A Fully Abstract Semantics for Constructor Systems*

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. Constructor-based term rewriting systems are a useful subclass of TRS, in particular for programming purposes. In this kind of systems constructors determine a universe of values, which are the expected output of the computations. Then it would be natural to think of a semantics associating each expression to the set of its reachable values. Somehow surprisingly, the resulting semantics has poor properties, for it is not compositional nor fully abstract when non-confluent systems are considered. In this paper we propose a novel semantics for expressions in constructor systems, which is compositional and fully abstract (with respect to sensible observation functions, in particular the set of reachable values for an expression), and therefore can serve as appropriate basis for semantic based analysis or manipulation of such kind of rewrite systems.

1 Introduction

Constructor based term rewriting systems (or simply *constructor systems*, CS in short) are an important subclass of term rewriting systems (TRS). The use of CS for programming has been frequently connected to the requirement of confluence. But by these days many proposals (see e.g.[13,3,10,9,11]) drop the requirement of confluence and/or termination.

On the other hand, it is widely accepted that an adequate semantics constitutes an excellent companion to any programming language. In the case of CS, an 'obvious' notion of semantics comes from defining the denotation of an expression e as the set of values reachable from e by rewriting. The notion of 'values' could be made concrete in different manners: constructor terms, outer constructor part of expressions or normal forms. Two questions arise:
– Is the semantics compositional? In our case: is the semantics of an expression determined by the semantics of its subexpressions?
– Does it capture observational equivalence? That is: for two semantically equivalent expressions e, e', is it ensured that we will *observe* the same behavior when e, e' are put in the same context? This depends on a criterion of what can be

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), Promesas-CAM (S-0505/TIC/0407) and STAMP (TIN2008-06622-C03-01/TIN).

R. Treinen (Ed.): RTA 2009, LNCS 5595, pp. 320–334, 2009.

observed from an expression. In the constructor discipline point of view, one is mostly interested again in observing which constructor terms (or outer stable constructor part) can be reached by rewriting.

Somehow surprisingly, the answer to both questions is negative for the 'obvious' semantics:

Example 1. Consider the constructors a, b, c, d and the non-confluent program

$$f(c(X)) \to d(X, X) \quad choice(X, Y) \to X \quad choice(X, Y) \to Y$$

The expressions $e \equiv c(choice(a, b))$ and $e' \equiv choice(c(a), c(b))$ reach by rewriting exactly the same constructor values, namely $c(a)$ and $c(b)$. However, this does not ensure that e, e' behave the same when put in the same context. For instance, $f(e)$ can be rewritten to the constructor values $d(a, a), d(a, b), d(b, a), d(b, b)$ while $f(e')$ only to $d(a, a)$ and $d(b, b)$. More in general, this works starts by remarking that *knowing the constructor values of an expression e is not enough information to know the constructor values of $\mathcal{C}[e]$ for any given context $\mathcal{C}$*. The same example shows that the remark remains true if we replace 'constructor value' by 'normal form' or 'outer constructor part'. Using standard terminology (see Sect. 4 for definitions) all those semantics are not compositional, sound nor fully abstract.

The aim of our work can be made clear now: to define a semantics for CS that is fully abstract (compositionality and soundness will come along the way) wrt the observability criterion of reachable constructor terms.

Our starting insight is that, to recover compositionality, the semantics must not collect a *flat* set of reachable values, like is $\{c(a), c(b)\}$ for $c(choice(a, b))$, but rather a more structured and 'packaged' representation, where constructors can be applied to sets, as to reflect more appropriately the matching capabilities of expressions. In our example, and disregarding for the moment some technical details, the denotation of $c(choice(a, b))$ will be the singleton 'package' $\{c(\{a, b\})\}$, reflecting the fact that $c(choice(a, b))$ can match $c(X)$ without reducing $choice(a, b)$, while the denotation of $choice(c(a), c(b))$ will be the two-element package $\{c(\{a\}), c(\{b\})\}$. Technically, things will be a bit more complicated (see Sect. 3), in particular due to the possibility of non-termination, that will require expressing some kind of *partial* values in the semantics.

Related work. Not too much attention has been paid to the issue of semantics of TRS, at least when compared to the huge amount of research in the fields of TRS and of semantics of programming languages in general. There are nevertheless some works to be mentioned.

In [7], Boudol develops a deep theory of the space of computations of left-linear TRS and provides a computational semantics based on continuous algebras. However, his semantics still associates an expression with a flat set of (possibly infinite) values, thus presenting the problems of our Ex. 1. Moreover, [6,16] demonstrate that there are problems with achieving full abstraction for non-terminating non-deterministic systems, if the semantics is based on fixpoints and infinite (limit) values (our semantics will avoid them). In [2] a compositional semantics for conditional TRS is presented. Compositionality is understood in

a different sense, related to the issue of joining programs. In addition, the considered programs are canonical (confluent and terminating). In [1], an abstract diagnosis scheme for functional programs modeled as TRS is developed, based on some notions of semantics that again collect results of individual computations. The semantics characterization of narrowing given in [12] includes a semantics for TRS, but most of the interesting results are for confluent ones. On the other hand, the cited papers give a more general treatment of variables, which have a passive role in our paper, behaving almost as constants.

With respect to the nesting of sets inside constructor symbols, a similar idea appears in [4], to improve the efficiency of functional logic computations, in [8] as part of the design of a functional programming implementation of functional logic languages, and in [14] as a mean for programming with non-determinism in a Haskell-like ambient. All these works are much more oriented to practice, far from the aims and results of our present work. Moreover, the setting is not the same: functional logic programming for the two first (with a *call-time choice* semantics [10], having essential differences with standard rewriting) and functional programming for the last one. In [5], sets of reachable values for CS are computed; however, only topmost constructor symbols are collected and furthermore systems must be confluent and terminating.

The rest of the paper is organized as follows. Sect. 2 contains some preliminaries about TRS. Sect. 3 is the technical core of the paper, where our semantics is defined and many strong properties are proved. In Sect. 4 we discuss in detail the question of full abstraction. Finally Sect. 5 discusses potential uses of our semantics and outlines future work. Omitted proofs can be found at `http://gpd.sip.ucm.es/fraguas/papers/rta2009Long.pdf`.

2 Preliminaries

We assume a first order signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *constructor* and *function* symbols resp., all them with associated arity. We write DC^n (FS^n resp.) for the set of constructor (function) symbols of arity n, and also Σ^n for any symbol of the signature of arity n. We also assume a numerable set $\mathcal{V}$ of variables. As usual notations we write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables. The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in \Sigma^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to DC^n (so $CTerm \subseteq Exp$).[1] We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects along the paper.

