TLP: page 1 of 52 © Cambridge University Press 2012
doi:10.1017/S147106841200004X

Singular and plural functions for functional logic programming

ADRIÁN RIESCO* and JUAN RODRÍGUEZ-HORTALÁ†
Dpto. Sistemas Informáticos y Computación, Facultad de Informática
Universidad Complutense de Madrid, Ciudad Universitaria–28040 Madrid
(*e-mail:* {ariesco, juanrh}@fdi.ucm.es)

submitted 11 February 2011; revised 17 November 2011; accepted 6 March 2012

Abstract

Modern functional logic programming (FLP) languages use non-terminating and non-confluent constructor systems (CSs) as programs in order to define non-strict and non-deterministic functions. Two semantic alternatives have been usually considered for parameter passing with this kind of functions: call-time choice and run-time choice. While the former is the standard choice of modern FLP languages, the latter lacks some basic properties – mainly compositionality – that have prevented its use in practical FLP systems. Traditionally it has been considered that call-time choice induces a singular denotational semantics, while run-time choice induces a plural semantics. We have discovered that this latter identification is wrong when pattern matching is involved, and thus in this paper we propose two novel compositional plural semantics for CSs that are different from run-time choice.

We investigate the basic properties of our plural semantics – compositionality, polarity, and monotonicity for substitutions, and a restricted form of the bubbling property for CSs – and the relation between them and to previous proposals, concluding that these semantics form a hierarchy in the sense of set inclusion of the set of values computed by them. Besides, we have identified a class of programs characterized by a simple syntactic criterion for which the proposed plural semantics behave the same, and a program transformation that can be used to simulate one of the proposed plural semantics by term rewriting. At the practical level, we study how to use the new expressive capabilities of these semantics for improving the declarative flavor of programs. As call-time choice is the standard semantics for FLP, it still remains the best option for many common programming patterns. Therefore, we propose a language that combines call-time choice and our plural semantics, which we have implemented in the Maude system. The resulting interpreter is then employed to develop and test several significant examples showing the capabilities of the combined semantics.

KEYWORDS: non-deterministic functions, semantics, program transformation, term rewriting, Maude

* Research partially supported by the Spanish projects *DESAFIOS10* (TIN2009-14599-C03-01) and *PROMETIDOS-CM* (S2009/TIC-1465).
† Research partially supported by the Spanish projects *FAST-STAMP* (TIN2008-06622-C03-01/TIN), *PROMETIDOS-CM* (S2009TIC-1465) and *GPD-UCM* (UCM-BSCH-GR58/08-910502).

1 Introduction

The combination of functional and logic features has been addressed in several proposals for multi-paradigm programming languages (Roy and Haridi 2004; Rodríguez-Hortalá and Sánchez-Hernández 2008; Hermenegildo *et al.* 2012; The Mercury Team 2012) with different variants – lazy or eager evaluation of functions, concurrent capabilities, support for object-oriented programming, *et al.* In this work we focus on the integration into a single language of the main features of lazy functional programming (FP) and logic programming (LP) that is described in Antoy and Hanus (2010). Term rewriting (Baader and Nipkow 1998) and term graph rewriting systems (Plump 1999) have often been used for modeling the semantics and operational behavior of that approach to functional logic programming (FLP) (DeGroot and Lindstrom 1986; Hanus 2007). In particular, the class of left-linear constructor-based term rewriting systems – or simply constructor systems (CS) – in which the signature is divided into two disjoint sets of constructor and function symbols, is used frequently to represent programs. There the notion of value as a term built using only constructor symbols – called constructor term or just c-term – arises naturally, and this way a term rewriting derivation from an expression to a c-term represents the reduction of that expression to one of its values in the language being modelled. This corresponds to a value-based semantic view, in which the purpose of computations is to produce values made of constructors. Besides, term graphs are used for modelling subexpression sharing, where several occurrences of the same subexpression are represented by several pointers to a single node in a term graph, resulting in a potential improvement of the time and space performance of programs. Sharing is at the core of implementations of lazy FP and FLP languages, and so several variations of term graph rewriting have been also used in formulations of the semantics of call-by-need in FP (Launchbury 1993; Plasmeijer and van Eekelen 1993; Ariola *et al.* 1995) and FLP (Echahed and Janodet 1998; Albert *et al.* 2005; López-Fraguas *et al.* 2007; López-Fraguas *et al.* available on request).

On the other hand, non-determinism is an expressive feature that has been used for a long time in programming (McCarthy 1963; Hughes and O'Donnell 1990; Dijkstra 1997) and system specification (Borovanský *et al.* 1998; Futatsugi and Diaconescu 1998; Clavel *et al.* 2007). In both fields, one of the appeals of term rewriting is its elegant way to express non-determinism through the use of non-confluent term rewriting systems, obtaining a clean and high-level representation of complex systems and programs. Non-determinism is integrated in FLP languages by means of a backtracking mechanism in the style of Prolog (Sterling and Shapiro 1986). It is introduced by employing possibly non-terminating and non-confluent CSs as programs, thus expressing non-strict and non-deterministic functions, which are one of the most distinctive features of the paradigm (González-Moreno *et al.* 1999; Antoy and Hanus 2002; Antoy and Hanus 2010).[1]

[1] Non-determinism also appears in FLP as a result of the utilisation of narrowing as the fundamental operational mechanism (Hanus 2005), but as usual in many works in the field, we will focus on rewriting aspects only, so our conclusions could be lifted to the narrowing case in subsequent works.

The point is that this combination of non-strictness and non-determinism gives rise to several semantic alternatives (Søndergaard and Sestoft 1992; Hussmann 1993). In particular, in Sondergaard and Sestoft (1992) the different language variants that result after adding non-determinism to a basic functional language were expounded, structuring the comparison as a choice among different options over several dimensions: strict/non-strict functions, angelic/demonic/erratic non-deterministic choices, and *singular/plural semantics* for parameter passing, also called *call-time choice/run-time choice* in Hussmann (1993). In the present paper we assume non-strict angelic non-determinism, so we focus on the last dimension only. To do that, let us take a look at the following example.

Example 1.1
Consider the program $\{f(c(X)) \rightarrow d(X,X), X \ ? \ Y \rightarrow X, X \ ? \ Y \rightarrow Y\}$ and the expression $f(c(0 \ ? \ 1))$. From an operational perspective we have to decide when it is time to fix the values for the arguments of functions:

- Under a *call-time choice* semantics, a value for each argument will be fixed on parameter passing and shared between every copy of that argument that arises during the computation. This corresponds to call-by-value in a strict setting and to call-by-need in a non-strict setting, in which a partial value instead of a total value is computed. So when applying the rule for f, the two occurrences of X in $d(X,X)$ will share the same value, hence $d(0,0)$ and $d(1,1)$ are correct values for $f(c(0 \ ? \ 1))$ in this semantics, while it is not the case either for $d(0,1)$ or $d(1,0)$.
- On the other hand, *run-time choice* corresponds to call-by-name, so the values of the arguments are fixed as they are used – i.e., as their evaluation is demanded by the matching process – and the copies of each argument created by parameter passing may evolve independently afterwards. Under this semantics not only $d(0,0)$ and $d(1,1)$ but also $d(0,1)$ and $d(1,0)$ are correct values for $f(c(0 \ ? \ 1))$.

In general, a call-time choice semantics produces less results than run-time choice. Modern functional-logic languages like Toy (López-Fraguas and Sánchez-Hernández 1999) or Curry (Hanus 2005) are heavily influenced by lazy functional programming and so they implement sharing in their operational mechanism, which results in call-by-need evaluation and the adoption of call-time choice. On the other hand, term rewriting is considered a standard formulation for run-time choice,[2] and is the basis for the semantics of languages like the Maude (Clavel *et al.* 2007).

But we may also see things from another perspective.

Example 1.2
Consider again the program in Example 1.1. From a denotational perspective we have to think about the domain used to instantiate the variables of program rules:

[2] In fact angelic non-strict run-time choice.

- Under a *singular semantics* variables will be instantiated with single values (which may be partial in a non-strict setting). *This is equivalent to having call-time choice parameter passing.*
- The alternative is having a *plural semantics*, in which the variables are instantiated with sets of values. Traditionally it has been considered that run-time choice has its denotational counterpart on a plural semantics, but we will see that this identification is wrong. Consider the expression $f(c(0)\ ?\ c(1))$, under run-time choice, that is, term rewriting, the evaluation of the subexpression $c(0)\ ?\ c(1)$ is needed in order to get an instance of the left-hand side of the rule for f. Hence, a choice between $c(0)$ and $c(1)$ is performed and so neither $d(0,1)$ nor $d(1,0)$ are correct values for $f(c(0)\ ?\ c(1))$. Nevertheless, under a plural semantics we may consider the set $\{c(0), c(1)\}$, which is a subset of the set of values for $c(0)\ ?\ c(1)$ in which every element matches the argument pattern $c(X)$. Therefore, the set $\{0,1\}$ can be used for parameter passing, obtaining a kind of "set expression" $d(\{0,1\},\{0,1\})$ that yields the values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$.

The conclusion is clear: *the traditional identification of run-time choice with a plural semantics is wrong when pattern matching is involved.*

Which of these is the more suitable perspective for FLP? This problem did not appear in Sondergaard and Sestoft (1992) because no pattern matching was present, nor in Hussmann (1993) because only call-time choice was adopted there. This fact was pointed out for the first time in Rodriguez-Hortala (2008), where the $\pi^{\alpha}CRWL$ logic – named $\pi CRWL$ in that work – was proposed as a novel formulation of a plural semantics with pattern-matching. This proves that one can conceive a meaningful plural semantics that is different to run-time choice, i.e., run-time choice is not the only plural semantics we should consider. We have seen that, using the program above, the expression $f(c(0\ ?\ 1))$ has more values than the expression $f(c(0)\ ?\ c(1))$ under run-time choice although they only differ in the subexpressions $c(0\ ?\ 1)$ and $c(0)\ ?\ c(1)$, which have the same values under all three, call-time choice, run-time choice, and plural semantics. That violates a fundamental property of FLP languages stating that any expression can be replaced by any other expression that could be reduced to exactly the same set of values. We will see that our plural semantics shares with $CRWL$[3] (González-Moreno *et al.* 1999) (the standard logic for call-time choice[4]), a compositionality property for values that makes it more suitable than run-time choice for a value-based language like current implementations of FLP. Nevertheless, run-time choice can be a good option for other kind of rewriting-based languages like the Maude, in which the notion of value is not necessarily present, at least in the sense it is in FLP languages.

In this paper we have put together our previous results about plural semantics, integrating our presentation of $\pi^{\alpha}CRWL$ from Rodriguez-Hortala (2008) with a user level introduction to the Maude-based transformational prototype for $\pi^{\alpha}CRWL$

[3] **C**onstructor-based **ReW**riting **L**ogic.
[4] In fact, angelic non-strict call-time choice.

(Riesco and Rodríguez-Hortalá 2010a). We have also included the results obtained in Riesco and Rodriguez-Hortala (2010b), which is devoted to the exploration of new expressive capabilities of our plural semantics. Although our plural semantics allows an elegant encoding of some problems – in particular those with an implicit manipulation of sets of values – call-time choice still remains the best option for many common programming patterns (González-Moreno *et al.* 1999; Antoy and Hanus 2002). Therefore, we propose a combined semantics for a language in which the user can specify, for each function symbol, that arguments that are considered "plural arguments" – thus being evaluated under our plural semantics – and the arguments that are "singular arguments" – thus being evaluated under call-time choice. This semantics is precisely specified by a modification of the *CRWL* logic, which retains the important properties of *CRWL* and $\pi^\alpha CRWL$, like compositionality. These new features were implemented by extending our Maude prototype, and then used to develop and test several significant examples showing the expressive capabilities of a combined semantics.

Apart from giving a unified and revised presentation, we have made several relevant advances. We have extended most of our results to deal with programs with extra variables, and above all, we have introduced the new plural semantics $\pi^\beta CRWL$ inspired by the proposal from Braßel and Berghammer (2009). The properties of this semantics and its relation to call-time choice, run-time choice, and $\pi^\alpha CRWL$ have been studied in depth and with technical accuracy. Our current implementation does not deal with extra variables because they cause an explosion in the search space when evaluated by term rewriting – we consider the development of a suitable plural narrowing mechanism that could effectively handle extra variables, a possible subject of future work.

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about term rewriting systems and the *CRWL* logic. In Section 3 we introduce $\pi^\alpha CRWL$ and $\pi^\beta CRWL$, two variations of *CRWL* to express plural semantics, and present some of their properties, in particular compositionality. In Section 4 we study the relation between call-time choice, run-time choice, and our plural semantics, focusing on the set of values computed by each semantics and concluding that these four semantics form a hierarchy in the sense of set inclusion. We also present a class of programs characterized by a simple syntactic criterion under which our two plural semantics are equivalent, and conclude the section providing a simple program transformation that can be used to simulate $\pi^\alpha CRWL$ with term rewriting. Section 5 begins with the presentation of our combinations of call-time choice and plural semantics that are formalized through the $CRWL^\sigma_{\pi^\alpha}$ and $CRWL^\sigma_{\pi^\beta}$ logics, which correspond to the combination of call-time choice with $\pi^\alpha CRWL$ and $\pi^\beta CRWL$, respectively. Then follows a user level introduction to our Maude prototype, which implements the $CRWL^\sigma_{\pi^\alpha}$ logic as it is based on the transformation from Section 4. The prototype is then employed to illustrate the use of the combined semantics for improving the declarative flavor of programs. This section concludes with a short sketch of the implementation of our prototype. Finally, in Section 6 we outline some possible lines for future work. For the sake of readability, some of the proofs have been moved to Riesco and Rodríguez-Hortalá

(2011), although the intutitions behind our main results have been presented in the text.

2 Preliminaries

We present in this section the main notions needed throughout the rest of the paper: Section 2.1 introduces constructor-based systems, while Section 2.2 describes the *CRWL* framework.

2.1 Constructor systems

We consider the first-order signature $\Sigma = CS \uplus FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols, respectively, all of them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity $n \in \mathbb{N}$. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions, and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. Given a set $\mathcal{A}$, we denote by $\mathcal{A}^*$ the set of finite sequences of elements of that set. We denote the empty sequence by $[]$. For any sequence $a_1 \ldots a_n \in \mathcal{A}^*$ and function $f : \mathcal{A} \rightarrow \{true, false\}$, we denote by $a_1 \ldots a_n \mid f$ the sequence constructed by taking in order every element from $a_1 \ldots a_n$ for which f holds. Finally, for any $1 \leqslant i \leqslant n$, $(a_1 \ldots a_n)[i]$ denotes a_i.

The set *Exp* of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. We use the symbol $\equiv$ for the syntactic equality between expressions, and in general for any syntactic construction. The set *CTerm* of *constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that *Exp* stands for evaluable expressions, i.e., expressions that can contain function symbols, while *CTerm* stands for data terms representing **values**. 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 $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\ ] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$, $e_1, \ldots, e_n \in Exp$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\ ][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

A position of an expression is a chain of natural numbers separated by dots that determines one of its subexpressions. Given an expression e by $\mathcal{O}(e)$ we denote the set of positions in e, which is defined as $\mathcal{O}(X) = \epsilon$; $\mathcal{O}(h(e_1, \ldots, e_n)) = \{\epsilon\} \cup \{i.o \mid i \in \{1, \ldots, n\} \wedge o \in \mathcal{O}(e_i)\}$, where $X \in \mathcal{V}$, $h \in \Sigma$, and ϵ denotes the empty or top position. We will write $o, p, q, u, v, \ldots$ for positions. By $e|_o$ we denote the subexpression of e at position $o \in \mathcal{O}(e)$, defined as $e|_\epsilon = e$; $h(e_1, \ldots, e_n)|_{i.o} = e_i|_o$. The set of variable positions in e is denoted as $\mathcal{O}_\mathcal{V}(e)$ and defined as $\mathcal{O}_\mathcal{V}(e) = \{o \in \mathcal{O}(e) \mid e|_o \in \mathcal{V}\}$.

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ϵ for the identity (or empty) substitution. We write $e\theta$ for the application of θ to e, and $\theta\theta'$ for the composition, defined by $e(\theta\theta') = (e\theta)\theta'$. The domain and variable range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \not\equiv X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some $i \in \{0,1\}$; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given $W \subseteq \mathcal{V}$ we write $\theta|_W$ for the restriction of θ to W, and $\theta|_{\backslash D}$ is a shortcut for $\theta|_{(\mathcal{V} \backslash D)}$. We will sometimes write $\theta = \sigma[W]$ instead of $\theta|_W = \sigma|_W$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We say that *e subsumes* e', and write $e \precsim e'$, if $e\sigma \equiv e'$ for some substitution σ.

A *constructor-based term rewriting system* (*CS*) or just CS or *program* $\mathcal{P}$ is a set of *rewrite rules* or *program rules* of the form $f(t_1,\dots,t_n) \rightarrow r$ where $f \in FS^n$, $e \in Exp$, and $(t_1,\dots,t_n)$ is a linear tuple of c-terms, where linearity means that variables occur only once in $(t_1,\dots,t_n)$. Notice that we allow r to contain *extra variables*, i.e., variables not occurring in $(t_1,\dots,t_n)$. To be precise, we say that $X \in \mathcal{V}$ is an extra variable in the rule $l \rightarrow r$ iff $X \in var(r) \setminus var(l)$, and by $vExtra(R)$ we denote the set of extra variables in a program rule R. For any program $\mathcal{P}$ the set $FS^{\mathcal{P}}$ of functions defined by $\mathcal{P}$ is $FS^{\mathcal{P}} = \{f \in FS \mid \exists(f(\overline{p}) \rightarrow r) \in \mathcal{P}\}$. We assume that every program $\mathcal{P}$ contains the rules $\{X\ ?\ Y \rightarrow X, X\ ?\ Y \rightarrow Y, \mathit{if\ true\ then}\ X \rightarrow X\}$, defining the behavior of the infix function $? \in FS^2$ and the mix-fix function *if then* $\in FS^2$ (used as *if* e_1 *then* e_2), and that those are the only rules for that function symbols. Besides ? is right-associative, so $e_1\ ?\ e_2\ ?\ e_3 \equiv e_1\ ?\ (e_2\ ?\ e_3)$. For the sake of conciseness we will often omit these rules when presenting a program.

Given a program $\mathcal{P}$, its associated *term rewriting relation* $\rightarrow_{\mathcal{P}}$ is defined as: $\mathcal{C}[l\sigma] \rightarrow_{\mathcal{P}} \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \rightarrow r \in \mathcal{P}$ and $\sigma \in Subst$. We write $\overset{*}{\rightarrow}_{\mathcal{P}}$ for the reflexive and transitive closure of the relation $\rightarrow_{\mathcal{P}}$. In the following, we will usually omit the reference to $\mathcal{P}$ or denote it by $\mathcal{P} \vdash e \rightarrow e'$ and $\mathcal{P} \vdash e \rightarrow^* e'$.

2.2 The CRWL framework

The *CRWL* framework (González-Moreno *et al.* 1996; González-Moreno *et al.* 1999) is considered a standard formulation of call-time choice by the FLP community (Hanus 2007; Antoy and Hanus 2010). To deal with non-strictness at the semantic level, Σ is enlarged with a new constant constructor symbol $\perp$. The sets $Exp_\perp$, $CTerm_\perp$, $Subst_\perp$, $CSubst_\perp$ of *partial expressions*, etc. are defined naturally. Our contexts will contain partial expressions from now on unless explicitly specified. Expressions, substitutions, etc. not containing $\perp$ are called *total*. Programs in *CRWL* still consist of rewrite rules with total expressions in both sides, so $\perp$ does not appear in programs. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\perp \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e']$ for all $e, e' \in Exp_\perp, \mathcal{C} \in Cntxt$. This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\perp$ we say $\theta \sqsubseteq \sigma$ if $X\theta \sqsubseteq X\sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in *CRWL* by means of a proof calculus able to derive *reduction statements* of the form $e \rightarrowtriangle t$, with $e \in Exp_\perp$ and $t \in CTerm_\perp$, meaning informally that t is (or approximates to) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$ under call-time choice. The *CRWL*-proof calculus is presented in Figure 1. Rules **RR** (restricted reflexivity) and **DC** (decomposition) are used to reduce any variable to itself, and to decompose the evaluation of constructor-rooted expressions. Rule **B** (bottom) allows us to avoid the evaluation of expressions in order to get a non-strict semantics. Finally, rule **OR**

$$\textbf{RR}\ \frac{}{X \rightarrowtriangle X}\ X \in \mathcal{V} \qquad \textbf{DC}\ \frac{e_1 \rightarrowtriangle t_1 \ldots e_n \rightarrowtriangle t_n}{c(e_1,\ldots,e_n) \rightarrowtriangle c(t_1,\ldots,t_n)}\ c \in CS^n$$

$$\textbf{B}\ \frac{}{e \rightarrowtriangle \perp} \qquad \textbf{OR}\ \frac{e_1 \rightarrowtriangle p_1\theta \ldots e_n \rightarrowtriangle p_n\theta \quad r\theta \rightarrowtriangle t}{f(e_1,\ldots,e_n) \rightarrowtriangle t}\ \begin{array}{l} f(p_1,\ldots,p_n) \to r \in \mathcal{P} \\ \theta \in CSubst_\perp \end{array}$$

Figure 1. Rules of *CRWL*.

