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 Søndergaard 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 \textit{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-Hortalá (2008), where the \( \pi\alpha\text{CRWL} \) logic – named \( \pi\text{CRWL} \) 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 \( \text{CRWL}^3 \) (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 \( \pi\alpha\text{CRWL} \) from Rodriguez-Hortalá (2008) with a user level introduction to the Maude-based transformational prototype for \( \pi\alpha\text{CRWL} \).  

\(^3\) Constructor-based ReWriting Logic.  
\(^4\) In fact, angelic non-strict call-time choice.
Singular and plural functions for functional logic programming

(Riesco and Rodríguez-Hortalá 2010a). We have also included the results obtained in Riesco and Rodriguez-Hortalá (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 πCRWL, 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 πβCRWL 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 πCRWL 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 πCRWL and πβCRWL, 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 πCRWL with term rewriting. Section 5 begins with the presentation of our combinations of call-time choice and plural semantics that are formalized through the CRWLσπα and CRWLσπβ logics, which correspond to the combination of call-time choice with πCRWL and πβCRWL, respectively. Then follows a user level introduction to our Maude prototype, which implements the CRWLσπα 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 Rodriguez-Hortalá.
(2011), although the intuitions 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 $\Sigma = CS \cup FS$, 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 $\sigma$ 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 $f : \mathcal{A} \to \{true, false\}$, we denote by $a_1, \ldots, a_n | f$ the sequence constructed by taking in order every element from $a_1, \ldots, a_n$ for which $f$ holds. Finally, for any $1 \leq i \leq n$, $(a_1, \ldots, a_n)[i]$ denotes $a_i$.

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

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 $C(e)$ we denote the set of positions in $e$, which is defined as $C(X) = e; C(h(e_1, \ldots, e_n)) = \{e\} \cup \{i.o \mid i \in \{1, \ldots, n\} \land o \in C(e)\}$, where $X \in \mathcal{V}$, $h \in \Sigma$, and $e$ 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 $o \in C(e)$, defined as $e|_o = e; h(e_1, \ldots, e_n)|_i.o = e|_o$. The set of variable positions in $e$ is denoted as $C_{\mathcal{V}}(e)$ and defined as $C_{\mathcal{V}}(e) = \{o \in C(e) \mid e|_o \in \mathcal{V}\}$.

Substitutions $\theta \in \text{Subst}$ are finite mappings $\theta : \mathcal{V} \longrightarrow \text{Exp}$, extending naturally to $\theta : \text{Exp} \longrightarrow \text{Exp}$. We write $e$ for the identity (or empty) substitution. We write $e\theta$ for the application of $\theta$ to $e$, and $\theta\theta'$ for the composition, defined by $e(\theta\theta') = (e\theta)\theta'$. The domain and variable range of $\theta$ are defined as $\text{dom}(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $\text{vran}(\theta) = \bigcup_{X \in \text{dom}(\theta)} \text{var}(X\theta)$. If $\text{dom}(\theta_0) \cap \text{dom}(\theta_1) = 0$, their disjoint union $\theta_0 \cup \theta_1$ is
defined by \((\theta_0 \cup \theta_1)(X) = \theta_i(X)\) if \(X \in \text{dom}(\theta_i)\) for some \(i \in \{0, 1\}\); \((\theta_0 \cup \theta_1)(X) = X\) otherwise. Given \(W \subseteq \mathcal{V}\) we write \(\theta|_W\) for the restriction of \(\theta\) to \(W\), and \(\theta|_{\mathcal{V}\setminus \varphi}\) is a shortcut for \(\theta|_{\mathcal{V}\setminus \varphi\setminus \varphi}\). We will sometimes write \(\theta = \sigma[W]\) instead of \(\theta|_W = \sigma|_W\).

C-substitutions \(\theta \in \text{CSubst} \) verify that \(X\theta \in \text{CTerm}\) for all \(X \in \text{dom}(\theta)\). We say that \(e\) subsumes \(e'\), and write \(e \leq e'\), if \(e\sigma \equiv e'\) for some substitution \(\sigma\).

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 \(f(t_1, \ldots, t_n) \rightarrow r\) where \(f \in \text{FS}^\mathcal{V}\), \(e \in \text{Exp}\), and \((t_1, \ldots, t_n)\) is a linear tuple of \(c\)-terms, where linearity means that variables occur only once in \((t_1, \ldots, t_n)\). Notice that we allow \(r\) to contain extra variables, i.e., variables not occurring in \((t_1, \ldots, t_n)\). To be precise, we say that \(X \in \mathcal{V}\) is an extra variable in the rule \(l \rightarrow r\) iff \(X \in \text{var}(r) \setminus \text{var}(l)\), and by \(v\text{Extra}(R)\) we denote the set of extra variables in a program rule \(R\). For any program \(\mathcal{P}\) the set \(\text{FS}^\mathcal{P}\) of functions defined by \(\mathcal{P}\) is \(\text{FS}^\mathcal{P} = \{f \in \text{FS} \mid \exists(f(\mathcal{P}) \rightarrow r) \in \mathcal{P}\}\). We assume that every program \(\mathcal{P}\) contains the rules \(\{X ? Y \rightarrow X, X ? Y \rightarrow Y, \text{if true then } X \rightarrow X\}\), defining the behavior of the infix function \(? \in \text{FS}^2\) and the mix-fix function \(\text{if then} \in \text{FS}^2\) (used as if \(e_1\) then \(e_2\)), and that those are the only rules for that function symbols. Besides \(?\) is right-associative, so \(e_1 ? e_2 ? e_3 \equiv e_1 \text{ if } e_2 ? e_3\). 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:

\[ C[\sigma] \rightarrow \mathcal{P} C[r\sigma] \text{ for any context } C, \text{ rule } l \rightarrow r \in \mathcal{P} \text{ and } \sigma \in \text{Subst}. \]

We write \(\rightarrow^* \mathcal{P}\) 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 \(\mathcal{P}, e \rightarrow e'\) and \(\mathcal{P}, e \rightarrow^* e'\).

### 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, \(\Sigma\) is enlarged with a new constant constructor symbol \(\bot\). The sets \(\text{Exp}_\bot\), \(\text{CTerm}_\bot\), \(\text{Subst}_\bot\) of partial expressions, etc. are defined naturally. Our contexts will contain partial expressions from now on unless explicitly specified. Expressions, substitutions, etc. not containing \(\bot\) are called total. Programs in CRWL still consist of rewrite rules with total expressions in both sides, so \(\bot\) does not appear in programs. Partial expressions are ordered by the approximation ordering \(\sqsubseteq\) defined as the least partial ordering satisfying \(\sqsubseteq\) and \(e \sqsubseteq e' \Rightarrow C[e] \sqsubseteq C[e']\) for all \(e, e' \in \text{Exp}_\bot, C \in \text{Ctxt}\). This partial ordering can be extended to substitutions: given \(\theta, \sigma \in \text{Subst}_{\bot}\) we say \(\theta \sqsubseteq \sigma\) if \(X\theta \sqsubseteq X\sigma\) for all \(X \notin \mathcal{V}\).

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 \(e \rightarrow t\), with \(e \in \text{Exp}_\bot\) and \(t \in \text{CTerm}_\bot\), 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
(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 \( \theta \in C\text{Subst}_\perp \)), and then reduce the corresponding right-hand side. The use of partial c-substitutions in \( \text{OR} \) is essential to express call-time choice, as only single partial values are used for parameter passing. Notice also that by the effect of \( \theta \) in \( \text{OR} \), 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 \( \mathcal{P} \vdash_{\text{CRWL}} e \rightarrow \triangleleft t \) to express that \( e \rightarrow t \) is derivable in the CRWL-calculus using the program \( \mathcal{P} \). Given a program \( \mathcal{P} \), the CRWL-denotation of an expression \( e \in \text{Exp}_\perp \) is defined as \( [e]_{\mathcal{P}}^{\triangledown} = \{ t \in \text{CTerm}_\perp \mid \mathcal{P} \vdash_{\text{CRWL}} e \rightarrow t \} \). In the following, we will usually omit the reference to \( \mathcal{P} \) 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 \( \text{OR} \) 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 \( C\text{Subst}_\perp^? = \{ \theta \in \text{Subst}_\perp \mid \forall X \in \text{dom}(\theta), \theta(X) = t_1 ? \ldots ? t_n \text{ such that } t_1, \ldots, t_n \in \text{CTerm}_\perp, n > 0 \} \), for which \( C\text{Subst}_\perp \subseteq C\text{Subst}_\perp^? \subseteq \text{Subst}_\perp \) obviously holds. The operator \( ? : C\text{Subst}_\perp^? \rightarrow C\text{Subst}_\perp \) constructs the \( C\text{Subst}_\perp^? \) corresponding to a non-empty sequence of \( C\text{Subst}_\perp \), and it is defined as follows:

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

where \( \rho_1 \ldots \rho_m = \theta_1 \ldots \theta_n \mid \lambda \theta. (X \in \text{dom}(\theta)) \). This operator is overloaded to handle non-empty sets \( \Theta \subseteq C\text{Subst}_\perp \) as \( ?\Theta = ?(\theta_1 \ldots \theta_n) \), where the sequence \( \theta_1 \ldots \theta_n \) corresponds to an arbitrary reordering of the elements of \( \Theta \) – for example using some standard order of terms in the line of Sterling and Shapiro (1986).
Lemma 1
For any $\theta_1, \ldots, \theta_n \in CSubst_{\bot}$, $\text{dom}(\theta_1 \ldots \theta_n) = \bigcup_i \text{dom}(\theta_i)$.