We consider also the extended signature $\Sigma_\perp = \Sigma \cup \{\perp\}$, where $\perp$ is a new 0-arity constructor symbol that stands for the undefined value. Over this signature

[1] We use the terminology Exp (for general expressions) instead of the more usual $Term$ in order to highlight the syntactic (and semantic) difference with $CTerm$ (data values).

we define the sets $Exp_\perp$ and $CTerm_\perp$ of *partial* expressions and c-terms resp. The intended meaning is that Exp and $Exp_\perp$ stand for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ and $CTerm_\perp$ stand for data terms representing total and partial values resp. The *shell* $|e|$ of an expression e represents its outer constructed part and is defined as: $|X| = X$; $|c(e_1, \ldots, e_n)| = c(|e_1|, \ldots, |e_n|)$; $|f(e_1, \ldots, e_n)| = \perp$. *Substitutions* $\theta \in Subst$ are mappings $\theta : \mathcal{V} \to Exp$, that extend naturally to $\theta : Exp \to Exp$.

One-hole contexts are defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in \Sigma^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

The approximation ordering $\sqsubseteq$ is defined on expressions as the least partial ordering satisfying: *i)* $\perp \sqsubseteq e$ for all $e \in Exp_\perp$, and *ii)* $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\perp, \mathcal{C} \in Cntxt$.

A *constructor system* $\mathcal{P}$ (*CS*, also called *program* along this paper) is a set of rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$, $var(e) \subseteq var(\overline{t})$, and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. Given a program $\mathcal{P}$, its associated rewrite relation $\to_\mathcal{P}$ is defined as: $\mathcal{C}[l\theta] \to_\mathcal{P} \mathcal{C}[r\theta]$ for any context $\mathcal{C}$, rule $l \to r \in \mathcal{P}$ and $\theta \in Subst$. We write $\overset{*}{\to}_\mathcal{P}$ for the reflexive and transitive closure of the relation $\to_\mathcal{P}$. In the following, we will usually omit the reference to $\mathcal{P}$.

3 A Semantics for CS

In this section we present our proposed semantics, which has a logic flavor as it is based on a proof calculus. The use of proof calculi to specify the semantics of rewriting formalisms is not unfrequent. Two well-known cases correspond to the frameworks of rewriting logic [15] and CRWL [10]. We have been inspired by the philosophy of the latter, according to the following roadmap:

- We first identify the 'finite pieces' of which the denotation of expressions should be made of. These will be the *s-cterms* introduced in 3.1, capturing technically the idea of 'packaging sets below constructor' mentioned in Sect. 1.
- Then, we devise in Sect. 3.2 a proof calculus able to prove statements of the form $e \twoheadrightarrow st$ expressing that st is a finite approximation of the denotation of e. Technically, expressions will be generalized to *s-expressions*.
- The proof calculus induces a natural notion of denotation of an expression: the set of its provable approximations. Working with finite approximations makes it unnecessary to use a background of cpo's and powerdomains. This was found greatly convenient in the CRWL framework, and it is even more so in our case, where recursive nestings of constructors and sets occur. Moreover, it is known ([6,16]) that an approach based on semantic domains with infinite (limit) elements and using fixpoint techniques has technical limitations.
- If the proof calculus is designed to have a 'compositional' aspect, then one can expect compositionality of the resulting semantics, and the proof calculus is in itself a great aid to prove it. We have pursued this design principle in our proof calculus; as a result, and we have been able to prove compositionality and other relevant properties of the semantics (Sect. 3.2).

• Now, since our aim is to develop a new semantics for standard rewriting, it is essential to show that our semantics is indeed related to rewriting: this is done in Sect. 3.3 by correctness and completeness results.
• Finally, with all the previous results and an extra little effort, we are able to prove full abstraction of our semantics (Sect. 4).

3.1 SCTerms: The Pieces of the Semantics

In this section we define new syntactic notions (of expressions, cterms, etc) in order to pack different values coming from non deterministic reductions at the syntactic level, by introducing sets in the corresponding syntax. Values become *s-cterms* that must be defined in mutual recursion with *elemental s-cterms*:

$$\begin{array}{l} ESCTerm \ni est ::= X \mid c(st_1, \ldots, st_n) \\ \qquad \text{for } X \in \mathcal{V}, c \in DC^n, st_1, \ldots, st_n \in SCTerm \\ SCTerm \ni st ::= \emptyset \mid \{est_1, \ldots, est_n\} \\ \qquad \text{for } n > 0, est_1, \ldots, est_n \in ESCTerm \end{array}$$

Thus, an s-cterm is a *finite* set of elemental s-cterms, that are variables or constructors applied to s-cterms. The aim of these values is to capture the reduction of a non deterministic expression like $c(choice(a, b))$ into the single value $\{c(\{a, b\})\}$. With the same idea, but allowing also function symbols, we define *elemental s-expressions* and *s-expressions* as:

$$\begin{array}{l} ESExp \ni ese ::= X \mid h(se_1, \ldots, se_n) \\ \qquad \text{for } X \in \mathcal{V}, h \in \Sigma^n, se_1, \ldots, se_n \in SExp \\ SExp \ni se ::= \emptyset \mid \{ese_1, \ldots, ese_n\} \\ \qquad \text{for } n > 0, ese_1, \ldots, ese_n \in ESExp \end{array}$$

In the inductive definitions of $SCTerm$ and $SExp$, the base case $\emptyset$ could be hidden in the brace notation $\{est_1, \ldots, est_n\}$ just permitting $n = 0$ (in fact, we will do it sometimes). We prefer to emphasize the presence of $\emptyset$, playing the role of the undefined value (similar to $\bot$ for Exp in Sect. 2). Therefore s-cterms and s-expressions should be understood as *partial*. *Total* s-expressions and s-cterms would not use $\emptyset$, but they do not play any significant role in the following.

We can flatten an s-expression se to obtain the set $flat(se)$ of partial expressions "contained" in it: $flat(\emptyset) = \{\bot\}$ and $flat(se) = \bigcup_{ese \in se} flat(ese)$ if $se \neq \emptyset$, where the flattening of elemental s-expressions is defined as: $flat(X) = \{X\}$ and $flat(h(se_1, \ldots, se_n)) = \{h(e_1, \ldots, e_n) | e_i \in flat(se_i) \text{ for } i = 1..n\}$. Notice that $flat(se)$ is always non-empty.

