then holds that if $i(\theta t) = i(\theta t')$ for all $\theta \in \Theta$, then $t \equiv_{D\Theta} t'$.

An equality relation $=_E$ is valid for the terms in $T_A(X)$ if, for all $t, t' \in T_A(X)$, holds that $t =_E t'$ implies that $i(\theta t) = i(\theta t')$ for all $\theta \in \Theta$. Thus, for valid equalities,

we have $t =_E t' \Rightarrow t \equiv_{D\Theta} t'$. Now assume that $\rightarrow_D$ is closed under replacement. Then $t \equiv_{D\Theta} t' \Rightarrow t \equiv_{D(\Theta\cup\Sigma)} t'$, where Σ is the set of the replacement functions $t \mapsto C[t]$ for all possible contexts C, even such that could be nondeterministically rewritten. By Theorem 21 $t \equiv_{D(\Theta\cup\Sigma)} t'$ yields $t \equiv_{\Theta\cup\Sigma} t'$. Thus, $t =_E t' \Rightarrow t \equiv_{\Theta\cup\Sigma} t'$ which 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 21 has the following corollary:

Corollary 22. *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 22 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 $x \text{ or } y \rightarrow x$, $x \text{ or } y \rightarrow y$. Consider again the term $x - x$. If this term is rewritten to 0, and we instantiate $x = 0 \text{ or } 1$, then the normal forms 1 and -1 of $(0 \text{ or } 1) - (0 \text{ or } 1)$ are lost. This is due to the non-commutation of the rule $x - x \rightarrow 0$ with the rules for *or* .

Another application of Theorem 21 concerns *weak head normal forms* (whnf's), which are λ-terms of the form $\lambda x.t$. In a lazy functional language, a whnf $\lambda x.t$ is not reduced even if t is reducible. This means that the deterministic reduction takes place according to a subrelation $\rightarrow_D^w$ of the full reduction relation $\rightarrow_D$. If $\rightarrow_D^w$ commutes with $\rightarrow$, then our theory can be applied directly to yield correctness of transformations contained in $\leftrightarrow_D^{w*}$. This is, however, a very restrictive equivalence since it does not allow transformations to take place under lambdas! We would, rather, like to be able to transform according to the full convertibility relation $\leftrightarrow_D^*$.

The latter can be obtained under natural conditions, provided that we consider two whnf's $\lambda x.t$ and $\lambda x.t'$ equivalent whenever they have the same extensional meaning, i.e., they define the same "function" (we do not presuppose any interpretation of lambda terms in some function space). This is a natural equivalence concept in typed programming languages, where the only means of "observing" a function-typed value is to apply it to an argument in order to obtain a printable value. In our setting, we can define $\lambda x.t$ and $\lambda x.t'$ to have the same extensional meaning whenever, for any term t'' which they both can be applied to, $S(\lambda x.t\,t'', f, \rightarrow) = S(\lambda x.t'\,t'', f, \rightarrow)$. We

then obtain, by considering the set of replacement functions $t \mapsto \lambda x.t\,t''$, for all terms t'' as above, the following corollary to Theorem 21:

Corollary 23. *If $\rightarrow_D \subseteq \rightarrow^*$, if $\rightarrow_D$ commutes with $\rightarrow$, and if $\rightarrow_D$ is closed under replacement, then $t \leftrightarrow_D^* t'$ implies that $\lambda x.t$ and $\lambda x.t'$ have the same extensional meaning.*

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 CRS rules, 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., [12, 25]. The results in this section can be seen as generalizations of some classical results to the nondeterministic case. Our theory is CRS based: thus, it covers also higher order programs.

First, we show a theorem about equivalence of abstract reduction systems (i.e., that they yield the same semantics for all elements). This theorem is based on *emulations* of reductions and will be applied to the reduction relations resulting from fold/unfold-transformations.

Definition 24. Let $\rightarrow_1$, $\rightarrow_2$ be binary relations over A, and let $f : A \rightarrow C$ be a monotone interpretation of both $\langle A, \rightarrow_1 \rangle$ and $\langle A, \rightarrow_2 \rangle$ into $\langle C, \sqsubseteq \rangle$. The $\rightarrow_1$-computation $\boldsymbol{a}$ is *emulated* by the $\rightarrow_2$-computation $\boldsymbol{a}'$ under f, if there exists a function $g : N \rightarrow N$ such that $f(\boldsymbol{a}_i) = f(\boldsymbol{a}'_{g(i)})$ for all $i \in N$, $g(0) = 0$, and $g(i) \rightarrow \infty$ when $i \rightarrow \infty$.