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 $CSubst^\alpha_{\bot}$ instead of $CSubst_{\bot}$ for parameter passing, achieving a plural semantics.$^5$ Besides, extra variables are instantiated by an arbitrary $\theta_e \in CSubst^\alpha_{\bot}$ 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 $\mathcal{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrow t$, 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 $[\![ e ]\!]^{pl}_{\mathcal{P}} = \{ t \in CTerm_{\bot} | \mathcal{P} \vdash_{\pi^{\alpha}CRWL} e \rightarrow t \}$. In the following, we will usually omit the reference to $\mathcal{P}$ and $\pi^{\alpha}pl$, 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 $f(c(0) ? c(1)) \rightarrow d(0,1)$ (some steps have been omitted for the sake of conciseness):

\[
\begin{array}{cccccccccccc}
0 \rightarrow 0 & \text{DC} & \text{DC} & \text{B} & \vdots & \vdots & \vdots & \vdots & \text{POR}$^\alpha$ & \vdots & \vdots & \text{DC} & \vdots & \vdots & \text{POR}$^\alpha$ \\
\hline
0 \rightarrow 0 & \text{DC} & \text{DC} & \text{B} & \vdots & \vdots & \vdots & \vdots & \text{POR}$^\alpha$ & \vdots & \vdots & \text{DC} & \vdots & \vdots & \text{POR}$^\alpha$ \\
\hline
f(c(0) ? c(1)) \rightarrow d(0,1)
\end{array}
\]

$^5$ In fact, angelic non-strict plural non-determinism.
One of the most important properties of $\pi^C$CRWL is compositionality, a property very close to the DET-additivity property for algebraic specifications of Hussmann (1993), or the referential transparency property of Søndergaard and Sestoft (1990). This property shows that the $\pi^C$CRWL-denotation of any expression put in a context only depends on the $\pi^C$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 $\pi^C$CRWL)

For any program, $\mathcal{C} \in \text{Cntxt}$ and $e \in \text{Exp}_\perp$:

$$\llbracket \mathcal{C}[e] \rrbracket^\pi_l = \bigcup_{\{t_1, \ldots, t_n\} \subseteq \llbracket e \rrbracket^\pi_l} \llbracket \mathcal{C}[t_1 \ldots t_n] \rrbracket^\pi_l$$

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

$$\llbracket e \rrbracket^\pi_l = \llbracket e' \rrbracket^\pi_l \text{ iff } \forall \mathcal{C} \in \text{Cntxt}. \llbracket \mathcal{C}[e] \rrbracket^\pi_l = \llbracket \mathcal{C}[e'] \rrbracket^\pi_l$$

Proof

We have to prove that, for any $t \in \text{CTerm}_\perp$, if $\mathcal{C}[e] \rightarrow t$ then $\exists \{s_1, \ldots, s_n\} \subseteq \llbracket e \rrbracket^\pi_l$ such that $\mathcal{C}[s_1 \ldots s_n] \rightarrow t$; and conversely, that given $\{s_1, \ldots, s_n\} \subseteq \llbracket e \rrbracket^\pi_l$ such that $\mathcal{C}[s_1 \ldots s_n] \rightarrow t$ then $\mathcal{C}[e] \rightarrow t$. Each of these statements can be proved by induction on the size of the starting proof (see Riesco and Rodríguez-Hortalá 2011 for details). \qed

Contrary to what happens to call-time choice (López-Fraguas et al. 2008; López-Fraguas et al. available on request), we cannot have a compositional result for single values like $\llbracket \mathcal{C}[e] \rrbracket = \bigcup_{t \in \llbracket e \rrbracket} \llbracket \mathcal{C}[t] \rrbracket$ for any arbitrary context $\mathcal{C}$ because $e$ could appear in a function call when put inside $\mathcal{C}$, and that function might demand more than one value from $e$ because of the plurality of $\pi^C$CRWL. We can see this considering the program from Example 1.1 extended with a function $\text{coin}$ defined by $\{\text{coin} \rightarrow 0, \text{coin} \rightarrow 1\}$, the context $f(c([]))$ and the expression $\text{coin}$: in order to
compute the value \( d(0, 1) \in \llbracket f(c(\text{coin})) \rrbracket \) we need \( \{0, 1\} \subseteq \llbracket \text{coin} \rrbracket \), while a single value of \( \text{coin} \) is not enough, which is reflected in the fact that \( d(0, 1) \notin \llbracket f(c(0 \not\in 1)) \rrbracket \) while \( d(0, 1) \notin \llbracket f(c(0)) \rrbracket \cup \llbracket f(c(1)) \rrbracket \). 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 \( \text{from} \) defined as \( \{\text{from}(X) \to X, \text{from}(X) \to s(\text{from}(X))\} \). Then using the same context as above and the expression \( \text{from}(z) \), in order to compute \( d(z, s(z)) \in \llbracket f(c(\text{from}(z))) \rrbracket \) we just need \( \{z, s(z)\} \subseteq \llbracket \text{from}(z) \rrbracket \), but not the infinite set of elements in \( \llbracket \text{from}(z) \rrbracket \). 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, \( \pi^\text{CRWL} \) enjoys other nice properties like the following polarity property.

**Proposition 1 (Polarity of \( \pi^\text{CRWL} \))**

For any program \( \mathcal{P}, e, e', e'' \in Exp_\bot, t, t', t'' \in CTerm_\bot \) if \( e \sqsubseteq e' \) and \( t'' \sqsubseteq t' \) then \( \mathcal{P} \vdash \pi^\text{CRWL} e \Rightarrow t \) implies \( \mathcal{P} \vdash \pi^\text{CRWL} e' \Rightarrow t' \) with a proof of the same size or smaller.

**Proof**

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

\( \pi^\text{CRWL} \) also has some monotonicity properties related to substitutions. These are formulated using the preorder \( \sqsubseteq_\protect{r} \) over \( CSubst_\bot^\pi \) defined by \( \theta \sqsubseteq_\protect{r} \theta' \iff \forall X \in \nu, \theta(X) = t_1 \ldots t_n \) 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 \( t \sqsubseteq_\protect{r} t' \); and the preorder \( \sqsubseteq^\text{pl}_{\protect{r}} \) over \( Subst_\bot \) defined by \( \sigma \sqsubseteq^\text{pl}_{\protect{r}} \sigma' \iff \forall X \in \nu, \llbracket \sigma(X) \rrbracket^\text{pl} \subseteq \llbracket \sigma'(X) \rrbracket^\text{pl} \).

**Proposition 2 (Monotonicity for substitutions of \( \pi^\text{CRWL} \))**

For any program, \( e, t \in CTerm_\bot, \sigma, \sigma' \in Subst_\bot, \theta, \theta' \in CSubst_\bot^\pi \):

1. **Strong monotonicity of \( Subst_\bot^\pi \):** If \( \forall X \in \nu, s \in CTerm_\bot \) given \( \mathcal{P} \vdash \pi^\text{CRWL} \sigma(X) \Rightarrow s \) with size \( K \) we also have \( \mathcal{P} \vdash \pi^\text{CRWL} \sigma'(X) \Rightarrow s \) with size \( K' \leq K \), then \( \vdash \pi^\text{CRWL} e \sigma \Rightarrow t \) with size \( L \) implies \( \vdash \pi^\text{CRWL} e \sigma' \Rightarrow t \) with size \( L' \leq L \).
2. **Monotonicity of \( CSubst_\bot^\pi \):** If \( \theta, \theta' \in CSubst_\bot^\pi \) and \( \theta \sqsubseteq_\protect{r} \theta' \) then \( \mathcal{P} \vdash \pi^\text{CRWL} e \theta \Rightarrow t \) with size \( K \) implies \( \mathcal{P} \vdash \pi^\text{CRWL} e \theta' \Rightarrow t \) with size \( K' \leq K \).
3. **Monotonicity of \( Subst_\bot^\pi \):** If \( \sigma \sqsubseteq^\text{pl}_{\protect{r}} \sigma' \) then \( \llbracket e \sigma \rrbracket^\text{pl} \subseteq \llbracket e \sigma' \rrbracket^\text{pl} \).
4. **Monotonicity of \( CSubst_\bot^\pi \):** If \( \theta \sqsubseteq_\protect{r} \theta' \) then \( \llbracket e \theta \rrbracket^\text{pl} \subseteq \llbracket e \theta' \rrbracket^\text{pl} \).

The properties of \( \pi^\text{CRWL} \) we have seen so far are shared with \( \text{CRWL} \), which is something natural taking into account that \( \pi^\text{CRWL} \) is a modification of that semantics. Nevertheless, there are some properties of \( \text{CRWL} \) – and as a consequence, of call-time choice – that do not hold for \( \pi^\text{CRWL} \). 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 $C \in \text{Cntxt}$, $e_1, e_2 \in \text{Exp}_\perp$ we have that $\llbracket C[e_1 ? e_2] \rrbracket = \llbracket C[e_1] ? C[e_2] \rrbracket$.” 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 $\pi^\text{CRWL}$.

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

It was very enlightening for us to discover that the correctness of bubbling does not hold for $\pi^\text{CRWL}$, and in fact in Rodríguez-Hortalá (2008) it was wrongly considered as true. This shows that $\text{CRWL}$ and $\pi^\text{CRWL}$ are more different than these may appear at first sight. In particular, regarding bubbling, the important difference is that while $\pi^\text{CRWL}$ is only compositional with respect to subsets of the denotation, $\text{CRWL}$ 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 $\pi^\text{CRWL}$ 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, $cC ::= [ ] | c(e_1, \ldots, cC, \ldots, e_n)$, with $c \in CS^n, e_1, \ldots, e_n \in \text{Exp}_\perp$. For c-contexts, $\pi^\text{CRWL}$ enjoys the same compositionality for single values as $\text{CRWL}$ – that property holds in $\text{CRWL}$ for arbitrary contexts – as shown in the following result.