The set $SSubst$ of *s-substitutions* consists of mappings $\sigma : \mathcal{V} \to SExp$ having a finite domain, where $dom(\sigma) = \{X \mid \sigma(X) \neq \{X\}\}$. Notice that s-substitutions replace variables by s-expressions (which are sets), and some care must be taken when extending s-substitutions to $ESExp$ and $SExp$:

$$\begin{array}{ll} \sigma : ESExp \to SExp & \sigma : SExp \to SExp \\ X\sigma = \sigma(X) & \{ese_1, \ldots, ese_n\}\sigma = \bigcup_{i \in \{1..n\}} ese_i\sigma \\ h(\overline{se})\sigma = \{h(\overline{se\sigma})\} & \end{array}$$

The set $SCSubst$ of *s-csubstitutions* consists of mappings $\sigma : \mathcal{V} \to SCTerm$ with a finite domain, that extend to $ESCTerm$ and $SCTerm$ analogously to the case of s-substitutions. One hole (elemental) *s-contexts* are defined as:

$$sCntxt \ni s\mathcal{C} ::= [\,] \mid \{\ldots, h(\ldots, s\mathcal{C}, \ldots), \ldots\} \quad \text{with } h \in \Sigma \text{ and } s\mathcal{C} \in sCntxt$$

The application of a context to an s-expression is defined in the natural way. Notice that s-contexts allow the hole to be only in the place of a sub-s-expression. For example, the possible s-contexts of $\{Y, c(\{X\})\}$ are $[\,]$ and $\{Y, c([\,])\}$, but not $\{[], c(\{X\})\}$ nor $\{Y, []\}$.

The preorder $\sqsubseteq$ is defined for s-expressions as the least preorder satisfying: $se \sqsubseteq se'$ if $\forall ese \in se.\exists ese' \in se'$ such that $ese \sqsubseteq ese'$, where for elemental s-expressions $\sqsubseteq$ is defined as the least preorder such that: $X \sqsubseteq X$ for any $X \in \mathcal{V}$ and $h(se_1, \ldots, se_n) \sqsubseteq h(se'_1, \ldots, se'_n)$ iff $se_i \sqsubseteq se'_i$ for $i = 1..n$. For s-substitutions, the preorder is defined as $\sigma \sqsubseteq \sigma'$ if $\sigma(X) \sqsubseteq \sigma'(X)$ for all $X \in \mathcal{V}$.

Programs remain as defined in Sect. 2. The proof calculus of the next section needs to use function rules transformed into the new syntactical framework of s-expressions. For this purpose we define the transformation of $e \in Exp$ into an s-expression $\tilde{e} \in SExp$ as: $\tilde{\bot} = \emptyset$; $\tilde{X} = \{X\}$ for any $X \in \mathcal{V}$; $\widetilde{h(e_1, \ldots, e_n)} = \{h(\tilde{e_1}, \ldots, \tilde{e_n})\}$, with $h \in \Sigma^n$. The transformation $\tilde{\mathcal{C}}$ of a context $\mathcal{C}$ is defined in the natural way, so that it verifies $\widetilde{\mathcal{C}[e]} = \tilde{\mathcal{C}}[\tilde{e}]$. On the other hand, $\tilde{\sigma}$ is defined as $\tilde{\sigma}(X) = \widetilde{\sigma(X)}$, for $\sigma \in Subst$.

3.2 A Proof Calculus

Our goal in this section is to devise a proof calculus that specifies which $SCTerm$s correspond to a given expression under a given CS. Inspired by the *CRWL* proof calculus of [10], to achieve a compositional aspect of the calculus expressions are evaluated in an innermost way, and the use of any transitivity rule is avoided. By the use of partial s-cterms as values, the 'compositional' innermost procedure of the calculus does not enforce strictness of functions, which is essential to achieve completeness of our semantics wrt term rewriting even for non-terminating CS.

Besides, during parameter passing the variables in the program rules will be instanciated with partial s-cterms. Then it is possible to end up evaluating expressions with some $SCTerm$ "inside" (as a subexpression), even when starting the computation from an ordinary $e \in Exp$. So, instead of dealing only with expressions from Exp, our calculus will compute the partial s-cterms corresponding to any given partial s-expression. Finally, the mapping $\tilde{\_}$ will be used in combination with our logic to get the $SCTerm$s corresponding to a given Exp.

To be precise, our proof calculus will prove reduction statements of the form $se \twoheadrightarrow st$ with $se \in SExp$ and $st \in SCTerm$, expressing that st represents a finite approximation to one of the possible structured sets of values for se.

The calculus is presented in Fig. 1. Rule E (empty) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) work with singleton sets and allow us to reduce any variable to itself, and to decompose the evaluation of

$$\textbf{(E)}\quad se \twoheadrightarrow \emptyset \qquad \textbf{(RR)}\quad \{X\} \twoheadrightarrow \{X\} \quad \text{if } X \in \mathcal{V}$$

$$\textbf{(DC)}\quad \frac{se_1 \twoheadrightarrow st_1 \ \ldots \ se_n \twoheadrightarrow st_n}{\{c(se_1,\ldots,se_n)\} \twoheadrightarrow \{c(st_1,\ldots,st_n)\}} \quad \text{if } c \in CS$$

$$\textbf{(More)}\quad \frac{se \twoheadrightarrow st_1 \ \ldots se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \ldots \cup st_n}$$

$$\textbf{(Less)}\quad \frac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{\{ese_1,\ldots,ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m} \quad \begin{array}{l}\text{if } n \geq 2, m > 0, \text{ for any} \\ \{esa_1,\ldots,esa_m\} \subseteq \{ese_1,\ldots,ese_n\}\end{array}$$

$$\textbf{(ROR)}\quad \frac{se_1 \twoheadrightarrow \tilde{p_1}\theta \ \ldots \ se_n \twoheadrightarrow \tilde{p_n}\theta \ \ \tilde{r}\theta \twoheadrightarrow st}{\{f(se_1,\ldots,se_n)\} \twoheadrightarrow st} \quad \begin{array}{l}\text{if } (f(p_1,\ldots,p_n) \to r) \in \mathcal{P} \\ \theta \in SCSubst\end{array}$$

Fig. 1. A proof calculus for constructor systems

a constructor-rooted elemental s-expression. Rule More allows us to compute more than one value for an s-expression, and to collect these values together. Rule Less allows us to discard some elemental s-expressions from the s-expression under evaluation. Finally rule ROR (run-time[2] outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of an $SCSubst\ \theta$) and then reduce the instantiated right-hand side. The use of $SCSubst$s is fundamental to get the exact behaviour of term rewriting, because then the branching information associated to the computation of each $\tilde{p_i}\theta$ is not lost in some kind of flattening to a set of c-terms, but kept into the structured representation of $SCTerm$s.

We write $\mathcal{P} \vdash se \twoheadrightarrow st$ to express that $se \twoheadrightarrow st$ is derivable in our calculus under the CS $\mathcal{P}$. The *denotation* of an s-expression se under $\mathcal{P}$ is defined as $[\![se]\!]^{\mathcal{P}} = \{st \in SCTerm \mid \mathcal{P} \vdash se \twoheadrightarrow st\}$. In the following we will usually omit the reference to $\mathcal{P}$.

