Programming with Singular and Plural Non-deterministic Functions [*]

Adrián Riesco Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{ariesco, juanrh}@fdi.ucm.es

Abstract

Non-strict non-deterministic functions are one of the most distinctive features of functional-logic languages. Traditionally, two semantic alternatives have been considered for this kind of functions: call-time choice and run-time choice. While the former is the standard choice of modern implementations of FLP, the latter lacks some basic properties—mainly compositionality—that have prevented its use in practical FLP implementations. Recently, a new compositional *plural semantics* for FLP has been proposed. Although this 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.

In this paper we explore the expressive possibilities of the combination of singular and plural non-determinism. After formalizing the intended semantics by means of a logic calculus, several significant examples exploiting the capabilities of the semantics are presented. These examples have been tested and developed in a Maude-based prototype whose implementation is outlined.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—Nondeterministic languages

General Terms Theory, Languages.

Keywords Non-deterministic functions, Semantics, Program transformation, Term rewriting, Maude

1. Introduction

Non-strict non-deterministic functions are one of the most distinctive features of functional-logic programming (*FLP*) languages [9, 10]. This is reflected in the use of possibly non-terminating and non-confluent constructor-based term rewrite systems (*CS's*) as programs. This combination of non-strictness and non-determinism gives rise to several semantic alternatives [11, 24], as we can see considering the program $\{f(c(X)) \to d(X, X), X \mathbin{?} Y \to X, X \mathbin{?} Y \to Y\}$ and the expression $f(c(0 \mathbin{?} 1))$. From an operational perspective we have to define 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 which arises during the computation. 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 $f(c(0 \mathbin{?} 1))$ in this semantics, while it is not the case for either $d(0, 1)$ or $d(1, 0)$. Modern functional-logic languages like Toy [14] or Curry [10] adopt call-time choice. On the other hand, under a *run-time choice* semantics the values of the arguments are fixed as they are used, 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 $f(c(0 \mathbin{?} 1))$. *Term rewriting* is considered a standard formulation for run-time choice, and is the basis for the semantics of languages like Maude [4].
But we may also see things from another perspective. From a denotational perspective we have to think about the domain used to instantiate the variables of the program rules. 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 $f(c(0) \mathbin{?} c(1))$ and the program above: under run-time choice, that is, term rewriting, the evaluation of $c(0) \mathbin{?} c(1)$ 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 $f(c(0) \mathbin{?} c(1))$. Nevertheless, under a plural semantics we may consider the set $\{c(0), c(1)\}$ which is a subset of the set of values for $c(0) \mathbin{?} c(1)$ 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 $d(1, 0)$. In conclusion: *the traditional identification of run-time choice with a plural semantics is wrong when pattern matching is involved*. This problem did not appear in [24] because no pattern matching was present, nor in [11] because only call-time choice was adopted.

In [23] this fact was pointed out for the first time, and the π*CRWL* logic was proposed as a novel formulation of a plural semantics with pattern matching. This logic shares with *CRWL* [9] (the standard logic for call-time choice) some compositionality properties

[*] This work has been partially supported by the Spanish projects *DESAFIOS* (TIN2006-15660-C02-01), *PROMESAS-CAM* (S-0505/TIC/0407), *FAST-STAMP* (TIN2008-06622-C03-01/TIN), and UCM-BSCH-GR58/08-910502.

that make it more suitable than run-time choice for a value-based language like current implementations of FLP. For example, we have seen that the expression $f(c(0\ ?\ 1))$ has more values than the expression $f(c(0)\ ?\ c(1))$ under run-time choice, even when the only difference between them is the subexpressions $c(0\ ?\ 1)$ and $c(0)\ ?\ c(1)$, which have the same values both under call-time choice, run-time choice, and plural semantics. This violates a fundamental property of FLP languages stating that any expression can be replaced by any other expression which could be reduced to exactly the same set of values. Nevertheless run-time choice can be a good option for other kind of rewriting based languages like Maude, in which the notion of value is not necessarily present, at least in the sense it is in FLP languages.

Although the semantics of π*CRWL* allows an elegant encoding of some problems [21, 23]—in particular those with an implicit manipulation of sets of values—, call-time choice still remains the best option for many common programming patterns. Therefore it would be nice to have a language in which both options could be available. That is our proposal in this paper, where we present a language where the user has the possibility to specify which arguments of each function symbol 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[1] instead of single elements, while the others will be evaluated under the usual singular/call-time choice semantics traditionally adopted for FLP. We have precisely formalized the semantics of this new language through a novel variant of *CRWL* called $CRWL_\pi^\sigma$. However, the main goal of this paper is not formalizing this semantics but exploring its expressive capabilities. Thereby we have extended our Maude based prototype [21] to support $CRWL_\pi^\sigma$, and used it to develop and test several programs that we think are significant examples of the possibilities of this combined semantics. This system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/PluralSemantics/Maude`.

The rest of the paper is organized as follows: after preliminaries, Sect. 3 presents the semantics of our system; Sect. 4 illustrates its use with several examples; Sect. 5 focus on the development of our prototype; and finally Sect. 6 relates the conclusions and outlines the future work.

2. Preliminaries

We consider a 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. We write $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions, and $X, Y, \ldots$ for variables of a numerable set $\mathcal{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. Given a set $\mathcal{A}$ we denote by $\mathcal{A}^*$ the set of finite sequences of elements of that set. For any sequence $a_1 \ldots a_n \in \mathcal{A}^*$ and function $f : \mathcal{A} \rightarrow \{true, false\}$ by $a_1 \ldots a_n \mid f$ we denote the sequence constructed taking in order every element from $a_1 \ldots a_n$ for which f holds. Besides, for any $1 \leq i \leq n$, $(a_1 \ldots a_n)[i]$ denotes a_i.

The set Exp of *expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing **values**.

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

We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathcal{C} ::= [\,] \mid h(e_1, \ldots, \mathcal{C}, \ldots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context $\mathcal{C}$ to an expression e, written by $\mathcal{C}[e]$, is defined inductively as $[\,][e] = e$ and $h(e_1, \ldots, \mathcal{C}, \ldots, e_n)[e] = h(e_1, \ldots, \mathcal{C}[e], \ldots, e_n)$.