(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 some substitution $\theta \in CSubst_\perp$), and then reduce the correspondingly instantiated right-hand side. The use of partial c-substitutions in **OR** is essential to express call-time choice, as only single partial values are used for parameter passing. Notice also that by the effect of θ in **OR**, extra variables on the right-hand side of a rule can be replaced by any partial c-term, but not by any expression as in term rewriting.

We write $\mathscr{P} \vdash_{CRWL} e \rightarrowtriangle t$ to express that $e \rightarrowtriangle t$ is derivable in the *CRWL*-calculus using the program $\mathscr{P}$. Given a program $\mathscr{P}$, the *CRWL-denotation* of an expression $e \in Exp_\perp$ is defined as $[\![e]\!]^{sg}_{\mathscr{P}} = \{t \in CTerm_\perp \mid \mathscr{P} \vdash_{CRWL} e \rightarrowtriangle t\}$. In the following, we will usually omit the reference to $\mathscr{P}$ when implied by the context.

3 Two plural semantics for constructor systems

In this section we present two semantic proposals for constructor systems that are plural in the sense described in the Introduction, but at the same time are different to the run-time choice semantics induced by term rewriting. We will formalize them by means of two modifications of the *CRWL*-proof calculus, that will now consider sets of partial values for parameter passing instead of single partial values. Thus, only the rule **OR** should be modified. To avoid the need to extend the syntax with new constructions to represent those "set expressions" that we mentioned in the Introduction, we will exploit the fact that $[\![e_1\ ?\ e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$ for any sensible semantics – in particular each of the semantics considered in this work. Therefore, the substitutions used for parameter passing will map variables to "disjunctions of values." We define the set $CSubst^?_\perp = \{\theta \in Subst_\perp \mid \forall X \in dom(\theta), \theta(X) = t_1\ ?\ \ldots\ ?\ t_n$ such that $t_1,\ldots,t_n \in CTerm_\perp, n > 0\}$, for which $CSubst_\perp \subseteq CSubst^?_\perp \subseteq Subst_\perp$ obviously holds. The operator $? : CSubst^*_\perp \to CSubst^?_\perp$ constructs the $CSubst^?_\perp$ corresponding to a non-empty sequence of $CSubst_\perp$, and it is defined as follows:

$$?(\theta_1 \ldots \theta_n)(X) = \begin{cases} X\ ?\ \rho_1(X)\ ?\ \ldots\ ?\ \rho_m(X) & \text{if } \exists\theta_i \text{ such that } X \notin dom(\theta_i) \\ \theta_1(X)\ ?\ \ldots\ ?\ \theta_n(X) & \text{otherwise} \end{cases}$$

where $\rho_1 \ldots \rho_m = \theta_1 \ldots \theta_n \mid \lambda\theta.(X \in dom(\theta))$. This operator is overloaded to handle non-empty sets $\Theta \subseteq CSubst_\perp$ as $?\Theta = ?(\theta_1 \ldots \theta_n)$, where the sequence $\theta_1 \ldots \theta_n$ corresponds to an arbitrary reordering of the elements of Θ – for example using some standard order of terms in the line of Sterling and Shapiro (1986).

Lemma 1
For any $\theta_1, \ldots, \theta_n \in CSubst_\bot$, $dom(?\{\theta_1 \ldots \theta_n\}) = \bigcup_i dom(\theta_i)$.

Proof
Simple calculations using the definition of $?\{\theta_1 \ldots \theta_n\}$ (see Riesco and Rodríguez-Hortalá 2011 for details). □

3.1 $\pi^\alpha CRWL$

Our first semantic proposal is defined by the $\pi^\alpha CRWL$-proof calculus in Figure 2. The only difference with the $CRWL$-proof calculus in Figure 1 is that the rule **OR** has been replaced by **POR**$^\alpha$ (alpha plural outer reduction), in which we may compute more than one partial value for each argument, and then use a substitution from $CSubst^?_\bot$ instead of $CSubst_\bot$ for parameter passing, achieving a plural semantics.[5] Besides, extra variables are instantiated by an arbitrary $\theta_e \in CSubst^?_\bot$ for the same reason. Just like $CRWL$, the calculus evaluates expressions in the innermost way, and avoids the use of any transitivity rule that would induce a step-wise semantics like, for example, term rewriting. The motivation for that is to get a compositional calculus in the values it computes, i.e., the semantics of an expression would only depend on the semantics of its constituents, in a simple way – we will give a formal characterization for that in Theorem 1. Note that the use of partial c-terms as values is crucial to prevent the innermost evaluation from making functions strict, thus losing lazy evaluation. Fortunately the rule **B** combined with the use of partial substitutions for parameter passing ensure a lazy behavior for both $\pi^\alpha CRWL$ and $CRWL$. Therefore, we could roughly describe the parameter passing of $CRWL$ as call-by-partial value, while $\pi^\alpha CRWL$ would perform call-by-set-of-partial values.

The calculus derives *reduction statements* of the form $\mathcal{P} \vdash_{\pi^\alpha CRWL} e \rightarrowtriangle t$, which expresses that t is (or approximates to) a possible value for e in this semantics under the program $\mathcal{P}$. For any $\pi^\alpha CRWL$-proof we define its *size* as the number of applications of rules of the calculus. The *$\pi^\alpha CRWL$-denotation* of an expression $e \in Exp_\bot$ under a program $\mathcal{P}$ in $\pi^\alpha CRWL$ is defined as $[\![e]\!]^{\alpha pl}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{\pi^\alpha CRWL} e \rightarrowtriangle t\}$. In the following, we will usually omit the reference to $\mathcal{P}$ and αpl, and even will skip $\vdash_{\pi^\alpha CRWL}$ when it is clearly implied by the context.

Example 3.1
Consider the program of Example 1.1 that is $\{f(c(X)) \rightarrow d(X,X),\ X\ ?\ Y \rightarrow X,\ X\ ?\ Y \rightarrow Y\}$. The following is a $\pi^\alpha CRWL$-proof for the statement $f(c(0)\ ?\ c(1)) \rightarrowtriangle d(0,1)$ (some steps have been omitted for the sake of conciseness):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,\mathbf{DC}}{c(0) \rightarrowtriangle c(0)}\,\mathbf{DC} \quad \dfrac{}{c(1) \rightarrowtriangle \bot}\,\mathbf{B} \quad \dfrac{\vdots}{c(0) \rightarrowtriangle c(0)}}{c(0)?c(1) \rightarrowtriangle c(0)}\,\mathbf{POR}^\alpha \quad \dfrac{\vdots}{c(0)?c(1) \rightarrowtriangle c(1)} \quad \dfrac{\dfrac{\vdots}{0?1 \rightarrowtriangle 0} \quad \dfrac{\vdots}{0?1 \rightarrowtriangle 1}}{d(0?1,0?1) \rightarrowtriangle d(0,1)}\,\mathbf{DC}}{f(c(0)?c(1)) \rightarrowtriangle d(0,1)}\,\mathbf{POR}^\alpha$$

[5] In fact, angelic non-strict plural non-determinism.

$$\textbf{RR}\ \frac{}{X \rightarrowtriangle X}\ X \in \mathcal{V} \qquad \textbf{DC}\ \frac{e_1 \rightarrowtriangle t_1 \ldots e_n \rightarrowtriangle t_n}{c(e_1, \ldots, e_n) \rightarrowtriangle c(t_1, \ldots, t_n)}\ c \in CS^n$$

$$\textbf{B}\ \frac{}{e \rightarrowtriangle \perp} \qquad \textbf{POR}^{\alpha}\ \frac{\begin{array}{ccc} e_1 \rightarrowtriangle p_1\theta_{11} & & e_n \rightarrowtriangle p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \rightarrowtriangle p_1\theta_{1m_1} & & e_n \rightarrowtriangle p_n\theta_{nm_n} \quad r\theta \rightarrowtriangle t \end{array}}{f(e_1, \ldots, e_n) \rightarrowtriangle t}$$

$$\begin{array}{l} \text{if } (f(\overline{p}) \to r) \in \mathcal{P}, \forall i \in \{1, \ldots, n\}\ \Theta_i = \{\theta_{i1}, \ldots, \theta_{im_i}\} \\ \theta = (\biguplus_{i=1}^{n} ?\Theta_i) \uplus \theta_e, \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\} \\ dom(\theta_{ij}) \subseteq var(p_i), \forall i \in \{1, \ldots, n\}\ m_i > 0 \\ dom(\theta_e) \subseteq vExtra(f(\overline{p}) \to r), \theta_e \in CSubst_{\perp}^{?} \end{array}$$

Figure 2. Rules of $\pi^{\alpha}CRWL$.

One of the most important properties of $\pi^{\alpha}CRWL$ is compositionality, a property very close to the DET-additivity property for algebraic specifications of Hussmann (1993), or the referencial transparency property of Søndergaard and Sestoft (1990). This property shows that the $\pi^{\alpha}CRWL$-denotation of any expression put in a context only depends on the $\pi^{\alpha}CRWL$ -denotation of that expression, and formalizes the idea that the semantics of the whole expression depends only on the semantics of its constituents, as we have informally pointed above.

Theorem 1 (*Compositionality of $\pi^{\alpha}CRWL$*)
For any program, $\mathscr{C} \in Cntxt$ and $e \in Exp_{\perp}$:

$$[\![\mathscr{C}[e]]\!]^{\alpha pl} = \bigcup_{\{t_1,\ldots,t_n\} \subseteq [\![e]\!]^{\alpha pl}} [\![\mathscr{C}[t_1\ ?\ldots\ ?\ t_n]]\!]^{\alpha pl}$$

for any arrangement of the elements of $\{t_1, \ldots, t_n\}$ in $t_1\ ?\ \ldots\ ?\ t_n$. As a consequence, for any $e' \in Exp_{\perp}$:

$$[\![e]\!]^{\alpha pl} = [\![e']\!]^{\alpha pl} \text{ iff } \forall \mathscr{C} \in Cntxt.[\![\mathscr{C}[e]]\!]^{\alpha pl} = [\![\mathscr{C}[e']]\!]^{\alpha pl}$$

Proof
We have to prove that, for any $t \in CTerm_{\perp}$, if $\mathscr{C}[e] \rightarrowtriangle t$ then $\exists\{s_1, \ldots, s_n\} \subseteq [\![e]\!]^{\alpha pl}$ such that $\mathscr{C}[s_1\ ?\ \ldots\ ?\ s_n] \rightarrowtriangle t$; and conversely, that given $\{s_1, \ldots, s_n\} \subseteq [\![e]\!]^{\alpha pl}$ such that $\mathscr{C}[s_1\ ?\ \ldots\ ?\ s_n] \rightarrowtriangle t$ then $\mathscr{C}[e] \rightarrowtriangle t$. Each of these statements can be proved by induction on the size of the starting proof (see Riesco and Rodríguez-Hortalá 2011 for details). □

Contrary to what happens to call-time choice (López-Fraguas *et al.* 2008; López-Fraguas *et al.* available on request), we cannot have a compositionality result for single values like $[\![\mathscr{C}[e]]\!] = \bigcup_{t \in [\![e]\!]} [\![\mathscr{C}[t]]\!]$ for any arbitrary context $\mathscr{C}$ because e could appear in a function call when put inside $\mathscr{C}$, and that function might demand more that one value from e because of the plurality of $\pi^{\alpha}CRWL$. We can see this considering the program from Example 1.1 extended with a function *coin* defined by $\{coin \to 0, coin \to 1\}$, the context $f(c([]))$ and the expression *coin*: in order to compute the value $d(0,1) \in [\![f(c(coin))]\!]$ we need $\{0,1\} \subseteq [\![coin]\!]$, while a single value of *coin* is not enough, which is reflected in the fact that $d(0,1) \in [\![f(c(0 \,?\, 1))]\!]$ while $d(0,1) \notin [\![f(c(0))]\!] \cup [\![f(c(1))]\!]$. On the other hand, note that we only need a finite subset of the denotation of the expression put in context, and not the whole denotation, which could be infinite, thus leading to $t_1 \ ? \ \ldots \ ? \ t_n$ being a malformed expression, as we only consider finite expressions in this work. To illustrate this we may consider again the program from Example 1.1, the symbols $z \in CS^0, s \in CS^1$ for the Peano natural numbers representation, and the function *from* defined as $\{from(X) \rightarrow X, from(X) \rightarrow s(from(X))\}$. Then using the same context as above and the expression $from(z)$, in order to compute $d(z, s(z)) \in [\![f(c(from(z)))]\!]$ we just need $\{z, s(z)\} \subseteq [\![from(z)]\!]$, but not the infinite set of elements in $[\![from(z)]\!]$. The intuition behind this is that, as we use c-terms as values and c-terms are finite, then any computation of a value is a finite process that only involves a finite amount of information: in this case a finite subset of the denotation of the expression put in context.

Besides compositionality, $\pi^{\alpha}CRWL$ enjoys other nice properties like the following polarity property.

Proposition 1 (*Polarity of* $\pi^{\alpha}CRWL$)
For any program $\mathcal{P}$, $e, e' \in Exp_{\perp}$, $t, t' \in CTerm_{\perp}$ if $e \sqsubseteq e'$ and $t' \sqsubseteq t$ then $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t$ implies $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e' \rightarrowtriangle t'$ with a proof of the same size or smaller.

Proof
By a simple induction on the structure of $e \rightarrowtriangle t$ using basic properties of $\sqsubseteq$ (see Riesco and Rodríguez-Hortalá 2011 for details). □

$\pi^{\alpha}CRWL$ also has some monotonicity properties related to substitutions. These are formulated using the preorder $\sqsubseteq_{\pi}$ over $CSubst^{?}_{\perp}$ defined by $\theta \sqsubseteq_{\pi} \theta'$ iff $\forall X \in \mathcal{V}$, given $\theta(X) = t_1 \ ? \ \ldots \ ? \ t_n$ and $\theta'(X) = t'_1 \ ? \ \ldots \ ? \ t'_m$ then $\forall t \in \{t_1, \ldots, t_n\} \exists t' \in \{t'_1, \ldots, t'_m\}$ such that $t \sqsubseteq t'$; and the preorder $\trianglelefteq^{\alpha pl}$ over $Subst_{\perp}$ defined by $\sigma \trianglelefteq^{\alpha pl} \sigma'$ iff $\forall X \in \mathcal{V}$, $[\![\sigma(X)]\!]^{\alpha pl} \subseteq [\![\sigma'(X)]\!]^{\alpha pl}$.

Proposition 2 (*Monotonicity for substitutions of* $\pi^{\alpha}CRWL$)
For any program, $e \in Exp_{\perp}$, $t \in CTerm_{\perp}$, $\sigma, \sigma' \in Subst_{\perp}$, $\theta, \theta' \in CSubst^{?}_{\perp}$:

(1) **Strong monotonicity of** $\boldsymbol{Subst_{\perp}}$: If $\forall X \in \mathcal{V}, s \in CTerm_{\perp}$ given $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} \sigma(X) \rightarrowtriangle s$ with size K we also have $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} \sigma'(X) \rightarrowtriangle s$ with size $K' \leqslant K$, then $\vdash_{\pi^{\alpha}CRWL} e\sigma \rightarrowtriangle t$ with size L implies $\vdash_{\pi^{\alpha}CRWL} e\sigma' \rightarrowtriangle t$ with size $L' \leqslant L$.
(2) **Monotonicity of** $\boldsymbol{CSubst_{\perp}}$: If $\theta, \theta' \in CSubst_{\perp}$ and $\theta \sqsubseteq \theta'$ then $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e\theta \rightarrowtriangle t$ with size K implies $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e\theta' \rightarrowtriangle t$ with size $K' \leqslant K$.
(3) **Monotonicity of** $\boldsymbol{Subst_{\perp}}$: If $\sigma \trianglelefteq^{\alpha pl} \sigma'$ then $[\![e\sigma]\!]^{\alpha pl} \subseteq [\![e\sigma']\!]^{\alpha pl}$.
(4) **Monotonicity of** $\boldsymbol{CSubst^{?}_{\perp}}$: If $\theta \sqsubseteq_{\pi} \theta'$ then $[\![e\theta]\!]^{\alpha pl} \subseteq [\![e\theta']\!]^{\alpha pl}$.

The properties of $\pi^{\alpha}CRWL$ we have seen so far are shared with *CRWL*, which is something natural taking into account that $\pi^{\alpha}CRWL$ is a modification of that semantics. Nevertheless, there are some properties of *CRWL* – and as a consequence, of call-time choice – that do not hold for $\pi^{\alpha}CRWL$. One of these is the correctness of the *bubbling* operational rule (Antoy *et al.* 2007), which can be formulated as “under any program and for any $\mathscr{C} \in Cntxt$, $e_1, e_2 \in Exp_\perp$ we have that $[\![\mathscr{C}[e_1 \ ?\ e_2]]\!] = [\![\mathscr{C}[e_1] \ ?\ \mathscr{C}[e_2]]\!]$.” Note that Examples 1.1 and 1.2 already show that this property does not hold for run-time choice, the following (counter) example proves that it is not the case for $\pi^\alpha CRWL$.

Example 3.2
Consider the program $\mathscr{P} = \{pair(X) \to (X,X), X \ ?\ Y \to X, X \ ?\ Y \to Y\}$ and the expressions $pair(0 \ ?\ 1)$ and $pair(0) \ ?\ pair(1)$, which correspond to a bubbling step using $\mathscr{C} = pair([])$. It is easy to check that $(0,1) \in [\![pair(0 \ ?\ 1)]\!]^{\alpha pl}$ while $(0,1) \notin [\![pair(0) \ ?\ pair(1)]\!]^{\alpha pl}$.

It was very enlightening for us to discover that the correctness of bubbling does not hold for $\pi^\alpha CRWL$, and in fact in Rodríguez-Hortalá (2008) it was wrongly considered as true. This shows that $CRWL$ and $\pi^\alpha CRWL$ are more different than these may appear at first sight. In particular, regarding bubbling, the important difference is that while $\pi^\alpha CRWL$ is only compositional with respect to subsets of the denotation, $CRWL$ is compositional with respect to single values of the denotation, as we saw above. Compositionality with respect to single values is stronger than compositionality with respect to subsets of the denotation, as the former implies the latter, and this is also exemplified by the fact that we need compositionality with respect to single values for bubbling to be correct, as we will see soon. On the other hand, compositionality with respect to subsets of the denotation is enough to obtain the result expressed at the end of Theorem 1, showing that expressions with the same values are indistinguishable, which corresponds to the value-based philosophy of FLP.

As the bubbling rule is devised to improve the efficiency of computations (Antoy *et al.* 2007), it would be nice to be able to use it in some situations, although it would only be for a restricted class of contexts. In this line, we have found that bubbling is still correct under $\pi^\alpha CRWL$ for a particular kind of contexts called *constructor contexts* or just *c-contexts*, which are contexts whose holes appear under a nested application of constructor symbols only, that is, $c\mathscr{C} ::= [\,] \mid c(e_1, \ldots, c\mathscr{C}, \ldots, e_n)$, with $c \in CS^n, e_1, \ldots, e_n \in Exp_\perp$. For c-contexts, $\pi^\alpha CRWL$ enjoys the same compositionality for single values as $CRWL$ – that property holds in $CRWL$ for arbitrary contexts – as shown in the following result.

Proposition 3 (*Compositionality of $\pi^\alpha CRWL$ for c-contexts*)
For any program, c-context $c\mathscr{C}$ and $e \in Exp_\perp$:

$$[\![c\mathscr{C}[e]]\!]^{\alpha pl} = \bigcup_{t \in [\![e]\!]^{\alpha pl}} [\![c\mathscr{C}[t]]\!]^{\alpha pl}$$

Proof
Very similar to the proof for the general compositionality of $\pi^\alpha CRWL$ from Theorem 1 (see Riesco and Rodríguez-Hortalá 2011 for details). □

As compositionality for single values is the key property needed for bubbling to be correct, we get the following result for bubbling in $\pi^\alpha CRWL$.

Proposition 4 (*Bubbling for c-contexts in $\pi^{\alpha}CRWL$*)

For any program, c-context $c\mathscr{C}$ and $e_1, e_2 \in Exp_{\perp}$, $[\![c\mathscr{C}[e1 \mathbin{?} e_2]]\!]^{\alpha pl} = [\![c\mathscr{C}[e_1] \mathbin{?} c\mathscr{C}[e_2]]\!]^{\alpha pl}$.

Proof

It is easy to prove that $\forall e_1, e_2 \in Exp_{\perp}$ we have $[\![e1 \mathbin{?} e_2]\!]^{\alpha pl} = [\![e_1]\!]^{\alpha pl} \cup [\![e_2]\!]^{\alpha pl}$ (see Riesco and Rodríguez-Hortalá 2011). But then:

$$\begin{array}{ll}
[\![c\mathscr{C}[e1 \mathbin{?} e_2]]\!]^{\alpha pl} & \\
= \bigcup_{t \in [\![e1 \,?\, e_2]\!]^{\alpha pl}} [\![c\mathscr{C}[t]]\!]^{\alpha pl} & \text{by Proposition 3} \\
= \bigcup_{t \in [\![e1]\!]^{\alpha pl} \cup [\![e_2]\!]^{\alpha pl}} [\![c\mathscr{C}[t]]\!]^{\alpha pl} & \\
= \bigcup_{t \in [\![e1]\!]^{\alpha pl}} [\![c\mathscr{C}[t]]\!]^{\alpha pl} \cup \bigcup_{t [\![e_2]\!]^{\alpha pl}} [\![c\mathscr{C}[t]]\!]^{\alpha pl} & \\
= [\![c\mathscr{C}[e_1]]\!]^{\alpha pl} \cup [\![c\mathscr{C}[e_2]]\!]^{\alpha pl} & \text{by Proposition 3} \\
= [\![c\mathscr{C}[e_1] \mathbin{?} c\mathscr{C}[e_2]]\!]^{\alpha pl} &
\end{array}$$

□

We end our presentation of $\pi^{\alpha}CRWL$ with an example showing how we can use $\pi^{\alpha}CRWL$ to model problems in which some collecting work has to be done.