Proposition 25. *Let $\boldsymbol{a}$ be a $\rightarrow_1$-computation, and let $\boldsymbol{a}'$ be a $\rightarrow_2$-computation. If there exists a function $g : N \rightarrow N$ such that $\boldsymbol{a}_i = \boldsymbol{a}'_{g(i)}$ for all $i \in N$, $g(0) = 0$, and $g(i) \rightarrow \infty$ when $i \rightarrow \infty$, then $\boldsymbol{a}$ is emulated by the $\rightarrow_2$-computation $\boldsymbol{a}'$ under any f.*

Proof. Immediate. □

Lemma 26. *Let $f : A \rightarrow C$ be a monotone interpretation of both $\langle A, \rightarrow_1 \rangle$ and $\langle A, \rightarrow_2 \rangle$ into $\langle C, \sqsubseteq \rangle$. If the $\rightarrow_1$-computation $\boldsymbol{a}$ is emulated by the $\rightarrow_2$-computation $\boldsymbol{a}'$, then $\bigsqcup f(\boldsymbol{a}) = \bigsqcup f(\boldsymbol{a}')$.*

Proof. Standard result. □

Theorem 27. *Let $f : A \rightarrow C$ be a monotone interpretation of both $\langle A, \rightarrow_1 \rangle$ and $\langle A, \rightarrow_2 \rangle$ into $\langle C, \sqsubseteq \rangle$. Then $S(a, f, \rightarrow_1) = S(a, f, \rightarrow_2)$ for all $a \in A$ if the following holds:*

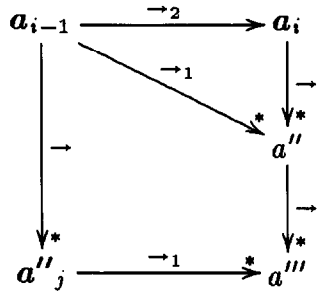

Fig. 6. Domination of a $\to_2$-computation by a computation emulating a $\to_1$-computation. The crucial part of the proof.

1. *every* $\to_1$*-computation* $\boldsymbol{a}$ *is emulated by a* $\to_2$*-computation* $em(\boldsymbol{a})$ *such that* $\bigsqcup f(\boldsymbol{a}) \in S(a, f, \to_1) \Rightarrow \bigsqcup f(em(\boldsymbol{a})) \in S(a, f, \to_2)$, *and*
2. *there exists a binary relation* $\to \subseteq \to_2$ *such that*:
   - $\to$ *commutes with* $\to_1$, *and*
   - *for all* $a, a' \in A$ *it holds that if* $a \to_2 a'$, *then there exists an* $a'' \in A$ *such that* $a' \to^* a''$ *and* $a \to_1^* a''$.

Proof. $S(a, f, \to_1) \subseteq S(a, f, \to_2)$: every $\to_1$-computation has an emulating $\to_2$-computation. By Lemma 26, their limits are equal. Thus, if $\bigsqcup f(\boldsymbol{a}) \in S(a, f, \to_1)$, then $\bigsqcup f(em(\boldsymbol{a})) = \bigsqcup f(\boldsymbol{a}) \in S(a, f, \to_2)$.

$S(a, f, \to_2) \subseteq S(a, f, \to_1)$: first we show, for any a-rooted $\to_2$-computation $\boldsymbol{a}$, the existence of a dominating a-rooted $\to_2$-computation $\boldsymbol{a}'$ and a $\to_1$-computation $\boldsymbol{a}''$ being emulated by $\boldsymbol{a}'$. We show, by induction over i, that for each $\boldsymbol{a}_i$ there exists a j such that $\boldsymbol{a}_i \to^* \boldsymbol{a}''_j : \boldsymbol{a}' = em(\boldsymbol{a}'')$ is then dominating $\boldsymbol{a}$. $i = 0$ is trivial. For $i > 0$, assume true for $i - 1$, and consider $\boldsymbol{a}_{i-1} \to_2 \boldsymbol{a}_i$. By assumption, we have $\boldsymbol{a}_{i-1} \to^* \boldsymbol{a}''_j$ for some j. Also, there exists an a'' such that $\boldsymbol{a}_i \to^* a''$ and $\boldsymbol{a}_{i-1} \to_1^* a''$. Then, by commutation, there exists an a''' such that $a'' \to^* a'''$ and $\boldsymbol{a}''_j \to_1^* a'''$. $\boldsymbol{a}''$ can thus be chosen so that, for some $k \geqslant j$, holds that $\boldsymbol{a}''_k = a'''$, and we have $\boldsymbol{a}_i \to^* \boldsymbol{a}''_k$. See Fig. 6.