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write $e\theta$ for the application of θ to e. The domain of θ is defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$. $[\overline{X/e}]$ denotes a substitution θ with $dom(\theta) = \overline{X}$ such that $\theta(X_i) = e_i$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$.

A *constructor-based term rewriting system* (*CS*), also called *program* throughout this paper, is a set of (constructor-based) rewrite rules of the form $f(\overline{t}) \rightarrow r$ where $f \in FS^n$, $r \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\overline{t}$. In the present work we restrict ourselves to CS's not containing *extra variables*, i.e., CS's for which $var(r) \subseteq var(f(\overline{t}))$ holds for any rewrite rule. We assume that every CS contains the rules $\{if\ true\ then\ X \rightarrow X, X\ ?\ Y \rightarrow X, X\ ?\ Y \rightarrow Y\}$, defining the behavior of $\_?\_ \in FS^2$ and $if\_then\_ \in FS^2$, both used in mixfix mode (the underscores indicate where the arguments are placed), and that those are the only rules for these function symbols. For the sake of conciseness we will often omit these rules when presenting a program.

Given a TRS $\mathcal{P}$, its associated *rewrite relation* $\mathcal{P} \vdash \_ \rightarrow \_$ is defined as: $\mathcal{P} \vdash \mathcal{C}[l\sigma] \rightarrow \mathcal{C}[r\sigma]$ for any context $\mathcal{C}$, rule $l \rightarrow r \in \mathcal{P}$ and $\sigma \in$ *Subst*. We will omit $\mathcal{P}$ when implied by the context.

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

3. The $CRWL_\pi^\sigma$ logic

To deal with non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\perp$. The sets $Exp_\perp$, $CTerm_\perp$, $Subst_\perp$, $CSubst_\perp$ of partial expressions, etc., are defined naturally. Notice that $\perp$ does not appear in programs. Our semantics is a combination of the *CRWL* [9] and π*CRWL* [23] logics. Therefore the semantics of a program $\mathcal{P}$ will be determined by means of a proof calculus able to derive reduction statements of the form $\mathcal{P} \vdash e \rightarrowtriangle t$, with $e \in Exp_\perp$ and $t \in CTerm_\perp$, meaning informally that t is (or approximates to) a *possible value* of e, obtained by iterated reduction of e using $\mathcal{P}$, applying singular or plural parameter passing according to the plurality map. Then the *denotation* of an expression $e \in Exp_\perp$ under a program $\mathcal{P}$ will be defined as $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\perp \mid \mathcal{P} \vdash e \rightarrowtriangle t\}$. In the following, we will usually omit the reference to $\mathcal{P}$.

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

$$
\textbf{OR}\ \frac{\begin{array}{ccc} e_1 \rightarrowtriangle p_1\theta_{11} & & e_n \rightarrowtriangle p_n\theta_{n1} \\ \ldots & \ldots & \ldots \\ e_1 \rightarrowtriangle p_1\theta_{1m_1} & & e_n \rightarrowtriangle p_n\theta_{nm_n} \quad r\theta \rightarrowtriangle t \end{array}}{f(e_1,\ldots,e_n) \rightarrowtriangle t}
$$
$$
\begin{array}{c}
(f(\overline{p}) \to r) \in \mathcal{P}, \theta = ?\{\theta_{11},\ldots,\theta_{1m_1}\} \uplus \ldots \uplus ?\{\theta_{n1},\ldots,\theta_{nm_n}\} \\
\forall i,j.\ \theta_{ij} \in CSubst_\perp \wedge dom(\theta_{ij}) = var(p_i) \\
\forall i.\ m_i > 0, \forall i \in sgArgs(f).\ m_i = 1
\end{array}
$$

Figure 1. Rules of $CRWL^\sigma_\pi$

All that is left is defining the new calculus $CRWL^\sigma_\pi$, in which we will consider sets of partial values for parameter passing instead of single partial values. These sets will be forced to be singleton for the singular arguments, thus getting call-time choice for them. To avoid the need to extend the syntax with new constructions to represent those "set expressions" that we talked about in Sect. 1, we will exploit the fact that $[\![e_1\ ?\ e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$ for any sensible semantics. Therefore the substitutions used for parameter passing will map variables to "disjunctions of values." We define the set $CSubst^?_\perp = \{\theta \in Subst_\perp \mid \forall X \in dom(\theta), \theta(X) = t_1\ ?\ \ldots\ ?\ t_n$ such that $t_1,\ldots,t_n \in CTerm_\perp, n > 0\}$, for which $CSubst_\perp \subseteq CSubst^?_\perp \subseteq Subst_\perp$ obviously holds. The operator $? : CSubst^*_\perp \to CSubst^?_\perp$ constructs the $CSubst^?_\perp$ corresponding to a non empty sequence of $CSubst_\perp$, and is defined as $?(\theta_1\ldots\theta_n)(X) = X$ if $X \notin \bigcup_{i\in\{1,\ldots,n\}} dom(\theta_i)$; $?(\theta_1\ldots\theta_n)(X) = \rho_1(X)\ ?\ \ldots\ ?\ \rho_m(X)$, where $\rho_1\ldots\rho_m = \theta_1\ldots\theta_n \mid \lambda\theta.(X \in dom(\theta))$, otherwise. Then $dom(?(\theta_1\ldots\theta_n)) = \bigcup_i dom(\theta_i)$. This operator is overloaded to handle non empty sets $\Theta \subseteq CSubst_\perp$ as $?\Theta = ?(\theta_1\ldots\theta_n)$ where the sequence $\theta_1\ldots\theta_n$ corresponds to an arbitrary reordering of the elements of Θ.

The $CRWL^\sigma_\pi$-proof calculus is presented in Fig. 1. Its first three rules have been inherited from *CRWL* (as they were inherited by π*CRWL*, too). Rule B (bottom) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) allow us to reduce any variable to itself, and to decompose the evaluation of a constructor-rooted expression. But the novelty is in the rule OR (outer reduction), that has been tuned to take account of the plurality map. As in its original formulation in *CRWL*, it expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing and then continue the evaluation with the correspondingly instantiated right-hand side. The difference is that now we may compute more than one partial value for each plural argument, and then use a substitution from $CSubst^?_\perp$ for parameter passing over that argument, achieving a plural behaviour. On the other hand, for singular arguments we are only allowed to compute a single value, thus performing parameter passing over it with a substitution from $CSubst_\perp$ (as obviously $?\{\theta\} = \theta$), and achieving a singular behaviour (call-time choice).