Example 2. Consider the CS of Ex. 1. We can use our calculus to prove the statement $f(c(\widetilde{choice}(a,b))) \twoheadrightarrow \widetilde{d(a,b)}$ (some steps have been omitted for the sake of conciseness, and *choice* is abbreviated to *ch*):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\,\text{DC} \quad \dfrac{}{\{b\} \twoheadrightarrow \emptyset}\,\text{E}}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{a\}}\,\text{ROR} \quad \dfrac{\ldots}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{b\}}\,\text{ROR}}{\{ch(\{a\},\{b\})\} \twoheadrightarrow \{a,b\}}\,\textsc{More}}{\{c(\{ch(\{a\},\{b\})\})\} \twoheadrightarrow \{c(\{a,b\})\}}\,\text{DC} \quad \dfrac{(*)}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\,\text{DC}}{f(c(\widetilde{ch(a,b)})) \equiv \{f(\{c(\{ch(\{a\},\{b\})\})\})\} \twoheadrightarrow \{d(\{a\},\{b\})\} \equiv \widetilde{d(a,b)}}\,\text{ROR}$$

where $(*)$ is the derivation:

$$\dfrac{\dfrac{\dfrac{}{\{a\} \twoheadrightarrow \{a\}}\,\text{DC}}{\{a,b\} \twoheadrightarrow \{a\}}\,\textsc{Less} \quad \dfrac{\ldots}{\{a,b\} \twoheadrightarrow \{b\}}}{\{d(\{a,b\},\{a,b\})\} \twoheadrightarrow \{d(\{a\},\{b\})\}}\,\text{DC}$$

[2] The prefix 'run-time' comes from 'run-time choice', a term often applied ([13,10]) to the parameter passing mechanism of term rewriting.

On the other hand, $\widetilde{d(a,b)}$ is not a correct value for $f(\widetilde{choice(c(a),c(b))})$, because in that expression the evaluation of $\widetilde{choice(c(a),c(b))}$ has to be performed in order to get a value matching the argument of the left-hand side of the only rule for f, and the only matching values for it are $\widetilde{c(a)}$, $\widetilde{c(b)}$ and $\{c(\emptyset)\}$, as for example $\{c(\{a\}),c(\{b\})\}$ does not match $\widetilde{c(X)}$.

Notice that, structurally, a denotation $[\![se]\!]$ is a possibly infinite set of s-cterms, Infinite denotations might appear with non-terminating programs. Notice, however, that the elements are s-cterms that are, by construction, finite objects. Thus, we avoid the presence of infinite values as elements of denotations, escaping from the problems mentioned in [6,16].

As we anticipated above, even when proving a reduction $\tilde{e} \twoheadrightarrow \tilde{t}$ for $e \in Exp$ and $t \in CTerm$, we may find premises of the shape $se \twoheadrightarrow st$ for $se \in SExp, st \in SCTerm$, because the substitutions used for parameter passing in ROR may introduce sets in $\tilde{r}\theta$, as we can see in the second premise of the first application of ROR, in Ex. 2. But in fact the kind of s-expressions that we may find in the proof for some reduction for an expression is more restricted. It is easy to prove that in any proof for any statement $\tilde{e} \twoheadrightarrow st$ with $e \in Exp$ we have that in any premise $se' \twoheadrightarrow st'$ for it, $se \in trSExp$, a set defined as $trSExp \ni tr ::= st \mid \{h(tr_1,\ldots,tr_n)\}$, with $st \in SCTerm, h \in \Sigma, tr_1,\ldots,tr_n \in trSExp$.
This suggests that we could have defined our logic to prove reductions $se \twoheadrightarrow st$ with $se \in trSExp$ and $st \in SCTerm$ only, but we think that it is more profitable to define it to deal with the more general case of $se \in SExp$. First of all, then we get a logic that can handle a more general kind of syntactic objects, and therefore that could be used to express other formalism apart from term rewriting. We could slightly modify the rule ROR to accept not only CSs but in general *"s-expression rewriting systems" (sCSs)*, consisting of rules $\{f(st_1,\ldots,st_n)\} \twoheadrightarrow se$. This way the original formulation of ROR becomes a particular case of the new version, that works with CSs adapted to sCSs by means of $\tilde{\_}$. This would be similar to what is done in [17] to express term rewriting, term graph rewriting and noncopying rewriting by means of the more general framework of marked term rewriting. We consider this an interesting possible subject of future work. On the other hand, working with reductions of s-expressions allows us to formulate more general and powerful results about the semantics, like the *polarity* property stated below, which become easier to prove because of their generality (that provides stronger induction hypotheses), and that can be then easily applied to the more restricted case. In all the results of this and the next section we assume a given CS and omit mentioning it.

Proposition 1 (Polarity). *Let $se, se' \in SExp$, $st, st' \in SCTerm$. If $se \sqsubseteq se'$ and $st' \sqsubseteq st$ then $st \in [\![se]\!]$ implies $st' \in [\![se']\!]$.*

Our semantics also enjoys the following monotonicity property related to s-substitutions. It is formulated for the preorder $\sqsubseteq$ and also for the preorder $\trianglelefteq$ over $SSubst$, defined by $\sigma \trianglelefteq \sigma'$ iff $\forall X \in \mathcal{V}, [\![\sigma(X)]\!] \subseteq [\![\sigma'(X)]\!]$.

Proposition 2 (Monotonicity of substitutions). *Let* $se \in SExp$, $\sigma, \sigma' \in SSubst$. *If* $\sigma \trianglelefteq \sigma'$ *or* $\sigma \sqsubseteq \sigma'$ *then* $[\![se\sigma]\!] \subseteq [\![se\sigma']\!]$.

One of the most important properties of our logic is its *compositionality*, a property very close to the DET-additivity property for algebraic specifications of [13], which will be one of the keys for full abstraction.

Theorem 1 (Compositionality). *For all* $s\mathcal{C} \in sCntxt, se \in SExp$,

$$[\![s\mathcal{C}[se]]\!] = \bigcup_{st \in [\![se]\!]} [\![s\mathcal{C}[st]]\!]$$

As a consequence: $[\![se]\!] = [\![se']\!] \Leftrightarrow \forall s\mathcal{C}.\ [\![s\mathcal{C}[se]]\!] = [\![s\mathcal{C}[se']]\!]$.

Regarding *closedness* under susbtitutions, as we use $SCSubst$ for parameter passing it is natural to have closedness of reductions under this type of substitutions. Besides, as rewriting is closed under $Subst$ it is expected to have some kind of closedness for $Subst$ too. But in general it is not true that for any $st \in SCTerm, \sigma \in SSubst$ we have $st\sigma \in SCTerm$, therefore it makes no sense to expect that $se \twoheadrightarrow st$ implies $se\sigma \twoheadrightarrow st\sigma$, as the reductions in our logic are from $SExp$ to $SCTerm$. Nevertheless we still can say something about that, as we can see in the following property.