It follows that $\bigsqcup f(\boldsymbol{a}) \sqsubseteq \bigsqcup f(\boldsymbol{a}')$. Now, consider any $u \in S(a, f, \to_2)$, and let $\boldsymbol{a}$ be a-rooted with $\bigsqcup f(\boldsymbol{a}) = u$. Then we have $\bigsqcup f(\boldsymbol{a}) = \bigsqcup f(\boldsymbol{a}') = \bigsqcup f(\boldsymbol{a}'')$. Do we have $\bigsqcup f(\boldsymbol{a}'') \in S(a, f, \to_1)$? If yes, then the result follows. Assume the opposite. Then there exists a $\to_1$-computation $\boldsymbol{a}'''$ strictly dominating $\boldsymbol{a}''$, with $\bigsqcup f(\boldsymbol{a}''') \in S(a, f, \to_1)$. But, by emulation of $\to_1$-computations, there then exists a $\to_2$-computation emulating $\boldsymbol{a}'''$ whose limit equals $\bigsqcup f(\boldsymbol{a}''')$. Thus, it strictly dominates $\boldsymbol{a}'$, and therefore also $\boldsymbol{a}$. This contradicts $\bigsqcup f(\boldsymbol{a}) \in S(a, f, \to_2)$. □.

We formulate fold/unfold-transformations for Combinatory Reduction Systems. As the semantics defined by a CRS R we consider the constructor tree semantics given by the monotone interpretation $[R]$ defined in Section 6.

Unfolding means to apply a rule $s' \to t'$ in a redex p in the RHS of the CRS rule $s \to t$, where $t/p = \theta(s')$ for some safe valuation θ, such that a new rule $s \to t[p \leftarrow \theta(t')]$

is formed. A CRS R containing these rules can now be transformed into a new CRS R' by replacing $s \to t$ with the new rule.

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

For simplicity, we state our results for the case where a single rule is folded or unfolded by a single rule. Unfolding always preserves the semantics, and having proved that unfolding by a single rule is correct a simple induction shows that it can be carried on arbitrarily. Folding is, however, not always correct. In the case of folding with a single rule it is sufficient to demand that a rule is not folded with itself. For folding with several rules, it is relatively straightforward to extend this condition to "restricted folding-unfolding" [11, 12]. Before we show the main theorems, we must however verify that some properties of first order substitutions carry over to valuations.

Lemma 28. *For all valuations ϕ, metaterms s,t and positions $p \in dom(t)$ holds that* $\phi(s[p \leftarrow t]) = \phi(s)[p \leftarrow \phi(t)]$.

Proof. The same lemma but for first order substitutions is stated in [15, Proposition 3.5]. The proof is by induction over the structure of s, and the extension to valuations is straightforward. □

Lemma 28 yields the following corollary, by choosing $t = s/p$ and considering the LHS and RHS at position p:

Corollary 29. *For all valuations ϕ, metaterms s and positions $p \in dom(t)$ holds that* $\phi(s)/p = \phi(s/p)$.

The following lemma also has a well-known counterpart for first order substitutions. It is straightforward to prove by structural induction:

Lemma 30. *For all valuations ϕ, metaterms s, and metasubstitutions $[x_i := t_i]$ such that no x_i occurs in $\phi(Z)$ for any metavariable Z in s, holds that* $\phi(s[x_i := t_i]) = \phi(s)[x_i := \phi(t_i)]$.

Here, we have written $[x_i := t_i]$ for the metasubstitution $[x_1 := t_1, \ldots, x_n := t_n]$. We will use this shorthand notation in the sequel. The following variation of Lemma 30, formulated for metasubstitutions, will also be used below:

Lemma 31. *If no y_j occurs in s, then* $(s[x_i := t_i])[y_j := u_j] = s[x_i := t_i[y_j := u_j]]$.