Example 1. *Consider the program* $\{f(X, c(Y)) \to d(X, X, Y, Y)\}$ *and a plurality map such that* $plurality(f) = sg\ pl$. *In Fig. 2 there is a* $CRWL^\sigma_\pi$*-proof for the statement* $f(0?1, c(0)?c(1)) \rightarrowtriangle d(0,0,0,1)$ *(some steps have been omitted for the sake of conciseness). Note that* $d(0,1,0,1)$ *is not a correct value for the expression* $f(0?1, c(0)?c(1))$ *under* $CRWL^\sigma_\pi$, *because the first argument of* f *is singular and therefore the two occurrences of* X *in the right-hand side of its rule share the same single value, fixed on parameter passing.*

$$
\frac{\begin{array}{c} \dfrac{(*)}{c(0)?c(1) \rightarrowtriangle c(0)} \\ \dfrac{\ldots}{c(0)?c(1) \rightarrowtriangle c(1)} \\ \dfrac{\ldots}{0?1 \rightarrowtriangle 0} \end{array} \quad \dfrac{\dfrac{\ldots}{0 \rightarrowtriangle 0}\ \dfrac{\ldots}{0 \rightarrowtriangle 0}\ \dfrac{\ldots}{0?1 \rightarrowtriangle 0}\ \dfrac{\ldots}{0?1 \rightarrowtriangle 1}}{d(0,0,0?1,0?1) \rightarrowtriangle d(0,0,0,1)}\,\mathrm{DC}}{f(0?1, c(0)?c(1)) \rightarrowtriangle d(0,0,0,1)}\,\mathrm{OR}
$$

where $(*)$ is the following proof:

$$
\frac{\dfrac{\dfrac{}{0 \rightarrowtriangle 0}\,\mathrm{DC}}{c(0) \rightarrowtriangle c(0)}\,\mathrm{DC} \quad \dfrac{}{c(1) \rightarrowtriangle \bot}\,\mathrm{B} \quad \dfrac{\ldots}{c(0) \rightarrowtriangle c(0)}}{c(0)?c(1) \rightarrowtriangle c(0)}\,\mathrm{OR}
$$

Figure 2. A sample $CRWL^\sigma_\pi$-proof

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 behaviour is significantly different (the corresponding redex has been underlined in each rewriting step):*

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

A first step resolving the choice between $c(0)$ *and* $c(1)$ *is unavoidable in order to get an expression matching the only rule for* f, *thus for any reachable c-term the two last arguments of* d *will be the same, contrary to what happens in* $CRWL^\sigma_\pi$ *under the given plurality map. Nevertheless its first two arguments can be different, contrary to what happens under* $CRWL^\sigma_\pi$. *In conclusion, it is easy to define a program and a plurality map for them such that term rewriting and* $CRWL^\sigma_\pi$ *are not comparable wrt the set of computed—reachable by a rewriting derivation, in case of term rewriting—c-terms.*

A useful intuition about $CRWL^\sigma_\pi$-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 with several examples in the next section.

One of the most important properties of $CRWL^\sigma_\pi$, inherited from *CRWL* and π*CRWL*, is its compositionality, which expresses the value-based philosophy underlying $CRWL^\sigma_\pi$: "all I know about an expression is its set of values." Thus expressions with the same set of values can be interchanged:

Theorem 1 (Compositionality)**.**
For all $\mathcal{C} \in Cntxt$, $e \in Exp_\perp$,

$$
[\![\mathcal{C}[e]]\!] = \bigcup_{\{t_1,\ldots,t_n\} \subseteq [\![e]\!]} [\![\mathcal{C}[t_1\ ?\ \ldots\ ?\ t_n]]\!]
$$

for any arrangement of the set $\{t_1, \ldots, t_n\}$ *in* $t_1 \ ? \ \ldots \ ? \ t_n$.
As a consequence: $[\![e]\!] = [\![e']\!] \Leftrightarrow \forall \mathcal{C}.\ [\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$.

Proof. A straightforward modification of the proof for Th. 1 in [23]. □

We conclude our discussion about $CRWL_\pi^\sigma$ with the following result stating that it is in fact a conservative extension of both *CRWL* (call-time choice, or equivalently, singular non-determinism) and π*CRWL* (plural non-determinism), as it was apparent from its rules.

Theorem 2 (Conservative extension)**.** *For any program* $\mathcal{P}$, $e \in Exp_\perp$*:*

1. *If every function is singular then* $[\![e]\!]^{\mathcal{P}}_{CRWL_\pi^\sigma} = [\![e]\!]^{\mathcal{P}}_{CRWL}$.
2. *If every function is plural then* $[\![e]\!]^{\mathcal{P}}_{CRWL_\pi^\sigma} = [\![e]\!]^{\mathcal{P}}_{\pi CRWL}$.

where $[\![e]\!]^{\mathcal{P}}_{CRWL_\pi^\sigma}$, $[\![e]\!]^{\mathcal{P}}_{CRWL}$ *and* $[\![e]\!]^{\mathcal{P}}_{\pi CRWL}$ *are the denotations for* e *under* $\mathcal{P}$ *given by* $CRWL_\pi^\sigma$, *CRWL and* π*CRWL, respectively.*

Proof. Whenever every function is singular then $CRWL_\pi^\sigma$'s OR is equivalent to its original definition in *CRWL*, henceforth each $CRWL_\pi^\sigma$ proof for a given reduction statement will also be a *CRWL* proof for that statement, and vice versa. Similarly, if every function is plural then $CRWL_\pi^\sigma$'s OR is equivalent to π*CRWL*'s POR. □

4. Programming with singular and plural functions

In this section we illustrate the use of the language and our prototype of $CRWL_\pi^\sigma$ by means of different examples. First, we introduce the concrete syntax and motivate the combination of singular and plural semantics with a simple example, while the next examples will illustrate how to combine singular and plural semantics in depth. The source code for these examples and the interpreter to test them can be found at `https://gpd.sip.ucm.es/trac/gpd/wiki/PluralSemantics/Maude`.