**Proposition 3 (Compositionality of $\pi^\text{CRWL}$ for c-contexts)**
For any program, c-context $cC$ and $e \in \text{Exp}_\perp$:

$$\llbracket cC[e] \rrbracket_{\pi^\text{pl}} = \bigcup_{t \in \llbracket e \rrbracket_{\pi^\text{pl}}} \llbracket cC[t] \rrbracket_{\pi^\text{pl}}$$

**Proof**
Very similar to the proof for the general compositionality of $\pi^\text{CRWL}$ 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 $\pi^\text{CRWL}$.
Proposition 4 (Bubbling for c-contexts in $\pi^\alpha$CRWL)

For any program, c-context $cE$ and $e_1, e_2 \in \text{Exp}_\perp$, $\llbracket cE[e_1 ? e_2] \rrbracket^{\pi^\alpha} = \llbracket cE[e_1] ? cE[e_2] \rrbracket^{\pi^\alpha}$.

Proof

It is easy to prove that $\forall e_1, e_2 \in \text{Exp}_\perp$ we have $\llbracket e_1 ? e_2 \rrbracket^{\pi^\alpha} = \llbracket e_1 \rrbracket^{\pi^\alpha} \cup \llbracket e_2 \rrbracket^{\pi^\alpha}$ (see Riesco and Rodríguez-Hortalá 2011). But then:

$$\llbracket cE[e_1 ? e_2] \rrbracket^{\pi^\alpha} = \bigcup_{t \in \llbracket e_1 ? e_2 \rrbracket^{\pi^\alpha}} \llbracket cE[t] \rrbracket^{\pi^\alpha}$$

by Proposition 3

$$= \bigcup_{t \in \llbracket e_1 \rrbracket^{\pi^\alpha} \cup \llbracket e_2 \rrbracket^{\pi^\alpha}} \llbracket cE[t] \rrbracket^{\pi^\alpha}$$

$$= \llbracket cE[e_1] \rrbracket^{\pi^\alpha} \cup \llbracket cE[e_2] \rrbracket^{\pi^\alpha}$$

by Proposition 3

$$= \llbracket cE[e_1] ? cE[e_2] \rrbracket^{\pi^\alpha}$$

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 $\text{branches}$ 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 $\text{employees}$, 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 $\text{twoclerks}$, which is based upon the function $\text{find}$, which forces the desired pattern $\text{e}(N,G,\text{clerk})$ over the set defined by the expression $\text{employees}($branches$)$:

$$\mathcal{P} = \{ \text{branches } \to \text{ madrid, }$$

$$\text{branches } \to \text{ vigo, }$$

$$\text{employees(madrid)} \to \text{e(pepe, man, clerk)},$$

$$\text{employees(madrid)} \to \text{e(paco, man, clerk)},$$

$$\text{employees(vigo)} \to \text{e(maria, woman, clerk)},$$

$$\text{employees(vigo)} \to \text{e(jaime, woman, clerk)},$$

$$\text{twoclerks} \to \text{find(employees(branches))},$$

$$\text{find(e(N,G,clerk))} \to (N,N) \}$$

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

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 \(\text{find2NG}\) that forces the desired pattern but now returning both the name and the gender of two clerks, by the rule \(\text{find2NG}((N,G,\text{clerk})) \rightarrow ((N,G),(N,G))\). Then, \(((\text{pepe},\text{man}),(\text{maria},\text{woman}))\) would be one of the values computed for the expression \(\text{find2NG}(\text{employees} (\text{branches}))\), as expected. Nevertheless, we can also compute the value \(((\text{pepe},\text{woman}),(\text{maria},\text{man}))\), which obviously does not correspond to the intended meaning of \(\text{find2NG}\), as can be seen in the following proof (using the abbreviations above and also \(m\) for \(\text{man}\), and \(w\) for \(\text{woman}\)).

\[
\begin{align*}
\text{find2NG}(\text{emps}(\text{brs})) & \rightarrow ((\text{pepe},w),(\text{maria},m)) \\
\text{twoclerks} & \rightarrow (\text{pepe},\text{maria}) \\
\text{DC} & \rightarrow (\text{pepe},\text{maria}) \\
\end{align*}
\]

\[
\begin{align*}
\text{emps}(\text{brs}) & \rightarrow e(\text{pepe},\bot,\text{clerk}) \\
\text{emps}(\text{brs}) & \rightarrow e(\text{maria},\bot,\text{clerk}) \\
\text{POR} & \rightarrow (\text{pepe} ? \text{maria} ? \text{maria}) \rightarrow (\text{pepe},\text{maria}) \\
\text{DC} & \rightarrow (\text{pepe} ? \text{maria} ? \text{maria}) \rightarrow (\text{pepe},\text{maria}) \\
\text{POR} & \rightarrow (\text{pepe} ? \text{maria} ? \text{maria}) \rightarrow (\text{pepe},\text{maria}) \\
\end{align*}
\]

This example is interesting because it shows a relevant flaw of \(\pi^b\text{CRWL}\), since there the matching substitutions \([N/\text{pepe},G/\text{man}]\) and \([N/\text{maria},G/\text{woman}]\) obtained for different evaluations of the argument \(\text{employees}(\text{branches})\) 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,
Singular and plural functions for functional logic programming

thus preventing an unintended information mix-up. Nevertheless, a better semantic behavior would have prevented “mixed” results like \((\text{pepe, woman}), (\text{maria, man})\), thus getting \((\text{maria, woman}), (\text{maria, woman})\) and \((\text{pepe, man}), (\text{pepe, man})\) as the only total values for \(\text{find2NG(employees(branches))}\), 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 \(d(0, 0) \not= d(1, 1)\) and \(d(0 \not= 1, 0 \not= 1)\) – for \(d \in CS^2\) and \(0, 1 \in CS^0\) – made by \(\pi^*\text{CRWL}\) 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 runtime 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 \([c(e_1, \ldots, e'_1 ? e'_2, \ldots, e_n)] = [c(e_1, \ldots, e'_1, \ldots, e_n)]\). 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 \(\text{CRWL}\) or \(\pi^*\text{CRWL}\), which – as we saw in the presentation of \(\pi^*\text{CRWL}\) 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 \(\text{CSubst}^\bot\), 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 \(\theta \sqcup [X/e_1]) \sqcup ([X/e_2]) \rightarrow_{\sqcup} \theta \sqcup [X/e_1 ? e_2]\) that corresponds to a bubbling step for substitutions. Using this we could, for example, perform the following bubbling derivation for substitutions.

\[
[X/0, Y/0] \sqcup [X/0, Y/1] \sqcup [X/1, Y/0] \sqcup [X/1, Y/1] \\
\to_{\sqcup} [X/0, Y/0 ? 1] \sqcup [X/1, Y/0] \sqcup [X/1, Y/1] \\
\to_{\sqcup} [X/0, Y/0 ? 1] \sqcup [X/1, Y/0 ? 1] \to_{\sqcup} [X/0 ? 1, Y/0 ? 1]
\]

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 \([X/0 ? 1, Y/0 ? 1] \in \text{CSubst}^\bot\) 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 \(\to_{\sqcup}\) step as these substitutions differ in more than one variable, thus failing to combine those c-substitutions into a single element from \(\text{CSubst}^\bot\). As can be seen in Figure 2, the key for getting a plural behavior in \(\pi^*\text{CRWL}\) 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 $\pi^\beta{\text{CRWL}}$ to this new semantics in which parameter passing is only performed with substitutions from $CSubst^1_\bot$ that come from a successful combination of c-substitutions using the relation $\rightarrow_\pi$, 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.

\[
\begin{align*}
f(c(X)) & \rightarrow d(X,X) & h(d(X,Y)) & \rightarrow d(X,X) \\
g(d(X,Y)) & \rightarrow l(X,X,Y,Y) & k(d(X,Y)) & \rightarrow d(X,Y)
\end{align*}
\]

- $f(c(0) \ ? c(1))$ and $f(c(0 \ ? 1))$ behave the same in both $\pi^\alpha{\text{CRWL}}$ and $\pi^\beta{\text{CRWL}}$. 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 $d(0,0), d(0,1), d(1,0), \text{and} \ d(1,1)$ 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 $\pi^\alpha{\text{CRWL}}$ and $\pi^\beta{\text{CRWL}}$. 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 $\pi^\alpha{\text{CRWL}}$ 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 $[X/0,Y/\bot]$ and $[X/1,Y/\bot]$ into $[X/0 \ ? 1,Y/\bot]$, which is just what is done in $\pi^\beta{\text{CRWL}}$ 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 $\pi^\alpha{\text{CRWL}}$, 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 $\pi^\beta{\text{CRWL}}$ would be able to combine the former set into the latter $CSubst^1_\bot$.

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 $\pi^\alpha{\text{CRWL}}$ and $\pi^\beta{\text{CRWL}}$. Once again $g(d(0 \ ? 1,0 \ ? 1))$ is not problematic, and for it we can get the values $l(0,0,0,0), l(0,0,0,1), \text{et al.}$ 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, even though $\pi^\alpha_{CRWL}$ only computes 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.

**Definition 1 (Compressible set of CSubst)***

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 $\Theta$ 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 $\rightarrow_{\perp}$ 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:

$$\{(X\theta, Y\theta) \mid \theta \in \{[X/0, Y/0], [X/1, Y/1]\}\} = \{(0, 0), (1, 1)\}$$

$$\neq \{(0, 0), (0, 1), (1, 0), (1, 1)\} = \{0, 1\} \times \{0, 1\}$$

$$= \{X\theta_x \mid \theta_x \in \{[X/0, Y/0], [X/1, Y/1]\}\} \times \{Y\theta_y \mid \theta_y \in \{[X/0, Y/0], [X/1, Y/1]\}\}$$
As we saw in Example 3.4, the function find2NG(employees(branches)) 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. \( \pi^\beta_{\text{CRWL}} \) cannot fix a wrong program, but at least is able to prevent “mixed” results like \(((\text{pepe}, \text{woman}),(\text{maria}, \text{man}))\).