Lemma 32. *For any valuations ϕ and θ there exists a valuation $\phi\theta$ such that* $(\phi\theta)(t) = \phi(\theta(t))$ *for all metaterms t.*

Proof. The result is a simple generalization from first order substitutions to valuations. It is proved by structural induction on metaterms. All cases are straightforward,

except possibly for metaterms $Z(x_1,\ldots,x_n)$: let $\theta(Z)=\lambda(y_1,\ldots,y_n).t'$. Define $(\phi\theta)(Z)=\lambda(z_1,\ldots,z_n).(\phi(\theta(Z(z_1,\ldots,z_n))))$ where, for any i, z_i occurs in neither t' nor any $\phi(Z)$, for the metavariables Z occurring in t'. Then

$$\begin{aligned}(\phi\theta)(Z(t_1,\ldots,t_n)) &= \phi(\theta(Z(z_1,\ldots,z_n)))[z_i:=(\phi\theta)(t_i)]\\ &= \phi(t'[y_i:=z_i])[z_i:=\phi(\theta(t_i))] \quad \text{(by induction, def. of } \theta(Z))\\ &= \phi((t'[y_i:=z_i])[z_i:=\theta(t_i)]) \quad \text{(from Lemma 30)}\\ &= \phi(t'[y_i:=\theta(t_i)]) \quad \text{(from Lemma 31)}\\ &= \phi(\theta(Z(t_1,\ldots,t_n))). \qquad \square\end{aligned}$$

We need the following lemma in order to apply Theorem 27:

Lemma 33. *If R' is obtained from R by either folding or unfolding, then $[R']=[R]$.*

Proof. $[R]$ can be different from $[R']$ only if there is a constructor w.r.t. one of the CRSs which is not a constructor w.r.t. the other one. Neither folding nor unfolding change the left-hand sides of the rules. Thus, a constructor w.r.t. R is a constructor w.r.t. R' and vice versa. $\square$

Theorem 34. *If $s\to t\in R$ is unfolded into $s\to t[p\leftarrow\theta(t')]$ by $s'\to t'$, and if $\to_{\{s'\to t'\}}$ commutes with $\to_{\{s''\to t''\}}$ for any rule $s''\to t''$ in the resulting CRS R', then $S_{R'}(u)=S_R(u)$ for all terms u.*

Proof. We show that Theorem 27 applies, with $\to_1=\to_{R'}$, $\to_2=\to_R$, $f=[R'](=[R])$, and $\to=\to_{\{s'\to t'\}}$.

We first show that for any $\to_{R'}$-computation $\boldsymbol{u}'$ there exists an emulating $\to_R$-computation $em(\boldsymbol{u}')$. The nontrivial emulation is that of a $s\to t[p\leftarrow\theta(t')]$-reduction $u_i'\to_{R'}u_i'[p'\leftarrow\phi(t[p\leftarrow\theta(t')])]$. A two-step reduction in R is $u_i'\to_R u_i'[p'\leftarrow\phi(t)]$ (since $u_i'/p'=\phi(s)$) and $u_i'[p'\leftarrow\phi(t)]\to_R u_i'[p'\leftarrow\phi(t)][p'\cdot p\leftarrow(\phi\theta)(t')]$ (since $u_i'[p'\leftarrow\phi(t)]/(p'\cdot p)=\phi(t)/p=\phi(t/p)=\phi(\theta(s'))=(\phi\theta)(s')$). This reduction does indeed emulate the $\to_{R'}$-reduction since $u_i'[p'\leftarrow\phi(t)][p'\cdot p\leftarrow(\phi\theta)(t')]=u_i'[p'\leftarrow\phi(t)[p\leftarrow(\phi\theta)(t')]]=u_i'[p'\leftarrow\phi(t)[p\leftarrow(\phi(\theta(t'))]]=u_i'[p'\leftarrow\phi(t[p\leftarrow\theta(t')])]$. See Fig. 7.

Now assume that $\bigsqcup[R'](\boldsymbol{u}')\in S_{R'}(u)$. Will $\bigsqcup[R](\boldsymbol{u})\in S_R(u)$ follow? Yes, since any $\to_R$-redex in $\boldsymbol{u}$ but not in $\boldsymbol{u}'$ eventually will disappear, due to the infinite emulation ($s\to t$-redexes in particular). Thus, there can be no $\to_R$-computation strictly dominating $\boldsymbol{u}$.

We now show that whenever $v\to_R v'$, there exists a v'' such that $v\to_{R'}^* v''$ and $v'\to_{\{s'\to t'\}}^* v''$. Since $\to_{\{s'\to t'\}}$ commutes with $\to_{R'}$, this completes the proof. Either $v\to_{\{s''\to t''\}}v'$, where $s''\to t''$ is distinct from $s\to t$. Then $s''\to t''\in R'$, and we can choose $v''=v'$. Otherwise $v\to_{\{s\to t\}}v'$. Then, for some position p' and valuation ϕ, we have $v/p'=\phi(s)$ and $v'=v[p'\leftarrow\phi(t)]$. Furthermore, $\phi(t)/p=(\phi\theta)(s')$, and