4.1 Clerks

This example shows how to perform a search in the database of a certain company. The different branches are defined by using the non-deterministic function `?`, that 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, built with the constructor `e`, that keeps their name, gender, and role in the company:

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

employees(madrid) -> e(john, men, clerk) ?
                     e(larry, men, boss) .
employees(vigo) -> e(mary, women, clerk) ?
                   e(james, men, boss) .
employees(badajoz) -> e(laura, women, clerk) ?
                      e(david, men, clerk) .
```

As pointed out above, `_?_` is a binary predefined function used in infix notation. The system also provides the `if_then_` function and the boolean values `tt` (for *true*) and `ff` (for *false*). This functions are defined by the rules `X ? Y -> X`, `X ? Y -> Y`, `if tt then E -> E`, and interpreted as plural functions for more flexibility: in this way they can be called in the right-hand sides of singular and plural functions while keeping the corresponding plurality of their arguments.

We define a function `twoclerks` which searches in the database for two employees working as clerks (where `p` is the constructor of pairs). The main function is `search`, which has been marked with the keyword `plural` to indicate that its argument has *plural* semantics, that is, it has to be used as a set of records from the database so, although the same variable `N` is used in the two components of the pair, each one can be instantiated with different values:

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

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

```
Maude> (eval twoclerks .)
Result: p(john,john)

Maude> (more .)
Result: p(john,mary)
```

This program works as we expected, even if all the functions are marked as plural (i.e., if π*CRWL* is used). However it could be improved in several directions. First of all, we are interested in getting two *different* clerks. To do that we define a function `newIns` that appends an element at the beginning of a list ensuring that the remaining elements of the list are different to the inserted element. This is checked by `diffL`, which returns the list in its second argument if it does not contain its first argument, and otherwise fails. Thus a disequality test is needed, but in our minimal framework we do not dispose of disequality constraints, common in FLP languages. Nevertheless we can implement a ground version of disequality through regular program rules, as it is done by `neq`.
By default, all function arguments have *singular* semantics, but it can be explicitly stated with the word `singular`:

```
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(john, larry) -> tt .
neq(john, mary) -> tt .
...
```

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

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

would have a funny behaviour if `filterWomen` had been declared a plural function, because then for `filterWomen(mary ? john)` we could compute `john` as a correct value.

On the other hand we have to explicitly mark any function that we want to be interpreted as *plural*, as shown in the functions `search` above and `vals` here, which generates lists of different values of its argument. Note the combination of plurality, to obtain more than one value from the argument of `vals`, and singularity, which is needed for the tests performed by `newIns`:

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

Finally, the same function could need some arguments to have *singular* semantics while others need *plural* semantics. In this case, the semantics of all the arguments must be given initially by using a sequence of s and p, where s stands for singular and p for plural semantics. To illustrate it we generalize our search function to look for any number of clerks, not just two. To do that we will use the function nVals below, that 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):

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

This nVals function is an example of how the use of plural arguments allows us to simulate some features that in a pure call-time choice context have to be defined at the meta level, in this case the collect [14] or findall [10] primitives of standard FLP systems.

Finally the function nClerks starts the search for the number of different clerks specified by the user. It uses the auxiliary function findClerk, that returns the name of a given clerk:

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

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

Now we can search for three different clerks, obtaining john, mary, and laura as the first possible result:

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

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

4.2 Dungeon

Ulysses has been captured and he wants to cheat his guardians using the bottomless bag of gold he carries from Troy. Thus, he needs to know if 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 .
```

These guardians have a complex behavior, circe exchanges Ulysses' trojan-gold by either an item(treasure-map) or the sirens-secret; calypso offers the item(chest-code) when she receives the sirens-secret; aeolus can combine two items[2]; 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 to escape, are marked as plural functions because they treat their corresponding argument as a set of pairs where the second component is an item or some piece of information, and the first one is the actor which provided it. The function discoverHow returns the set of pairs of that shape that can be obtained starting from those contained in its argument, and then chatting to the guardians. Hence it returns either its argument or the result of exchanging the current information with some guardian and then iterating the process. That exchange is performed with discStepHow, that non-deterministically offers some of the items or information available, to one of the guardians:

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

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

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

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

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

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

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

There the functions genPairs and genPairsBad follow the same pattern as discoverHow, but this time are designed to generate values made up with pairs and the supplied argument. Besides this functions share the same "step function" genPairsStep. Nevertheless their behaviour is very different, as we can see evaluating the expressions genPairs(z) and genPairsBad(z): the point is that the value p(p(z,z),z) can be computed for the former but

[2] 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.

not for the latter, because `z` and `p(z,z)` are values generated in different recursive calls to `genPairsBad`. But this poses no problem for `genPairs`, because the extra `? P` in its definition makes it possible to combine those values.

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

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

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

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

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

```
Maude> (more .)
Result: p(circe,item(treasure-map))
Maude> (more .)
Result: p(circe,sirens-secret)
Maude> (more .)
Result: p(calypso,item(chest-code))
...
Maude> (more .)
Result: p(polyphemus,key)
```

In this example the function `discoverHow` is an instance of an interesting pattern of plural function: a function that performs deduction by repeatedly combining the information we have fed it with the information it infers in one step of deduction. Therefore in its definition the function `?` has to be understood again as the set union operator, as it is used to add elements to the set of deduced information. On the other hand the use of a singular argument in `askWho` is unavoidable to be able to keep track of the guardian who answers the question, while its second argument has to be plural because it represents the knowledge accumulated so far.
Several variants of this problem can be conceived, in particular currently it is simplified because the items are not lost after each exchange —this 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 behaviours of the principals can be described through the patterns in 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 the future, and that the addition of equality and disequality constraints to our framework could be decisive to deal with those problems.