It is also easy to check that \( \pi^\beta_{\text{CRWL}} \) has the same behavior as \( \pi^\alpha_{\text{CRWL}} \) for Example 3.3, as sets like \( \{[N/\text{pepe}, G/\bot], [N/\text{maria}, G/\bot]\} \) are compressible. Similarly, in Example 3.5, functions \( f \) and \( h \) behave the same under both semantics, and \( \pi^\beta_{\text{CRWL}} \) also behaves for \( h \) and \( k \) as specified there, because \( \{[X/0, Y/0], [X/1, Y/1]\} \) is not compressible, just like \( \{[N/\text{pepe}, G/\text{man}], [N/\text{maria}, G/\text{woman}]\} \), while for \( \Theta = \{[X/0, Y/0], [X/0, Y/1], [X/1, Y/0], [X/1, Y/1]\} \) we have that \( \Theta \) 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 \( \Theta \subseteq \text{CSubst}_\bot \) for \( \{X_1, \ldots, X_n\} = \bigcup_{\theta \in \Theta} \text{dom}(\theta) \) 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 \( \Theta \) 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: \( \Theta \) 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 \( \pi^\beta_{\text{CRWL}} \) is smaller than \( \pi^\alpha_{\text{CRWL}} \) 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 \( \pi^\alpha_{\text{CRWL}} \) with no previous compressibility test. We will see more about the relations between call-time choice, run-time choice, \( \pi^\alpha_{\text{CRWL}} \), and \( \pi^\beta_{\text{CRWL}} \) in Section 4.

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

**Theorem 2 (Basic properties of \( \pi^\beta_{\text{CRWL}} \))**

The basic properties of \( \pi^\alpha_{\text{CRWL}} \) also hold for \( \pi^\beta_{\text{CRWL}} \) under any program, i.e., the corresponding versions of Theorem 1 and Propositions 1–4 also hold for \( \pi^\beta_{\text{CRWL}} \).

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

**Proof**

In each proof for the \( \pi^\alpha_{\text{CRWL}} \) versions of these results we start from a given \( \pi^\alpha_{\text{CRWL}} \)-proof and build another one using a bigger expression with respect to \( \subseteq \), a
more powerful substitution, interchanging an expression with an alternative of some of its values. Therefore, we can use the same technique for \( \pi^\beta \text{CRWL} \) to replicate any \( \text{POR}^\beta \) step in the starting \( \pi^\beta \text{CRWL} \)-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 \( \pi^\alpha \text{CRWL} \).

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, \( \pi^\alpha \text{CRWL} \), is a pretty simple extension of \( \text{CRWL} \) 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 \( \pi^\beta \text{CRWL} \), in which a compressibility test is added to prevent wrong combination of substitutions. This could suggest that \( \pi^\alpha \text{CRWL} \) is only a preliminary attempt that should now be put aside and forgotten. Nevertheless, \( \pi^\alpha \text{CRWL} \) will still be very useful for us, again because of its simplicity, as we will see in subsequent sections.

Finally, note that both \( \pi^\alpha \text{CRWL} \) and \( \pi^\beta \text{CRWL} \) have been devised starting from \( \text{CRWL} \) 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 \( \text{CRWL} \), \( \pi^\alpha \text{CRWL} \), and \( \pi^\beta \text{CRWL} \) 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.\footnote{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, \( \pi^C \text{CRWL} \), and \( \pi^R \text{CRWL} \) 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 \(|\bot| = \bot, |X| = X, c(e_1, \ldots, e_n) = c(|e_1|, \ldots, |e_n|), |f(e_1, \ldots, e_n)| = \bot,\) 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:

- \( \llbracket e \rrbracket_{sg} = \{ t \in CTerm\bot \mid \mathcal{P} \vdash_{\text{CRWL}} e \rightarrow t \} \)
- \( \llbracket e \rrbracket_{pl} = \{ t \in CTerm\bot \mid \mathcal{P} \vdash e \rightarrow^* t \subseteq |e'| \} \)
- \( \llbracket e \rrbracket_{\pi^C} = \{ t \in CTerm\bot \mid \mathcal{P} \vdash_{\pi^C} e \rightarrow t \} \)
- \( \llbracket e \rrbracket_{\pi^R} = \{ t \in CTerm\bot \mid \mathcal{P} \vdash_{\pi^R} e \rightarrow t \} \)

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

As \( \pi^C \text{CRWL} \) and \( \pi^R \text{CRWL} \) are modifications of CRWL, the relation between these three semantics is straightforward.

**Theorem 3**

For any CRWL-program \( \mathcal{P}, e \in Exp\bot \)

\[
\llbracket e \rrbracket_{sg} \subseteq \llbracket e \rrbracket_{pl} \subseteq \llbracket e \rrbracket_{\pi^C}
\]

None of the converse inclusions holds in general.

**Proof**

Given a CRWL-proof for \( \vdash_{\text{CRWL}} e \rightarrow t \) we can build a \( \pi^C \text{CRWL} \)-proof for \( \vdash_{\pi^C} e \rightarrow t \) 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 \( \llbracket e \rrbracket_{sg} \subseteq \llbracket e \rrbracket_{pl} \). On the other hand, we can turn any \( \pi^R \text{CRWL} \)-proof into a \( \pi^C \text{CRWL} \)-proof just replacing any POR \( \alpha \) step by the corresponding POR \( \beta \), as POR \( \beta \) has stronger premises than POR \( \alpha \), and the same consequence. Therefore \( \llbracket e \rrbracket_{pl} \subseteq \llbracket e \rrbracket_{\pi^C} \).

Regarding the failure of the converse inclusions in the general case, consider the program \( \{ \text{pair}(X) \rightarrow d(X, X), g(d(X, Y)) \rightarrow d(X, Y) \} \) for which it is easy to check that \( \llbracket \text{pair}(0?1) \rrbracket_{sg} \not\supseteq d(0, 1) \in \llbracket \text{pair}(0?1) \rrbracket_{pl} \) and \( \llbracket g(d(0, 0)?d(1, 1)) \rrbracket_{pl} \not\supseteq d(0, 1) \in \llbracket g(d(0, 0)?d(1, 1)) \rrbracket_{\pi^C} \).

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 \( P \), \( e \in \text{Exp} \), \( \llbracket e \rrbracket^r \subseteq \llbracket e \rrbracket^t \). 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 \( \pi^\beta\text{CRWL} \) 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 \( \pi^\beta\text{CRWL} \) computes more values in general.

Theorem 5
For any CRWL-program \( P \), \( e \in \text{Exp} \), \( \llbracket e \rrbracket^t \subseteq \llbracket e \rrbracket^\beta \). 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 \( \pi^\beta\text{CRWL} \).

Lemma 3 (One-step soundness of \( \rightarrow \) with respect to \( \pi^\beta\text{CRWL} \))
For any CRWL-program \( P \), \( e, e' \in \text{Exp} \) if \( e \rightarrow e' \) then \( \llbracket e' \rrbracket^\beta \subseteq \llbracket e \rrbracket^\beta \).

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

Definition 3 (Denotation of substitutions)
For any CRWL-program \( P \), \( \sigma \in \text{Subst} \) the \( \pi^\beta\text{CRWL} \)-denotation of \( \sigma \) under \( P \) is
\[
\llbracket \sigma \rrbracket^\beta_P = \{ \emptyset \in \mathbb{CSubst} \mid \forall X \in \mathcal{V}, \ P \vdash \pi^{\beta\text{CRWL}} \sigma(X) \rightarrow \emptyset(X) \}
\]

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

Lemma 4
For any finite not empty \( \Theta \subseteq \llbracket \sigma \rrbracket^\beta \) we have \( ?\Theta \leq^\beta \sigma \).

Besides, it is clear that in any \( \pi^\beta\text{CRWL} \)-proof that uses some \( \sigma \in \text{Subst} \) only a finite amount of information is contained in \( \sigma \). Therefore, in \( \llbracket \sigma \rrbracket^\beta \) is employed, just like in any proof for a statement \( \vdash \pi^\beta\text{CRWL} \ e \rightarrow t \) 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 $\vdash \pi^\beta_{CRWL} e \rightarrow t$ could not have been proved. These intuitions are formalized in the following result.

**Lemma 5**

For any $\sigma \in \text{Subst}_\bot, e \in \text{Exp}_\bot, t \in \text{CTerm}_\bot$ if $\vdash \pi^\beta_{CRWL} e\sigma \rightarrow t$ then $\exists \Theta \subseteq [\sigma]^{\beta\bot}$ finite and not empty such that $\vdash \pi^\beta_{CRWL} e(?\Theta) \rightarrow t$

**Proof (sketch)**

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

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

Otherwise if $X \notin \text{dom}(\sigma)$ then given $\bar{Y} = \text{dom}(\sigma)$ we define $\theta = [Y/\bot]$. 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 $e\sigma \rightarrow t$, using the property that for any $\Theta, \Theta' \subseteq \text{CSubst}_\bot$, if $\Theta \subseteq \Theta'$ then $?\Theta \subseteq (?\Theta')$, combined with the monotonicity under substitutions of $\pi^\beta_{CRWL}$ (see Riesco and Rodríguez-Hortalá 2011 for details). □

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 $\text{CSubst}_\bot$, 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 $\vdash_{CRWL} d(X,X)[X/0\ ?\ 1] \rightarrow d(0,1)$ for which there is no substitution $\theta$ in $\text{CSubst}_\bot$ — the kind of substitutions used for parameter passing in $CRWL$ — such that $\vdash_{CRWL} d(X,X)\theta \rightarrow d(0,1)$. 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 $\text{CSubst}_\bot$ 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 $\Theta \subseteq C\text{Subst}_\bot$ finite such that $\{X_1, \ldots, X_n\} = \bigcup_{\theta \in \Theta} \text{dom}(\theta)$, 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 $\Theta \subseteq C\text{Subst}_\bot$ finite such that $\{X_1, \ldots, X_n\} = \bigcup_{\theta \in \Theta} \text{dom}(\theta)$

(a) $cc(\Theta) \subseteq C\text{Subst}_\bot$ and it is finite.
(b) $\Theta \subseteq cc(\Theta)$. As a result, $?\Theta \subseteq cc(\Theta)$.
(c) $\bigcup_{\mu \in cc(\Theta)} \text{dom}(\mu) = \{X_1, \ldots, X_n\}$.
(d) $cc(\Theta)$ is compressible.