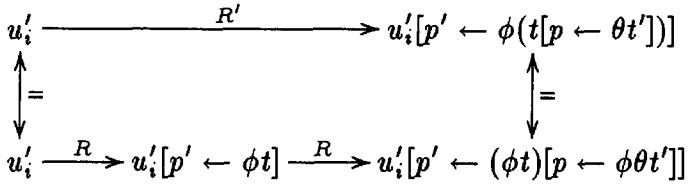

Fig. 7. Emulation of a computation in R' by a computation in R.

thus $v' \rightarrow_{\{s' \rightarrow t'\}} v''$ where $v'' = v[p' \leftarrow \phi\ (t)][p' \cdot p \leftarrow (\phi\theta)(t')]$. v'' is equal to $v[p' \leftarrow \phi(t[p \leftarrow \theta(t')])]$: thus, $v \rightarrow_{\{s \rightarrow t[p \leftarrow \theta(t')]\}} v''$, i.e., $v \rightarrow_{R'} v''$. □

It can be seen from the proofs of Theorems 34 and 27 that $S_{R'}(u) \subseteq S_R(u)$ always holds, regardless of any commutation properties.

We prove correctness of folding under the simplifying condition that a rule is not folded with itself. This corresponds to "restricted folding–unfolding" in [11, 12].

Theorem 35. *If $s \rightarrow t \in R$ is folded into $s \rightarrow t[p \leftarrow \theta(s')]$ by $s' \rightarrow t'$, where $s' \rightarrow t' \neq s \rightarrow t$, and if $\rightarrow_{\{s' \rightarrow t'\}}$ commutes with $\rightarrow_{\{s'' \rightarrow t''\}}$ for any rule $s'' \rightarrow t''$ in R, then $S_{R'}(u) = S_R(u)$ for all terms u.*

Proof. The proof is dual to the proof of Theorem 34. We show that Theorem 27 applies, with $\rightarrow_1 = \rightarrow_R$, $\rightarrow_2 = \rightarrow_{R'}$, $f = [R](=[R'])$, and $\rightarrow = \rightarrow_{\{s' \rightarrow t'\}}$. Note that $\rightarrow_{\{s' \rightarrow t'\}} \in \rightarrow_{R'}$, since $s' \rightarrow t'$ is not folded with itself.

First we show that for any $\rightarrow_R$-computation $\boldsymbol{u}$ there exists an emulating $\rightarrow_{R'}$-computation $\boldsymbol{u}'$. The nontrivial emulation is that of a $s \rightarrow t$-reduction $u_i \rightarrow_R u_i[p' \leftarrow \phi(t)]$. A two-step reduction in R' is $u_i \rightarrow_{R'} u_i[p' \leftarrow \phi(t[p \leftarrow \theta(s')])] \rightarrow_{R'} u_i[p' \leftarrow \phi(t[p \leftarrow \theta(s')])][p' \cdot p \leftarrow (\phi\theta)(t')]$. This reduction is indeed an emulation since $u_i[p' \leftarrow \phi(t[p \leftarrow \theta(s')])][p' \cdot p \leftarrow (\phi\theta)(t')] = u_i[p' \leftarrow \phi(t[p \leftarrow \theta(s')])][p' \cdot p \leftarrow \phi(\theta(t'))] = u_i[p' \leftarrow \phi(t[p \leftarrow \theta(s')][p \leftarrow \theta(t')])] = u_i[p' \leftarrow \phi(t[p \leftarrow \theta(t')])] = u_i[p' \leftarrow \phi(t[p \leftarrow (t/p)])] = u_i[p' \leftarrow \phi(t)]$. Here, we have used the identity $t[p \leftarrow t'][p \leftarrow t''] = t[p \leftarrow t'']$ which follows from [15, Proposition 3.1] and the fact that $t/p = \theta(t')$ when $s \rightarrow t$ is folded by $s' \rightarrow t'$ at p. $\bigsqcup[R'](\boldsymbol{u}') = \bigsqcup[R](\boldsymbol{u})$ follows.

Now assume that $\bigsqcup[R](\boldsymbol{u}) \in S_R(u)$. Will $\bigsqcup[R'](\boldsymbol{u}') \in S_{R'}(u)$ follow? Yes, since any $\rightarrow_{R'}$-redex in $\boldsymbol{u}'$ but not in $\boldsymbol{u}$ eventually will disappear, due to the infinite emulation ($s \rightarrow t[p \leftarrow \theta(s')]$-redexes in particular). Thus, there can be no $\rightarrow_{R'}$-computation strictly dominating $\boldsymbol{u}'$.