4.3 Exams

In this example we show how to use both plural and singular semantics to check if a group of students can pass an exam (that is, to obtain a mark bigger or equal than `s(s(s(s(s(z)))))`) by sharing their knowledge about the subjects that each one has studied. First, we define the students and their initial knowledge; `james` has studied the subjects `t1.1` and `t1.3`, `lyla` has studied `t1.2` and `t3`, and `harry` has studied `t2.1` and `t4`. This knowledge is represented in the function `subjects1` where the alternative function `?` is understood once more as the set union, in such a way that a set of database records is represented. Note that the possibility of nesting `?` inside the constructor `knows` allows us to get a more compact representation of this database than we could have if the alternatives where allowed at the top level only:

```
team1 -> james ? lyla ? harry ? alice .
subjects1 -> knows(james, t1.1 ? t1.3) ?
             knows(lyla, t1.2 ? t3) ?
             knows(harry, t2.1 ? t4) .
```

We define the function `knowsAll` to check if all the subjects required in the list given as first argument are contained in the set of subjects represented in its second argument, where each subject can have been studied by a different student, and hence is marked as *plural*. This function is just an special case of `listInSet`, which checks if every element of a given list is contained in a given set, represented by a plural argument. As pointed out in the Clerks example, we do not dispose of equality constraints in our minimal framework, hence `eq` corresponds here to its ground version implemented by using regular program rules:

```
knowsAll is sp .
knowsAll(Questions, Knowledge) ->
        listInSet(Questions, Knowledge) .

listInSet is sp .
listInSet(nil, K) -> tt .
listInSet(cons(K1, Ks), K) ->
        if eq(K, K1) then listInSet(Ks, K)  .
```

Using this function and the same semantics we define `mark`, that indicates the subjects needed to solve each question and its marks; for example, the question `q1` requires the subjects `t1.1` and `t1.2` and two marks are obtained when answered correctly, while the question `q2` requires `t1.3` and `t3` and one mark is obtained when answered:

```
mark is sp .
mark(q1, K) ->
     if knowsAll(cons(t1.1, cons(t1.2, nil)), K)
     then s(s(z)) .
mark(q2, K) ->
     if knowsAll(cons(t1.3, cons(t3, nil)), K)
     then s(z) .
...
```

The function `answer` checks the knowledge of each particular student. Given a student and a database in the format of `subjects1` it returns a non-deterministic value representing the set of subjects that the students knows, as represented in the database. For example if we evaluate `answer(lyla, subjects1)` we will first get the results `t1.2`, then `t3` if we ask for more, and finally `No more results` if we ask for a third result. As usual, as it performs tests, its arguments are marked as singular:

```
answer is singular .
answer(Student, knows(S, Subject)) ->
        if eq(Student, S) then Subject .
```

The function `markAll` generates a list with the marks obtained in each question. Its first argument is an exam represented as a list of questions, hence it is annotated as singular in order to use the same exam during the whole marking process. On the other hand we annotate its second argument as plural because it represents the set of subjects the students have studied:

```
markAll is sp .
markAll(nil, K) -> nil .
markAll(cons(Q, Qs), K) ->
        cons(mark(Q, K), markAll(Qs, K)) .
```

In the same way, examine is declared as spp, fixing the exam and allowing different combinations of the team and the studies. We do so because we assume that the students will be cheating and passing on results between them, hence that combinations represent the "collective knowledge" owned by the students. The sum function just folds a list of naturals into its total sum:

```
examine is spp .
examine(Exam,Team,Studies) ->
 p(Exam,sum(markAll(Exam,answer(Team,Studies)))) .
```

The main function of the program is passes, that checks if the team can obtain the minimum mark to pass the exam. There the function someElems, that non-deterministically drops some elements of its input list, is used to express that the students may pass the exam without answering all the questions:

```
passes is spp .
passes(Exam,Team,Studies) ->
   minMark(examine(someElems(Exam),
                     Team,Studies)) .

minMark(p(Exam, s(s(s(s(s(N))))))) ->
   p(Exam, s(s(s(s(s(N)))))) .

someElems(Xs) -> nil .
someElems(cons(X, Xs)) -> cons(X, someElems(Xs)) .
someElems(cons(X, Xs)) -> someElems(Xs) .
```

Finally, we define the different exams:

```
exams ->
cons(q1,cons(q2.1,cons(q3.2,cons(q4.1,
     cons(q5,nil)))))
? cons(q1,cons(q2.1,cons(q3.2,cons(q4.2,
       cons(q6,nil)))))
? cons(q1,cons(q2.2,cons(q3,cons(q4.1,
       cons(q5,nil))))) .
```

Now we can check which are the possible answers that the students can give in order to pass one of the exams:

```
Maude> (eval passes(exams, team1, subjects1) .)
The term cannot be reduced to a cterm.
```

As we can see, they should work harder because they cannot pass any of the exams. We can try to make some of them study another subject, for example t5, with the command:

```
Maude> (eval passes(exams, team1, subjects1 ?
                             knows(lyla, t5)) .)
Result: p(cons(q1,cons(q2.1,
           cons(q4.1,cons(q5,nil)))),
                        s(s(s(s(s(zero))))))
```

Now, they can pass the first exam by answering the questions q1, q2.1, q4.1, and q5. Note that we have made lyla study an extra subject by just combining the new record knows(lyla, t5) with subjects1 through the function ?. However in a run-time choice semantics we would have needed to nest t5 inside the original knows record for lyla in subjects1, in order to get the same result.

We can also check if it is possible to pass this exam answering other questions or if it is possible to pass another exam. We do it with the command:

```
Maude> (more .)
Result: p(cons(q1,cons(q3,cons(q4.1,cons(q5,nil)))),
                             s(s(s(s(s(zero))))))
```

That is, it is also possible to pass the third exam. If we ask for more results, we realize that no other solutions are reachable:

```
Maude> (more .)
No more results.
```

4.4 Discussion: to be singular or to be plural?

After these examples, now we (hopefully) 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 and that plural arguments denote single values. But this does not work for any situation, for example consider the function findClerk whose plurality is singular, although its argument intuitively denotes a set of records from the database. We can also consider that its argument denotes a single record, and that findClerk defines how to extract the name from a single employee, which motives the final plurality choice. In this case the program behaves the same both declaring findClerk as 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 that the singular ones [23]. Hence here it is better to declare findClerk as singular.