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

**Lemma 6**
For any $\sigma \in \text{Subst}_\bot$ and any $\Theta \subseteq [\sigma]^{\beta \pi}$ finite and not empty we have that $cc(\Theta) \subseteq [\sigma]^{\beta \pi}$ too.

This is precisely the result we need to strengthen Lemma 5, so it now becomes applicable for $\pi^{\beta \text{CRWL}}$, as it allows us to shift from any subset of the $\pi^{\beta \text{CRWL}}$-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 $\sigma \in \text{Subst}_\bot$, $e \in \text{Exp}_\bot$, $t \in C\text{Term}_\bot$ if $\vdash \pi^{\beta \text{CRWL}} e\sigma \rightarrow t$ then $\exists \Theta \subseteq [\sigma]^{\beta \pi}$ finite, not empty, and compressible such that $\vdash \pi^{\beta \text{CRWL}} e(?\Theta) \rightarrow t$.

**Proof**
By Lemma 5 we get some $\Theta \subseteq [\sigma]^{\beta \pi}$ finite and not empty such that $\vdash \pi^{\beta \text{CRWL}} e(?\Theta) \rightarrow t$. Then by Lemma 6 we get that $cc(\Theta) \subseteq [\sigma]^{\beta \pi}$ too, and that it is finite, not empty (as $\Theta \subseteq cc(\Theta)$ and $\Theta$ is not empty), compressible and $?\Theta \subseteq cc(\Theta)$ by Proposition 5. But then we can apply the monotonicity of Theorem 2 to get $\vdash \pi^{\beta \text{CRWL}} e(?cc(\Theta)) \rightarrow t$, 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 $\pi^{\beta \text{CRWL}}$) for steps performed at the root of the expression, i.e., of the shape $f(p)\sigma \rightarrow r\sigma$. Thus, given some $t \in [r\sigma]^{\beta \pi}$, our goal is proving that $t \in [f(p)\sigma]^{\beta \pi}$. First of all by Lemma 7 we get some compressible $\Theta \subseteq [\sigma]^{\beta \pi}$ such that $t \in [r(?\Theta)]^{\beta \pi}$. If we could use it to prove that $t \in [f(p)(?\Theta)]^{\beta \pi}$ then by Lemma 4 we would get $?\Theta \subseteq [p]^{\beta \pi} \sigma$, so by the monotonicity of Theorem 2 we could obtain $t \in [f(p)\sigma]^{\beta \pi}$ as we wanted. As $p \in C\text{Term}_\bot$ and $\Theta \subseteq C\text{Subst}_\bot$ we can easily prove that $\forall p_i \in p, \theta_j \in \Theta$ we have $\vdash \pi^{\beta \text{CRWL}} p_i(?\Theta) \rightarrow p_i\theta_j$. All this can be used to perform the following step, assuming $\Theta = \{\theta_1, \ldots, \theta_m\}$.
for $\theta' = (\{i\}?\Theta_i) \cup \theta_e$ where $\forall i \in \{1, \ldots, n\}, \Theta_i = \{\theta_j|_{\text{var}(p_i)} : \theta_j \in \Theta\}, \theta_e = (?\Theta)|_{\forall^c}$ for $\forall^c = v\text{Extra}(f(p) \rightarrow r)$. It can be easily proved that having $\Theta$ compressible implies that each $\Theta_i$ is also compressible – so the $\text{POR}^\beta$ step above is valid – and that $r\theta' \equiv r(?\Theta)$.

Therefore, we have just proved the soundness with respect to $\pi^\beta\text{CRWL}$ of term rewriting steps performed at the root of the starting expression. So, all that is left is using the compositionality of $\pi^\beta\text{CRWL}$ 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 $t \in \llbracket e \rrbracket^\star$, by definition $\exists e' \in \text{Exp}$ such that $t \sqsubseteq |e'|$ and $e \rightarrow^\ast e'$. We can extend Lemma 3 to $\rightarrow^\ast$ by a simple induction on the length of $e \rightarrow^* e'$, hence $\llbracket e' \rrbracket^\beta \subseteq \llbracket e \rrbracket^\beta$. As $\forall e \in \text{Exp}_1, |e| \in \llbracket e \rrbracket^\beta$ (by a simple induction on the structure of $e$), then $t \sqsubseteq |e'| \in \llbracket e' \rrbracket^\beta \subseteq \llbracket e \rrbracket^\beta$, hence $t \in \llbracket e \rrbracket^\beta$ by the polarity of Theorem 2. Example 1.1 shows that the converse inclusion does not hold in general. □

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

**Corollary 4.1**

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

$$\llbracket e \rrbracket^\text{sg} \subseteq \llbracket e \rrbracket^\ast \subseteq \llbracket e \rrbracket^\beta \subseteq \llbracket e \rrbracket^\varphi$$

Hence for any $t \in \text{CTerm}$, $\mathcal{P} \vdash_{\text{CRWL}} e \rightarrow t$ implies $\mathcal{P} \vdash e \rightarrow^\ast t$, which implies $\mathcal{P} \vdash \pi^\beta\text{CRWL} e \rightarrow t$, which implies $\mathcal{P} \vdash \pi^\ast\text{CRWL} e \rightarrow t$.

**Proof**

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

Concerning the second part, assume $\vdash_{\text{CRWL}} e \rightarrow t$, in other words, $t \in \llbracket e \rrbracket^\ast$. Then by the first part $t \in \llbracket e \rrbracket^\ast$, hence $e \rightarrow^* e'$ such that $t \sqsubseteq |e'|$. But as $t \in \text{CTerm}$, 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, $e \rightarrow^\ast e' \equiv t \in \text{CTerm}$, which implies $t \in \llbracket e \rrbracket^\ast$ 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 $t \in \llbracket e \rrbracket^\beta \subseteq \llbracket e \rrbracket^\varphi$ by the first part, and so both $\vdash \pi^\beta\text{CRWL} e \rightarrow t$ and $\vdash \pi^\ast\text{CRWL} e \rightarrow t$. □
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 $\llbracket e \rrbracket^{\beta}_{pl} \subseteq \llbracket e \rrbracket^{\alpha}_{pl}$ for any expression and program, therefore we just have to find a class of programs such that $\llbracket e \rrbracket^{\alpha}_{pl} \subseteq \llbracket e \rrbracket^{\beta}_{pl}$ 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 $[X/0?1,Y/\bot]$ that can be obtained from combining the compressible set $\{[X/0,Y/\bot],[X/1,Y/\bot]\}$. 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 $[X/0,Y/\bot] \sqsubseteq [X/0,Y/0]$ and $[X/1,Y/\bot] \sqsubseteq [X/1,Y/1]$.

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 $\bot$ the value obtained for the other variables in the matching substitution, thus getting a smaller – with respect 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 $\text{POR}^{\alpha}$ step using a possibly non-compressible substitutions into a $\text{POR}^{\beta}$ step using a compressible substitutions that would be then a valid $\text{POR}^{\beta}$ step as well. Note that in this case extra variables pose no problem, as the only difference between $\text{POR}^{\alpha}$ and $\text{POR}^{\beta}$ 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 substitutes, they always behave the same under both $\pi^{\alpha}CRWL$ and $\pi^{\beta}CRWL$.

In the following definition we formally define the class $\mathcal{C}^{\alpha\beta}$ of programs in which the ideas above are materialized.
Definition 5 (Class of programs $\mathcal{C}^{\alpha\beta}$)

The class of programs $\mathcal{C}^{\alpha\beta}$ is defined by

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

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 $\mathcal{C}^{\alpha\beta}$: 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, $\pi^\alpha_{\text{CRWL}}$, and $\pi^\beta_{\text{CRWL}}$ 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, $\pi^\alpha_{\text{CRWL}}$, and $\pi^\beta_{\text{CRWL}}$ behave the same – CRWL remains as the smaller semantics in this case: just consider the program $\{\text{pair}(X) \rightarrow d(X,X)\}$ and the expression $\text{pair}(0 ? 1)$ for which $d(0,1)$ cannot be computed by CRWL, but it can be by any of the other three semantics.

Anyway, the class $\mathcal{C}^{\alpha\beta}$ 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 $\pi^\alpha_{\text{CRWL}}$ and $\pi^\beta_{\text{CRWL}}$ for programs in the class $\mathcal{C}^{\alpha\beta}$.

Theorem 6 (Equivalence of $\pi^\alpha_{\text{CRWL}}$ and $\pi^\beta_{\text{CRWL}}$ for the class $\mathcal{C}^{\alpha\beta}$)

For any program $\mathcal{P} \in \mathcal{C}^{\alpha\beta}$, $e \in \text{Exp}_{\bot}$,

$$\llbracket e \rrbracket_{\mathcal{P}} = \llbracket e \rrbracket_{\mathcal{P}^\beta}$$

This equivalence between $\pi^\alpha_{\text{CRWL}}$ and $\pi^\beta_{\text{CRWL}}$ will be very useful for us for several reasons. First of all, as we will see in Section 4.3, $\pi^\alpha_{\text{CRWL}}$ can be simulated by term rewriting through a simple program transformation, which implies that the same transformation can be used to simulate $\pi^\beta_{\text{CRWL}}$ for the class of programs $\mathcal{C}^{\alpha\beta}$, thanks to the equivalence from Theorem 6. On the other hand, the class $\mathcal{C}^{\alpha\beta}$ is defined by a simple syntactic criterion, which allows its application to mechanized program analysis. Finally, this equivalence grows in importance after realizing that the class $\mathcal{C}^{\alpha\beta}$ 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 $\mathcal{C}^{\alpha\beta}$.