We now show that whenever $v \rightarrow_{R'} v'$, there exists a v'' such that $v \rightarrow_R^* v''$ and $v' \rightarrow_{\{s' \rightarrow t'\}}^* v''$. Since $\rightarrow_{\{s' \rightarrow t'\}}$ commutes with $\rightarrow_R$, this completes the proof. Either $v \rightarrow_{\{s'' \rightarrow t''\}} v'$, where $s'' \rightarrow t''$ is distinct from $s \rightarrow t$. Then $s'' \rightarrow t'' \in R$, and we can choose $v'' = v'$. Otherwise $v \rightarrow_{\{s \rightarrow t[p \leftarrow \theta(s')]\}} v'$. Then, for some position p' and valuation ϕ, we have $v/p' = \phi(s)$ and $v' = v[p' \leftarrow \phi(t[p \leftarrow \theta(s')])]$. Furthermore, $\phi(t[p \leftarrow \theta(s')])/p = (\phi\theta)(s')$, and thus $v' \rightarrow_{\{s' \rightarrow t'\}} v''$, where $v'' = v[p' \leftarrow \phi\ (t)][p' \cdot p$

$\leftarrow(\phi\theta)(t')]$, which equals $v[p' \leftarrow \phi\ (t)]$ since $\phi(t)/p = (\phi\theta)(t')$ due to the folding of $s \rightarrow t$ at p by $s' \rightarrow t'$. Thus, $v \rightarrow_{\{s \rightarrow t\}} v''$, i.e., $v \rightarrow_R v''$. □

It can be seen from the proofs of Theorems 35 and 27 that $S_{R'}(u) \supseteq S_R(u)$ always holds, regardless of any commutation properties. Thus, folding (without commutation) may increase the number of possible results for a nondeterministic computation, whereas unfolding may decrease this number.

A particular case where Theorems 34 and 35 apply is when the CRS rule $s' \rightarrow t'$ to be folded (or unfolded) with is orthogonal to all other rules in R: then it is also orthogonal to all rules in R', and thus $\rightarrow_{\{s' \rightarrow t'\}}$ commutes with both $\rightarrow_R$ and $\rightarrow_{R'}$.

What makes the application of Theorem 27 so straightforward in the proof of Theorem 35 is the fact that the rule being folded with is present in the transformed system. This is also true for restricted folding-unfolding. We leave it as an exercise to the reader to extend Theorem 35 to restricted folding–unfolding.

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 **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.

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

$$a.P|x.Q \rightarrow a.(P|x.Q), \tag{2}$$

$$c!y.P|c?x.Q \rightarrow P|Q\{x/y\}. \tag{3}$$

The first group of rules (2) 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 (3) are communication rules. P, Q, x, y are variables above: thus, the rules (4) are pure term rewriting rules whereas the rules (4) are akin to β-reduction. Furthermore, we introduce auxiliary term rewriting rules, viz.:

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

$$P|(Q|R) \rightarrow (P|Q)|R, \tag{5}$$

$$P|Q \rightarrow Q|P. \tag{6}$$

They express equivalence modulo AC for parallel composition. (Cf. "chemical abstract machines" [7].) Their permutative nature gives rise to cyclic, infinite computa-

tions which are in some sense artificial. However, these computations will be strictly dominated whenever progress can be made through some of the first rules. Thus, they will not contribute to the semantics.

The rules (2)–(6) can easily be put into CRS format ((3) obtains the format, keeping the infix notation, $c!Y.P|c?[x]Q(x) \rightarrow P|Q(Y)$). With a constructor tree semantics according to Section 6, the "values" of type *Proc* are formed from the constructors ".", **0** and values of type *Event*. The constructor tree semantics thus, in this case, yields the set of possible traces (n.b. of fair computations) as the semantics for a process typed expression.

This process language can be enriched with a computational component described by an orthogonal CRS. For instance, a higher order functional language can be added, which is defined by, say, the following CRS rules:

- Delta (ground) rules describing strict operations, like for instance addition "+" of integers:

$$0+0 \rightarrow 0$$
$$0+1 \rightarrow 1$$
$$\vdots$$

- a conditional *if*, defined by the first order TRS rules:

$$if(\mathbf{true}, x, y) \rightarrow x$$
$$if(\mathbf{false}, x, y) \rightarrow y$$

- (Possibly) recursive definitions, defined by rules of the form

$$f(x_1, \ldots, x_n) \rightarrow t$$

  where the function names f are selected distinct from other function symbols in the system,
- β-reduction according to (1).