Example 3.3

We want to represent the database of a bank in which we hold some data about its employees. This bank has several branches and we want to organize the information according to them. To do that we define a non-deterministic function *branches* to represent the set of branches: a set is then identified with a non-deterministic expression. We also use this technique to define non-deterministic function *employees*, which conceptually returns, for a given branch, the set of records containing the information regarding the employees that work in that branch. Now we want to search for the names of two clerks, which may be working in different branches. To do this we define the function *twoclerks*, which is based upon the function *find*, which forces the desired pattern $e(N, G, clerk)$ over the set defined by the expression $employees(branches)$:

$$\begin{array}{l}
\mathscr{P} = \{branches \to madrid, \\
\quad branches \to vigo, \\
\quad employees(madrid) \to e(pepe, man, clerk), \\
\quad employees(madrid) \to e(paco, man, clerk), \\
\quad employees(vigo) \to e(maria, woman, clerk), \\
\quad employees(vigo) \to e(jaime, woman, clerk), \\
\quad twoclerks \to find(employees(branches)), \\
\quad find(e(N, G, clerk)) \to (N, N)\}
\end{array}$$

With term rewriting $twoclerks \to find(employees(branches)) \not\to^{*} (pepe, maria)$, because in that expression the evaluation of *branches* is needed and thus one of the branches must be chosen. On the other hand, with $\pi^{\alpha}CRWL$ the value $(pepe, maria)$ can be computed for *twoclerks* (some steps have been omitted for the sake of conciseness, *emps* abbreviates *employees*, and *brs* abbreviates *branches*):

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{emps(brs) \rightarrowtriangle e(pepe, \perp, clerk)}\,\mathbf{POR}^\alpha \quad \dfrac{\vdots}{(pepe\ ?\ maria, pepe\ ?\ maria) \rightarrowtriangle (pepe, maria)}\,\mathbf{DC} \quad \dfrac{\vdots}{emps(brs) \rightarrowtriangle e(maria, \perp, clerk)}\,\mathbf{POR}^\alpha}{find(emps(brs)) \rightarrowtriangle (pepe, maria)}\,\mathbf{POR}^\alpha}{twoclerks \rightarrowtriangle (pepe, maria)}\,\mathbf{POR}^\alpha}{}$$

where

$$\dfrac{\dfrac{\dfrac{}{madrid \rightarrowtriangle madrid}\,\mathbf{DC}}{brs \rightarrowtriangle madrid}\,\mathbf{POR}^\alpha \quad \dfrac{\vdots}{e(pepe, man, clerk) \rightarrowtriangle e(pepe, \perp, clerk)}\,\mathbf{DC}}{emps(brs) \rightarrowtriangle e(pepe, \perp, clerk)}\,\mathbf{POR}^\alpha$$

3.2 $\pi^\beta CRWL$

So far we have presented our first proposal for a plural semantics for constructor systems, seen some interesting properties, and how to use it to solve collecting problems. Nevertheless, this semantics also has some weak points that will be illustrated by the following example.

Example 3.4
Starting from the program of Example 3.3, we want to search for the names of two clerks paired with their corresponding genders. Therefore, following the same ideas, we define a function *find2NG* that forces the desired pattern but now returning both the name and the gender of two clerks, by the rule $find2NG(e(N, G, clerk)) \to ((N, G), (N, G))$. Then, $((pepe, man), (maria, woman))$ would be one of the values computed for the expression $find2NG(employees(branches))$, as expected. Nevertheless, we can also compute the value $((pepe, woman), (maria, man))$, which obviously does not correspond to the intended meaning of *find2NG*, as can be seen in the following proof (using the abbreviations above and also m for *man*, and w for *woman*).

$$\dfrac{\dfrac{\vdots}{emps(brs) \rightarrowtriangle e(pepe, m, clerk)} \quad \dfrac{\vdots}{emps(brs) \rightarrowtriangle e(maria, w, clerk)} \quad \dfrac{\vdots}{\begin{array}{l}((pepe\ ?\ maria, m\ ?\ w), (pepe\ ?\ maria, m\ ?\ w))\\ \quad \rightarrowtriangle ((pepe, w), (maria, m))\end{array}}}{find2NG(emps(brs)) \rightarrowtriangle ((pepe, w), (maria, m))}\,\mathbf{POR}^\alpha$$

This example is interesting because it shows a relevant flaw of $\pi^\alpha CRWL$, since there the matching substitutions $[N/pepe, G/man]$ and $[N/maria, G/woman]$ obtained for different evaluations of the argument $employees(branches)$ are wrongly intermingled. Anyway, the program is not well conceived, as it does not specify that each of the (N, G) pairs correspond to a particular clerk in the database, thus preventing an unintended information mix-up. Nevertheless, a better semantic behavior would have prevented “mixed” results like $((pepe, woman), (maria, man))$, thus getting $((maria, woman), (maria, woman))$ and $((pepe, man), (pepe, man))$ as the only total values for $find2NG(employees(branches))$, which does not fix the program but at least avoids wrong information mix-up.[6]

This problem was also pointed out in Braßel and Berghammer (2009), where an identification between $d(0, 0)\ ?\ d(1, 1)$ and $d(0\ ?\ 1, 0\ ?\ 1)$ – for $d \in CS^2$ and $0, 1 \in CS^0$ – made by $\pi^{\alpha}CRWL$ for relevant contexts was reported. In the technical setting presented in that paper another plural semantics that avoids this problem is proposed, although its technical relation with call-time or run-time choice is neither formally stated nor proved. In that work, that particular plurality is achieved by allowing bubbling steps for constructor applications by means of a rule that could be expressed in our syntax as $[\![c(e_1, \dots, e'_1\ ?\ e'_2, \dots, e_n)]\!] = [\![c(e_1, \dots, e'_1, \dots, e_n)\ ?\ c(e_1, \dots, e'_2, \dots, e_n)]\!]$. This kind of rules are well suited for a step-wise semantics like the one presented in Braßel and Berghammer (2009), but are more difficult to integrate with a goal-oriented proof calculus in the style of $CRWL$ or $\pi^{\alpha}CRWL$, which – as we saw in the presentation of $\pi^{\alpha}CRWL$ above – perform a kind of innermost evaluation of expressions by exploiting the use of partial values to get a compositional calculus for a lazy semantics.

Hence, in order to adapt this idea to our framework, we could *switch from bubbling under constructors to bubbling of* $CSubst^?_\perp$, *allowing the combination of substitutions that only differ in the value they assign to a single variable*. This can be realized by defining a binary operator $\sqcup$ to combine partial c-substitutions and a reduction notion $\rightarrow_\sqcup$ defined by the rule $(\theta \uplus [X/e_1]) \sqcup (\theta \uplus [X/e_2]) \rightarrow_\sqcup \theta \uplus [X/e_1\ ?\ e_2]$ that corresponds to a bubbling step for substitutions. Using this we could, for example, perform the following bubbling derivation for substitutions.

$$\begin{aligned}
&\underline{[X/0, Y/0] \sqcup [X/0, Y/1]} \sqcup [X/1, Y/0] \sqcup [X/1, Y/1] \\
&\rightarrow_\sqcup [X/0, Y/0\ ?\ 1] \sqcup \underline{[X/1, Y/0] \sqcup [X/1, Y/1]} \\
&\rightarrow_\sqcup \underline{[X/0, Y/0\ ?\ 1] \sqcup [X/1, Y/0\ ?\ 1]} \rightarrow_\sqcup [X/0\ ?\ 1, Y/0\ ?\ 1]
\end{aligned}$$

This derivation shows a criterion that determines that the set of c-substitutions $\{[X/0, Y/0], [X/0, Y/1], [X/1, Y/0], [X/1, Y/1]\}$ can be safely combined into $[X/0\ ?\ 1, Y/0\ ?\ 1] \in CSubst^?_\perp$ with no wrong substitution mix-up. On the other hand, for $[X/0, Y/0] \sqcup [X/1, Y/1]$ we should not be able to perform any $\rightarrow_\sqcup$ step as these substitutions differ in more than one variable, thus failing to combine those c-substitutions into a single element from $CSubst^?_\perp$. As can be seen in Figure 2, the key for getting a plural behavior in $\pi^{\alpha}CRWL$ is finding a way to combine different matching substitutions obtained from the evaluation of the same expression, therefore this new combination method should give rise to another plural semantic proposal. We conjecture that the resulting semantics expresses the same plural semantics proposed in Braßel and Berghammer (2009) – the one resulting in that setting when only variables of sort Ch (as defined in that paper) are used – although

[6] In Section 5 we will see how to combine singular and plural *function arguments* to solve generalization of this problem.

we will not give any formal result relating to both proposals. Let us call $\pi^\beta CRWL$ to this new semantics in which parameter passing is only performed with substitutions from $CSubst^?_\perp$ that come from a successful combination of c-substitutions using the relation $\rightarrow_\sqcup$, and consider the behavior of different plural semantics in the following example.

Example 3.5
Consider the constructors $c \in CS^1$, $d \in CS^2$, $l \in CS^4$, and $0, 1 \in CS^0$, and the following program.

$$\begin{array}{ll} f(c(X)) \rightarrow d(X,X) & h(d(X,Y)) \rightarrow d(X,X) \\ g(d(X,Y)) \rightarrow l(X,X,Y,Y) & k(d(X,Y)) \rightarrow d(X,Y) \end{array}$$

- $f(c(0)\ ?\ c(1))$ and $f(c(0\ ?\ 1))$ behave the same in both $\pi^\alpha CRWL$ and $\pi^\beta CRWL$. In this case there is only one variable involved in the matching substitution and thus no substitution mix-up like the ones seen before may appear. That is, for both expressions we only have to combine the substitutions $[X/0]$ and $[X/1]$, thus reaching the values $d(0,0)$, $d(0,1)$, $d(1,0)$, and $d(1,1)$ in both semantics.
- More surprisingly, we also get the same behavior for $h(d(0,0)\ ?\ d(1,1))$ and $h(d(0\ ?\ 1, 0\ ?\ 1))$ in both $\pi^\alpha CRWL$ and $\pi^\beta CRWL$. There the suspicious expression is $h(d(0,0)\ ?\ d(1,1))$, which generates the matching substitutions $[X/0, Y/0]$ and $[X/1, Y/1]$ that are wrongly combined by $\pi^\alpha CRWL$ into the substitution $?\{[X/0, Y/0], [X/1, Y/1]\} = [X/0\ ?\ 1, Y/0\ ?\ 1]$, used to instantiate the right-hand side of the rule for h. But this mistake has no consequence because only X appears on the right-hand side of the rule for h, therefore it has the same effect as combining $[X/0, Y/\perp]$ and $[X/1, Y/\perp]$ into $[X/0\ ?\ 1, Y/\perp]$, which is just what is done in $\pi^\beta CRWL$ as we will see later on.

  On the other hand, $h(d(0\ ?\ 1, 0\ ?\ 1))$ is not problematic as it generates the matching substitutions $[X/0, Y/0]$, $[X/0, Y/1]$, $[X/1, Y/0]$, and $[X/1, Y/1]$ that already cover all the possible instantiations of X and Y caused by its combination in $\pi^\alpha CRWL$, the substitution $[X/0\ ?\ 1, Y/0\ ?\ 1]$. The point is that in a sense both $\{[X/0, Y/0], [X/0, Y/1], [X/1, Y/0], [X/1, Y/1]\}$ and $[X/0\ ?\ 1, Y/0\ ?\ 1]$ have the same power. This will also be reflected by the fact that $\pi^\beta CRWL$ would be able to combine the former set into the latter $CSubst^?_\perp$.

  Again, we can reach the values $d(0,0)$, $d(0,1)$, $d(1,0)$, and $d(1,1)$ for each expression in both semantics.
- It is for the expressions $g(d(0,0)\ ?\ d(1,1))$ and $g(d(0\ ?\ 1, 0\ ?\ 1))$ that we can see a different behavior of $\pi^\alpha CRWL$ and $\pi^\beta CRWL$. Once again $g(d(0\ ?\ 1, 0\ ?\ 1))$ is not problematic, and for it we can get the values $l(0,0,0,0)$, $l(0,0,0,1)$, *et al.* and all the combinations of 0 and 1 in both semantics. But for $g(d(0,0)\ ?\ d(1,1))$ we have that, for example, to compute $l(0,0,0,1)$ we need the expression $d(0,0)\ ?\ d(1,1)$ to generate both 0 and 1 for Y in the matching substitutions. The only (total) matching substitutions that can be obtained from the evaluation of $d(0,0)\ ?\ d(1,1)$ are $[X/0, Y/0]$ and $[X/1, Y/1]$, which

cannot be combined by $\pi^{\beta}CRWL$, hence we cannot get both 0 and 1 for Y in the combined substitution. As a consequence $l(0,0,0,0)$ and $l(1,1,1,1)$ are the only values computed for $g(d(0,0)\ ?\ d(1,1))$ by $\pi^{\beta}CRWL$. On the other hand, $\pi^{\alpha}CRWL$ computes all the combinations of 0 and 1 – like it did for $g(d(0\ ?\ 1, 0\ ?\ 1))$ – as it is able to combine $\{[X/0, Y/0], [X/1, Y/1]\}$ into $[X/0\ ?\ 1, Y/0\ ?\ 1]$.

- A more exotic discovery is that $k(d(0,0)\ ?\ d(1,1))$ does not behave the same as for call-time choice, run-time choice, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$, even though it only uses a right-linear program rule, and it is a known fact that call-time choice and run-time choice are equivalent for right-linear programs (Hussmann 1993). $CRWL$ (call-time choice), term rewriting (run-time choice), and $\pi^{\beta}CRWL$ only compute the values $d(0,0)$ and $d(1,1)$ for $k(d(0,0)\ ?\ d(1,1))$ in the case of $\pi^{\beta}CRWL$ because it fails to combine $[X/0, Y/0]$ and $[X/1, Y/1]$. Nevertheless, $\pi^{\alpha}CRWL$ is able to combine those substitutions into $[X/0\ ?\ 1, Y/0\ ?\ 1]$, thus getting additional values $d(0,1)$ and $d(1,0)$ for the expression $k(d(0,0)\ ?\ d(1,1))$. However, we still strongly conjecture that $\pi^{\beta}CRWL$ – as formulated below – is equivalent to call-time choice and run-time choice for right-linear programs.

The previous example motivates the interest of a formal definition of $\pi^{\beta}CRWL$. It would be nice if it were by means of a proof calculus similar to $CRWL$ and $\pi^{\alpha}CRWL$because then their comparison would be easier, and maybe they could even share some of their properties, in particular compositionality. The above ideas regarding bubbling derivations for substitutions have given us the right intuitions, but these derivations are not so easy to handle as the following characterization of *compressible sets of c-substitutions* illustrates, which will be the only sets of substitutions that will be combined by $\pi^{\beta}CRWL$.

Definition 1 (*Compressible set of* $CSubst_{\perp}$)

A finite set $\Theta \subseteq CSubst_{\perp}$ is *compressible* iff for $\{X_1, \ldots, X_n\} = \bigcup_{\theta \in \Theta} dom(\theta)$

$$\{(X_1\theta, \ldots, X_n\theta) \mid \theta \in \Theta\} = \{X_1\theta_1 \mid \theta_1 \in \Theta\} \times \ldots \times \{X_n\theta_n \mid \theta_n \in \Theta\}$$

Note that this property is easily computable for Θ finite, as we only consider finite domain substitutions.

Example 3.6

Let us see how the notion of compressible set of c-substitutions can be used to replace the relation $\rightarrow_{\sqcup}$ sketched above. We have seen that the substitutions $[X/0, Y/0]$ and $[X/1, Y/1]$ should not be combined to prevent a wrong substitution mix-up. This is reflected in the fact that the set $\{[X/0, Y/0], [X/1, Y/1]\}$ is not compressible because:

$$\begin{aligned}
&\{(X\theta, Y\theta) \mid \theta \in \{[X/0, Y/0], [X/1, Y/1]\}\} = \{(0,0), (1,1)\}\\
&\neq \{(0,0), (0,1), (1,0), (1,1)\} = \{0,1\} \times \{0,1\}\\
&= \{X\theta_x \mid \theta_x \in \{[X/0, Y/0], [X/1, Y/1]\}\} \times \{Y\theta_y \mid \theta_y \in \{[X/0, Y/0], [X/1, Y/1]\}\}
\end{aligned}$$

$$\textbf{RR}\ \frac{}{X \twoheadrightarrow X}\ X \in \mathcal{V} \qquad \textbf{DC}\ \frac{e_1 \twoheadrightarrow t_1 \ldots e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)}\ c \in CS^n$$

$$\textbf{B}\ \frac{}{e \twoheadrightarrow \perp} \qquad \textbf{POR}^{\beta}\ \frac{\begin{array}{ccc} e_1 \twoheadrightarrow p_1\theta_{11} & & e_n \twoheadrightarrow p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \twoheadrightarrow p_1\theta_{1m_1} & & e_n \twoheadrightarrow p_n\theta_{nm_n} \quad r\theta \twoheadrightarrow t \end{array}}{f(e_1,\ldots,e_n) \twoheadrightarrow t}$$

$$\begin{array}{l} \text{if } (f(\overline{p}) \to r) \in \mathcal{P},\ \forall i \in \{1,\ldots,n\}\ \Theta_i = \{\theta_{i1},\ldots,\theta_{im_i}\} \\ \text{is compressible, } \theta = (\biguplus_{i=1}^{n} ?\Theta_i) \uplus \theta_e, \forall i \in \{1,\ldots,n\}, \\ j \in \{1,\ldots,m_i\} dom(\theta_{ij}) \subseteq var(p_i), \forall i \in \{1,\ldots,n\}\ m_i > 0 \\ dom(\theta_e) \subseteq vExtra(f(\overline{p}) \to r), \theta_e \in CSubst^{?}_{\perp} \end{array}$$

Figure 3. Rules of $\pi^{\beta}CRWL$.

On the other hand, for $\Theta = \{[X/0, Y/0], [X/0, Y/1], [X/1, Y/0], [X/1, Y/1]\}$ the substitutions it contains can be safely combined, therefore we should have that Θ is compressible, as it happens:

$$\begin{aligned} &\{(X\theta, Y\theta) \mid \theta \in \Theta\} = \{(0,0),(0,1),(1,0),(1,1)\} \\ &= \{0,1\} \times \{0,1\} = \{X\theta_x \mid \theta_x \in \Theta\} \times \{Y\theta_y \mid \theta_y \in \Theta\} \end{aligned}$$

Our last proposal for a plural semantics for CSs is based on the notion of compressible set of c-substitutions, and it is defined by the $\pi^{\beta}CRWL$-proof calculus in Figure 3. Note that the only difference with $\pi^{\alpha}CRWL$ is that the rule $\mathbf{POR}^{\alpha}$ is replaced by $\mathbf{POR}^{\beta}$, which now demands the different matching substitutions obtained from the evaluation of each function argument to be compressible. Apart from that, compressible sets of partial c-substitutions are combined just like in $\pi^{\alpha}CRWL$by means of the ? operator.

This calculus, like $CRWL$ and $\pi^{\alpha}CRWL$, also derives *reduction statements* of the form $\mathcal{P} \vdash_{\pi^{\beta}CRWL} e \twoheadrightarrow t$, which expresses that t is (or approximates to) a possible value for e in this semantics under the program $\mathcal{P}$. Then the $\pi^{\beta}CRWL$-*denotation* of an expression $e \in Exp_{\perp}$ under the program $\mathcal{P}$ in $\pi^{\beta}CRWL$ is defined as $[\![e]\!]^{\beta pl}_{\mathcal{P}} = \{t \in CTerm_{\perp} \mid \mathcal{P} \vdash_{\pi^{\beta}CRWL} e \twoheadrightarrow t\}$. In the following, we will usually omit the reference to $\mathcal{P}$ when implied by the context.

Example 3.7
Consider the program of Example 3.4, $\pi^{\beta}CRWL$ is able to avoid computing the value $((pepe, woman), (maria, man))$ for the expression $find2NG(employees(branches))$ because the set of matching substitutions $\{[N/pepe, G/man], [N/maria, G/woman]\}$ is not compressible as can be easily checked by applying Definition 1 in a way similar to Example 3.6. Nevertheless, the values $((maria, woman), (maria, woman))$ and $((pepe, man), (pepe, man))$ can be computed for $find2NG(employees(branches))$ by using the sets of substitutions $\{[N/pepe, G/man]\}$ and $\{[N/maria, G/woman]\}$, respectively, for parameter passing, which are compressible, as they are singletons. As we saw in Example 3.4, the function $find2NG$ is wrongly conceived because it does not specify that in each pair (N, G) the name N and the genre G must

correspond to the same clerk. $\pi^\beta CRWL$ cannot fix a wrong program, but at least is able to prevent "mixed" results like $((pepe, woman), (maria, man))$.