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, $\pi^\alpha_{\text{CRWL}}$ 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 $\pi^\alpha_{\text{CRWL}}$. 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 \( \text{var}(r) \subseteq \text{var}(l) \) 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 \( \mathcal{P} = \{ f(c(X)) \rightarrow d(X,X) \} \) from Example 1.1, which results in the following program \( \hat{\mathcal{P}} \).

\[
\hat{\mathcal{P}} = \{ f(Y) \rightarrow \text{if match}(Y) \text{ then } d(\text{project}(Y),\text{project}(Y)), \text{match}(c(X)) \rightarrow \text{true}, \text{project}(c(X)) \rightarrow X \}
\]

In the resulting program \( \hat{\mathcal{P}} \) 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, \( \text{match} \) and \( \text{project} \). As a consequence, when we evaluate by term rewriting under \( \hat{\mathcal{P}} \) 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

\[
f(c(0) \ ? \ c(1)) \rightarrow \text{if } \text{match}(c(0) \ ? \ c(1)) \text{ then } d(\text{project}(c(0) \ ? \ c(1)),\text{project}(c(0) \ ? \ c(1)))
\]

is sound, thus replicating the argument of \( f \) freely without demanding any evaluation, this way keeping its \( \pi\text{CRWL} \)-denotation untouched: this is the key to achieve completeness with respect to \( \pi\text{CRWL} \). Note that the guard \( \text{if } \text{match}(c(0) \ ? \ c(1)) \) 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) → true” for defining an emptiness test for the classical representation of lists in functional programming, we would get “null(Y) → true,” which is clearly unsound because it allows us to rewrite \( \text{null(cons(0,nil))} \) into \( \text{true} \). Later on, after resolving the guard, different evaluations of the occurrences of \( \text{project}(c(0) \ ? \ c(1)) \) 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 \( \pi\text{CRWL} \).
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 \( \text{match} \) and \( \text{project} \) rules for the replacement.

**Definition 6** (\( \pi^CRWL \) to term rewriting transformation, simple version)

Given a program \( \mathcal{P} \), our transformation proceeds rule by rule. For every program rule \( (f(p_1,\ldots,p_n) \rightarrow r) \in \mathcal{P} \) such that \( f \notin \{?,\text{if then}\} \) we define its transformation as follows:

\[
pST(f(p_1,\ldots,p_n) \rightarrow r) = f(Y_1,\ldots,Y_n) \rightarrow \text{match}(Y_1,\ldots,Y_n) \text{ then } r[X_{ij}/\text{project}_{ij}(Y_i)]
\]

where

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

For \( f \notin \{?,\text{if then}\} \) 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:

\[
\begin{align*}
f(c(0)?c(1)) & \rightarrow \text{match}(c(0)?c(1)) \text{ then } d(\text{project}(c(0)?c(1)),\text{project}(c(0)?c(1))) \\
& \rightarrow^* \text{if true then } d(\text{project}(c(0)?c(1)),\text{project}(c(0)?c(1))) \\
& \rightarrow d(\text{project}(c(0)?c(1)),\text{project}(c(0)?c(1))) \rightarrow^* d(c(0)), d(c(1)) \rightarrow^* d(0,1)
\end{align*}
\]

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 \( \pi^CRWL \) 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 \( \pi^CRWL \) under the original program.

**Theorem 7**

For any \( \text{CRWL}\)-program \( \mathcal{P} \), and any \( e \in \text{Exp}_{\bot} \) built up on the signature of \( \mathcal{P} \), we have

\[
\llbracket e \rrbracket_{pST(\_\_)}^{\pi^l} \subseteq \llbracket e \rrbracket_{\mathcal{P}}^{\pi^l}
\]

As a consequence \( \llbracket e \rrbracket_{pST(\mathcal{P})}^{\pi^l} \subseteq \llbracket e \rrbracket_{\mathcal{P}}^{\pi^l} \).

**Proof (sketch)**

The first part states the soundness within \( \pi^CRWL \) of the transformation. Assuming \( \pi^CRWL\)-proof for statement \( pST(\mathcal{P}) \vdash \pi^CRWL e \rightarrow t \) for some \( t \in \text{CTerm}_\bot \), we can then build another \( \pi^CRWL\)-proof for \( \mathcal{P} \vdash \pi^CRWL e \rightarrow t \) by induction on the size of the starting proof – measured as the number of rules of \( \pi^CRWL \) 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 \( \llbracket e \rrbracket_{pST(\mathcal{P})}^{\pi^l} \subseteq \llbracket e \rrbracket_{pST(\mathcal{P})}^{\pi^l} \subseteq \llbracket e \rrbracket_{\mathcal{P}}^{\pi^l} \).
by term rewriting under the transformed program any value computed for that expression by $\pi^\ast CRWL$ under the original program.

Theorem 8
For any CRWL-program $P$, and any $e \in Exp, t \in CTerm_\bot$ built up on the signature of $P$, if $P \vdash_{\pi^\ast CRWL} e \rightarrow t$ then exists some $e' \in Exp$ built using symbols of the signature of $pST(P)$ such that $pST(P) \vdash e \rightarrow^* e'$ and $t \sqsubseteq |e'|$. In other words, $\llbracket e \rrbracket_{P} \subseteq \llbracket e \rrbracket_{pST(P)}$.

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

Then for any $e \in Exp_\bot, t \in CTerm_\bot$ constructed using just symbols in the signature of $P \cup \mathcal{M}$ we have $P \cup \mathcal{M} \vdash_{\pi^\ast CRWL} e \rightarrow t$ implies $\hat{P} \cup \mathcal{M} \vdash e \rightarrow^* e'$ 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 $\hat{P} \cup \mathcal{M} = pST(P)$ be, where $\mathcal{M}$ is the set containing the rules for the new functions match and project, and $\hat{P}$ contains the new versions of the original rules of $P$.

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

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

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

$$\llbracket e \rrbracket_{P} = \llbracket e \rrbracket_{pST(P)}$$

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

Proof
The fist part holds by a combination of Theorems 7 and 8.

For the second part if $P \vdash_{\pi^\ast CRWL} e \rightarrow t$ then $t \in \llbracket e \rrbracket_{P} = \llbracket e \rrbracket_{pST(P)}$ by the first part, hence $\exists e' \in Exp$ such that $pST(P) \vdash e \rightarrow^* e'$ 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 $pST(\mathcal{P}) \vdash e \rightarrow^* e' \equiv t$. On the other hand, if $pST(\mathcal{P}) \vdash e \rightarrow^* t$ then as $t \sqsubseteq t \equiv |t|$ (again because $t$ is a total c-term) we have $t \in [e]_{pST(\mathcal{P})}^{\pi\alpha} = [e]_{pST(\mathcal{P})}^{\pi\gamma}$, and so $\mathcal{P} \vdash \pi\gamma\text{CRWL} e \rightarrow t$. □

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

**Corollary 4.3 (Restricted adequacy of $pST(\_)$ for simulating $\pi\beta\text{CRWL}$)**

For any program $\mathcal{P} \in \mathcal{C}^{\alpha\beta}$, $e \in \text{Exp}$ is built using symbols of the signature of $\mathcal{P}$

$$[e]_{pST(\mathcal{P})}^{\pi\beta} = [e]_{pST(\mathcal{P})}^{\pi\gamma}$$

Hence, $\forall t \in \text{CTerm}$ we have that $\mathcal{P} \vdash \pi\beta\text{CRWL} e \rightarrow t$ iff $pST(\mathcal{P}) \vdash e \rightarrow^* t$.

**Proof**

A straightforward combination of Corollary 4.2 and Theorem 6. □

This last result illustrates the interest of $\pi\gamma\text{CRWL}$. Because of its simplicity, $\pi\gamma\text{CRWL}$ 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 $\pi\gamma\text{CRWL}$. Besides, thanks to the restricted equivalence between $\pi\gamma\text{CRWL}$ and $\pi\beta\text{CRWL}$, which would also be an implementation of $\pi\beta\text{CRWL}$ for the class $\mathcal{C}^{\alpha\beta}$, and the membership check of program to the class $\mathcal{C}^{\alpha\beta}$ could be also mechanized because $\mathcal{C}^{\alpha\beta}$ 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 $\mathcal{C}^{\alpha\beta}$ is illustrated.

### 4.3.2 An optimized transformation

As we have already mentioned in our comments after presenting the class $\mathcal{C}^{\alpha\beta}$ 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 $\text{null}(\text{nil}) \rightarrow \text{true}$, the transformation returns the rules $\{\text{null}(Y) \rightarrow \text{if match}(Y) \text{ then true, match}(\text{nil}) \rightarrow \text{true}\}$, 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 $\text{pair}(X) \rightarrow d(X, X)$ for which $\{\text{pair}(Y) \rightarrow \text{if match}(Y) \text{ then } d(\text{project}(Y), \text{project}(Y)), \text{match}(X) \rightarrow \text{true, project}(X) \rightarrow X\}$ 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** ($\pi^a$CRWL to term rewriting transformation, optimized version)

Given a program $\mathcal{P}$, our transformation proceeds rule by rule. For every program rule $(f(p_1, \ldots, p_n) \rightarrow r) \in \mathcal{P}$ we define its transformation as follows:

\[
p_{ST}(f(p_1, \ldots, p_n) \rightarrow r) = \begin{cases} 
  f(p_1, \ldots, p_n) \rightarrow r & \text{if } m = 0 \\
  f(\tau(p_1), \ldots, \tau(p_n)) \rightarrow \text{match}(Y_1, \ldots, Y_m) \\
  \text{then } r[X_{ij}/\text{project}_{ij}(Y_i)] & \text{otherwise}
\end{cases}
\]

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p.(p \notin \mathcal{F} \land \text{var}(p) \neq \emptyset)$.