Proposition 3 (Closedness under substitutions). *Let* $se \in SExp$ *and* $st \in [\![se]\!]$. *Then: a)* $\forall \theta \in SCSubst, st\theta \in [\![se\theta]\!]$. *b)* $\forall \sigma \in SSubst, [\![st\sigma]\!] \subseteq [\![se\sigma]\!]$.

All these properties are powerful tools to reason about the denotation of s-expressions. And this reasoning power is transferred to the term rewriting universe through the adequacy results that we will see in the next section, thus opening paths for the development of new reasoning techniques for CSs.

3.3 Relation with Rewriting

The nice properties of our logic will be useless unless they are accompanied by strong adequacy results with respect to the term rewriting relation. We first address the *completeness* of our logic, i.e., that the semantics of any expression captures any c-term reachable from it by rewriting. As a first result we have:

Proposition 4. *For all* $e, e' \in Exp$, *if* $e \to^* e'$ *then* $[\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$.

The keys for its proof are Th. 1 and the following Lemma 1, expressing that any reduction $se\sigma \twoheadrightarrow st$ needs to use only a finite amount of the information contained in σ, formalized through the notion of denotation of an $SSubst$, defined as $[\![\sigma]\!] = \{\theta \in SCSubst \mid \forall X \in \mathcal{V}, \sigma(X) \twoheadrightarrow \theta(X)\}$.

Lemma 1. *Let* $\sigma \in SSubst, se \in SExp, st \in SCTerm$. *If* $se\sigma \twoheadrightarrow st$ *then there exists* $\theta \in [\![\sigma]\!]$ *such that* $se\theta \twoheadrightarrow st$.

Now we can apply Prop. 4 to get the following strong completeness result.

Theorem 2 (Completeness). *For all* $e, e' \in Exp, t \in CTerm$,
a) $e \to^* e'$ *implies* $|\widetilde{e'}| \in [\![\tilde{e}]\!]$ *b)* $e \to^* t$ *implies* $\widetilde{t} \in [\![\tilde{e}]\!]$

$\_ \vdash \_ \lessdot \_ \subseteq CS \times SCTerm \times Exp$	$\_ \vdash \_ \lessdot \_ \subseteq CS \times ESCTerm \times Exp$
$\mathcal{P} \vdash st \lessdot e$ if $\forall est \in st, \mathcal{P} \vdash est \lessdot e$	$\mathcal{P} \vdash X \lessdot e$ if $\mathcal{P} \vdash e \rightarrow^* X$
	$\mathcal{P} \vdash c(\overline{st}) \lessdot e$ if $\mathcal{P} \vdash e \rightarrow^* c(\overline{e})$ for some $\overline{e}$ such that $\forall e_i \in \overline{e}, \mathcal{P} \vdash st_i \lessdot e_i$

Fig. 2. Domination relation

Proof. For *a)*, it is easy to prove that $\forall e \in Exp, \tilde{e} \twoheadrightarrow \widetilde{|e|}$, by induction on the structure of e. But then $\widetilde{|e'|} \in [\![\tilde{e'}]\!] \subseteq [\![\tilde{e}]\!]$ by Prop. 4. For *b)*, just notice that $\forall t \in CTerm, |t| \equiv t$, and so $\tilde{e} \twoheadrightarrow \widetilde{|t|} \equiv \tilde{t}$, by *a)*.

We also want our logic to be *correct*, in the sense that the semantics of any expression does not compute more c-terms than those reachable by rewriting. One key ingredient will be the *domination relation* $\_ \lessdot \_$ defined in Fig. 2 (we will omit the prefix "$\mathcal{P} \vdash$" when deducible from the context). With this relation we try to transfer to the rewriting world the finer distinction between sets of values that the structured representation of $SCTerm$ allows us to perform. This way under the CS of Ex. 1 we have $\{c(\{a, b\})\} \lessdot c(choice(a, b))$ but not $\{c(\{a, b\})\} \lessdot choice(c(a), c(b))$. The domination relation $\_ \lessdot \_$ has a strong relation with our semantics, as stated in the following result:

Lemma 2 (Domination). *For all $e \in Exp, st \in SCTerm$: $st \in [\![\tilde{e}]\!]$ iff $st \lessdot e$.*

Notice that $\_ \lessdot \_$ only talks about reductions for $\tilde{e}$ with $e \in Exp$, and so it cannot be used to formulate properties like those seen in Sect. 3.2, although it inherits them through Lemma 2. But he good thing about $\_ \lessdot \_$ is that it already has a strong connection with rewriting, as it is defined by means of rewriting derivations. Hence we can perform a simple induction on the structure of $SCTerm$ and $ESCTerm$ to prove the following result, which uses the notion of flattening defined in Sect. 3.1.

Lemma 3. *Let $st \in SCTerm$, $est \in ESCTerm$, $e \in Exp$, and assume $t \in flat(st)$. If $st \lessdot e$ then $e \rightarrow^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.*

And now we are ready to state and prove our main correctness result.

Theorem 3 (Correctness). *Let $e \in Exp, st \in SCTerm, t \in CTerm_\perp$:*

a) If $st \in [\![\tilde{e}]\!]$ and $t \in flat(st)$, then $e \rightarrow^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.
b) If $\tilde{t} \in [\![\tilde{e}]\!]$, then $e \rightarrow^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$.
c) Besides, in a) or b), if $t \in CTerm$, then $e \rightarrow^* t$.

Proof. We get *a)* just chaining Lemma 2 and Lemma 3. Concerning *b)*, we can prove that $\forall t \in CTerm_\perp, flat(\tilde{t}) = \{t\}$ by induction on $CTerm_\perp$, and chain it with *a)*. Finally *c)* is a consequence of *a)* and *b)*, because if $t \in CTerm$ then it is maximal wrt $\sqsubseteq$, hence $t \sqsubseteq |e'|$ implies $t \equiv |e'|$. But that implies there is no $\perp$ in $|e'|$, therefore $e' \in CTerm$ and $e' \equiv |e'| \equiv t$, and so $e \rightarrow^* e' \equiv t$.

Correctness is the point where left linearity of programs is essential. Without left linearity, the results of Sect. 3.2 still hold, but the semantics becomes incorrect wrt rewriting. For instance, if $\mathcal{P} \equiv \{f(X, X) \rightarrow a\}$ and $e \equiv f(a, b)$ then it can be shown that $\tilde{a} \in [\![\tilde{e}]\!]$ but $e \not\rightarrow^* a$, contradicting Th. 3.