It is also easy to check that $\pi^\beta CRWL$ has the same behavior as $\pi^\alpha CRWL$ for Example 3.3, as sets like $\{[N/pepe, G/\perp], [N/maria, G/\perp]\}$ are compressible. Similarly, in Example 3.5, functions f and h behave the same under both semantics, and $\pi^\beta CRWL$ also behaves for h and k as specified there, because $\{[X/0, Y/0], [X/1, Y/1]\}$ is not compressible, just like $\{[N/pepe, G/man], [N/maria, G/woman]\}$, while for $\Theta = \{[X/0, Y/0], [X/0, Y/1], [X/1, Y/0], [X/1, Y/1]\}$ we have that Θ is compressible as seen in Example 3.6.

The following result shows that part of the equality that defines compressibility always holds trivially, thus simplifying the definition of compressible set of c-substitutions.

Lemma 2
For any finite set $\Theta \subseteq CSubst_\perp$ for $\{X_1, \ldots, X_n\} = \bigcup_{\theta \in \Theta} dom(\theta)$ we have

$$\{(X_1\theta, \ldots, X_n\theta) \mid \theta \in \Theta\} \subseteq \{X_1\theta_1 \mid \theta_1 \in \Theta\} \times \ldots \times \{X_n\theta_n \mid \theta_n \in \Theta\}$$

As a consequence Θ is compressible iff

$$\{(X_1\theta, \ldots, X_n\theta) \mid \theta \in \Theta\} \supseteq \{X_1\theta_1 \mid \theta_1 \in \Theta\} \times \ldots \times \{X_n\theta_n \mid \theta_n \in \Theta\}$$

This gives another criterion to prove compressibility: Θ is compressible iff $\forall \theta_1, \ldots, \theta_n \in \Theta$. $\exists \theta \in \Theta$ such that $\forall i. X_i\theta_i \equiv X_i\theta$ (which implies that $(X_1\theta_1, \ldots, X_n\theta_n) \equiv (X_1\theta, \ldots, X_n\theta)$).

In a way this result exemplifies why $\pi^\beta CRWL$ is smaller than $\pi^\alpha CRWL$ in the sense that in general it computes less values for a given expression under a given program, as $\{X_1\theta_1 \mid \theta_1 \in \Theta\} \times \ldots \times \{X_n\theta_n \mid \theta_n \in \Theta\}$ corresponds to the substitution $?\Theta$ that is always used for parameter passing in $\pi^\alpha CRWL$with no previous compressibility test. We will see more about the relations between call-time choice, run-time choice, $\pi^\alpha CRWL$, and $\pi^\beta CRWL$ in Section 4.

We have just seen how $\pi^\beta CRWL$ corrects the excessive permissiveness of the combinations of substitutions performed by $\pi^\alpha CRWL$, but will it be able to do it while keeping the nice properties of $\pi^\alpha CRWL$– in particular compositionality – at the same time. Fortunately, the answer is yes as shown by the following result.

Theorem 2 (*Basic properties of* $\pi^\beta CRWL$)
The basic properties of $\pi^\alpha CRWL$ also hold for $\pi^\beta CRWL$ under any program, i.e, the corresponding versions of Theorem 1 and Propositions 1–4 also hold for $\pi^\beta CRWL$.

For Proposition 2 in particular we replace $\trianglelefteq^{\alpha pl}$ with $\trianglelefteq^{\beta pl}$, which is defined in terms of $\pi^\beta CRWL$ instead of $\pi^\alpha CRWL$, i.e., $\sigma \trianglelefteq^{\beta pl} \sigma'$ iff $\forall X \in \mathcal{V}, [\![\sigma(X)]\!]^{\beta pl} \subseteq [\![\sigma'(X)]\!]^{\beta pl}$. Nevertheless, in the following we will often omit the superscripts αpl and βpl in $\trianglelefteq^{\alpha pl}$ and $\trianglelefteq^{\beta pl}$ when those are implied by the context.

Proof
In each proof for the $\pi^\alpha CRWL$ versions of these results we start from a given $\pi^\alpha CRWL$-proof and build another one using a bigger expression with respect to $\sqsubseteq$, a

more powerful substitution, interchanging an expression with an alternative of some of its values. Therefore, we can use the same technique for $\pi^\beta CRWL$ to replicate any $\mathbf{POR}^\beta$ step in the starting $\pi^\beta CRWL$-proof by using the substitution used there for parameter passing, which must be compressible by hypothesis, and that we are able to obtain by using a similar reasoning to that performed in the proof for the corresponding result for $\pi^\alpha CRWL$. □

In this section we have presented two different proposals for a plural semantics for non-deterministic constructor systems that are different from run-time choice. The first one, $\pi^\alpha CRWL$, is a pretty simple extension of $CRWL$ that comes up naturally from allowing the combination of several matching substitution through the operator ? for c-substitutions. But it is precisely the simplicity of that combination that leads to a wrong information mix-up in some situations. These problems are solved in $\pi^\beta CRWL$, in which a compressibility test is added to prevent wrong combination of substitutions. This could suggest that $\pi^\alpha CRWL$ is only a preliminary attempt that should now be put aside and forgotten. Nevertheless, $\pi^\alpha CRWL$ will still be very useful for us, again because of its simplicity, as we will see in subsequent sections.

Finally, note that both $\pi^\alpha CRWL$ and $\pi^\beta CRWL$ have been devised starting from $CRWL$ and then adding some criterion for combining different matching substitutions for the same argument, so any number of alternative plural – and even also compositional, possibly – semantics for constructor systems could be conceived just by defining new combination procedures.

4 Hierarchy, equivalence, and simulation

In this section we will first compare different characteristics of the semantics considered so far, with a special emphasis in the set of computed c-terms. Then we will present a class of programs characterized by a simple syntactic criterion under which our two plural semantics are equivalent. Finally, we will conclude the section presenting a program transformation that can be used to simulate our plural semantics by using term rewriting.

4.1 A hierarchy of semantics

We have already seen that $CRWL$, $\pi^\alpha CRWL$, and $\pi^\beta CRWL$ enjoy similar properties like polarity, monotonicity for substitutions, and, above all, compositionality, which implies that two expressions have the same denotation if and only if they have the same denotation when put under the same arbitrary context. This is not the case for run-time choice as we saw when switching from $f(c(0\ ?\ 1))$ to $f(c(0)\ ?\ c(1))$ in Examples 1.1 and 1.2, taking into account that for expressions $c(0\ ?\ 1)$ and $c(0)\ ?\ c(1)$ the same values are computed under run-time choice, i.e., the same c-terms are reached by a term rewriting derivation.[7]

[7] In fact compositionality can be achieved for run-time choice by using a different set of values instead of the partial c-terms considered in this work. Those values essentially are recursively nested applications

But our main goal in this section is to study the relationship between call-time choice, run-time choice, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$ with respect to the denotations they define, which express a set of values computed by each semantics. To do this we will lean on a traditional notion from the $CRWL$ framework, the notion of *shell* $|e|$ *of an expression* e, which represents the outer constructor (thus partially computed) part of e, defined as $|\bot| = \bot$, $|X| = X$, $c(e_1,\dots,e_n) = c(|e_1|,\dots,|e_n|)$, $|f(e_1,\dots,e_n)| = \bot$, for $X \in \mathcal{V}, c \in CS, f \in FS$. Now we can define our notion of denotation of an expression in each of the semantics considered.

Definition 2 (*Denotations*)
For any program $\mathcal{P}$, $e \in Exp$ we define the denotation of e under the different semantics as follows:

- $[\![e]\!]^{sg}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \rightarrowtriangle t\}$.
- $[\![e]\!]^{rt}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash e \to^* e' \wedge t \sqsubseteq |e'|\}$.
- $[\![e]\!]^{\alpha pl}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t\}$.
- $[\![e]\!]^{\beta pl}_{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{\pi^{\beta}CRWL} e \rightarrowtriangle t\}$.

In the following, we will usually omit the reference to $\mathcal{P}$ when implied by the context.

As $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ are modifications of $CRWL$, the relation between these three semantics is straightforward.

Theorem 3
For any $CRWL$-program $\mathcal{P}$, $e \in Exp_\bot$

$$[\![e]\!]^{sg} \subseteq [\![e]\!]^{\beta pl} \subseteq [\![e]\!]^{\alpha pl}$$

None of the converse inclusions holds in general.

Proof
Given a $CRWL$-proof for $\vdash_{CRWL} e \rightarrowtriangle t$ we can build a $\pi^{\alpha}CRWL$-proof for $\vdash_{\pi^{\alpha}CRWL}$ $e \rightarrowtriangle t$ just replacing every **OR** step by the corresponding **POR**$^{\beta}$ step, as it is easy to see that any singleton set of c-substitutions is compressible, and that $?\{\theta\} = \theta$. As a consequence $[\![e]\!]^{sg} \subseteq [\![e]\!]^{\beta pl}$. On the other hand, we can turn any $\pi^{\beta}CRWL$-proof into a $\pi^{\alpha}CRWL$-proof just replacing any **POR**$^{\alpha}$ step by the corresponding **POR**$^{\alpha}$, as **POR**$^{\beta}$ has stronger premises than **POR**$^{\alpha}$, and the same consequence. Therefore $[\![e]\!]^{\beta pl} \subseteq [\![e]\!]^{\alpha pl}$.

Regarding the failure of the converse inclusions in the general case, consider the program $\{pair(X) \to d(X,X), g(d(X,Y)) \to d(X,Y)\}$ for which it is easy to check that $[\![pair(0?1)]\!]^{sg} \not\ni d(0,1) \in [\![pair(0?1)]\!]^{\beta pl}$ and $[\![g(d(0,0)?d(1,1))]\!]^{\beta pl} \not\ni d(0,1) \in [\![g(d(0,0)?d(1,1))]\!]^{\alpha pl}$. □

Concerning the relation between call-time choice and run-time choice, it was already explored in the previous works of the authors (López-Fraguas *et al.* 2007, 2010), and we recast it here in the following theorem.

of constructor symbols to sets of values structured in the same way, therefore intrinsically more complicated than plain c-terms, and anyway not considered in the present work (see López-Fraguas *et al.* 2009a for details).

Theorem 4
For any $CRWL$-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{sg} \subseteq [\![e]\!]^{rt}$. The converse inclusion does not hold in general (as shown by Example 1.1).

On the other hand, we cannot rely on any precedent to study relation between $\pi^{\beta}CRWL$ and run-time choice. Therefore, putting run-time choice in the right place in the semantics inclusion chain from Theorem 3 will be one of the contributions of this work. We anticipate that the conclusion is that $\pi^{\beta}CRWL$ computes more values in general.

Theorem 5
For any $CRWL$-program $\mathcal{P}$, $e \in Exp$, $[\![e]\!]^{rt} \subseteq [\![e]\!]^{\beta pl}$. The converse inclusion does not hold in general.

It is easy to prove the last statement of Theorem 5, as in fact Example 1.2 is a valid counter-example for that, but proving the first part is far more complicated. The key for this proof is the following lemma stating that every term rewriting step is sound with respect to $\pi^{\beta}CRWL$.

Lemma 3 (*One-step soundness of* $\rightarrow$ *with respect to* $\pi^{\beta}CRWL$)
For any $CRWL$-program $\mathcal{P}$, $e, e' \in Exp$ if $e \rightarrow e'$ then $[\![e']\!]^{\beta pl} \subseteq [\![e]\!]^{\beta pl}$.

Note that any term rewriting step is of the shape $\mathcal{C}[f(\overline{p})\sigma] \rightarrow \mathcal{C}[r\sigma]$ for some $\sigma \in Subst$ and some program rule $f(\overline{p}) \rightarrow r$. If we could prove Lemma 3 for any step performed at the root of the starting expression, i.e., $f(\overline{p})\sigma \rightarrow r\sigma$ implies $[\![r\sigma]\!]^{\beta pl} \subseteq [\![f(\overline{p})\sigma]\!]^{\beta pl}$, then we could use the compositionality of $\pi^{\beta}CRWL$ from Theorem 2 to propagate the result $[\![r\sigma]\!]^{\beta pl} \subseteq [\![f(\overline{p})\sigma]\!]^{\beta pl}$ to $[\![\mathcal{C}[r\sigma]]\!]^{\beta pl} \subseteq [\![\mathcal{C}[f(\overline{p})\sigma]]\!]^{\beta pl}$. To do that we will use the following notion of $\pi^{\beta}CRWL$-denotation of a substitution.

Definition 3 (*Denotation of substitutions*)
For any $CRWL$-program $\mathcal{P}$, $\sigma \in Subst_{\perp}$ the $\pi^{\beta}CRWL$-denotation of σ under $\mathcal{P}$ is

$$[\![\sigma]\!]^{\beta pl}_{\mathcal{P}} = \{\theta \in CSubst_{\perp} \mid \forall X \in \mathcal{V},\ \mathcal{P} \vdash_{\pi^{\beta}CRWL} \sigma(X) \rightarrowtriangle \theta(X)\}$$

Denotations of substitutions enjoy several interesting properties. For example, every $\sigma \in Subst_{\perp}$ is more powerful than any combination of substitutions from its denotation by means of the ? operator, in the sense that σ is bigger than the combination with respect to the preorder $\unlhd^{\beta pl}$ – which implies that if we apply σ to an arbitrary expression, we get an expression with a bigger denotation that if we apply the combination, thanks to the monotonicity of $Subst_{\perp}$ enjoyed by $\pi^{\beta}CRWL$. This is something natural because c-substitutions in $[\![\sigma]\!]^{\beta pl}$ only contain a finite part of the possibly infinite set of values generated for each expression in the range of σ.

Lemma 4
For any finite not empty $\Theta \subseteq [\![\sigma]\!]^{\beta pl}$ we have $?\Theta \unlhd^{\beta pl} \sigma$.

Besides, it is clear that in any $\pi^{\beta}CRWL$-proof that uses some $\sigma \in Subst_{\perp}$ only a finite amount of information is contained in σ. Therefore, in $[\![\sigma]\!]^{\beta pl}$ is employed, just like in any proof for a statement $\vdash_{\pi^{\beta}CRWL} e \rightarrowtriangle t$ only a finite amount of information in e is used. This follows because t is a finite element and the $\pi^\beta CRWL$-proof is also finite, otherwise the statement $\vdash_{\pi^\beta CRWL} e \rightarrowtriangle t$ could not have been proved. These intuitions are formalized in the following result.

Lemma 5

For any $\sigma \in Subst_\perp, e \in Exp_\perp, t \in CTerm_\perp$ if $\vdash_{\pi^\beta CRWL} e\sigma \rightarrowtriangle t$ then $\exists \Theta \subseteq [\![\sigma]\!]^{\beta pl}$ finite and not empty such that $\vdash_{\pi^\beta CRWL} e(?\Theta) \rightarrowtriangle t$

Proof (sketch)

First we prove the case where $e \equiv X \in \mathcal{V}$. If $X \in dom(\sigma)$ then we define some $\theta \in CSubst_\perp$ as

$$\theta(Y) = \begin{cases} t & \text{if } Y \equiv X \\ \perp & \text{if } Y \in (dom(\sigma) \setminus \{X\}) \\ Y & \text{if } Y \notin dom(\sigma) \end{cases}$$

Otherwise if $X \notin dom(\sigma)$ then given $\overline{Y} = dom(\sigma)$ we define $\theta = [\overline{Y/\perp}]$. In both cases it is easy to see that taking $\Theta = \{\theta\}$ then the conditions of the lemma are granted. To prove the general case where e is not restricted to be a variable, we perform an easy induction over the structure of $e\sigma \rightarrowtriangle t$, using the property that for any $\Theta, \Theta' \subseteq CSubst_\perp$, if $\Theta \subseteq \Theta'$ then $?\Theta \sqsubseteq_\pi ?\Theta'$, combined with the monotonicity under substitutions of $\pi^\beta CRWL$ (see Riesco and Rodríguez-Hortalá 2011 for details). $\square$

This result is very interesting because it expresses a particular property of our plural semantics, as it can be also proved true for the corresponding definition of $\pi^\alpha CRWL$-denotation of a substitution. The key in this result is that the substitution obtained for rebuilding the starting derivation is a substitution from $CSubst^?_\perp$, which are precisely the kind of substitutions used for parameter passing in our plural semantics. On the other hand, this is not true for $CRWL$, and it is one of the reasons why in general call-time choice computes less values than run-time choice: just consider the derivation $\vdash_{CRWL} d(X,X)[X/0\ ?\ 1] \rightarrowtriangle d(0,1)$ for which there is no substitution θ in $CSubst_\perp$ – the kind of substitutions used for parameter passing in $CRWL$ – such that $\vdash_{CRWL} d(X,X)\theta \rightarrowtriangle d(0,1)$. Nevertheless, if we restrict to deterministic programs, this property becomes true for $CRWL$ – and besides in that case run-time choice and call-time choice are equivalent too (see López-Fraguas *et al.* 2007, 2010 for details).

Although Lemma 5 is a nice result, we still need an extra ingredient to be able to use it for proving Lemma 3, thus enabling an easy proof for Theorem 5. The point is that we cannot use an arbitrary substitution from $CSubst^?_\perp$ for parameter passing in $\pi^\beta CRWL$ but only a substitution that would be also compressible in order to ensure that no wrong substitution mix-up is performed, which is precisely the main feature of $\pi^\beta CRWL$. Therefore, although a version of Lemma 5 for $\pi^\alpha CRWL$ can be used for proving that term rewriting is sound with respect to $\pi^\alpha CRWL$ – as in fact it was done in Rodriguez-Hortala (2008) – for proving its soundness with respect to $\pi^\beta CRWL$ we will still need to do a little extra effort. And the missing piece is the following notion of compressible completion of a set of c-substitutions, which adds some additional c-substitutions to its input set in order to ensure that the resulting set is then compressible.

Definition 4 (*Compressible completion*)
Given $\Theta \subseteq CSubst_\perp$ finite such that $\{X_1,\ldots,X_n\} = \bigcup_{\theta\in\Theta} dom(\theta)$, its compressible completion $cc(\Theta)$ is defined as

$$cc(\Theta) = \{[X_1/X_1\theta_1,\ldots,X_n/X_n\theta_n] \mid \theta_1,\ldots,\theta_n \in \Theta\}$$

Every compressible completion enjoys the following basic properties, which explain why we call it "completion" and also "compressible."

Proposition 5 (*Properties of* $cc(\Theta)$)
For any $\Theta \subseteq CSubst_\perp$ finite such that $\{X_1,\ldots,X_n\} = \bigcup_{\theta\in\Theta} dom(\theta)$

(a) $cc(\Theta) \subseteq CSubst_\perp$ and it is finite.
(b) $\Theta \subseteq cc(\Theta)$. As a result, $?\Theta \sqsubseteq_\pi ?cc(\Theta)$.
(c) $\bigcup_{\mu\in cc(\Theta)} dom(\mu) = \{X_1,\ldots,X_n\}$.
(d) $cc(\Theta)$ is compressible.

But, for the current task, the most interesting property of compressible completions is the following.

Lemma 6
For any $\sigma \in Subst_\perp$ and any $\Theta \subseteq [\![\sigma]\!]^{\beta pl}$ finite and not empty we have that $cc(\Theta) \subseteq [\![\sigma]\!]^{\beta pl}$ too.

This is precisely the result we need to strengthen Lemma 5, so it now becomes applicable for $\pi^\beta CRWL$, as it allows us to shift from any subset of the $\pi^\beta CRWL$-denotation of a substitution to its compressible completion, which will be also more powerful than the starting subset, thanks to Proposition 5(b).

Lemma 7
For any $\sigma \in Subst_\perp, e \in Exp_\perp, t \in CTerm_\perp$ if $\vdash_{\pi^\beta CRWL} e\sigma \rightarrowtriangle t$ then $\exists\Theta \subseteq [\![\sigma]\!]^{\beta pl}$ finite, not empty, and compressible such that $\vdash_{\pi^\beta CRWL} e(?\Theta) \rightarrowtriangle t$.