It is easy to verify that these rules do form an orthogonal CRS (indeed, it is even left-normal, which means that a leftmost–outermost reduction strategy is normalizing [22]). Furthermore, all these rules will be orthogonal to the rules (2)–(6) defining the semantics for the process primitives. Thus, we can perform fold/unfold-transformations with respect to any of these rules, according to Theorems 34 and 35, without changing the semantics for the combined language.

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 commutation of the reduction relations makes the computational part "referentially transparent" even when the environment is non-deterministic. 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 expressions. 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 CRSs 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 is to support the design and implementation of lazy recursive languages with process primitives. In order to do this, two things must be observed: first, lazy languages use a *normalizing* reduction strategy rather than a cofinal one. We believe that for such languages the correctness can be proved for combined reduction strategies, employing a normalizing (rather than cofinal) strategy for certain terms. Second, laziness also implies sharing of already computed results (call-by-need rather than call-by-name), which must be modeled in a formalism such as explicit substitutions [1] or graph reduction systems [2, 6].

The difference between call-by-need and call-by-name is indeed crucial and must be taken into account. Returning to our initial example in Section 1, with $f(x)=x-x$, the unfolding $f(0\ or\ 1)\rightarrow(0\ or\ 1)-(0\ or\ 1)$ is indeed correct for call-by-name since then the function is adequately described by the literally translated rewrite rule $f(x)\rightarrow x-x$. In the case of call-by-need, however, the sharing of the argument must be modeled. Using explicit substitutions, the function f is now described by the rewrite rule $f(x)\rightarrow(y-y)[y\leftarrow x]$, where $[y\leftarrow x]$ is a term (rather than a metaoperation on terms) representing the explicit substitution of x for y in the term to the left (an additional set of rewrite rules describes how and when the explicit substitution can be applied; to properly model call-by-need, these should defer the substitution to take place until x is reduced to whnf). y can be thought of as a reference pointing to the closure for the argument. In our example, the correct unfolding for call-by-need is then $f(0\ or\ 1)\rightarrow(y-y)[y\leftarrow(0\ or\ 1)]$. When this expression is evaluated, 0 *or* 1 must be executed first to yield a whnf, before the result can be substituted for y. This yields the correct semantics for the unfolded term.

The correctness of folding was proved under the condition that a rule would not be used to fold itself, which corresponds to the restricted folding-unfolding condition by Courcelle [11, 12]. As observed by Sands [32], this condition is quite restrictive since it essentially prohibits the formation of new recursive definitions. This limits the applicability to program transformations and partial evaluation. We are, however, quite confident that the techniques developed in this paper can be applied to prove the correctness of certain combined unfold/fold-transformations, introducing new recursive definitions, which are common in these applications.

Acknowledgements

I want to thank Vincent van Oostrom and Jan Willem Klop for their kind assistance regarding higher order rewriting formalisms, and the anonymous referees for their valuable comments.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitutions, in: Proc. 7th Ann. ACM Symp. on Principles of Programming Languages, San Francisco, 1990, pp. 31–46.
[2] Z.M. Ariola, Arvind, Properties of a first-order functional language with sharing, Theoret. Comput. Sci. 146 (1995) 69–108.
[3] J. Armstrong, R. Virding, M. Williams, Concurrent Programming in ERLANG, Prentice-Hall, Englewood Cliffs, NJ, 1993.
[4] A. Arnold, P. Naudin, M. Nivat, On semantics of nondeterministic recursive program schemes, in: M. Nivat, J.C. Reynolds (Eds.), Algebraic Methods in Semantics, Cambridge University Press, Cambridge, 1985, Ch. 1, pp. 1–33.
[5] H.P. Barendregt, The Lambda Calculus – Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, Amsterdam, 1981.
[6] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, Term graph rewriting, in: Proc. PARLE'87, vol. 2, Lecture Notes in Computer Science, vol. 259, Springer, Berlin, 1987, pp. 149–158.
[7] G. Berry, G. Boudol, The chemical abstract machine, Theoret. Comput. Sci. 96 (1992) 217–248.
[8] G. Boudol, Computational semantics of term rewriting systems, in: M. Nivat, J.C. Reynolds (Eds.) Algebraic Methods in Semantics, Cambridge University Press, Cambridge, 1985, Ch. 5, pp. 170–236.
[9] R.M. Burstall, J. Darlington, A transformation system for developing recursive programs, J. Assoc. Comput. Mach. 24(1) (1977) 44–67.
[10] B. Courcelle, Infinite trees in normal form and recursive equations having a unique solution, Math. System Theory 13 (1979) 131–180.
[11] B. Courcelle, Equivalence and transformations of regular systems, Theoret. Comput. Sci. 42 (1986) 1–122.
[12] B. Courcelle, Recursive applicative program schemes, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier 1990, Ch. 9, pp. 459–492.
[13] N. Dershowitz, S. Kaplan, D.A. Plaisted, Rewrite, rewrite, rewrite, rewrite, rewrite,..., Theoret. Comput. Sci. 83 (1) (1991) 71–96.
[14] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, J. Assoc. Comput. Math. 24 (1) (1977) 68–95.
[15] G. Huet, Confluent reductions: abstract properties and applications to term rewriting systems, J. Assoc. Comput. Mach. 27 (4) (1980) 797–821.
[16] G. Heut, B. Lang, Proving and applying program transformations expressed with second-order patterns, Acta Inform. 11 (1) (1978) 31–55.
[17] J. Hughes, A. Moran, Making choices lazily, in: Proc. 7th Conf. on Functional Programming Languages and Computer Architecture, ACM Press, New York, 1995.
[18] T. Johnsson, Lambda lifting: transforming programs to recursive equations, in: J.-P. Jouannaud (Ed.), Proc. Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, vol. 201, Nancy, France, 1985, Springer, Berlin, 1985, pp. 190–203.
[19] N.D. Jones, Challenging problems in partial evaluation and mixed computation, in: D. Bjørner, A.P. Ershov, N.D. Jones (Eds.), Partial Evaluation and Mixed Computation, Proc. IFIP TC2 Workshop, Gammel Avernas, Demark, 1987, North-Holland, Amsterdam, 1988, pp. 1–14.
[20] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Hertfordshire, UK, 1993.
[21] R. Kennaway, J.W. Klop, R. Sleep, F.-J. de Vries, Transfinite reductions in orthogonal term rewriting systems, Information and Comput. 119 (1) (1995) 18–38.