4 Full Abstraction

The semantics $[\![se]\!]^{\mathcal{P}}$ of Sect. 3 is defined for s-expressions, but induces naturally a notion of semantics for ordinary expressions $e \in Exp$:

$$[\![e]\!]^{\mathcal{P}}_S = [\![\tilde{e}]\!]^{\mathcal{P}}(= \{st \in SCTerm | \tilde{e} \twoheadrightarrow st\})$$

In this section we discuss full abstraction in the context of CS and show that $[\![\_]\!]_S$ achieves it, in contrast to semantics directly based on sets of results, informally described in Sect. 1. In a fully abstract semantics, two expressions have the same semantics if and only if they are observationally indistinguishable. For this to be meaningful, one must choose a criterion of observability. The problem of full abstraction was first investigated by Plotkin [18] for PCF (a simple functional programming language), and since then is a standard topic when dealing with program semantics (see e.g. [19]). It is common to adjust its definition to the characteristics of the language under consideration. In the context of functional-like programming languages like PCF the condition for full abstraction is usually stated as:

$$[\![e]\!] = [\![e']\!] \Leftrightarrow \mathcal{O}(\mathcal{C}[e]) = \mathcal{O}(\mathcal{C}[e']), \text{for any context } \mathcal{C} \tag{1}$$

where $\mathcal{O}$ is the observation function of interest. Programs do not need to be mentioned, because programs and expressions can be identified by contemplating the evaluation of e under $\mathcal{P}$ as the evaluation of a large λ-expression or *let*-expression embodying $\mathcal{P}$ and e. Contexts pose no problems either. In our case, since programs (CS) are kept different from expressions, some care must be taken. It might happen that $\mathcal{P}$ has not enough syntactical elements and rules to build interesting distinguishing contexts. For instance, if in Ex. 1 we drop the definition of f, then we cannot built a context that distinguishes $c(choice(a, b))$ from $choice(c(a), c(b))$. This would imply that soundness or full abstraction would not be intrinsic to the semantics, but would depend on the program. What we need is requiring the right part of (1) to hold for all contexts that might be obtained by extending $\mathcal{P}$ with new auxiliary functions. To be more precise, we say that $\mathcal{P}'$ is a *safe extension* of $(\mathcal{P}, e)$ if $\mathcal{P}' = \mathcal{P} \cup \mathcal{P}''$, where $\mathcal{P}''$ does not include defining rules for any function symbol occurring in $\mathcal{P}$ or e. Any sensible notion of semantics should verify $[\![e]\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{P}'}$ when $\mathcal{P}'$ safely extends (P, e). This happens indeed for all the semantics considered below.

Things are now prepared to give the following definition:

Definition 1 (Observations, full abstraction).
(a) A semantic function (a semantics*, in short) is a function $[\![\_]\!]$ associating a semantic value (taken from a set $\mathcal{D}$) to each expression e under a given program $\mathcal{P}$. We write $[\![e]\!]^{\mathcal{P}}$ for such value.*
(b) An observation function *is a function $\mathcal{O}$ associating a set of observation values (or* observables*, taken from a set Obs) to each expression e under a given program $\mathcal{P}$. We write $\mathcal{O}^{\mathcal{P}}(e)$ for it.*
(c) A semantics is fully abstract *wrt $\mathcal{O}$ iff for any $\mathcal{P}$ and $e, e' \in Expr$, the following two conditions are equivalent:*

(i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ *(ii)* $\mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}(\mathcal{C}[e'])$ *for any* $\mathcal{P}'$ *safely extending* $(\mathcal{P}, e)$, $(\mathcal{P}, e')$ *and any* $\mathcal{C}$ *built with the signature of* $\mathcal{P}'$.
In words: semantic equality is equivalent to observational indistinguishability.
(d) A notion weaker than full abstraction is: a semantics is sound *wrt* $\mathcal{O}$ *iff the condition* (i) *above implies the condition* (ii).
In words: semantic equality implies observational indistinguishability.
(e) A semantics is compositional *iff for any* $\mathcal{P}$ *and* $e, e' \in Expr$, *the following two conditions are equivalent:*
(i) $[\![e]\!]^{\mathcal{P}} = [\![e']\!]^{\mathcal{P}}$ *(ii)* $[\![\mathcal{C}[e]]\!]^{\mathcal{P}} = [\![\mathcal{C}[e']]\!]^{\mathcal{P}}$ *for any* $\mathcal{C}$ *built with the signature of* $\mathcal{P}$.
In words: the semantics of an expression depends only on the semantics of its subexpressions. Notice that $(ii) \Rightarrow (i)$ *holds trivially (take* $\mathcal{C} = [\ ]$*).*

In the next definition we collect some notions of semantics and observables for the case of CS. Our new contributed semantics are $[\![\_]\!]_S$ and $[\![\_]\!]_{S'}$; the rest are the 'obvious' semantics of Sect. 1. As usual, we omit the program $\mathcal{P}$ in notations.

Definition 2 (Semantics and observations for CSs).
We consider the following semantics for expressions $e \in Exp$:

$$[\![e]\!]_{rw} = \{e' \mid e \to^* e'\} \qquad [\![e]\!]_{nf} = \{e' \mid e \to^* e', e' \text{in normal form}\}$$
$$[\![e]\!]_t = \{t \in CTerm \mid e \to^* t\} \qquad [\![e]\!]_{t_\perp} = \{t \in CTerm_\perp \mid \exists e'.(e \to^* e' \wedge t \sqsubseteq |e'|)\}$$
$$[\![e]\!]_S = [\![\tilde{e}]\!] \qquad [\![e]\!]_{S'} = \bigcup_{st \in [\![e]\!]_S} st.$$

and two observation functions for expressions: $\mathcal{O}_t(e) = [\![e]\!]_t$ *and* $\mathcal{O}_{t_\perp}(e) = [\![e]\!]_{t_\perp}$.

Some remarks:

- It is clear that $[\![e]\!]_t \subseteq [\![e]\!]_{nf} \subseteq [\![e]\!]_{rw}$, and also $[\![e]\!]_t \subseteq [\![e]\!]_{t_\perp}$.
- Notice that some of the sets above can play at the same time the role of semantic values and of observation values.
- $[\![e]\!]_S$ was introduced earlier in this section. $[\![e]\!]_{S'}$ is a simplified variant, making more readable the semantics of expressions, because $[\![e]\!]_S$ is a set of finite sets of *est*s, while $[\![e]\!]_{S'}$ is simply a set of *est*s. However $[\![\_]\!]_S$ has been technically more convenient for proving properties of the semantics, due to its more direct connection to the proof calculus. Both semantics are essentially the same:

Proposition 5. *For any* $e, e' \in Exp$, $[\![e]\!]_S = [\![e']\!]_S \Leftrightarrow [\![e]\!]_{S'} = [\![e']\!]_{S'}$

The next result shows that, although $\mathcal{O}_t, \mathcal{O}_{t_\perp}$ define different observations, it is irrelevant which is chosen, as far as full abstraction is concerned.