Proof
By Lemma 5 we get some $\Theta \subseteq [\![\sigma]\!]^{\beta pl}$ finite and not empty such that $\vdash_{\pi^\beta CRWL} e(?\Theta) \rightarrowtriangle t$. Then by Lemma 6 we get that $cc(\Theta) \subseteq [\![\sigma]\!]^{\beta pl}$ too, and that it is finite, not empty (as $\Theta \subseteq cc(\Theta)$ and Θ is not empty), compressible and $?\Theta \sqsubseteq_\pi ?cc(\Theta)$ by Proposition 5. But then we can apply the monotonicity of Theorem 2 to get $\vdash_{\pi^\beta CRWL} e(?cc(\Theta)) \rightarrowtriangle t$, so we are done. □

We can now use this result to prove a particularization of Lemma 3 (one-step soundness of $\rightarrow$ with respect to $\pi^\beta CRWL$) for steps performed at the root of the expression, i.e., of the shape $f(\overline{p})\sigma \rightarrow r\sigma$. Thus, given some $t \in [\![r\sigma]\!]^{\beta pl}$, our goal is proving that $t \in [\![f(\overline{p})\sigma]\!]^{\beta pl}$. First of all by Lemma 7 we get some compressible $\Theta \subseteq [\![\sigma]\!]^{\beta pl}$ such that $t \in [\![r(?\Theta)]\!]^{\beta pl}$. If we could use it to prove that $t \in [\![f(\overline{p})(?\Theta)]\!]^{\beta pl}$ then by Lemma 4 we would get $?\Theta \trianglelefteq^{\beta pl} \sigma$, so by the monotonicity of Theorem 2 we could obtain $t \in [\![f(\overline{p})\sigma]\!]^{\beta pl}$ as we wanted. As $\overline{p} \subseteq CTerm_\perp$ and $\Theta \subseteq CSubst_\perp$ we can easily prove that $\forall p_i \in \overline{p}, \theta_j \in \Theta$ we have $\vdash_{\pi^\beta CRWL} p_i(?\Theta) \rightarrowtriangle p_i\theta_j$. All this can be used to perform the following step, assuming $\Theta = \{\theta_1,\ldots,\theta_m\}$.

$$\dfrac{\begin{array}{ccc} p_1(?\Theta) \rightarrowtriangle p_1\theta_1 \equiv p_1\theta_1|_{var(p_1)} & & p_n(?\Theta) \rightarrowtriangle p_n\theta_1 \equiv p_n\theta_1|_{var(p_n)} & \\ \dots & \dots & \dots & \\ p_1(?\Theta) \rightarrowtriangle p_1\theta_m \equiv p_1\theta_m|_{var(p_1)} & & p_n(?\Theta) \rightarrowtriangle p_n\theta_m \equiv p_n\theta_m|_{var(p_n)} & r\theta' \equiv r(?\Theta) \rightarrowtriangle t \end{array}}{f(p_1, \dots, p_n)(?\Theta) \rightarrowtriangle t}\ \mathbf{POR}^{\beta}$$

for $\theta' = (\biguplus ?\Theta_i) \uplus \theta_e$ where $\forall i \in \{1, \dots, n\}.\Theta_i = \{\theta_j|_{var(p_i)} \mid \theta_j \in \Theta\}$, $\theta_e = (?\Theta)|_{\mathscr{V}_e}$ for $\mathscr{V}_e = vExtra(f(\overline{p}) \to r)$. It can be easily proved that having Θ compressible implies that each Θ_i is also compressible – so the $\mathbf{POR}^{\beta}$ step above is valid – and that $r\theta' \equiv r(?\Theta)$.

Therefore, we have just proved the soundness with respect to $\pi^{\beta}CRWL$ of term rewriting steps performed at the root of the starting expression. So, all that is left is using the compositionality of $\pi^{\beta}CRWL$ from Theorem 2 for propagating this result for steps performed in an arbitrary context. A detailed proof for Lemma 3 can be found in Riesco and Rodríguez-Hortalá (2011).

And now we are finally ready to prove Theorem 5.

Proof for Theorem 5

Given some $t \in [\![e]\!]^{rt}$, by definition $\exists e' \in Exp$ such that $t \sqsubseteq |e'|$ and $e \to^* e'$. We can extend Lemma 3 to $\to^*$ by a simple induction on the length of $e \to^* e'$, hence $[\![e']\!]^{\beta pl} \subseteq [\![e]\!]^{\beta pl}$. As $\forall e \in Exp_{\perp}, |e| \in [\![e]\!]^{\beta pl}$ (by a simple induction on the structure of e), then $t \sqsubseteq |e'| \in [\![e']\!]^{\beta pl} \subseteq [\![e]\!]^{\beta pl}$, hence $t \in [\![e]\!]^{\beta pl}$ by the polarity of Theorem 2. Example 1.1 shows that the converse inclusion does not hold in general. $\square$

The evident corollary for all these results is the following inclusion chain.

Corollary 4.1

For any $CRWL$-program $\mathscr{P}$, $e \in Exp$

$$[\![e]\!]^{sg} \subseteq [\![e]\!]^{rt} \subseteq [\![e]\!]^{\beta pl} \subseteq [\![e]\!]^{\alpha pl}$$

Hence for any $t \in CTerm$, $\mathscr{P} \vdash_{CRWL} e \rightarrowtriangle t$ implies $\mathscr{P} \vdash e \to^* t$, which implies $\mathscr{P} \vdash_{\pi^{\beta}CRWL} e \rightarrowtriangle t$, which implies $\mathscr{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t$.

Proof

The first part holds just combining Theorems 3, 4, and 5.

Concerning the second part, assume $\vdash_{CRWL} e \rightarrowtriangle t$, in other words, $t \in [\![e]\!]^{sg}$. Then by the first part $t \in [\![e]\!]^{rt}$, hence $e \to^* e'$ such that $t \sqsubseteq |e'|$. But as $t \in CTerm$, it is total and then t is maximal with respect to $\sqsubseteq$ (a known property of $\sqsubseteq$ easy to check by induction on the structure of expressions), and so $t \sqsubseteq |e'|$ implies $t \equiv |e'|$, which implies $t \equiv e'$, as t is total (easy to check by induction on the structure of t). Therefore, $e \to^* e' \equiv t \in CTerm$, which implies $t \in [\![e]\!]^{rt}$ by definition, as for c-terms t we have $t \sqsubseteq t \equiv |t|$ (a property of shells proved by induction on the structure of t), but then $t \in [\![e]\!]^{\beta pl} \subseteq [\![e]\!]^{\alpha pl}$ by the first part, and so both $\vdash_{\pi^{\beta}CRWL} e \rightarrowtriangle t$ and $\vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t$. $\square$

4.2 Restricted equivalence of $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$

In this section we will present a class of programs for which $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ behave the same, thus yielding exactly the same denotation for any expression. In the previous section we saw that $[\![e]\!]^{\beta pl} \subseteq [\![e]\!]^{\alpha pl}$ for any expression and program, therefore we just have to find a class of programs such that $[\![e]\!]^{\alpha pl} \subseteq [\![e]\!]^{\beta pl}$ also holds for programs in that class.

The intuitions and ideas behind the characterization of that class of programs come from Example 3.5. The program used there contains two functions f and h defined by the rules $\{f(c(X)) \to d(X,X), h(d(X,Y)) \to d(X,X)\}$, with $d \in CS^2$, under which it is easy to check that $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ behave the same for the expressions $f(c(0)\ ?\ c(1))$ and $h(d(0,0)\ ?\ d(1,1))$.

- Regarding $f(c(0)\ ?\ c(1))$, it is pretty natural for both plural semantics to behave the same, as no wrong information mix-up can be performed when combining two substitutions with singleton domain, like $[X/0]$ and $[X/1]$, coming when evaluating $c(0)\ ?\ c(1)$ to get an instance of $c(X)$.
- The case for $h(d(0,0)\ ?\ d(1,1))$ is more surprising at first look because then we can obtain the matching substitutions $[X/0, Y/0]$ and $[X/1, Y/1]$, which cannot be safely combined because the set $\{[X/0, Y/0], [X/1, Y/1]\}$ is not compressible. But as seen in Example 3.5 this poses no problem because the wrongly intermingled substitution $[X/0\ ?\ 1, Y/0\ ?\ 1]$ used by $\pi^{\alpha}CRWL$ has the same effect over the right-hand side $d(X,X)$ of the rule for h as the substitution $[X/0\ ?\ 1, Y/\perp]$ that can be obtained from combining the compressible set $\{[X/0, Y/\perp], [X/1, Y/\perp]\}$. This compressible set not only can be used for parameter passing by $\pi^{\beta}CRWL$but also can be generated by evaluating the arguments of $h(d(0,0)\ ?\ d(1,1))$ to get an instance of the left-hand side of the rule for h as $[X/0, Y/\perp] \sqsubseteq [X/0, Y/0]$ and $[X/1, Y/\perp] \sqsubseteq [X/1, Y/1]$.

What the functions f and h have in common is that, for each argument of the left-hand side of each of their program rules, at most one variable in that argument appears also on the right-hand side. If we only have to care about one variable then we can lower to $\perp$ the value obtained for the other variables in the matching substitution, thus getting a smaller – with respect to to $\sqsubseteq$ – matching substitution corresponding to a smaller value that then can be computed, thanks to the polarity of $\pi^{\alpha}CRWL$ from Proposition 1. The effect of this is that we would get a compressible substitution that can be used by $\pi^{\alpha}CRWL$ to turn a **POR**$^{\alpha}$ step using a possibly non-compressible substitutions into a **POR**$^{\alpha}$ step using a compressible substitutions that would be then a valid **POR**$^{\beta}$ step as well. Note that in this case extra variables pose no problem, as the only difference between **POR**$^{\alpha}$ and **POR**$^{\beta}$ is the way they handle the matching substitutions obtained by the evaluation of function arguments. Then, as extra variables are instantiated freely and independently of the matching substituions, they always behave the same under both $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$.

In the following definition we formally define the class $\mathscr{C}^{\alpha\beta}$ of programs in which the ideas above are materialized.

Definition 5 (*Class of programs* $\mathscr{C}^{\alpha\beta}$)
The class of programs $\mathscr{C}^{\alpha\beta}$ is defined by

$$\mathscr{P} \in \mathscr{C}^{\alpha\beta} \text{ iff } \forall (f(p_1,\ldots,p_n) \to r) \in \mathscr{P}. \forall i \in \{1,\ldots,n\}. \#(var(p_i) \cap var(r)) \leqslant 1$$

where, given a set S, $\#(S)$ stands for the cardinality of S. Note that any program rule in which every argument on its left-hand side is ground or a variable passes the test that characterizes $\mathscr{C}^{\alpha\beta}$: for ground arguments no parameter passing is performed, only matching, so we conjecture that if the arguments on the left-hand side of each program rule are ground, then $CRWL$, term rewriting, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$ behave the same. On the other hand, for variable arguments we have the converse situation, so matching is trivial and parameter passing is an important thing, so we conjecture that if the arguments on the left-hand side of each program rule are variables, then term rewriting, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$ behave the same – $CRWL$ remains as the smaller semantics in this case; just consider the program $\{pair(X) \to d(X,X)\}$ and the expression $pair(0 \ ? \ 1)$ for which $d(0,1)$ cannot be computed by $CRWL$, but it can be by any of the other three semantics.

Anyway, the class $\mathscr{C}^{\alpha\beta}$ is defined by a simple syntactic criterion, which can be easily implemented in any mechanized program analysis tool, and that we have implemented in our prototype from Section 5.

The following theorem formalizes the expected equivalence between $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ for programs in the class $\mathscr{C}^{\alpha\beta}$.

Theorem 6 (*Equivalence of* $\pi^{\alpha}CRWL$ *and* $\pi^{\beta}CRWL$ *for the class* $\mathscr{C}^{\alpha\beta}$)
For any program $\mathscr{P} \in \mathscr{C}^{\alpha\beta}$, $e \in Exp_{\perp}$

$$[\![e]\!]^{\alpha pl}_{\mathscr{P}} = [\![e]\!]^{\beta pl}_{\mathscr{P}}$$

This equivalence between $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ will be very useful for us for several reasons. First of all, as we will see in Section 4.3, $\pi^{\alpha}CRWL$ can be simulated by term rewriting through a simple program transformation, which implies that the same transformation can be used to simulate $\pi^{\beta}CRWL$ for the class of programs $\mathscr{C}^{\alpha\beta}$, thanks to the equivalence from Theorem 6. On the other hand, the class $\mathscr{C}^{\alpha\beta}$ is defined by a simple syntactic criterion, which allows its application to mechanized program analysis. Finally, this equivalence grows in importance after realising that the class $\mathscr{C}^{\alpha\beta}$ contains many relevant programs: as a matter of fact, all the programs considered in Section 5 – where we explore the expressive capabilities of our plural semantics – belong to the class $\mathscr{C}^{\alpha\beta}$.

4.3 Simulating plural semantics with term rewriting

In López-Fraguas *et al.* (2007, 2009a, 2010) it was shown that neither $CRWL$ can be simulated by term rewriting with a simple program transformation, nor vice versa. Nevertheless, $\pi^{\alpha}CRWL$ can be simulated by term rewriting using the transformation presented in the current section, which can then be used as the basis for a first implementation of $\pi^{\alpha}CRWL$. First, we will present a naive version of this transformation, and show its adequacy; later, we will propose some simple optimizations for it.

In this section we will restrict ourselves to programs not containing extra variables, i.e., such that for any program rule $l \rightarrow r$ we have that $var(r) \subseteq var(l)$ holds, a restriction usually adopted in texts devoted to term rewriting systems (Baader and Nipkow 1998; TeReSe 2003) for which term rewriting with extra variables is normally considered as an extension of standard term rewriting. Besides, in practical implementations, extra variables are usually handled by using narrowing (López-Fraguas and Sánchez-Hernández 1999; Hanus 2006) or additional conditions to restrict their possible instantiations (Clavel *et al.* 2007) in order to avoid a state space explosion in the search process. Therefore, we leave the extension of our work to completely deal with extra variables as a subject of future work.

4.3.1 A simple transformation

The main idea in our transformation is to postpone the pattern-matching process in order to prevent an early resolution of non-determinism. Instead of presenting the transformation directly, we will first illustrate this concept by applying the transformation over the program $\mathscr{P} = \{f(c(X)) \rightarrow d(X,X)\}$ from Example 1.1, which results in the following program $\hat{\mathscr{P}}$.

$$\hat{\mathscr{P}} = \{\, f(Y) \rightarrow if\ match(Y)\ then\ d(project(Y), project(Y)),\\ match(c(X)) \rightarrow true, project(c(X)) \rightarrow X\ \ \}$$

In the resulting program $\hat{\mathscr{P}}$ the only rule for function f has been transformed, so matching is transferred from the left-hand side to the right-hand side of the rule by means of auxiliary functions, *match* and *project*. As a consequence, when we evaluate by term rewriting under $\hat{\mathscr{P}}$ the function call to f, in the expression $f(c(0)\ ?\ c(1))$ we are not forced anymore to solve the non-deterministic choice between $c(0)$ and $c(1)$ before parameter passing, because any expression matches the variable pattern Y. Therefore, the term rewriting step

$$f(c(0)\ ?\ c(1)) \rightarrow if\ match(c(0)\ ?\ c(1))\ then\ d(project(c(0)\ ?\ c(1)), project(c(0)\ ?\ c(1)))$$

is sound, thus replicating the argument of f freely without demanding any evaluation, this way keeping its $\pi^{\alpha}CRWL$-denotation untouched: this is the key to achieve completeness with respect to $\pi^{\alpha}CRWL$. Note that the guard $if\ match(c(0)\ ?\ c(1))$ is needed to ensure that at least one of the values of the argument matches the original pattern, otherwise the soundness of the step could not be granted. For example, if we drop this condition in the translation of the rule "$null(nil) \rightarrow true$" for defining an emptiness test for the classical representation of lists in functional programming, we would get "$null(Y) \rightarrow true$," which is clearly unsound because it allows us to rewrite $null(cons(0, nil))$ into $true$. Later on, after resolving the guard, different evaluations of the occurrences of $project(c(0)\ ?\ c(1))$ will solve the non-deterministic choice implied by ?, and project the argument of c, thus leading us to the final values $d(0,0)$, $d(1,1)$, $d(0,1)$, and $d(1,0)$, which are the expected values for the expression in the original program under $\pi^{\alpha}CRWL$.

In the following definition we formalize the transformation by means of the function pST, which for any program rule returns a rule to replace it, and a set of auxiliary *match* and *project* rules for the replacement.

Definition 6 ($\pi^{\alpha}CRWL$ to term rewriting transformation, simple version)
Given a program $\mathcal{P}$, our transformation proceeds rule by rule. For every program rule $(f(p_1,\ldots,p_n) \to r) \in \mathcal{P}$ such that $f \notin \{?, \mathit{if\ then}\}$ we define its transformation as follows:

$$\begin{aligned} &pST(f(p_1,\ldots,p_n) \to r) \\ &\quad = f(Y_1,\ldots,Y_n) \to \mathit{if}\ \mathit{match}(Y_1,\ldots,Y_n)\ \mathit{then}\ r[\overline{X_{ij}/\mathit{project}_{ij}(Y_i)}] \end{aligned}$$

where

- $\forall i \in \{1,\ldots,n\}$, $\{X_{i1},\ldots,X_{ik_i}\} = var(p_i) \cap var(r)$ and $Y_i \in \mathcal{V}$ is fresh.
- $\mathit{match} \in FS^n$ is a fresh function defined by the rule $\mathit{match}(p_1,\ldots,p_n) \to \mathit{true}$.
- Each $\mathit{project}_{ij} \in FS^1$ is a fresh symbol defined by the single rule $\mathit{project}_{ij}(p_i) \to X_{ij}$.

For $f \in \{?, \mathit{if\ then}\}$ the transformation leaves its rules untouched.

It is easy to check that if we use the program $\mathcal{P}$ from Example 1.1 as input for this transformation then it outputs the program $\hat{\mathcal{P}}$ from the discussion above, under which we can perform the following term rewriting derivation:

$$\begin{aligned} &\underline{f(c(0)?c(1))} \to \mathit{if}\ \underline{\mathit{match}(c(0)?c(1))}\ \mathit{then}\ d(\mathit{project}(c(0)?c(1)), \mathit{project}(c(0)?c(1))) \\ &\to^* \underline{\mathit{if}\ \mathit{true}\ \mathit{then}\ d(\mathit{project}(c(0)?c(1)), \mathit{project}(c(0)?c(1)))} \\ &\to d(\mathit{project}(\underline{c(0)?c(1)}), \mathit{project}(\underline{c(0)?c(1)})) \to^* d(\underline{\mathit{project}(c(0))}, \underline{\mathit{project}(c(1))}) \to^* d(0,1) \end{aligned}$$

We do not only claim that this transformation is sound but also have technical results about the strong adequacy of our transformation $pST(\_)$ for simulating the $\pi^{\alpha}CRWL$ logic using term rewriting. The first one is a soundness result, stating that if we rewrite an expression under the transformed program, then we cannot get more results than those that we can get in $\pi^{\alpha}CRWL$ under the original program.

Theorem 7
For any $CRWL$-program $\mathcal{P}$, and any $e \in Exp_{\perp}$ built up on the signature of $\mathcal{P}$, we have

$$[\![e]\!]^{\alpha pl}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{\alpha pl}_{\mathcal{P}}$$

As a consequence $[\![e]\!]^{rt}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{\alpha pl}_{\mathcal{P}}$.

Proof (sketch)
The first part states the soundness within $\pi^{\alpha}CRWL$ of the transformation. Assuming $\pi^{\alpha}CRWL$-proof for statement $pST(\mathcal{P}) \vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t$ for some $t \in CTerm_{\perp}$, we can then build another $\pi^{\alpha}CRWL$-proof for $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrowtriangle t$ by induction on the size of the starting proof – measured as the number of rules of $\pi^{\alpha}CRWL$ used. Full details for that proof can be found in Riesco and Rodríguez-Hortalá (2011).

Concerning the second part, it follows from combining the first part with Corollary 4.1, because then we can chain $[\![e]\!]^{rt}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{\alpha pl}_{pST(\mathcal{P})} \subseteq [\![e]\!]^{\alpha pl}_{\mathcal{P}}$. □

Regarding completeness of the transformation, we obtained the following result stating that, for any expression one can build in the original program, we can refine by term rewriting under the transformed program any value computed for that expression by $\pi^\alpha CRWL$ under the original program.

Theorem 8
For any $CRWL$-program $\mathscr{P}$, and any $e \in Exp, t \in CTerm_\perp$ built up on the signature of $\mathscr{P}$, if $\mathscr{P} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$ then exists some $e' \in Exp$ built using symbols of the signature of $pST(\mathscr{P})$ such that $pST(\mathscr{P}) \vdash e \rightarrow^* e'$ and $t \sqsubseteq |e'|$. In other words, $[\![e]\!]^{\alpha pl}_{\mathscr{P}} \subseteq [\![e]\!]^{rt}_{pST(\mathscr{P})}$.

The proof for this result is technically very involved. First of all we have to slightly generalize Theorem 8 to consider not only the functions of the original program but also the auxiliary *match* and *project* functions generated by the transformation in order to obtain strong enough induction hypothesis.

Lemma 8
Given a $CRWL$-program $\mathscr{P}$ let $\hat{\mathscr{P}} \uplus \mathscr{M} = pST(\mathscr{P})$, where $\mathscr{M}$ is the set containing the rules for the new functions *match* and *project*, and $\hat{\mathscr{P}}$ contains the new versions of the original rules of $\mathscr{P}$ – note that by an abuse of notation, the rules for $?$, *if then* presented in Section 2.1 belong implicitly to both $\mathscr{P} \uplus \mathscr{M}$ and $\hat{\mathscr{P}} \uplus \mathscr{M}$.

Then for any $e \in Exp_\perp, t \in CTerm_\perp$ constructed using just symbols in the signature of $\mathscr{P} \uplus \mathscr{M}$ we have $\mathscr{P} \uplus \mathscr{M} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$ implies $\hat{\mathscr{P}} \uplus \mathscr{M} \vdash e \rightarrow^* e'$ such that $t \sqsubseteq |e'|$.

The proof for Lemma 8 is pretty complicated and it relies on several auxiliary notions; a detailed proof can be found in Riesco and Rodríguez-Hortalá (2011). Then Theorem 8 follows as an almost trivial consequence of Lemma 8.

Proof for Theorem 8
Let $\hat{\mathscr{P}} \uplus \mathscr{M} = pST(\mathscr{P})$ be, where $\mathscr{M}$ is the set containing the rules for the new functions *match* and *project*, and $\hat{\mathscr{P}}$ contains the new versions of the original rules of $\mathscr{P}$.