[22] J.W. Klop, Combinatory Reduction Systems, Ph.D. thesis, CWI, Amsterdam, 1980. Mathematical Centre Tracts Nr. 127.
[23] J.W. Klop, Term rewriting systems, in: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 2, Oxford University Press, Oxford, 1992, Ch. 1, pp. 1–116.
[24] J.W. Klop, V. van Oostrom, F. van Raamsdonk, Combinatory reduction systems: introduction and survey, Theoret. Comput. Sci. 121 (1993) 279–308.
[25] L. Kott, Unfold/fold program transformations, in: M. Nivat, J.C. Reynolds (Eds.), Algebraic Methods in Semantics, Cambridge University Press, 1985, Ch. 12, pp. 411–434.
[26] J.-J. Lévy, An algebraic interpretation of the $\lambda\beta k$-calculus; and an application of a labelled λ-calculus, Theoret. Comput. Sci. 2 (1) (1976) 97–114.
[27] M. Nivat, On the interpretation of recursive polyadic program schemes, Symp. Math. 15 (1975) 255–281.
[28] S. Peyton Jones, A.J. Gordon, S. Finne, Concurrent Haskell, in: Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, 1996, pp. 295–308.
[29] J.-C. Raoult, J. Vuillemin, Operational and semantic equivalence between recursive programs, J. Assoc. Comput. Mach. 2 (1) (1976) 97–114.
[30] J.H. Reppy, CML: a higher order concurrent language, in: Proc. SIGPLAN'91 Conf. on Programming Language Design and Implementation, 1991, pp. 293–305.
[31] D. Sahlin, Partial evaluation of AKL, in: Proc. 1st Internat. Workshop on Concurrent Constraint Programming, Cà Dolfin, Venice, 1995.
[32] D. Sands, Total correctness by local improvement in the transformation of functional programs, ACM Trans. Program. Lang. Syst. 18 (2) (1996) 175–234.
[33] H. Søndergaard, P. Sestoft, Referential transparency, definiteness and unfoldability, Acta Inform. 27 (1990) 505–517.
[34] B. Thomsen, L. Leth, A. Giacalone, Some Facile Chemistry, Tech. Rep. ECRC-92-14, European Computer-Industry Research Centre, 1992.
[35] D.A. Turner (Ed), Research Topics in Functional Programming, Addison-Wilsey, Reading, MA, 1989.
[36] V. van Oostrom, Private communication.
[37] C.P. Wadsworth, The relation between computational and denotational properties for Scott's D_∞-models of the lambda calculus, SIAM J. Comput. 5 (3) (1976) 488–521.
[38] C.P. Wadsworth, Approximate reduction and lambda calculus models, SIAM J. Comput. 7 (3) (1978) 337–356.