Proposition 6. *Assume a given semantics* $[\![\_]\!]$. *Then:*

$$[\![\_]\!] \text{ is fully abstract wrt } \mathcal{O}_t \Leftrightarrow [\![\_]\!] \text{ is fully abstract wrt } \mathcal{O}_{t_\perp}.$$

Now we show that the first four semantics, $[\![]\!]_{rw}$, $[\![\_]\!]_{nf}$, $[\![\_]\!]_t$, $[\![\_]\!]_{t_\perp}$ do not have good properties, as was informally discussed in Sect. 1. We use $\mathcal{O}_t$ in the result but, according to the previous result, $\mathcal{O}_{t_\perp}$ could be used instead.

Proposition 7.

(a) $[\![\_]\!]_{rw}, [\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$ *are not fully abstract wrt* $\mathcal{O}_t$.
(b) Moreover, $[\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$ *are not compositional, nor sound wrt* $\mathcal{O}_t$.
(c) $[\![\_]\!]_{nf}$ *(*$[\![\_]\!]_t$ *resp.) remains not compositional nor sound wrt* $\mathcal{O}_t$ *even if programs are restricted to be confluent (confluent and terminating, resp.).*

Proof. (a) For $[\![\_]\!]_{nf}, [\![\_]\!]_t, [\![\_]\!]_{t_\perp}$, (a) is implied by (b). For $[\![\_]\!]_{rw}$, just add a new function $g(X) \to f(X)$ to Ex. 1. It is easy to see that $f(a)$ and $g(a)$ are contextually indistinguishable wrt $\mathcal{O}_t$, but $[\![f(a)]\!]_{rw} \neq [\![g(a)]\!]_{rw}$.
(b) Example 1 of Sect. 1 serves for all the three semantics.
(c) For $[\![\_]\!]_{nf}$, consider the program $\{f \to f, g \to c(f), h(c(X)) \to a\}$. We have $[\![f]\!]_{nf} = [\![g]\!]_{nf} = \emptyset$, but $[\![h(f)]\!]_{nf} = \emptyset \neq \{a\} = [\![h(g)]\!]_{nf}$, proving at the same time not compositionality and unsoundness. For $[\![\_]\!]_t$, replace the above program by $\{f \to h(a), g \to c(f), h(c(X)) \to a\}$.

Finally, we show that our semantics do not present those problems.

Theorem 4 (Compositionality and full abstraction of $[\![\_]\!]_S$).

$[\![\_]\!]_S$ *and* $[\![\_]\!]_{S'}$ *are compositional and fully abstract wrt* $\mathcal{O}_t$ *and* $\mathcal{O}_{t_\perp}$.

Proof. We prove the results for $[\![\_]\!]_S$ and $\mathcal{O}_t$. For $[\![\_]\!]_{S'}$ and $\mathcal{O}_{t_\perp}$, just use propositions 5 and 6. Compositionality follows from definition of $[\![\_]\!]_S$ and compositionality of $[\![\_]\!]$ (Th. 1).
For full abstraction, let $\mathcal{P}$ be any CS, and $e, e' \in Expr$. We must prove:

$$[\![e]\!]_S^{\mathcal{P}} = [\![e']\!]_S^{\mathcal{P}} \Leftrightarrow \forall \mathcal{P}', \mathcal{C} \ . \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) = \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$$

where $\mathcal{P}'$ ranges over safe extensions of $(\mathcal{P}, e)$ and $(\mathcal{P}, e')$, and $\mathcal{C}$ over contexts built with the signature of $\mathcal{P}'$.
$\boxed{\Rightarrow}$ Assume $[\![e]\!]_S^{\mathcal{P}} = [\![e']\!]_S^{\mathcal{P}}$. Since $\mathcal{P}'$ is a safe extension, we know that $[\![e]\!]_S^{\mathcal{P}'} = [\![e']\!]_S^{\mathcal{P}'}$. We prove $\mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e]) \subseteq \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$ (the other inclusion is similar). Let $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e])$, which means $\mathcal{C}[e] \to_{\mathcal{P}'}^* t$. By Th. 2, $\tilde{t} \in [\![\mathcal{C}[e]]\!]_S^{\mathcal{P}'} = [\![\mathcal{C}[e']]\!]_S^{\mathcal{P}'}$, where the last equality is justified by compositionality. Then, since $t \in flat(\tilde{t})$, we conclude from Th. 3 that $\mathcal{C}[e'] \to_{\mathcal{P}'}^* t$, that is, $t \in \mathcal{O}_t^{\mathcal{P}'}(\mathcal{C}[e'])$, as desired.

For the other implication we need two auxiliary constructions enabling to build a context that distinguishes two expressions having different semantics:

• Given $st \in SCTerm$, we define a c-term $\widehat{st}$ 'mirroring' st as follows:

$$\widehat{\emptyset} = \langle\,\rangle_0 \qquad \widehat{\{X\}} = X \ (X \in \mathcal{V}) \qquad \widehat{\{c(\overline{st_i})\}} = c(\overline{\widehat{st_i}})$$
$$\widehat{\{est_1, \ldots, est_n\}} = \langle \widehat{\{est_1\}}, \ldots, \widehat{\{est_n\}} \rangle_n \ (n > 1)$$

where $\langle\_\rangle_n$ $(n \geq 0)$ are new tuple-forming constructor symbols. It is assumed here that $SCTerm, ESCTerm$ are equipped with any standard ordering.

• Given $st \in SCTerm$, the program $\mathcal{P}_{st}$ defines new functions $f_{st}, \ldots$ as follows:

$$f_\emptyset(X) \to \langle\,\rangle_0 \qquad f_{\{X\}}(U) \to U \ (X \in \mathcal{V}) \qquad f_{\{c(\overline{st_i})\}}(c(\overline{U_i})) \to c(\overline{f_{st_i}(U_i)})$$
$$f_{\{est_1, \ldots, est_n\}}(U) \to \langle f_{\{est_1\}}(U), \ldots, f_{\{est_n\}}(U) \rangle_n \ (n > 1)$$

The roles of $\widehat{st}, \mathcal{P}_{st}$ are made clear by the following lemma:

Lemma 4. *For any* $\mathcal{P}, st, e$*:* $st \in [\![e]\!]^{\mathcal{P}}_S \Leftrightarrow f_{st}(e) \rightarrow^*_{\mathcal{P}'} \widehat{st}$*, where* $\mathcal{P}' \equiv \mathcal{P} \cup \mathcal{P}_{st}$.