If $e \in Exp, t \in CTerm_\perp$ are built using symbols on the signature of $\mathscr{P}$, then $\mathscr{P} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$ implies $\mathscr{P} \uplus \mathscr{M} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$, which implies $\hat{\mathscr{P}} \uplus \mathscr{M} \vdash e \rightarrow^* e'$ such that $t \sqsubseteq |e'|$ by Lemma 8, that is, $pST(\mathscr{P}) \vdash e \rightarrow^* e'$. □

To conclude, the following corollary summarizes the adequacy of the simulation performed by our program transformation.

Corollary 4.2 (*Adequacy of $pST(\_)$ for simulating $\pi^\alpha CRWL$*)
For any program $\mathscr{P}$, $e \in Exp$ built using symbols of the signature of $\mathscr{P}$

$$[\![e]\!]^{\alpha pl}_{\mathscr{P}} = [\![e]\!]^{rt}_{pST(\mathscr{P})}$$

Hence, $\forall t \in CTerm$ we have that $\mathscr{P} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$ iff $pST(\mathscr{P}) \vdash e \rightarrow^* t$.

Proof
The fist part holds by a combination of Theorems 7 and 8.
For the second part if $\mathscr{P} \vdash_{\pi^\alpha CRWL} e \twoheadrightarrow t$ then $t \in [\![e]\!]^{\alpha pl}{}_{\mathscr{P}} = [\![e]\!]^{rt}_{pST(\mathscr{P})}$ by the first part, hence $\exists e' \in Exp$ such that $pST(\mathscr{P}) \vdash e \rightarrow^* e'$ and $t \sqsubseteq |e'|$. But as $t \in CTerm$ then t is maximal with respect to $\sqsubseteq$ and so $t \equiv |e'|$, which implies $t \equiv e'$ (these are

known properties of shells and $\sqsubseteq$), therefore $pST(\mathscr{P}) \vdash e \to^* e' \equiv t$. On the other hand, if $pST(\mathscr{P}) \vdash e \to^* t$ then as $t \sqsubseteq t \equiv |t|$ (again because t is a total c-term) we have $t \in [\![e]\!]^{rt}_{pST(\mathscr{P})} = [\![e]\!]^{\alpha pl}_{\mathscr{P}}$, and so $\mathscr{P} \vdash_{\pi^\alpha CRWL} e \rightarrowtriangle t$. $\square$

As promised at the end of the previous subsection, we can now use the restricted equivalence between $\pi^\alpha CRWL$ and $\pi^\beta CRWL$ from Theorem 6 to extend the adequacy results of the simulation of $\pi^\alpha CRWL$ with term rewriting to $\pi^\beta CRWL$for the class of programs $\mathscr{C}^{\alpha\beta}$.

Corollary 4.3 (*Restricted adequacy of pST(_) for simulating $\pi^\beta CRWL$*)
For any program $\mathscr{P} \in \mathscr{C}^{\alpha\beta}$, $e \in Exp$ is built using symbols of the signature of $\mathscr{P}$

$$[\![e]\!]^{\beta pl}_{\mathscr{P}} = [\![e]\!]^{rt}_{pST(\mathscr{P})}$$

Hence, $\forall t \in CTerm$ we have that $\vdash_{\pi^\beta CRWL} e \rightarrowtriangle t$ iff $pST(\mathscr{P}) \vdash e \to^* t$.

Proof
A straightforward combination of Corollary 4.2 and Theorem 6. $\square$

This last result illustrates the interest of $\pi^\alpha CRWL$. Because of its simplicity, $\pi^\alpha CRWL$ sometimes combines matching substitutions in a wrong way, but it is precisely the same simplicity that allows it to be simulated by term rewriting through a simple program transformation. As a result we can use any available implementation of term rewriting, like the Maude system, to devise an implementation of $\pi^\alpha CRWL$. Besides, thanks to the restricted equivalence between $\pi^\alpha CRWL$ and $\pi^\beta CRWL$, which would also be an implementation of $\pi^\beta CRWL$ for the class $\mathscr{C}^{\alpha\beta}$, and the membership check of program to the class $\mathscr{C}^{\alpha\beta}$ could be also mechanized because $\mathscr{C}^{\alpha\beta}$ is defined by a simple syntactic criterion. We will see how these ideas are developed in the next sections, where the Maude-based implementation of our plural semantics is presented, and the interest of the class $\mathscr{C}^{\alpha\beta}$ is illustrated.

4.3.2 An optimized transformation

As we have already mentioned in our comments after presenting the class $\mathscr{C}^{\alpha\beta}$ in Definition 5, we expect that for ground or variable arguments run-time choice and our plural semantics behave the same. We can take advantage of this for applying some optimizations to the program transformation from Definition 6.

- When applied to $null(nil) \to true$, the transformation returns the rules $\{null(Y) \to \mathit{if}\ match(Y)\ then\ true,\ match(nil) \to true\}$, which behave the same as the original rule. The conclusion is that when a given pattern is ground then no parameter passing will be done for that pattern, and thus no transformation is needed.
- Something similar happens with $pair(X) \to d(X, X)$ for which $\{pair(Y) \to \mathit{if}\ match(Y)\ then\ d(project(Y), project(Y)), match(X) \to true, project(X) \to X\}$ is returned. In this case the pattern is a variable to which any expression matches without any evaluation, and the projection functions are trivial, so no transformation is needed either.

We can apply these ideas to get the following refinement of our original program transformation.

Definition 7 (*$\pi^{\alpha}CRWL$ to term rewriting transformation, optimized version*)
Given a program $\mathscr{P}$, our transformation proceeds rule by rule. For every program rule $(f(p_1,\dots,p_n) \to r) \in \mathscr{P}$ we define its transformation as follows:

$$pST(f(p_1,\dots,p_n) \to r) = \begin{cases} f(p_1,\dots,p_n) \to r & \text{if } m = 0 \\ f(\tau(p_1),\dots,\tau(p_n)) \to \begin{array}{l} \mathit{if}\ \ match(Y_1,\dots,Y_m) \\ \quad \mathit{then}\ r\overline{[X_{ij}/project_{ij}(Y_i)]} \end{array} & \text{otherwise} \end{cases}$$

where $\rho_1 \dots \rho_m = p_1 \dots p_n \mid \lambda p.(p \notin \mathscr{V} \wedge var(p) \neq \emptyset)$.
- $\forall \rho_i$, $\{X_{i1},\dots,X_{ik_i}\} = var(\rho_i) \cap var(r)$ and $Y_i \in \mathscr{V}$ is fresh.
- $\tau : CTerm \to CTerm$ is defined by $\tau(p) = p$ if $p \notin \{\rho_1,\dots,\rho_m\}$; otherwise $\tau(\rho_i) = Y_i$.
• $match \in FS^m$ fresh is defined by the rule $match(\rho_1,\dots,\rho_m) \to true$.
• Each $project_{ij} \in FS^1$ is a fresh symbol defined by the rule $project_{ij}(\rho_i) \to X_{ij}$.

Note that this transformation is well defined because each $\rho_i \in \{\rho_1,\dots,\rho_m\}$ contains at least one variable, and so it can be distinguished from any other ρ_j by using syntactic equality, thanks to left linearity of program rules, therefore τ is well defined.

We will not give any formal proof for the adequacy of this optimized transformation. Nevertheless, note how this transformation leaves untouched the rules for ? and *if then* without defining a special case for them. The simple transformation from Definition 6 worked well for these rules that suggests that we are doing the right thing.

We end this section with an example application of the optimized transformation, over the program from Example 3.3. As expected, the transformed program behaves under term rewriting like the original one under $\pi^{\alpha}CRWL$.

Example 4.1
The only rule modified is the one for *find*, for which we get the following program:

$$\begin{array}{l} \{find(Y) \to \mathit{if}\ \ match(Y)\ \mathit{then}\ (project(Y), project(Y)), \\ \ match(e(N,G,clerk)) \to true, \\ \ project(e(N,G,clerk)) \to N\} \end{array}$$

under which we can perform this term rewriting derivation for *twoclerks*

$$\begin{array}{l} \underline{twoclerks} \to \underline{find(employees(branches))} \\ \to \mathit{if}\ \ match(\underline{employees(branches)}) \\ \qquad\qquad \mathit{then}\ \ (project(employees(branches)), project(employees(branches))) \\ \to^{*} \underline{\mathit{if}\ \ match(e(pepe,man,clerk))} \\ \qquad\qquad \underline{\mathit{then}\ \ (project(employees(branches)), project(employees(branches)))} \\ \to^{*} (project(\underline{employees(branches)}), project(\underline{employees(branches)})) \\ \to^{*} (\underline{project(e(pepe,man,clerk))}, \underline{project(e(maria,woman,clerk))}) \\ \to^{*} (pepe, maria) \end{array}$$

5 Programming with singular and plural functions

So far we have presented two novel proposals for the semantics of lazy non-determi-nistic functions, studied some of its properties, and explored their relation to previous proposals like call-time choice and run-time choice. Nevertheless, we have seen just a couple of program examples using the semantics, so until now we have hardly tested the way we can exploit the new expressive capabilities offered by our plural semantics to improve the declarative flavor of programs. The present section is devoted to the exploration of these expressive capabilities by means of several programs that try to illustrate the virtues of our new plural semantics.

In Riesco and Rodríguez-Hortalá (2010a) the authors already explored the capabilities of $\pi^{\alpha}CRWL$ by using the Maude system (Clavel *et al.* 2007) to develop an interpreter for this semantics based on the program transformation from Section 4.3. The resulting interpreter was then used for experimenting with $\pi^{\alpha}CRWL$, showing how it allows an elegant encoding of some problems, in particular those with an implicit manipulation of sets of values. However, call-time choice still remains the best option for many common programming patterns (González-Moreno *et al.* 1999; Antoy and Hanus 2002), and that is why it is the semantic option adopted by modern functional-logic programming systems like Toy (López-Fraguas and Sánchez-Hernández 1999) or Curry (Hanus 2006). Therefore, it would be nice to have a language in which both options could be available. In this section we propose such a language, where the user has the possibility to specify the arguments of each function symbol that will be considered "plural arguments." These arguments will be evaluated using our plural semantics, which intuitively means that they will be treated like sets of elements of the corresponding type[8] instead of single elements, while the others will be evaluated under the usual singular/call-time choice semantics traditionally adopted for FLP. Thereby in Riesco and Rodríguez-Hortalá (2010b) we extended our Maude-based prototype to support this combination of singular and plural arguments, and used it to develop and test several programs that we think are significant examples of the possibilities of the combined semantics. The source code for these examples and the interpreter to test the same can be found at http://gpd.sip.ucm.es/PluralSemantics.

As we have two different plural semantics available, we get two different semantics resulting from their combination with call-time choice that we have precisely formalized by means of two novel variants of $CRWL$ called $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$, corresponding to the combination of call-time choice with $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$, respectively. Our prototype is based on the program transformation from Section 4.3, therefore it is an implementation of $CRWL^{\sigma}_{\pi^{\alpha}}$, and so $CRWL^{\sigma}_{\pi^{\beta}}$ is only supported for programs in the class $\mathscr{C}^{\alpha\beta}$ described in Section 4.2. After those calculus, we introduce the concrete syntax of our interpreter and motivate the combination of singular and plural semantics with a simple example, while the next examples illustrate how to combine singular and plural arguments in depth. Then after a short discussion

[8] As types are not considered through this work; here we mean the type naturally intended by the programmer.

$$\mathbf{OR}^{\sigma}_{\pi\alpha}\quad \frac{\begin{array}{ccc} e_1 \rightarrowtriangle p_1\theta_{11} & & e_n \rightarrowtriangle p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \rightarrowtriangle p_1\theta_{1m_1} & & e_n \rightarrowtriangle p_n\theta_{nm_n} \quad r\theta \rightarrowtriangle t \end{array}}{f(e_1,\ldots,e_n) \rightarrowtriangle t}$$

$$\begin{array}{l} \text{if } (f(\overline{p}) \rightarrow r) \in \mathcal{P},\, \forall i \in \{1,\ldots,n\}\ \Theta_i = \{\theta_{i1},\ldots,\theta_{im_i}\} \\ \theta = (\biguplus_{i=1}^{n} ?\Theta_i) \uplus \theta_e, \forall i \in \{1,\ldots,n\}, j \in \{1,\ldots,m_i\}\ dom(\theta_{ij}) \subseteq var(p_i) \\ dom(\theta_e) \subseteq vExtra(f(\overline{p}) \rightarrow r), \theta_e \in CSubst^{?}_{\perp} \\ \forall i \in \{1,\ldots,n\}\ m_i > 0, \forall i \in sgArgs(f).m_i = 1 \end{array}$$

$$\mathbf{OR}^{\sigma}_{\pi\beta}\quad \frac{\begin{array}{ccc} e_1 \rightarrowtriangle p_1\theta_{11} & & e_n \rightarrowtriangle p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \rightarrowtriangle p_1\theta_{1m_1} & & e_n \rightarrowtriangle p_n\theta_{nm_n} \quad r\theta \rightarrowtriangle t \end{array}}{f(e_1,\ldots,e_n) \rightarrowtriangle t}$$

$$\begin{array}{l} \text{if } (f(\overline{p}) \rightarrow r) \in \mathcal{P},\, \forall i \in \{1,\ldots,n\}\ \Theta_i = \{\theta_{i1},\ldots,\theta_{im_i}\} \text{ is compressible} \\ \theta = (\biguplus_{i=1}^{n} ?\Theta_i) \uplus \theta_e, \forall i \in \{1,\ldots,n\}, j \in \{1,\ldots,m_i\}\ dom(\theta_{ij}) \subseteq var(p_i) \\ dom(\theta_e) \subseteq vExtra(f(\overline{p}) \rightarrow r), \theta_e \in CSubst^{?}_{\perp} \\ \forall i \in \{1,\ldots,n\}\ m_i > 0, \forall i \in sgArgs(f).m_i = 1 \end{array}$$

Figure 4. The rules $\mathbf{OR}^{\sigma}_{\pi^\alpha}$ and $\mathbf{OR}^{\sigma}_{\pi^\beta}$.

about the use of singular and plural arguments, we conclude this section with a brief outline of the implementation of our prototype.

5.1 The logics $CRWL^{\sigma}_{\pi^\alpha}$ and $CRWL^{\sigma}_{\pi^\beta}$

We assume a mapping *plurality* : $FS \rightarrow \{sg, pl\}^*$ called *plurality map* such that, for every $f \in FS^n$, $plurality(f) = b_1 \ldots b_n$ sets its plurality behavior: if $b_i = sg$ then the ith argument of f will be interpreted with a singular semantics, otherwise it will be interpreted under a plural semantics. In this line $sgArgs(f) = \{i \in \{1,\ldots,ar(f)\} \mid plurality(f)[i] = sg\}$ and $plArgs(f) = \{i \in \{1,\ldots,ar(f)\} \mid plurality(f)[i] = pl\}$ are the sets of singular and plural arguments of some $f \in FS$. In particular we say that f is a *singular function* if $sgArgs(f) = \{1,\ldots,ar(f)\}$ and that it is a *plural function* when $plArgs(f) = \{1,\ldots,ar(f)\}$. A related notion is that of singular and plural variables of a pattern: $sgVars(f(\overline{p})) = \bigcup_{i \in sgArgs(f)} var(p_i)$ and $plVars(f(\overline{p})) = \bigcup_{i \in plArgs(f)} var(p_i)$.

Thus, we employ the plurality map to express the function arguments that are considered singular arguments and plural arguments. With this at hand we now define the combined semantics $CRWL^{\sigma}_{\pi^\alpha}$ and $CRWL^{\sigma}_{\pi^\beta}$ as the result of taking the rules of *CRWL* and replacing the rule **OR** by either the rule $\mathbf{OR}^{\sigma}_{\pi^\alpha}$ or rule $\mathbf{OR}^{\sigma}_{\pi^\beta}$ from Figure 4. As any variant of *CRWL*, these calculi derive reduction statements of the form $\mathscr{P} \vdash_{CRWL^{\sigma}_{\pi^\alpha}} e \rightarrowtriangle t$ and $\mathscr{P} \vdash_{CRWL^{\sigma}_{\pi^\beta}} e \rightarrowtriangle t$ that express that t is (or approximates to) a possible value for e in $CRWL^{\sigma}_{\pi^\alpha}$ or $CRWL^{\sigma}_{\pi^\beta}$, respectively, under the program $\mathscr{P}$. The denotations $[\![e]\!]^{s\alpha p}_{\mathscr{P}}$ and $[\![e]\!]^{s\beta p}_{\mathscr{P}}$ established by these semantics are defined as usual – see Definition 2.

Just like in $\pi^\alpha CRWL$ and $\pi^\beta CRWL$, we consider sets of partial values for parameter passing instead of single partial values, but the novelty is that now these sets are forced to be singleton for singular arguments. This is reflected in the new rules $\mathbf{OR}^\sigma_{\pi^\alpha}$ and $\mathbf{OR}^\sigma_{\pi^\beta}$, corresponding to $\mathbf{POR}^\alpha$ and $\mathbf{POR}^\beta$ respectively, that now have been tuned to take account of the plurality map; as for singular arguments, we are only allowed to compute a single value, thus performing parameter passing over it with a substitution from $CSubst_\perp$ (as obviously $?\{\theta\} = \theta$), and achieving a singular behavior (call-time choice).

Example 5.1
Consider the program $\{f(X, c(Y)) \to d(X, X,\ Y, Y)\}$ and a plurality map such that $plurality(f) = sg\ pl$. The following is a $CRWL^\sigma_{\pi^\alpha}$-proof for the statement $f(0\ ?\ 1, c(0)\ ?\ c(1)) \rightarrowtriangle d(0,0,0,1)$ (some steps have been omitted for the sake of conciseness).

$$\dfrac{\begin{array}{c}(*)\\ \hline c(0)\ ?\ c(1) \rightarrowtriangle c(0)\\ \vdots\\ \hline c(0)\ ?\ c(1) \rightarrowtriangle c(1)\\ \vdots\\ \hline 0\ ?\ 1 \rightarrowtriangle 0\end{array} \qquad \dfrac{\dfrac{\vdots}{0 \rightarrowtriangle 0}\quad \dfrac{\vdots}{0 \rightarrowtriangle 0}\quad \dfrac{\vdots}{0\ ?\ 1 \rightarrowtriangle 0}\quad \dfrac{\vdots}{0\ ?\ 1 \rightarrowtriangle 1}}{d(0,0,0\ ?\ 1, 0\ ?\ 1) \rightarrowtriangle d(0,0,0,1)}\ \mathbf{DC}}{f(0\ ?\ 1, c(0)\ ?\ c(1)) \rightarrowtriangle d(0,0,0,1)}\ \mathbf{OR}^\sigma_{\pi^\alpha}$$

where $(*)$ is the following proof:

$$\dfrac{\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\ \mathbf{DC}}{c(0) \rightarrowtriangle c(0)}\ \mathbf{DC} \quad \dfrac{}{c(1) \rightarrowtriangle \perp}\ \mathbf{B} \quad \dfrac{\vdots}{c(0) \rightarrowtriangle c(0)}}{c(0)\ ?\ c(1) \rightarrowtriangle c(0)}\ \mathbf{OR}^\sigma_{\pi^\alpha}$$

Note that $d(0,1,0,1)$ is not a correct value for the expression $f(0\ ?\ 1, c(0)\ ?\ c(1))$ under $CRWL^\sigma_{\pi^\alpha}$because the first argument of f is singular and therefore the two occurrences of X on the right-hand side of its rule share the same single value fixed on parameter passing. Besides, as this program is in the class $\mathscr{C}^{\alpha\beta}$, it behaves the same under $\pi^\alpha CRWL$ and $\pi^\beta CRWL$, and therefore also under $CRWL^\sigma_{\pi^\alpha}$ and $CRWL^\sigma_{\pi^\beta}$, so the previous proof and comments also hold for $CRWL^\sigma_{\pi^\beta}$.

On the other hand, if we take the same program and evaluate $f(0\ ?\ 1, c(0)\ ?\ c(1))$ under term rewriting – which ignores the plurality map – its behavior is significantly different:

$$\begin{array}{l} f(0\ ?\ 1, \underline{c(0)\ ?\ c(1)}) \to \underline{f(0\ ?\ 1, c(0))} \to d(\underline{0\ ?\ 1}, 0\ ?\ 1, 0, 0)\\ \to d(0, \underline{0\ ?\ 1}, 0, 0) \to d(0, 1, 0, 0)\end{array}$$

The first step resolving the choice between $c(0)$ and $c(1)$ is unavoidable in order to get an expression matching for the only rule for f, thus for any reachable c-term the last two arguments of d will be the same, contrary to what happens in $CRWL^\sigma_{\pi^\alpha}$ and $CRWL^\sigma_{\pi^\beta}$ under the given plurality map. Nevertheless, its first two arguments can be different, contrary to what happens under $CRWL^\sigma_{\pi^\alpha}$ and $CRWL^\sigma_{\pi^\beta}$. In conclusion, it is easy to define a program and a plurality map for them such that neither $CRWL^{\sigma}_{\pi^{\alpha}}$ nor $CRWL^{\sigma}_{\pi^{\beta}}$ are comparable to term rewriting with respect to set inclusion of the computed values.

A useful intuition about programs comes from considering the singular arguments as fixed individual values, while thinking about the plural ones as sets. We could have chosen to specify the plurality or singularity of functions instead of that of its arguments, but the use of arguments with different plurality arises naturally in programs, in the same way it is natural to have arguments of different types. We will illustrate this fact later by means of different examples.