- $\forall \rho_i, \{X_{i1}, \ldots, X_{ik_i}\} = \text{var}(\rho_i) \cap \text{var}(r)$ and $Y_i \in \mathcal{F}$ is fresh.
- $\tau : \mathcal{CTerm} \rightarrow \mathcal{CTerm}$ is defined by $\tau(p) = p$ if $p \notin \{\rho_1, \ldots, \rho_m\}$; otherwise $\tau(\rho_i) = Y_i$.
- $\text{match} \in \mathcal{FS}^m$ fresh is defined by the rule $\text{match}(\rho_1, \ldots, \rho_m) \rightarrow \text{true}$.
- Each $\text{project}_{ij} \in \mathcal{FS}^1$ is a fresh symbol defined by the rule $\text{project}_{ij}(\rho_i) \rightarrow X_{ij}$.

Note that this transformation is well defined because each $\rho_i \in \{\rho_1, \ldots, \rho_m\}$ contains at least one variable, and so it can be distinguished from any other $\rho_j$ by using syntactic equality, thanks to left linearity of program rules, therefore $\tau$ 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 $?a$ and $\text{if then}$ 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 $\pi^a$CRWL.

**Example 4.1**

The only rule modified is the one for $\text{find}$, for which we get the following program:

\[
\{\text{find}(Y) \rightarrow \text{if } \text{match}(Y) \text{ then } (\text{project}(Y), \text{project}(Y)), \\
\text{match}(e(N,G,\text{clerk})) \rightarrow \text{true}, \\
\text{project}(e(N,G,\text{clerk})) \rightarrow N\}
\]

under which we can perform this term rewriting derivation for $\text{twoclerks}$

\[
\begin{align*}
\text{twoclerks} & \rightarrow \text{find}(\text{employees}(\text{branches})) \\
& \rightarrow \text{if } \text{match}(\text{employees}(\text{branches})) \\
& \text{then } (\text{project}(\text{employees}(\text{branches})), \text{project}(\text{employees}(\text{branches}))) \\
& \rightarrow^* \text{if } \text{match}(e(\text{pepe}, \text{man}, \text{clerk})) \\
& \text{then } (\text{project}(\text{employees}(\text{branches})), \text{project}(\text{employees}(\text{branches}))) \\
& \rightarrow^* (\text{project}(\text{employees}(\text{branches})), \text{project}(\text{employees}(\text{branches}))) \\
& \rightarrow^* (\text{project}(e(\text{pepe}, \text{man}, \text{clerk})), \text{project}(e(\text{maria}, \text{woman}, \text{clerk}))) \\
& \rightarrow^* (\text{pepe}, \text{maria})
\end{align*}
\]
5 Programming with singular and plural functions

So far we have presented two novel proposals for the semantics of lazy non-deterministic 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 \url{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 \textsc{CRWL} called \textsc{CRWL}$_{\alpha\pi}$ and \textsc{CRWL}$_{\alpha\beta}$, 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 \textsc{CRWL}$_{\alpha\pi}$, and so \textsc{CRWL}$_{\alpha\beta}$ is only supported for programs in the class $\mathcal{C}_{\alpha\beta}$ 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.
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\(_{\pi^n}\) and CRWL\(_{\pi^n,\theta}\)

We assume a mapping \(\text{plurality} : \text{FS} \rightarrow \{\text{sg}, \text{pl}\}^*\) called \textit{plurality map} such that, for every \(f \in \text{FS}^n\), \(\text{plurality}(f) = b_1 \ldots b_n\) sets its plurality behavior: if \(b_i = \text{sg}\) then the \(i\)th argument of \(f\) will be interpreted with a singular semantics, otherwise it will be interpreted under a plural semantics. In this line \(\text{sgArgs}(f) = \{i \in \{1, \ldots, \text{ar}(f)\} \mid \text{plurality}(f)[i] = \text{sg}\}\) and \(\text{plArgs}(f) = \{i \in \{1, \ldots, \text{ar}(f)\} \mid \text{plurality}(f)[i] = \text{pl}\}\) are the sets of singular and plural arguments of some \(f \in \text{FS}\).

In particular we say that \(f\) is a \textit{singular function} if \(\text{sgArgs}(f) = \{1, \ldots, \text{ar}(f)\}\) and that it is a \textit{plural function} when \(\text{plArgs}(f) = \{1, \ldots, \text{ar}(f)\}\). A related notion is that of singular and plural variables of a pattern: \(\text{sgVars}(f(\overline{\theta})) = \bigcup \{i \in \text{sgArgs}(f) \mid \text{var}(p_i)\}\) and \(\text{plVars}(f(\overline{\theta})) = \bigcup \{i \in \text{plArgs}(f) \mid \text{var}(p_i)\}\).

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 \(\text{CRWL}_{\pi^n}\) and \(\text{CRWL}_{\pi^n,\theta}\) as the result of taking the rules of \(\text{CRWL}\) and replacing the rule \(\text{OR}\) by either the rule \(\text{OR}_{\pi^n}\) or rule \(\text{OR}_{\pi^n,\theta}\) from Figure 4. As any variant of \(\text{CRWL}\), these calculi derive statement forms of the form \(\mathcal{P} \vdash \text{CRWL}_{\pi^n}\) \(e \rightarrow t\) and \(\mathcal{P} \vdash \text{CRWL}_{\pi^n,\theta}\) \(e \rightarrow t\) that express that \(t\) is (or approximates to) a possible value for \(e\) in \(\text{CRWL}_{\pi^n}\) or \(\text{CRWL}_{\pi^n,\theta}\), respectively, under the program \(\mathcal{P}\). The denotations \(\llbracket e \rrbracket_{\pi^n,\theta}\) and \(\llbracket e \rrbracket_{\pi^n,\theta}\) established by these semantics are defined as usual – see Definition 2.

![Figure 4](image-url). The rules \(\text{OR}_{\pi^n}\) and \(\text{OR}_{\pi^n,\theta}\).
Just like in \(\pi^\alpha\text{CRWL}\) and \(\pi^\beta\text{CRWL}\), 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 \(\text{OR}^\alpha\), and \(\text{OR}^\beta\), corresponding to \(\text{POR}^\alpha\) and \(\text{POR}^\beta\) 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 \(\text{CSubst}_\bot\) (as obviously \(?\{\theta\} = \theta\) ), and achieving a singular behavior (call-time choice).

**Example 5.1**

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

\[
\begin{array}{c}
\text{(1)} \\
\vdots \\
\text{(2)} \\
\vdots \\
\vdots \\
\text{(3)} \\
\end{array}
\]

\[
\begin{array}{c}
\text{OR}^\alpha
\end{array}
\]

where (1) is the following proof:

\[
\begin{array}{c}
\text{OR}^\alpha
\end{array}
\]

Note that \(d(0, 1, 0, 1)\) is not a correct value for the expression \(f(0 \ ? 1, c(0) \ ? c(1))\) under \(\text{CRWL}^\alpha\), 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 \(\mathcal{C}_{\alpha\beta}\), it behaves the same under \(\pi^\alpha\text{CRWL}\) and \(\pi^\beta\text{CRWL}\), and therefore also under \(\text{CRWL}^\alpha\), and \(\text{CRWL}^\beta\), so the previous proof and comments also hold for \(\text{CRWL}^\alpha\) and \(\text{CRWL}^\beta\).

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:

\[
\begin{array}{c}
\text{OR}^\alpha
\end{array}
\]

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 \(\text{CRWL}^\alpha\) and \(\text{CRWL}^\beta\) under the given plurality map. Nevertheless, its first two arguments can be different, contrary to what happens under \(\text{CRWL}^\alpha\) and \(\text{CRWL}^\beta\). In conclusion, it
36

A. Riesco and J. Rodríguez-Hortalá

is easy to define a program and a plurality map for them such that neither CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\alpha nor CRWL\textsubscript{\sigma}\,\textsubscript{\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\textsubscript{\sigma}\,\textsubscript{\pi}\alpha and CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\beta inherit the properties of \pi\alpha\,\textsubscript{CRWL} from Section 3.1, for the same reason \pi\beta\,\textsubscript{CRWL} inherits the properties of \pi\alpha\,\textsubscript{CRWL}. The most important among these properties is their compositionality, which expresses the value-based philosophy underlying CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\alpha and CRWL\textsubscript{\sigma}\,\textsubscript{\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\textsubscript{\sigma}\,\textsubscript{\pi}\alpha and CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\beta, just like it happens for \pi\alpha\,\textsubscript{CRWL} and \pi\beta\,\textsubscript{CRWL}: in fact Example 3.2 can be reused to prove it. Nevertheless, just like for \pi\alpha\,\textsubscript{CRWL} and \pi\beta\,\textsubscript{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\textsubscript{\sigma}, which are contexts whose holes appear only under a nested application of constructor symbols or singular function arguments: s\textsubscript{\sigma} ::= [ ] | c(e\textsubscript{1},...,s\textsubscript{\sigma},...,e\textsubscript{n}) | f(e\textsubscript{1},...,s\textsubscript{\sigma},...,e\textsubscript{n}), with c \in CS\textsuperscript{n}, f \in FS\textsuperscript{n} such that the subcontext appears in a singular argument of f, and e\textsubscript{1},...,e\textsubscript{n} \in Exp\textsubscript{\bot}. 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\textsubscript{\sigma}\,\textsubscript{\pi}\alpha and CRWL\textsubscript{\sigma}\,\textsubscript{\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\textsubscript{\bot}:

1. If the program contains no extra variables and every function is singular, then \llbracket e \rrbracket\textsuperscript{\sigma}\,\textsuperscript{\pi} = \llbracket e \rrbracket\textsuperscript{\sigma} = \llbracket e \rrbracket\textsuperscript{\sigma}\,\textsuperscript{\beta}.
2. If every function is plural, then \llbracket e \rrbracket\textsuperscript{\sigma}\,\textsuperscript{\pi} = \llbracket e \rrbracket\textsuperscript{\pi}\,\textsuperscript{\alpha} and \llbracket e \rrbracket\textsuperscript{\sigma}\,\textsuperscript{\beta} = \llbracket e \rrbracket\textsuperscript{\pi}\,\textsuperscript{\beta}.