We can now proceed with the proof of the pending implication.
$\boxed{\Leftarrow}$ Assume that $\mathcal{O}^{\mathcal{P}'}_t(\mathcal{C}[e]) = \mathcal{O}^{\mathcal{P}'}_t(\mathcal{C}[e'])$ for any safe extension $\mathcal{P}'$ and context $\mathcal{C}$. We prove $[\![e]\!]^{\mathcal{P}}_S \subseteq [\![e']\!]^{\mathcal{P}}_S$ (the other inclusion is similar). Let $st \in [\![e]\!]^{\mathcal{P}}_S$. Let $\mathcal{P}' \equiv \mathcal{P} \cup \mathcal{P}_{st}$, which is a safe extension of $(P, e), (P, e')$. Lemma 4 ensures $f_{st}(e) \rightarrow^*_{\mathcal{P}'} \widehat{st}$, which means $\widehat{st} \in \mathcal{O}^{\mathcal{P}'}_t(f_{st}(e))$. Now, since $\mathcal{P}'$ is a safe extension, observational equivalence of e, e' implies $\widehat{st} \in \mathcal{O}^{\mathcal{P}'}_t(f_{st}(e'))$, which means $f_{st}(e') \rightarrow^*_{\mathcal{P}'} \widehat{st}$. Again by Lemma 4, we conclude that $st \in [\![e']\!]^{\mathcal{P}}_S$, as desired.

5 Conclusions

In this paper we have provided a semantics for constructor systems that is fully abstract with respect to natural notions of observation that extract the outer constructor part of outcomes as relevant information of computations. To the best of our knowledge, this is the first time that full abstraction has been achieved for this class of programs and observations. Along the way to this result we have made some contributions: after noticing that 'obvious' semantics directly based on rewrite sequences lack compositionality, our main insight has been that it can be recovered by recursively packaging sets of results below constructor symbols. That insight has been realized at the technical level by introducing s-cterms as suitable semantic values, and giving a proof calculus able to derive reachable s-cterms from a given expression. Previous to full abstraction, we have proved a bunch of good properties of the semantics: polarity, compositionality, closedness under substitutions, correctness and completeness with respect to rewriting.

We expect our semantics to be a useful tool for CS-based program manipulation. We remark that, for instance, to justify the correctness of a CS-transformation by proving preservation of reachable values could be incorrect if transformations are to be used locally. Our semantics could be a better option, as illustrated by the following simple example: consider a program piece made of the rules $f \rightarrow c(g),\ g \rightarrow e,\ g \rightarrow e'$, where e, e' are expressions that costly reduce to a, b respectively. A 'obvious' partial evaluation might suggest replacing f's definition by the optimized one $f \rightarrow c(a),\ f \rightarrow c(b)$, leading to a transformed program $\mathcal{P}'$, presumably equivalent to $\mathcal{P}$. This is wrong: if $\mathcal{P}$ defines also h by the rule $h(c(X)) \rightarrow d(X, X)$, then $h(f)$ behaves different with both definitions of f. This is detected in our semantics (using e.g. the variant $[\![\_]\!]_{S'}$ of Def. 2), because $[\![f]\!]^{\mathcal{P}}_{S'} \ni c(\{a, b\}) \notin [\![f]\!]^{\mathcal{P}'}_{S'}$. Imagine, however, that the original program piece was $f \rightarrow g,\ g \rightarrow c(e),\ g \rightarrow c(e')$. In this case, the 'obvious' partial evaluation of f would lead again to $f \rightarrow c(a),\ f \rightarrow c(b)$. Is this right now? Yes, because $[\![f]\!]^{\mathcal{P}}_{S'} = [\![f]\!]^{\mathcal{P}'}_{S'}$, and full abstraction of $[\![\_]\!]_{S'}$ makes the rest. A deeper investigation of these issues is planned for the future.

There are other aspects not yet accomplished that can be subject of future work. We plan to investigate full abstraction for other notions of observations, in two different senses: by giving a more active role to variables (as happens in [2,1,12]) taking into account that, for instance, variables can be subject of narrowing substitutions; and by replacing our 'may-convergent' or 'angelic' view of non-determinism (in which two expressions may have the same

semantics even if one admits divergent reductions while the other does not) by a 'must-convergence' view where divergence plays a role. Dropping the constructor restriction is also interesting, replacing the role of constructor values by appropriate alternatives. Finally, incorporating s-expressions to the syntax of programs, as discussed in Sect. 3.2, could lead to more expressive programs.

References

1. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract diagnosis of functional programs. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)
2. Alpuente, M., Falaschi, M., Ramis, M., Vidal, G.: Narrowing approximations as an optimization for equational logic programs. In: Penjam, J., Bruynooghe, M. (eds.) PLILP 1993. LNCS, vol. 714, pp. 391–409. Springer, Heidelberg (1993)
3. Antoy, S.: Optimal non-deterministic functional logic computations. In: Hanus, M., Heering, J., Meinke, K. (eds.) ALP 1997 and HOA 1997. LNCS, vol. 1298, pp. 16–30. Springer, Heidelberg (1997)
4. Antoy, S., Iranzo, P.J., Massey, B.: Improving the efficiency of non-deterministic computations. ENTCS, 64 (2002)
5. Bert, D., Echahed, R., Østvold, B.M.: Abstract Rewriting. In: Cousot, P., Filé, G., Falaschi, M., Rauzy, A. (eds.) WSA 1993. LNCS, vol. 724, pp. 178–192. Springer, Heidelberg (1993)
6. Boudol, G.: Une semantique pour les arbres non deterministes. In: Astesiano, E., Böhm, C. (eds.) CAAP 1981. LNCS, vol. 112, pp. 147–161. Springer, Heidelberg (1981)
7. Boudol, G.: Computational semantics of term rewriting systems. In: Algebraic methods in semantics, pp. 169–236. Cambridge University Press, Cambridge (1986)
8. Braßel, B., Huch, F.: On a tighter integration of functional and logic programming. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 122–138. Springer, Heidelberg (2007)
9. Clavel, M., et al. (eds.): All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. J. of Logic Programming 40(1), 47–87 (1999)
11. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
12. Hanus, M., Lucas, S.: An evaluation semantics for narrowing-based functional logic languages. J. Funct. and Logic Prog. 2001(2) (2001)
13. Hussmann, H.: Non-Determinism in Algebraic Specifications and Algebraic Programs. Birkhäuser Verlag, Basel (1993)
14. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Bundles pack tighter than lists. In: Draft Proc. Trends in Funct. Prog, TFP 2007 (2007)
15. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. 285(2), 121–154 (2002)
16. Nyström, S.-O.: There is no fully abstract fixpoint semantics for non-deterministic languages with infinite computations. Inf. Process. Lett. 60(6), 289–293 (1996)
17. Ohlebusch, E.: Advanced topics in term rewriting. Springer, Heidelberg (2002)
18. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. 5(3), 225–255 (1977)
19. Reynolds, J.: Theories of Programing Languages. Cambridge University Press, Cambridge (1998)