Regarding properties of these semantics (see Riesco and Rodríguez-Hortalá 2011 for more details), both $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ inherit the properties of $\pi^{\alpha}CRWL$ from Section 3.1, for the same reason $\pi^{\beta}CRWL$ inherits the properties of $\pi^{\alpha}CRWL$. The most important among these properties is their compositionality, which expresses the value-based philosophy underlying $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$, "all I know about an expression is its set of values," and that holds for the corresponding reformulation of Theorem 1 – as it can be proved by a straightforward modification of the proof for that theorem. Bubbling is also incorrect for both $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$, just like it happens for $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$: in fact Example 3.2 can be reused to prove it. Nevertheless, just like for $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$, bubbling is correct for a particular kind of contexts, in this case not only for c-contexts but also for the bigger class of *singular contexts* $s\mathscr{C}$, which are contexts whose holes appear only under a nested application of constructor symbols or singular function arguments: $s\mathscr{C} ::= [\ ] \mid c(e_1,\ldots,s\mathscr{C},\ldots,e_n) \mid f(e_1,\ldots,s\mathscr{C},\ldots,e_n)$, with $c \in CS^n$, $f \in FS^n$ such that the subcontext appears in a singular argument of f, and $e_1,\ldots,e_n \in Exp_{\perp}$. For singular contexts we get a compositionality result for singular contexts analogous to that of Proposition 3 – following the same scheme as the proof for Theorem 1 – that can be used to easily prove the correctness of bubbling for singular contexts.

We conclude our discussion about $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ with the following result stating that these are in fact conservative extensions of both $CRWL$ (call-time choice, or equivalently singular non-determinism) and their corresponding plural semantics, as it was apparent from their rules.

Theorem 9 (*Conservative extension*)
Under any program and for any $e \in Exp_{\perp}$:

(1) If the program contains no extra variables and every function is singular, then $[\![e]\!]^{s\alpha p} = [\![e]\!]^{sg} = [\![e]\!]^{s\beta p}$.
(2) If every function is plural, then $[\![e]\!]^{s\alpha p} = [\![e]\!]^{\alpha pl}$ and $[\![e]\!]^{s\beta p} = [\![e]\!]^{\beta pl}$.

Proof
If every function is singular and the program contains no extra variables, then $\mathbf{OR}^{\sigma}_{\pi^{\alpha}}$ and $\mathbf{OR}^{\sigma}_{\pi^{\beta}}$ are equivalent to $\mathbf{OR}$, so $CRWL$, $CRWL^{\sigma}_{\pi^{\alpha}}$, and $CRWL^{\sigma}_{\pi^{\beta}}$, behave the same. Note that the absence of extra variables is essential, as for example from program $\{f \to d(X,X)\}$ we get $[\![f]\!]^{sg} \not\ni d(0,1) \in [\![f]\!]^{s\alpha p} = [\![f]\!]^{s\beta p}$.

Similarly, if every function is plural then $\mathbf{OR}^{\sigma}_{\pi^{\alpha}}$ and $\mathbf{OR}^{\sigma}_{\pi^{\beta}}$ are equivalent to $\mathbf{POR}^{\alpha}$ and $\mathbf{POR}^{\beta}$, respectively. Note that extra variables pose no problem in this case, as

```
(plural SAMPLE-PROGRAM is
   f is plural .
   f(c(X)) -> p(X, X) .
 endp)
```

Figure 5. Concrete syntax of programs.

any of these plural semantics is able to instantiate them with an arbitrary substitution from $CSubst^{?}_{\perp}$. □

5.2 Commands

In this section we introduce concrete syntax of our language and the commands provided by our interpreter. The system is started by loading in the Maude the file `plural.maude`, available at http://gpd.sip.ucm.es/PluralSemantics. It starts an input/output loop that allows the user to introduce commands by enclosing them in parens. Programs start with the keyword `plural`, followed by the module name and the keyword `is`, and finish with `endp`, as exemplified in Figure 5. The body of each program is a list of statements of the form e_1 `->` e_2 `.`, indicating that the program rule $e_1 \rightarrow e_2$ is a part of the program.

The plurality map is specified by means of `is` annotations for each function of the program. These annotations have the form f `is` *plurality* `.`, where *plurality* can take the values `singular` for singular functions, `plural` for plural functions, or a sequence composed by the characters `s` and `p` specifying in more detail the plurality behavior for each function argument, along the lines of the beginning of Section 5.1: If the ith element of this chain is the character `s` then the ith argument of f will be a singular argument, otherwise it will be considered a plural argument. *Functions are considered singular by default when no* `is` *annotation is provided.*

The system is able to evaluate any expression built with the symbols of the program under the semantics specified by the $CRWL^{\sigma}_{\pi^{\alpha}}$ logic. *The prototype does not support programs with extra variables* for two main reasons. First of all it is based on the transformation from Section 4.3, whose adequacy has been only proved for programs without extra variables. But the main reason is the lack of a suitable narrowing mechanism for plural variables, which is the resort usually employed by FLP systems to deal with the space explosion caused by extra variables (López-Fraguas and Sánchez-Hernández 1999; Hanus 2005; Hanus 2007). We consider the development of a plural narrowing mechanism, an interesting subject of future work, but for now and for the rest of the paper, we restrict ourselves to programs not containing extra variables.

The system provides by default the constant c-terms `tt` (for *true*) and `ff` (for *false*), and two more handy functions: the binary function `_?_`, which is used with infix notation, and the `if_then_` function, which is used with mix-fix notation, defined by the following rules:

```
X ? Y -> X .
X ? Y -> Y .
if tt then E -> E .
```

Note that, since no `is` annotation is provided, both functions are singular.

Once a module has been introduced, the user can evaluate expressions with the command

```
(eval [[depth = DEPTH]] EXPRESSION .)
```

where EXPRESSION is the expression to be evaluated and DEPTH is a bound in the number of steps. If this last value is omitted, the search is assumed to be unbounded. If the term can be reduced to a c-term, it will be printed and the user can use

```
(more .)
```

until no more solutions are found.

It is also possible to switch between two evaluation strategies, depth-first and breadth-first, with the following commands:

```
(depth-first .)
(breadth-first .)
```

Finally, the system can be rebooted with the command

```
(reboot .)
```

5.3 Examples

In this section we show how to use the above-mentioned commands by means of two examples.

5.3.1 Clerks

First we show how to implement in our tool the program from Example 3.3, slightly extended by adding a new branch to the bank. The different branches are defined by using the non-deterministic function ?, which here has to be understood as the set union operator. In the same line, for each branch the function employees returns the set of its employees:

```
branches -> madrid ? vigo ? badajoz .

employees(madrid) -> e(pepe, men, clerk) ? e(paco, men, boss) .
employees(vigo) -> e(maria, women, clerk) ? e(jaime, men, boss) .
employees(badajoz) -> e(laura, women, clerk) ? e(david, men, clerk) .
```

Now, we define a function twoclerks, which searches in the database for the names of two employees working as clerks. It calls the function find, which has been marked with the keyword plural in order to express that its argument will be understood as a set of records from the database of the bank. Therefore, although the same variable N is used in the two components of the pair on the right-hand side of its rule, each one can be instantiated with different values:

```
twoclerks -> find(employees(branches)) .
find is plural .
find(e(N,G,clerk)) -> p(N,N) .
```

Once the module has been loaded in our system,[9] we can use the `eval` command to evaluate expressions, and the command `more` to find the next solutions:

```
Maude> load clerks.plural
Module introduced.
Both alpha and beta plural semantics supported for this program.

Maude> (eval twoclerks .)
Result: p(pepe,pepe)

Maude> (more .)
Result: p(pepe,maria)
```

This program works as we expected, even if all the functions are marked as plural (i.e., if $\pi^{\alpha}CRWL$ is used). However, it can be improved in several directions. First of all, we are interested in getting two *different* clerks. To do that we will define a function `vals` that generates a list containing different values of its argument. This function will use an auxiliary function `newIns` that appends an element at the beginning of the list ensuring that the remaining elements of the list are different from the new one. This is checked by `diffL`, which returns the list in its second argument when it does not contain its first argument, or otherwise fails. Thus, a disequality test is needed, but in our minimal framework we do not dispose of disequality constraints, common in FLP languages (Hanus 2007; Antoy and Hanus 2010). Nevertheless, we can implement a ground version of disequality through regular program rules as it is done here in the function `neq`.

```
newIns is singular .
newIns(X, Xs) -> cons(X, diffL(X, Xs)) .

diffL(X, nil) -> nil .
diffL(X, cons(Y, Xs)) ->
     if neq(X, Y) then cons(Y, diffL(X, Xs)) .

neq(pepe, paco) -> tt .
neq(pepe, maria) -> tt .
. . .
```

Note that we need `newIns`, `diffL`, and `neq` to be singular because these essentially perform tests, and when performing a test we naturally want the returning value to be the same that has been tested. For example, the following program

```
isWoman(maria) -> tt .
isWoman(laura) -> tt .
. . .
filterWomen(P) -> if isWoman(P) then P
```

would have a funny behavior if `filterWomen` had been declared a plural function because then for `filterWomen(maria ? pepe)` we could compute `pepe` as a correct value.

[9] The tool also indicates whether the program belongs to the class $\mathscr{C}^{\alpha\beta}$ – remember that in that case $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ would be equivalent and so both would be supported by the system – or not.

On the other hand, the function `vals` is marked as *plural* because it is devised to generate lists of different values of its argument. Note the combination of plurality to obtain more than one value from the argument of `vals`, and singularity, which is needed for the tests performed by `newIns`:

```
vals is plural .
vals(X) -> newIns(X, vals(X)) .
```

We now generalize our search function to look for any number of clerks, not just two. To do that we will use the function `nVals` below, which returns a list of different values corresponding to different evaluations of its second argument. Therefore, that second argument has to be declared as plural, while its first argument is singular, as it fixes the number of values claimed (that is, the length of the returning list in the Peano notation for natural numbers):

```
nVals is sp .
nVals(N, E) -> take(N, vals(E)) .

take(s(N), cons(X, Xs)) -> cons(X, take(N, Xs)) .
take(z, Xs) -> nil .
```

This `nVals` function is an example of how the use of plural arguments allows us to simulate some features that in a pure call-time choice context have to be defined at the meta level, in this case the `collect` (López-Fraguas and Sánchez-Hernández 1999) or the `findall` (Hanus 2005) primitives of standard FLP systems.

Finally, the function `nClerks` starts the search for a number of different clerks specified by the user. It uses the auxiliary function `findClerks`, which returns the name of the clerks:

```
nClerks is singular .
nClerks(N) -> nVals(N, findClerk(employees(branches))) .

findClerk is singular .
findClerk(e(N,G,clerk)) -> N .
```

Now we can search for three different clerks, obtaining `pepe`, `maria`, and `laura` as the first possible result:

```
Maude> (eval nClerks(s(s(s(z)))) .)
Result: cons(pepe,cons(maria,cons(laura,nil)))
```

As anticipated in Example 3.4, we can use this technique to solve the problem of finding the names of the clerks paired with their genre, but avoiding the wrong information mix-up caused by a purely plural approach using the style of the plural `find` function above, under $\pi^{\alpha}\mathit{CRWL}$. To do this we just have to define a new auxiliary function `findClerksNG`, which this time returns a pair composed by the name of the clerk and his or her genre.

```
nClerksNG is singular .
nClerksNG(N) -> nVals(N, findClerkNG(employees(branches))) .

findClerkNG is singular .
findClerkNG(e(N,G,clerk)) -> p(N, G) .
```

The fact that findClerksNG is singular, just like findClerks, ensures that the names and genres will be correctly paired. Besides, note that the whole Clerks program presented here belongs to the class $\mathscr{C}^{\alpha\beta}$, therefore its evaluation under $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ is the same and any wrong information mix-up is prevented. We can check this by searching again for three different clerks:

```
Maude> (eval nClerksNG(s(s(s(z)))) .)
Result: cons(p(pepe,men),cons(p(maria,women),cons(p(laura,women),nil)))
```

In the next example we will see more clearly how to decide the plurality of functions. Remember that the key idea is that singular arguments are used *to fix their values*, while plural arguments are needed when we want to use *sets of values*.

5.3.2 Dungeon

Ulysses has been captured and he wants to cheat his guardians using the gold he carries from Troy. Thus, he needs to know whether there is an escape (what we define as obtaining the key of its jail) and, if possible, which is the path to freedom (we define each step of this path as a pair composed of a guardian and the item Ulysses obtains from him).

He uses the function ask to interchange items and information with his guardians. Since each guardian provides different information, we have to assure that they are not mixed, and thus its first argument will be singular; on the other hand, he may offer different items to the same guardian, thus the second argument will be plural: this function needs plurality sp:

```
ask is sp .
```

The guardians have a complex behavior, circe exchanges Ulysses' trojan-gold by either the sirens-secret or an item(treasure-map); calypso, once she receives the sirens-secret, offers the item(chest-code); aeolus can combine two items;[10] and polyphemus gives Ulysses the key once he can give him the combination of the treasure-map and the chest-code:

```
ask(circe, trojan-gold) -> item(treasure-map) ? sirens-secret .
ask(calypso, sirens-secret) -> item(chest-code) .
ask(aeolus, item(M)) -> combine(M,M) .
ask(polyphemus, combine(treasure-map, chest-code)) -> key .
```

In the same line, askWho has as arguments a (*fixed*) guardian and a message (probably with many items) for him, so it also has plurality sp. This function returns the next step in the Ulysses' path to freedom, that is, a pair with the guardian and the items obtained from him with the function ask:

```
askWho is sp .
askWho(Guardian, Message) -> p(Guardian, ask(Guardian, Message)) .
```

[10] Note that we say *two* items when the function only shows *one*. This rule uses the expressive power of plural semantics to allow the combination of different items.

The following functions, which are in charge of computing the actions that must be performed in order to escape, are marked as plural because they treat their corresponding arguments as sets of pairs where the second component is an item or some piece of information, and the first one is the actor which provided it. The function `discoverHow` returns the set of pairs of that shape that can be obtained starting from those contained in its argument, and then chatting to the guardians. Hence, it returns either its argument or the result of exchanging the current information with some guardian and then iterating the process. That exchange is performed with `discStepHow`, which non-deterministically offers some of the items or information available to one of the guardians:

```
discoverHow is plural .
discoverHow(T) -> T ? discoverHow(discStepHow(T) ? T) .

discStepHow is plural .
discStepHow(p(W, M)) -> askWho(guardians, M) .

guardians -> circe ? calypso ? aeolus ? polyphemus .
```

Note that the additional disjunction `? T` in the recursive call to `discStepHow` is needed to be able to combine the old information with the new one resulting after one exchanging step. This point can be illustrated better with the following program:

```
genPairs is plural .
genPairs(P) -> P ? genPairs(genPairsStep(P) ? P) .

genPairsStep is plural .
genPairsStep(P) -> p(P, P) .

genPairsBad is plural .
genPairsBad(P) -> P ? genPairsBad(genPairsStep(P)) .
```

There the functions `genPairs` and `genPairsBad` follow the same pattern as the `discoverHow`, but this time are designed to generate values made up with pairs and the supplied argument. Besides, these functions share the same "step function" `genPairsStep`. Nevertheless, their behavior is very different, as we can see evaluating the expressions `genPairs(z)` and `genPairsBad(z)`: the point is that the value `p(p(z,z),z)` can be computed for the former but not for the latter because `z` and `p(z,z)` are values generated in different recursive calls to `genPairsBad`. But this poses no problem for `genPairs` because the extra `? P` in its definition makes it possible to combine those values.

Finally, the search is started with the function `escapeHow`, which initializes the search with the trojan gold provided by Ulysses:

```
escapeHow -> discoverHow(p(ulysses, trojan-gold)) .
```

Once the module is introduced, we can start the search with the following command:

```
Maude> (eval escapeHow .)
Result: p(ulysses,trojan-gold)
```

When this first result has been computed, we can ask the tool for more with the command `more`, which progressively will show the path followed by Ulysses to escape:

```
Maude> (more .)
Result: p(circe,item(treasure-map))

Maude> (more .)
Result: p(circe,sirens-secret)

Maude> (more .)
Result: p(calypso,item(chest-code))

. . .

Maude> (more .)
Result: p(polyphemus,key)
```

In this example the function `discoverHow` is an instance of an interesting pattern of plural function: a function that performs deduction by repeatedly combining the information we have fed it with the information it infers in one step of deduction. Therefore, in its definition the function `?` has to be understood again as the set union operator, as it is used to add elements to the set of deduced information. On the other hand, the use of a singular argument in `askWho` is unavoidable to be able to keep track of the guardian who answers the question, while its second argument has to be plural because it represents the knowledge accumulated so far.

Several variants of this problem can be conceived, in particular currently it is simplified because the items are not lost after each exchange – that is why Ulysses' bag is bottomless. Anyway, we think that this version of the problem is relevant because, in fact, it corresponds to a small model of an intruder analysis for a security protocol, where Ulysses is the intruder, the guardians are the honest principals, the key is the secret, and complex behaviors of the principals can be described through the patterns on left-hand sides of program rules. In this case we assume that the intruder is able to store any amount of information, and that this information can be used many times. Nevertheless, we also think that different variants of the problem should be tackled in future, and that the addition of equality and disequality constraints to our framework could be decisive to deal with those problems.

With this program we conclude our presentation of some examples that show the expressive capabilities of our plural semantics. In these examples we have tried to find a way of using $CRWL_{\pi^{\alpha}}^{\sigma}$ for programming, so it could be more than just a semantic eccentricity. Although we have found some interesting uses of our plural semantics, in particular the meta-like function `nVals`, and the deduction programming pattern corresponding to `discoverHow`, we cannot still say that we have found a "killer application" for our plural semantics. Only time will tell us if these semantics are useful, because these proposals are still too young to have a reasonable benchmark collection. Our prototype opens the door to experimenting with these new semantics, and in that sense it contributes to the development of such collection. Anyway, we admit that our plural semantics probably will only be useful in some fragments of the programs, and that is why we have proposed to combine it with the usual singular semantics of FLP. As a final remark, the reader can check manually or by using our prototype that all the program examples in this section belong to the $\mathscr{C}^{\alpha\beta}$ class – and hence they behave the same both under $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$– which motivates the relevance of that class of programs. But again, as the collection of examples is very small, this does not give a strong argument about the usefulness of this class of programs, but just an encouraging indicator.

5.4 Discussion: to be singular or to be plural?

After these examples, we (hopefully) should have some intuitions about how to decide the plurality of function arguments. Our first resort is considering that plural arguments are used to represent sets of values, while singular arguments denote single values. But this does not work for any situation, for example, consider the function `findClerk` whose plurality is singular, although its argument intuitively denotes a set of records from the database. On the other hand, we may consider that its argument denotes a single record, and that `findClerk` defines how to extract the name from a single employee, which motivates the final plurality choice. In this case the program behaves the same declaring `findClerk` either singular or plural, because the variables in its arguments are used only once. As a rule of thumb we should try to have as little plural arguments as possible because these arguments increase the search space more than the singular ones, as using a plural semantics we can compute more values than under a singular semantics as seen in Section 4.1. Hence, in this case it is better to declare `findClerk` as singular.

Thus, having a more formal criterion about the equivalence of plurality maps would be useful to minimize the search space of our programs and understand them better. A static adaptation of the determinism analysis of Caballero and López-Fraguas (2003) could be useful, as it would help us to detect deterministic functions of our programs for which the plurality map would not matter as we expect to easily extend the equivalence results of singular/call-time choice and run-time choice for deterministic programs of López-Fraguas *et al.* (2007, 2010) to our plural semantics. We also should try to develop equational laws about non-determinism. In fact, a first step in this line is the discussion about the correctness of bubbling for singular contexts from Section 5.1. Anyway, all these are subjects of future work.

5.5 Implementation

The system described in the previous sections has been implemented in the Maude system (Clavel *et al.* 2007) a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. The fundamental ingredients for this implementation are a core language into which all programs are transformed, and an interpreter for the operational semantics of the core language that is used to execute programs.

The transformation into core language treats each program rule separately and applies two different transformation stages to them. The first one applies the modification of the transformation described in Definition 7, but now taking into account only those arguments marked as *pl* in the plurality map described in Section 5.1. Then in the second stage, which consists in the modification of the sharing transformation of López-Fraguas *et al.* (2009c, Def. 1), we introduce a let-binding for each singular variable that also appears on the right-hand side, therefore obtaining subexpression-sharing, and as a consequence, a singular behavior for those arguments.

Once source programs have been transformed into core programs, we can execute them by using a heap-based operational semantics for the core language (Riesco and Rodríguez-Hortalá 2010b). A heap is just a mapping from variables to expressions that represents a graph structure, as the image of each variable is interpreted as a subgraph definition. The nodes of that implied graph are defined according to those let-bindings introduced by the transformation into core language. The operational semantics manipulates this heap, and contains rules for removing useless bindings, propagating the terms associated to a variable, and for creating new bindings for each singular argument when their corresponding let-bindings are found. Finally, in order to turn the operational semantics of the core language into an effective operational mechanism for $CRWL^{\sigma}_{\pi^{\alpha}}$, we have adapted the natural rewriting strategy in Escobar (2004) to deal with these heaps, ensuring that the evaluation is performed on-demand. The program transformation into core language, the interpreter, and our adaptation of natural rewriting have been implemented in the Maude with an intensive exploiting of its reflection capabilities, thus obtaining an executable interpreter for $CRWL^{\sigma}_{\pi^{\alpha}}$. More details about our implementation can be found in Riesco and Rodríguez-Hortalá (2010b).