Thus having a more formal criterion about the equivalence of plurality maps would be useful to minimize the search space of our programs and understand them better. A static adaptation of the determinism analysis of [3] 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 [15] to our plural semantics. We also should try to develop equational laws about non-determinism in the line of the "bubbling" rule as formulated in [16], stating that $[\![\mathcal{C}[e_1 \ ? \ e_2]]\!] = [\![\mathcal{C}[e_1]]\!] \cup [\![\mathcal{C}[e_2]]\!]$. Although this rule does not hold in $CRWL_\pi^\sigma$ —just consider the expression $pair(0 \ ? \ 1)$ for the plural function $pair$ defined as $pair(X) \rightarrow (X, X)$—, we conjecture that it would be the case for its restriction to singular contexts (where the hole has no plural argument position above itself). All these are interesting subjects of future work.

5. Implementation

We describe in this section the modification of the core language of [17], upon which we have built our implementation. Source programs are transformed into core programs by a combination of the sharing transformation of [17] and the plural transformation of [23] that now takes the plurality map into account. These core programs are then evaluated using a heap-based operational semantics, to which the natural rewriting on-demand strategy [6] has been adapted. All these new features have been added to our Maude based prototype [21], thus getting an implementation of $CRWL_\pi^\sigma$.

5.1 Program transformations

In [17] a run-time choice rewriting framework enhanced with a *let* primitive to express sharing was proposed for the combination of singular/call-time choice and run-time choice semantics. There the syntax is extended to handle *let*-expressions, defined as $LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$. We will often use the notation $let\ \overline{X = a}\ in\ e$ to abbreviate $let\ X_1 = a_1\ in\ \ldots\ in\ let\ X_n = a_n\ in\ e$. These $LExp$ are governed by the run-time let rewriting relation $\rightarrow^{rt}$ that defines its operational behaviour. Intuitively, $LExp$'s are evaluated by $\rightarrow^{rt}$ like in regular term rewriting with the exception that in any reduction of a *let*-expressions $let\ X = e_1\ in\ e_2$, all the occurrences of X in e_2 share the same value, produced by e_1.

We will use a simplified version of that framework as the *core language* for our implementation. The point is that the transforma-

$pST(f(p_1,\ldots,p_n) \to r)$

$$= \begin{cases} f(p_1,\ldots,p_n) \to r & \text{if } \rho_1 \ldots \rho_m \text{ is empty} \\ f(\tau(p_1),\ldots,\tau(p_n)) \to \begin{array}{l} if\ match(Y_1,\ldots,Y_m) \\ \quad then\ r[\overline{X_{ij}/project_{ij}(Y_i)}] \end{array} & \text{otherwise} \end{cases}$$

where $\rho_1 \ldots \rho_m = p_1 \ldots p_n \mid \lambda p_l.(p_l \notin \mathcal{V} \wedge var(p_l) \neq \emptyset \wedge l \in plArgs(f))$.
- $(Y_1,\ldots,Y_m)$ is a linear tuple of fresh variables, one for each ρ_i. A mapping Φ is then defined so $\Phi(\rho_i)$ returns the fresh variable corresponding to ρ_i.
- $\forall \rho_i, \{X_{i1},\ldots,X_{ik_i}\} = var(\rho_i) \cap var(r)$.
- $\tau : CTerm \to CTerm$ is defined by $\tau(p_l) = p_l$ if $p_l \notin \rho_1 \ldots \rho_m$; otherwise $\tau(p_l) = \Phi(p_l)$.
- $match \in FS^m$ fresh is defined by the rule $match(\rho_1,\ldots,\rho_m) \to true$.
- Each $project_{ij} \in FS^1$ is a fresh symbol defined by the single rule $project_{ij}(\rho_i) \to X_{ij}$.

$$sST(f(p_1,\ldots,p_n) \to r) = f(p_1,\ldots,p_n) \to let\ \overline{Y = X}\ in\ e[\overline{X/Y}]$$

where $\overline{X} = var(e) \cap sgVars(f(p_1,\ldots,p_n))$ and $\overline{Y}$ is a linear tuple of fresh variables.

Figure 3. Plural and Singular Semantics Transformations

tion that we will present below only introduces *let* bindings of the shape $let\ Y = X$ in the right hand side of program rules. Therefore our *core programs* will be sets of program rules of the form $f(\overline{t}) \to let\ \overline{Y = X}\ in\ r$ with $f \in FS^n$, $r \in Exp$, $\overline{t}$ a linear n-tuple of c-terms and $\overline{X}, \overline{Y} \subset \mathcal{V}$. As we said in the preliminaries, we will restrict ourselves to programs not containing extra variables.

Our transformation is presented in Fig. 3, and translates source programs —regular constructor systems— into core programs that can be evaluated using the run-time let rewriting relation $\to^{rt}$ [17]. This transformation treats each program rule separately, applying two different transformation stages to them. Assume a program rule $(f(p_1,\ldots,p_n) \to r)$, first of all, if the rule has some plural arguments the modification of the *pST* transformation [23] shown in Fig. 3 is applied to them, postponing their pattern matching to prevent an early resolution of non-determinism. For example, given the Dungeon example in Sect. 4.2, the following rule for the plural function `discStepHow`

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

is transformed into the following set of rules:

```
discStepHow(V) ->
  if match9(V)
  then askWho(guardians, project9-0-0(V)).
match9(p(W, M)) -> tt .
project9-0-0(p(W, M)) -> M .
```

that is, the expression is substituted for a fresh variable `V` to postpone the pattern matching, the function `match9` is used as a guard to check that the expression really matches the expected pattern, and `project9-0-0` extracts the appropriate component.

Then we proceed with the singular transformation sST from Fig. 3, a modification of the sharing transformation of [17, Def. 1]. This transformation introduces a *let* binding for each singular variable also present in the right hand side. Thereby we assure that the let construct is in the outermost position of the right-hand side of the transformed rule, as we anticipated before. For example, in the Clerks program the singular rule for `newIns`:

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

is transformed into the rule:

```
newIns(X,Xs) -> let V1 = X, V0 = Xs
                in cons(V1,diffL(V1,V0)).
```

We will not give a formal proof of the soundness of the simulation of $CRWL_\pi^\sigma$ through the transformations of Fig. 3 and the use of $\to^{rt}$, although the formal results regarding the original transformations and the conservativeness of $\to^{rt}$ wrt. $\to$ found in [17, 23], already give a clue about its adequacy. Anyway, as mentioned in Sect. 1, our main goal is to experiment with $CRWL_\pi^\sigma$ so those technical results are beyond the scope of this paper.