**Proof**

If every function is singular and the program contains no extra variables, then OR\textsuperscript{\sigma}\,\textsuperscript{\pi}\alpha and OR\textsuperscript{\sigma}\,\textsuperscript{\pi}\beta are equivalent to OR, so CRWL, CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\alpha, and CRWL\textsubscript{\sigma}\,\textsubscript{\pi}\beta, behave the same. Note that the absence of extra variables is essential, as for example from program \{f \rightarrow d(X,X)\} we get \llbracket f \rrbracket\textsuperscript{\sigma} \not\subseteq d(0,1) \in \llbracket f \rrbracket\textsuperscript{\sigma}\,\textsuperscript{\pi} = \llbracket f \rrbracket\textsuperscript{\pi}\,\textsuperscript{\beta}.

Similarly, if every function is plural then OR\textsuperscript{\sigma}\,\textsuperscript{\pi}\alpha and OR\textsuperscript{\sigma}\,\textsuperscript{\pi}\beta are equivalent to POR\textsuperscript{\sigma} and POR\textsuperscript{\beta}, respectively. Note that extra variables pose no problem in this case, as
any of these plural semantics is able to instantiate them with an arbitrary substitution from $CSubst$. □

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 $e_1 \rightarrow e_2$, 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 $i$th element of this chain is the character s then the $i$th 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 $CRWL^\alpha$ 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:

\[
\begin{align*}
X \ ? \ Y & \rightarrow X \\
X \ ? \ Y & \rightarrow Y \\
\text{if } tt \text{ then } E & \rightarrow E
\end{align*}
\]

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

\[
\text{(eval \[\text{depth = DEPTH\]} \text{EXPRESSION .)}
\]

where \text{EXPRESSION} is the expression to be evaluated and \text{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

\[
\text{(more .)}
\]

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:

\[
\text{(depth-first .)}
\]
\[
\text{(breadth-first .)}
\]

Finally, the system can be rebooted with the command

\[
\text{(reboot .)}
\]

### 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:

\[
\begin{align*}
\text{branches} & \rightarrow \text{madrid} \ ? \ \text{vigo} \ ? \ \text{badajoz} . \\
\text{employees(madrid)} & \rightarrow e(\text{pepe, men, clerk}) \ ? e(\text{paco, men, boss}) . \\
\text{employees(vigo)} & \rightarrow e(\text{maria, women, clerk}) \ ? e(\text{jaime, men, boss}) . \\
\text{employees(badajoz)} & \rightarrow e(\text{laura, women, clerk}) \ ? e(\text{david, men, clerk}) .
\end{align*}
\]

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 \text{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 \text{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:

\[
\begin{align*}
\text{twoclerks} & \rightarrow \text{find(employees(branches)) .} \\
\text{find \ is \ plural .} \\
\text{find(e(N,G,clerk))} & \rightarrow p(N,N) .
\end{align*}
\]
Once the module has been loaded in our system, 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 $\pi CRWL$ 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`.

```plaintext
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

```plaintext
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 $\omega^\alpha \pi^\beta$ – remember that in that case $CRWL_{\omega^\alpha \pi^\beta}$ and $CRWL_{\omega^\beta}$ would be equivalent and so both would be supported by the system – or not.
On the other hand, the function \textit{vals} is marked as \textit{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 \textit{vals}, and singularity, which is needed for the tests performed by \textit{newIns}:

\begin{verbatim}
vals is plural .
vals(X) -> newIns(X, vals(X)) .
\end{verbatim}

We now generalize our search function to look for any number of clerks, not just two. To do that we will use the function \textit{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):

\begin{verbatim}
nVals is sp .
nVals(N, E) -> take(N, vals(E)) .
\end{verbatim}

\begin{verbatim}
take(s(N), cons(X, Xs)) -> cons(X, take(N, Xs)) .
take(z, Xs) -> nil .
\end{verbatim}

This \textit{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 \texttt{collect} (López-Fraguas and Sánchez-Hernández 1999) or the \texttt{findall} (Hanus 2005) primitives of standard FLP systems.

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

\begin{verbatim}
nClerks is singular .
nClerks(N) -> nVals(N, findClerk(employees(branches))) .
\end{verbatim}

\begin{verbatim}
findClerk is singular .
findClerk(e(N,G,clerk)) -> N .
\end{verbatim}

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

\begin{verbatim}
Maude> (eval nClerks(s(s(s(z)))) .)
Result: cons(pepe,cons(maria,cons(laura,nil)))
\end{verbatim}

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 \textit{find} function above, under \texttt{\pi^2CRWL}. To do this we just have to define a new auxiliary function \textit{findClerksNG}, which this time returns a pair composed by the name of the clerk and his or her genre.

\begin{verbatim}
nClerksNG is singular .
nClerksNG(N) -> nVals(N, findClerkNG(employees(branches))) .
\end{verbatim}

\begin{verbatim}
findClerkNG is singular .
findClerkNG(e(N,G,clerk)) -> p(N, G) .
\end{verbatim}
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 $C^{2\beta}$, therefore its evaluation under $CRWL_{\alpha}^\sigma$ and $CRWL_{\beta}^\pi$ 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;\footnote{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.} 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)) .
```
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 \texttt{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 \texttt{discStepHow}, which non-deterministically offers some of the items or information available to one of the guardians:

\begin{verbatim}
discoverHow is plural.

discStepHow is plural.
discStepHow(p(W, M)) -> askWho(guardians, M).
\end{verbatim}

Note that the additional disjunction \( ?T \) in the recursive call to \texttt{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:

\begin{verbatim}
genPairs is plural.
genPairsStep is plural.
genPairsStep(P) -> p(P, P).
genPairsBad is plural.
genPairsBad(P) -> P ? genPairsBad(genPairsStep(P)).
\end{verbatim}

There the functions \texttt{genPairs} and \texttt{genPairsBad} follow the same pattern as the \texttt{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” \texttt{genPairsStep}. Nevertheless, their behavior is very different, as we can see evaluating the expressions \texttt{genPairs(z)} and \texttt{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 \texttt{genPairsBad}. But this poses no problem for \texttt{genPairs} because the extra \( ?P \) in its definition makes it possible to combine those values.

Finally, the search is started with the function \texttt{escapeHow}, which initializes the search with the trojan gold provided by Ulysses:

\begin{verbatim}
escapeHow -> discoverHow(p(ulysses, trojan-gold)).
\end{verbatim}

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

\begin{verbatim}
Maude> (eval escapeHow .)
Result: p(ulysses, trojan-gold)
\end{verbatim}
When this first result has been computed, we can ask the tool for more with the command \texttt{more}, which progressively will show the path followed by Ulysses to escape:

\begin{verbatim}
Maude> (more .)
Result: p(circe,item(treasure-map))
Maude> (more .)
Result: p(circe,sirens-secret)
Maude> (more .)
Result: p(calypso,item(chest-code))
\ldots
Maude> (more .)
Result: p(polyphemus,key)
\end{verbatim}

In this example the function \texttt{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 \texttt{?} 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 \texttt{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$_{\pi\alpha}$, 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 \texttt{nVals}, and the deduction programming pattern corresponding to \texttt{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 $C_{\alpha\beta}$ 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 $\text{findClerk}$ 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 $\text{findClerk}$ 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 $\text{findClerk}$ 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 $\text{findClerk}$ 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 \( CRWL_{\pi\alpha} \), 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 \( CRWL_{\pi\alpha} \). 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 \( CRWL^\sigma_\pi \) 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 \( CRWL^\sigma_\pi \) 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 \( CRWL^\sigma_\pi \) and \( CRWL^\sigma_\beta \) 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 \( CRWL^\sigma_\pi \), 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 \( CRWL^\sigma_\pi \) 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 $CRWL \subseteq \text{term rewriting} \subseteq \pi^\beta CRWL \subseteq \pi^\alpha CRWL$, corresponding to the chain singular/call-time choice $\subseteq$ run-time choice $\subseteq \beta$-plural $\subseteq \alpha$-plural. Besides, we have determined that for the class of programs $\mathcal{C}^{C\alpha\beta}$, 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 $\mathcal{C}^{C\alpha\beta}$. Regarding the combined semantics $CRWL^e_{\pi^\alpha}$ and $CRWL^e_{\pi^\beta}$, 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 $CSubst_\bot$, 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 $CSubst_\bot$, and its related results. The
characterization of the class of programs $C_{\alpha \beta}$, for which $\pi^{\alpha}CRLW$ and $\pi^{\beta}CRLW$ are equivalent, and the formal proof for that equivalence are also novel, obviously.

- Finally, the logic $CRLW_{\pi^\sigma}$ 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 $CRLW_{\pi^\pi}$.

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 $CRLW_{\pi^\pi}$ and $CRLW_{\pi^\pi}$ 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 $CRLW_{\pi^\pi}$ and $CRLW_{\pi^\pi}$ 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 $CRLW_{\pi^\pi}$ 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 $CRLW_{\pi^\pi}$ 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 $CRWL_{\sigma}^\pi$ 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 $CRWL_{\sigma}^\pi$. 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 $CRWL$, $\pi^CRWL$, 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


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