We decided to follow the line of employing the transformation from Definition 7 and then using a language that implements non-deterministic term rewriting to run the transformed program, because our motivation was to obtain a simple proof of concept prototype that could be used to experiment with the new semantics, but obtaining an optimized implementation is out of scope of this work. The addition of the natural rewriting on-demand strategy was necessary to reduce the search space to a reasonable size, but we are aware of other approaches that could improve the efficiency of the system. In particular, we could have adapted the techniques in Antoy *et al.* (2002), Braßel and Huch (2007), and Braßel *et al.* (2011) that rely on turning the function $? \in FS^2$ into a constructor to explicitly represent non-deterministic computations in a deterministic language, which results in additional advantages like a kind of backtracking memoization called "sharing across non-determinism." That would have allowed us to use the Maude functional modules, which are much more efficient than the non-deterministic system modules that are used in our current implementation. But as functional modules perform eager evaluation, we should also employ the context-sensitive rewriting (Lucas 1998) features of the Maude – offered as `strat` annotations – to get the lazy evaluation that corresponds to our semantics. That would have entailed adapting the techniques from Antoy *et al.* (2002), Braßel and Huch (2007), and Braßel *et al.* (2011) from a call-time choice setting to the run-time choice semantics of term rewriting, and also using the techniques in Lucas (1997) to introduce the `strat` annotations needed to ensure lazy evaluation. This is a possible road map that could be followed in case a more optimized implementation of $CRWL^{\sigma}_{\pi^{\alpha}}$ is to be developed. The monad transformer of Fischer *et al.* (2009) is another alternative in the same line, as it also provides a representation of non-determinism with support for memoization in a deterministic language, in this case Haskell, which could be used as the basis for the implementation of $CRWL^{\sigma}_{\pi^{\alpha}}$ by modifying the transformation from FLP programs with a call-time choice into Haskell from Braßel *et al.* (2010). As that work is placed in a higher order setting, our plural semantics should be first extended with higher order capabilities, following the line of González-Moreno *et al.* (1997).

6 Concluding remarks and future work

The starting point of this work is the observation that the traditional identification between a run-time choice and a plural denotational semantics is wrong in a non-deterministic functional language with pattern matching. To illustrate this, we have provided formulations for two different plural semantics that are different from a run-time choice: the $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ semantics. We argue that the run-time choice semantics induced by term rewriting is not the best option for a value-based programming language like current implementations of FLP because of its lack of compositionality. Nevertheless, our plural semantics are compositional for a simple notion of value – the notion of partial c-term – just like the usual call-time choice semantics adopted by modern FLP languages, following the value-based philosophy of the FLP paradigm: "all I care about an expression is the set of its values." This, together with the fact that our concrete formulations for these plural semantics are variants of the $CRWL$ logic– a standard formulation for singular/call-time choice semantics in FLP – turns the problem of devising a combined semantics for singular and plural non-determinism into a trivial task, getting the $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ logics as a result. The combination of singular and plural semantics in the same language is interesting and follows naturally when programming, as it allows us to reuse known programming patterns from the more usual singular/call-time choice semantics, standard in modern FLP systems, while we are still able to use the new capabilities of the novel plural semantics for some interesting fragments of the program. In these logics, apart from the program, the user may specify for each function its arguments that will be marked as singular and plural, resulting in different parameter passing mechanisms. A simple intuition that works in most situations can be considering plural arguments as sets of values and singular arguments as individual values. We have not only proposed such semantic combinations but have also provided a prototype implementation for $CRWL^{\sigma}_{\pi^{\alpha}}$ using the Maude system (see Riesco and Rodríguez-Hortalá 2010a, 2010b for details about the implementation), in which the program transformation to simulate $\pi^{\alpha}CRWL$ with term rewriting – a standard formulation for run-time choice – also presented in this work as a crucial ingredient. The resulting system, available at http://gpd.sip.ucm.es/PluralSemantics is an interpreter for the $CRWL^{\sigma}_{\pi^{\alpha}}$ logic that we have used to develop several program examples that exploit the new expressive capabilities of the combined semantics to improve the declarative flavor of programs.

Along the way we have also made several contributions at the foundational level. We have studied the technical properties of $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$, providing formal proofs for its compositionality and also for other interesting properties like polarity, several monotonicity properties for substitutions, and a restricted form of bubbling for constructor contexts. Then we have compared different semantics for non-determinism considered in this work with respect to the set of computed values, concluding that these form the inclusion chain $CRWL \subseteq$ term rewriting $\subseteq \pi^{\beta}CRWL \subseteq \pi^{\alpha}CRWL$, corresponding to the chain singular/call-time choice $\subseteq$ run-time choice $\subseteq$ β-plural $\subseteq$ α-plural. Besides, we have determined that for the class of programs $\mathscr{C}^{\alpha\beta}$, characterized by a simple syntactic criterion, our plural semantics proposals $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ are equivalent. We have also provided a formal proof of the adequacy of the (non-optimized version of the) transformation used by our prototype to simulate $\pi^{\alpha}CRWL$ with term rewriting. As a consequence, this transformation can be used to simulate $\pi^{\beta}CRWL$ for programs in the class $\mathscr{C}^{\alpha\beta}$. Regarding the combined semantics $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$, it is easy to see that these inherit the good properties of $CRWL$, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$, and we have also proved that the combined semantics are conservative extensions of both singular/call-time choice and their corresponding plural semantics.

These questions were first approached in previous works by the authors (Rodríguez-Hortalá 2008; Riesco and Rodríguez-Hortalá 2010a, 2010b); however, in this paper we do not only give a revised and unified presentation but also include several important novel results.

- All the technical results from those works have been extended to deal with programs with extra variables except those results regarding the simulation of $\pi^{\alpha}CRWL$ with term rewriting from Section 4.3. The new technical results have been also proved for programs with extra variables. Besides, we have fixed some errata from the original works, in particular the formulation of bubbling for $\pi^{\alpha}CRWL$, the definition of the operator ? over sequences of $CSubst_{\perp}$, and also some other minor mistakes in the proofs. The formulations of bubbling for constructor and singular contexts are novel contributions of this paper.
- The plural semantics $\pi^{\beta}CRWL$, inspired in the proposal from Braßel and Berghammer(2009), is introduced in this work for the first time. We give clear explanations of some problematic situations where $\pi^{\alpha}CRWL$ performs a wrong information mix-up, and how our attempts to fix those problems, inspired in the solutions from Braßel and Berghammer (2009), led us to the current formulation of $\pi^{\beta}CRWL$, which leans on the notion of compressible set of partial c-substitutions.
- As the formulations of $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ are very similar, it is not difficult to check that $\pi^{\beta}CRWL$ also enjoys the same basic properties of $\pi^{\alpha}CRWL$. Nevertheless, it was more difficult to place $\pi^{\beta}CRWL$ in the semantic inclusion chain from Rodríguez-Hortalá (2008), being a key idea the notion of compressible completion of a set of $CSubst_{\perp}$, and its related results. The

characterization of the class of programs $\mathscr{C}^{\alpha\beta}$, for which $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$ are equivalent, and the formal proof for that equivalence are also novel, obviously.

- Finally, the logic $CRWL^{\sigma}_{\pi^{\beta}}$ is also a novel contribution of this work, but in this case its definition is straightforward, because it follows the same pattern as the definition for $CRWL^{\sigma}_{\pi^{\alpha}}$.

Previous to our work, not much work has been done in the combination of singular and plural non-determinism in functional or functional-logic programming, since the mainstream approaches (Wadler 1985; López-Fraguas and Sánchez-Hernández 1999; Hanus 2005) only support the usual singular semantics. Closer are the combinations of call-time and run-time choices of López-Fraguas *et al.* (2009a, 2009b) , which anyway follow a different approach, as the plural sides of $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ are essentially different to run-time choice. Anyway, we still think that the combination of call-time choice and run-time choice is not very suitable for value-based languages because of the lack of compositionality for values under run-time choice. The monad transformer of Fischer *et al.* (2009), devised to improve the laziness of non-deterministic monads while retaining a call-time choice semantics, is based on a `share` combinator, which plays a role similar to the let-bindings of our core language. The authors seem to be interested in staying in a pure call-time choice framework, but maybe a combination of call-time and run-time choice could be achieved there too, getting something similar to López-Fraguas *et al.* (2009a), but again essentially different from $CRWL^{\sigma}_{\pi^{\alpha}}$ and $CRWL^{\sigma}_{\pi^{\beta}}$ for the same reason. Besides, that work is focused on the implementation issues of FLP in concrete deterministic functional languages, while in ours we start from the more abstract world of CSs and are fundamentally concerned in exploring the language design space.

We contemplate several interesting subjects of future work. As pointed in Sections 4.3 and 5.2, the development of a suitable plural narrowing mechanism would be the key for finding an effective way of handling extra and free variables. Besides, in our examples it has arisen the necessity of equality and disequality constraints (whose ground versions have been simulated by using regular functions) that will ease and shorten the definition of programs, and increase the expressiveness of the setting. Both subjects would be interesting at theoretical and practical levels, as we could then improve our prototype by extending it with those new features.

Similarly, adding higher order capabilities by the extension of $CRWL^{\sigma}_{\pi^{\alpha}}$ in the line of González-Moreno *et al.* (1997), and implementing them by means of the classic transformation of Warren (1982), would also be interesting and it is standard in the field of FLP. Then, for example, we could define a more generic version of `discoverHow` with an additional argument for the function used to perform a deduction step (`discStepHow` in our dungeon problem). This higher order version of $CRWL^{\sigma}_{\pi^{\alpha}}$ could also be used to face the challenges regarding the implementation of type classes in FLP through the classical transformational technique of Wadler and Blott (1989) pointed out by Lux (2009). Although some solutions based on the frameworks of López-Fraguas *et al.* (2009a, 2009b) were already proposed in

Rodriguez-Hortala (2009), we think that an alternative based on $CRWL^{\sigma}_{\pi^{\alpha}}$ would be better, thanks to its clean and compositional semantics. More novel would be using the matching-modulo capacities of the Maude to enhance the expressiveness of the semantics, after a corresponding revision of the theory of $CRWL^{\sigma}_{\pi^{\alpha}}$. Besides, as mentioned at the end of Section 5.5, some additional research must be done to improve the performance of the interpreter, especially because of the increase of the size of the search space due to the use of plural arguments. As we pointed out there, an explicit representation of non-determinism on a deterministic language seems promising (Antoy *et al.* 2002; Braßel and Huch 2007; Fischer *et al.* 2009; Braßel *et al.* 2011), in particular the memoization capabilities of these approaches could be exploited to deal with the non-determinism overhead caused by plural arguments. Some possible concretization of this idea could be using the Maude functional modules with `strat` annotations using the techniques in Lucas (1997), or adapting the transformation in Braßel *et al.* (2010).

As suggested in Section 5.4, finding a criterion for the equivalence of plurality maps and defining more equational laws for non-determinism, besides the restricted forms of bubbling proposed here, would improve the understanding of programs, which could finally lead to the development of more interesting program examples that could illustrate the interest of the semantics. In this line we also find interesting the relation between different notions of determinism entailed by $CRWL$, $\pi^{\alpha}CRWL$, and $\pi^{\beta}CRWL$, and the relation between confluence of term rewriting and those notions of determinism. We already made some advances in this line in previous works (López-Fraguas *et al.* 2007, 2010).

To conclude, an investigation of the technical relation between $\pi^{\beta}CRWL$ and the plural semantics from Braßel and Berghammer (2009), which inspired it, would be very interesting. We conjecture a strong semantic equivalence between them.

References

Albert, E., Hanus, M., Huch, F., Oliver, J. and Vidal, G. 2005. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation 40,* 1, 795–829.

Antoy, S., Brown, D. and Chiang, S. 2007. Lazy context cloning for non-deterministic graph rewriting. Proceedings of the Termgraph'06. *Electronic Notes in Theoretical Computer Science – ENTCS, 176,* 1, 61–70.

Antoy, S. and Hanus, M. 2002. Functional logic design patterns. In *FLOPS*, Z. Hu and M. Rodríguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 2441. Springer, New York, USA, 67–87.

Antoy, S. and Hanus, M. 2010. Functional logic programming. *Communications of ACM 53,* 4, 74–85.

Antoy, S., Iranzo, P. J. and Massey, B. 2002. Improving the efficiency of non-deterministic computations. *Electronic Notes in Theoretical Computer Science 64*, 73–94.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. 1995. The call-by-need lambda calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1995*. ACM, 233–246.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E. and Ringeissen, C. 1998. An overview of ELAN. In *Proceedings of the Second International Workshop on Rewriting Logic and Its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4,* C. Kirchner and H. Kirchner, Eds. Electronic Notes in Theoretical Computer Science, vol. 15. Elsevier, 329–344. http://www.elsevier.nl/locate/entcs/volume15.html.

Brassel, B. and Berghammer, R. 2009. Functional (logic) programs as equations over order-sorted algebras. *Informal Proceedings of 19th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2009.*

Brassel, B., Fischer, S., Hanus, M. and Reck, F. 2010. Transforming functional logic programs into monadic functional programs. In *WFLP*, J. Mariño, Ed. Lecture Notes in Computer Science, vol. 6559. Springer, New York, USA, 30–47.

Brassel, B., Hanus, M., Peemöller, B. and Reck, F. 2011. KiCS2: A new compiler from Curry to Haskell. In *Proceedings of the 20th International Conference on Functional and Constraint Logic Programming, WFLP 2011*, H. Kuchen, Ed. Lecture Notes in Computer Science, vol. 6816. Springer, New York, USA, 1–18.

Brassel, B. and Huch, F. 2007. On a tighter integration of functional and logic programming. In *APLAS*, Z. Shao, Ed. Lecture Notes in Computer Science, vol. 4807. Springer, New York, USA, 122–138.

Caballero, R. and López-Fraguas, F. 2003. Improving deterministic computations in lazy functional logic languages. *Journal of Functional and Logic Programming 2003,* 1, 23.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. 2007. *All About Maude: A High-Performance Logical Framework.* Lecture Notes in Computer Science, vol. 4350. Springer, New York, USA.

DeGroot, D. and Lindstrom, G. E. 1986. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, Upper Saddle River, NJ, USA.

Dijkstra, E. W. 1997. *A Discipline of Programming.* Prentice Hall, Upper Saddle River, NJ, USA.

Echahed, R. and Janodet, J.-C. 1998. Admissible graph rewriting and narrowing. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming.* MIT Press, Manchester, UK, 325–340.

Escobar, S. 2004. Implementing natural rewriting and narrowing efficiently. In *Proceedings of the 7th International Symposium on Functional and Logic Programming, FLOPS 2004*, Y. Kameyama and P. Stuckey, Eds. Lecture Notes in Computer Science, vol. 2998. Springer, New York, USA, 147–162.

Fischer, S., Kiselyov, O. and Shan, C.-C. 2009. Purely functional lazy non-deterministic programming. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming.* ACM, New York, NY, USA, 11–22.

Futatsugi, K. and Diaconescu, R. 1998. *CafeOBJ Report.* AMAST Series in Computing, vol. 6. World Scientific, Singapore.

González-Moreno, J. C., Hortalá-González, T., López-Fraguas, F. and Rodríguez-Artalejo, M. 1996. A rewriting logic for declarative programming. In *Proceedings of the European Symposium on Programming, ESOP 1996.* Lecture Notes in Computer Science, vol. 1058. Springer, New York, USA, 156–172.

González-Moreno, J. C., Hortalá-González, T., López-Fraguas, F. and Rodríguez-Artalejo, M. 1999. An approach to declarative programming-based on a rewriting logic. *Journal of Logic Programming 40,* 1, 47–87.

González-Moreno, J., Hortalá-González, M. and Rodríguez-Artalejo, M. 1997. A higher order rewriting logic for functional logic programming. In *Proceedings of the International Conference on Logic Programming, ICLP 1997.* MIT Press, Cambridge, MA, USA, 153–167.

Hanus, M. 2005. *Functional Logic Programming: From Theory to Curry*. Tech. Rep. Christian-Albrechts-Universität, Kiel, Germany.

Hanus, M. (Ed.) 2006. Curry: An integrated functional logic language (version 0.8.2). Accessed 3 October 2010. URL: *http://www.informatik.uni-kiel.de/~curry/report.html.*

Hanus, M. 2007. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming, ICLP 2007*. LNCS, vol. 4670. Springer, New York, USA, 45–75.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F. and Puebla, G. 2012. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252.

Hughes, J. and O'Donnell, J. 1990. Expressing and reasoning about non-deterministic functional programs. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Workshops in Computing, Springer, London, UK, 308–328.

Hussmann, H. 1993. *Non-Determinism in Algebraic Specifications and Algebraic Programs.* Birkhäuser Verlag, Berlin, Germany.

Launchbury, J. 1993. A natural semantics for lazy evaluation. In *Proceedings of the ACM Symposium on Principles of Programming Languages, POPL 1993*. ACM Press, New York, USA, 144–154.

López-Fraguas, F., Martin-Martin, E., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. Available on request. Rewriting and narrowing for constructor systems with call-time choice semantics. Submitted to *Theory and Practice of Logic Programming*.

López-Fraguas, F., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2007. A simple rewrite notion for call-time choice semantics. In *Proceedings of the Principles and Practice of Declarative Programming*. ACM Press, New York, USA, 197–208.

López-Fraguas, F., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2008. Rewriting and call-time choice: The HO case. In *Proceedings of the 9th International Symposium on Functional and Logic Programming, FLOPS 2008*. Lecture Notes in Computer Science, vol. 4989. Springer, New York, USA, 147–162.

López-Fraguas, F., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2009a. A fully abstract semantics for constructor systems. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications, RTA 2009*. Lecture Notes in Computer Science, vol. 5595. Springer, Berlin, Germany, 320–334.

López-Fraguas, F., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2009b. A lightweight combination of semantics for non-deterministic functions. *CoRR abs/0903.2205.*

López-Fraguas, F., Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2009c. A flexible framework for programming with non-deterministic functions. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2009.* ACM, New York, USA, 91–100.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proceedings of the Rewriting Techniques and Applications, RTA 1999.* Lecture Notes in Computer Science, vol. 1631. Springer, New York, USA, 244–247.

Lucas, S. 1997. Needed reductions with context-sensitive rewriting. In *ALP/HOA*, M. Hanus, J. Heering and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1298. Springer, Berlin, Germany, 129–143.

Lucas, S. 1998. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming 1998,* 1, 1–61.

Lux, W. 2009. Curry mailing list: Type-classes and call-time choice vs. run-time choice. Accessed 3 October 2010. URL: http://www.informatik.uni-kiel.de/∼curry/listarchive/0790.html.

McCarthy, J. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirshberg, Eds. North-Holland, Amsterdam, Netherlands, 33–70.

Plasmeijer, R. J. and van Eekelen, M. C. J. D. 1993. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Boston, MA, USA.

Plump, D. 1999. Term graph rewriting. In *Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages, and Tools*. World Scientific, River Edge, NJ, USA, 3–61.

Riesco, A. and Rodríguez-Hortalá, J. 2010a. A natural implementation of plural semantics in Maude. In *Proceedings of the 9th Workshop on Language Descriptions, Tools, and Applications, LDTA 2009*, T. Ekman and J. Vinju, Eds. Electronic Notes in Computer Science, vol. 253(7). Elsevier, Maryland Heights, MO, USA, 165–175.

Riesco, A. and Rodríguez-Hortalá, J. 2010b. Programming with singular and plural non-deterministic functions. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*. ACM, New York, USA, 83–92.

Riesco, A. and Rodríguez-Hortalá, J. 2011. *Singular and Plural Functions for Functional Logic Programming: Detailed Proofs*. Tech. Rep. SIC-9/11, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid. November. URL: http://gpd.sip.ucm.es/PluralSemantics.

Rodríguez-Hortalá, J. 2008. A hierarchy of semantics for non-deterministic term rewriting systems. In *Proceedings Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008*. Leibniz International Proceedings in Informatics. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Rodríguez-Hortalá, J. 2009. Curry mailing list: Re: Type-classes and call-time choice vs. run-time choice. Accessed 3 October 2010. URL: http://www.informatik.uni-kiel.de/∼curry/listarchive/0801.html.

Rodríguez-Hortalá, J. and Sánchez-Hernández, J. 2008. Functions and lazy evaluation in prolog. *Electronic Notes in Theoretical Computer Science 206*, 153–174.

Roy, P. V. and Haridi, S. 2004. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA.

Søndergaard, H. and Sestoft, P. 1990. Referential transparency, definiteness and unfoldability. *Acta Informatica 27,* 6, 505–517.

Søndergaard, H. and Sestoft, P. 1992. Non-determinism in functional languages. *The Computer Journal 35,* 5, 514–523.

Sterling, L. and Shapiro, E. 1986. *The Art of Prolog*. MIT Press, MA, USA.

TeReSe. 2003. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge, UK.

The Mercury Team. 2012. *The Mercury Language Reference Manual, Version 11.07.1*. The Mercury Project, URL: http://www.mercury.csse.unimelb.edu.au/rss.xml.

Wadler, P. 1985. How to replace failure by a list of successes. In *Proceedings of the Functional Programming and Computer Architecture*. LNCS, vol. 201. Springer, Berlin, Germany, 113–128.

Wadler, P. and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 60–76.

Warren, D. H. 1982. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence 10*, J. Hayes, D. Michie and Y.-H. Pao, Eds. Ellis Horwood, Chichester, UK, 441–454.