5.2 An Operational Semantics for the core language

In order to simplify the management of the scopes generated by *let* bindings, instead of using $\to^{rt}$ core programs will be evaluated using a heap-based operational semantics. 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. This can be used to represent $LExp$ because any *let*-expression in fact corresponds to a directed acyclic graph with variables or symbols of the signature in its nodes. Similar heap structures were used in Launchbury's natural semantics for functional programming [12], the operational semantics for FLP of [1, 2] or the formalization of term graph rewriting of [20]. But contrary to what it is done there, and in line with [17], we do not require a previous "normalization" step ensuring that every argument of an application of a symbol from Σ is a variable.
The advantage of this approach is that this way all the bindings are kept in the same scope, easing their manipulation. We will use the metavariables Γ, Δ to denote heaps and, in order to clarify the notation, will name every variable bound in the heap with the subscript $\_p$, where p connotes "pointer." Thus $\Gamma[X_p]$ denotes the expression associated in Γ to the variable X_p, and $\Gamma[X_{p_1} \mapsto e_1,\ldots,X_{p_n} \mapsto e_n]$ denotes a heap Δ with $\Delta[X_{p_i}] = e_i$ and $\Delta[Y] = \Gamma[Y]$ for every $Y \notin \{X_{p_1},\ldots,X_{p_n}\}$. Substitutions are applied to the heap as $[\overline{X \mapsto e}]\theta = [\overline{X \mapsto e\theta}]$.

Our operational semantics manipulates configurations $\Gamma : R_p$, where R_p stands for a pointer to the root of the graph represented in Γ. Therefore to evaluate an expression e we will start from $[R_p \mapsto e] : R_p$ and reduce it until reaching a configuration $\Gamma : R_p$ where the subgraph represented in $\Gamma[R_p]$ corresponds to a c-term. Fig. 4 shows the reduction rules for this configurations. The key rule is FAPP. Our transformation assures that $\overline{X} = sgVars(f(\overline{t}))$

FAPP $\Gamma[X_p \mapsto \mathcal{C}[f(\overline{t})\sigma]] : R_p \to_g \Gamma[X_p \mapsto \mathcal{C}[r([\overline{Y/Y_p}] \uplus \sigma)], \overline{Y_p/\sigma(X)}] : R_p$
if $(f(\overline{t}) \to let\ \{\overline{Y = X}\}\ in\ r) \in \mathcal{P}, \sigma \in Subst, dom(\sigma) = var(f(\overline{t}))$, for $\overline{Y_p}$ fresh
BINDC $\Gamma[X_p \mapsto c(\overline{e})] : R_p \to_g (\Gamma[X_p/c(\overline{Y_p})])[\overline{Y_p \mapsto e}] : R_p$ if $X_p \neq R_p$ and $c \in CS$, for $\overline{Y_p}$ fresh
BINDV $\Gamma[X_p \mapsto Z] : R_p \to_g \Gamma[X_p/Z]$ if $X_p \neq R_p$ and $Z \in \mathcal{V}$.
TRASH $\Gamma[X_p \mapsto e] : R_p \to_g \Gamma : R_p$ if $X_p \notin (\bigcup_{(Y_p \mapsto e') \in \Gamma} var(e')) \cup \{R_p\}$

Figure 4. Operational semantics for the core language

and $dom(\sigma) \setminus \overline{X} = plVars(f(\overline{t}))$, therefore the substitution $[\overline{Y/Y_p}] \uplus \sigma$ will perform parameter passing for the plural variables while refreshing the singular variables, for each of them a new entry in the heap has been made pointing to the result of applying σ to its corresponding variable in $\overline{t}$. Rules BINDC and BINDV propagate through the heap the outer computed part of expressions. Finally rule TRASH throws away needless bindings.
Again, we will not prove any formal result relating the relations $\to_g$ and $\to^{rt}$ because, as mentioned before, it is beyond the scope of the paper. Nevertheless their equivalence is apparent, although the treatment of the BIND rules is a little different, and $\to^{rt}$ can handle a more general kind of let bindings (not only those of the shape $let\ X = Y\ in\ e$).

A small additional effort is needed to turn $\to_g$ into an effective operational mechanism for $CRWL^{\sigma}_{\pi}$: an optimal evaluation strategy should be attached to it to ensure that the evaluation is performed on-demand. We have opted for the natural rewriting strategy [6], which was also the choice in our previous prototype. The matching definitional trees used to implement it remain the same, because left-hand sides do so, but the function used to apply the strategy to terms had to be adapted to deal with heaps instead of terms. Then the evaluation proceeds as follows, first we repeatedly apply BINDC, BINDV and TRASH until they cannot be applied anymore, and then we apply FAPP over the subexpression pointed by the natural rewriting strategy, and iterate this process until $\Gamma[R_p]$ contains the representation of a c-term. This is sensible because the BIND or TRASH steps do not change the graph represented in the heap but just reconfigure its representation to enable the application of more FAPP steps. Besides the following result states that this reconfiguration process always terminates.

Proposition 1. *The relation $\to_{g_{\setminus \text{FAPP}}}$ defined by the rules of Fig. 4 except* FAPP *is terminating.*

Proof. We define $sM(\Gamma, X_p) = sM(\Gamma, \Gamma[X_p])$, $sM(\Gamma, X) = \{X\}$ if X is not defined in Γ, and $sM(\Gamma, h(\overline{e})) = \{h\} \oplus \bigoplus_{e_i \in \overline{e}} sM(\Gamma, e_i)$ (where $\oplus$ is the union of multisets), and finally $sM([X_{p_1} \mapsto e_1, \dots, X_{p_n} \mapsto e_n]) = \bigoplus_i sM(\Gamma, X_{p_i})$. Thus $sM(\Gamma)$ is a multiset of symbols in the heap Γ, with one occurrence of each symbol per each reference to them. Hence each $\to_{g_{\setminus \text{FAPP}}}$ step necessarily decreases the cardinality of $sM(\Gamma)$. □

5.3 Maude implementation

The current implementation of this language is an extension of the work presented in [21], where only the plural semantics transformation was available. This extension has been achieved in a very short time thanks to the modularity of the implementation—that keeps program transformations, execution, and user interaction separated—and the use of Maude as specification language because, on the one hand, provides powerful features to transform programs [5] and, on the other hand, allows an almost immediate translation of the semantic rules to code [25].

Maude [4] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [19]. Rewriting logic is a good semantic framework for formally specifying programming languages as rewrite theories [4, Chap. 20]. Moreover, since those specifications usually can be executed in Maude, they in fact become interpreters for these languages.

Exploiting the fact that rewriting logic is reflective [5], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [4, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Furthermore, the Maude system provides another module, `LOOP-MODE` [4, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our program transformation, its execution, and its user interactions are implemented in Maude itself.

All these capabilities are used in our tool. First, Maude allows to represent the operational semantics described in Sect. 5.2 in a natural way [25]. Moreover, we have adapted the on-demand natural rewriting strategy shown in [21] to deal with this semantics by defining functions that select the expression in the heap that must be evaluated, and reducing then the subexpression indicated by the natural rewriting strategy, that has been adapted to take into account the dereferencing caused by the heap. On the other hand, program transformations applied in our system (see Sect. 5.1) can be easily handled by using the reflective capabilities of Maude, that allow to manipulate rules as usual data. Thus, the rules introduced by the user can be traversed and modified in a direct way, new rules (such as the *match* and *project* rules of the plural transformation) can be added, and the obtained module can be directly executed.

6. Conclusions and future work

In this paper we have explored the expressive capabilities of the combination of singular and plural arguments in non-deterministic functions. Several examples have been presented, showing that this combination allows us to define programs in a very natural way, and a Maude based prototype has been made available to test them.

Previously to ours, not much work has been done in the combination of singular and plural non-determinism in functional or functional-logic programming, since mainstream approaches [10, 14, 26] only support the usual singular/call-time choice semantics. More close are the combinations of call-time and run-time choice of [13, 17], which anyway follow a different approach as the plural side of $CRWL^{\sigma}_{\pi}$ is essentially different to run-time choice. The monad transformer of [7], 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 *lets* 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 [17] but again essentially different to $CRWL^{\sigma}_{\pi}$ for the same reason. Besides that work is focused in implementation issues of FLP in concrete deterministic functional languages, while in ours we start from the more abstract world of CS's and are fundamentally concerned in exploring the language design space.

Our examples have also shown the direction that should be followed to improve the prototype. First, it has arisen the necessity of equality and disequality constraints (whose ground version have been simulated by using regular functions), that will ease and shorten the definition of programs. Adding higher order capabilities by an extension of $CRWL^{\sigma}_{\pi}$ in the line of [8], and implementing them by means of the classic transformation of [28], would also be interesting and it is standard in the field of FLP. Then, for example, we could define a more generic version of `discoverHow` with an additional argument for the function used to perform a deduction step (`discStepHow` in our dungeon problem). This higher order version of $CRWL^{\sigma}_{\pi}$ could also be used to face the challenges regarding the implementation of type classes in FLP through the classical transformational technique of [27] pointed out by Lux in [18]. Although some solutions based on the frameworks of [13, 17] were already proposed in [22] 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 Maude to enhance the expressiveness of the semantics, after a corresponding revision of the theory of $CRWL^{\sigma}_{\pi}$. Besides, some additional research must be done to improve the performance of the interpreter by means of some kind of sharing across non-determinism in the line of [2] and in some sense [7], as both are based on similar data structures where non-determinism is nested under constructors.

Finally, as suggested in Sect. 4.4, finding criterion for the equivalence of plurality maps and defining equational laws for non-determinism would improve the understanding of programs.

Acknowledgements: We would like to thank the anonymous referees for their very useful comments and suggestions.

References

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

[2] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. 5th Asian Symposium on Programming Languages and Systems (APLAS'07)*, Springer LNCS 4807, pages 122–138, 2007.

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

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, Springer LNCS 4350, 2007.

[5] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2): 70–91, 2007.

[6] S. Escobar. Implementing natural rewriting and narrowing efficiently. In *Proc. 7th International Symposium on Functional and Logic Programming (FLOPS'04)*, Springer LNCS 2998, pages 147–162, 2004.

[7] S. Fischer, O. Kiselyov, and C.-c. Shan. Purely functional lazy non-deterministic programming. In *Proc. 14th ACM SIGPLAN international conference on Functional programming (ICFP '09)*, pages 11–22. ACM, 2009.

[8] J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

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

[10] M. Hanus. Functional logic programming: From theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[11] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[12] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM, 1993.

[13] F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A lightweight combination of semantics for non-deterministic functions. In *Proc. 18th Workshop on Logic-based methods in Programming Environments (WLPE'08)*, *CoRR*, abs/0903.2205, 2009.

[14] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[15] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM, 2007.

[16] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, Springer LNCS 4989, pages 147–162, 2008.

[17] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A flexible framework for programming with non-deterministic functions. In *Proc. 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM'09)*, pages 91–100. ACM, 2009.

[18] W. Lux. Curry mailing list: Type-classes and call-time choice vs. run-time choice. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`, August 2009.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[20] R. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[21] A. Riesco and J. Rodríguez-Hortalá. A natural implementation of plural semantics in Maude. In *Proc. 9th Workshop on Language Descriptions Tools and Applications (LDTA'09)*, 2009.

[22] J. Rodríguez-Hortalá. Curry mailing list: Re: Type-classes and call-time choice vs. run-time choice. `http://www.informatik.uni-kiel.de/~curry/listarchive/0801.html`, August 2009.

[23] J. Rodríguez-Hortalá. A hierarchy of semantics for non-deterministic term rewriting systems. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, 2008.

[24] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

[25] A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27:113–172, 2005.

[26] P. Wadler. How to replace failure by a list of successes. In *Proc. Functional Programming and Computer Architecture*. Springer LNCS 201, 1985.

[27] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Pro.16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[28] D. H. Warren. Higher-order extensions to Prolog: are they needed? In J. Hayes, D. Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., 1982.