Error: Macro TOC(None) failed

`'NoneType' object has no attribute 'endswith'`

Overview

In a recent work [1][2] a new semantics for *non-deterministic lazy functional(-logic) programming (**FLP**)* was presented, in which the treatment of parameter passing was different to previous proposals like call-time choice (CRWL[3]) or run-time choice (term rewriting [4]). In that work the semantics is formalized through a variation of the CRWL logic, and a program transformation to simulate this new logic with term rewriting is proposed.

Here you can find a first implementation of this new semantics [5], [6] developed in the Maude system, based on the mentioned program transformation and the Natural Rewriting on-demand strategy [7].

A plural semantics for functional-logic programming

Consider the program `P = {f(c(X)) -> d(X,X), X ? Y -> X, X ? Y -> Y}` and the expression `f(c(0?1))`. Let us see what are the values for that expression under the traditional semantics for non-deterministic functions.

- Under **call-time choice** parameter passing to compute a value for the term `f(c(0?1))` we must first compute a (partial) value for `c(0?1)`, and then we may continue the computation with `f(c(0))` or `f(c(1))` which yield `d(0,0)` or `d(1,1)`. Note that `d(0,1)` and `d(1,0)` are not correct values for `f(c(0?1))` in that setting.

  > From the point of view of a denotational semantics, call-time choice parameter passing is equivalent to having a **singular semantics**, in which the substitutions used to instantiate the program rules for function application are such that the variables of the program rules range over single objects of the set of values considered.

- On the other hand, under **run-time choice** parameter passing, which corresponds to call-by-name, each argument is copied without any evaluation and so the different copies of any argument may evolve in different ways afterwards. Therefore the step `f(c(0?1)) --> d(0?1, 0?1)` is sound, and then not only `d(0,0)` and `d(1,1)` but also `d(0,1)` and `d(1,0)` are valid values for `f(c(0?1))`.

  > Traditionally it has been considered that run-time choice has its denotational counterpart on a **plural semantics**, in which the variables of the programs rules take their values over sets of objects of the set of values considered. Hence in the example we consider the set `{c(0), c(1)}` which is a subset of the set of values for `c(0?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})`, which evaluation yields the values `d(0,0)`, `d(1,1)`, `d(0,1)` and `d(1,0)`.

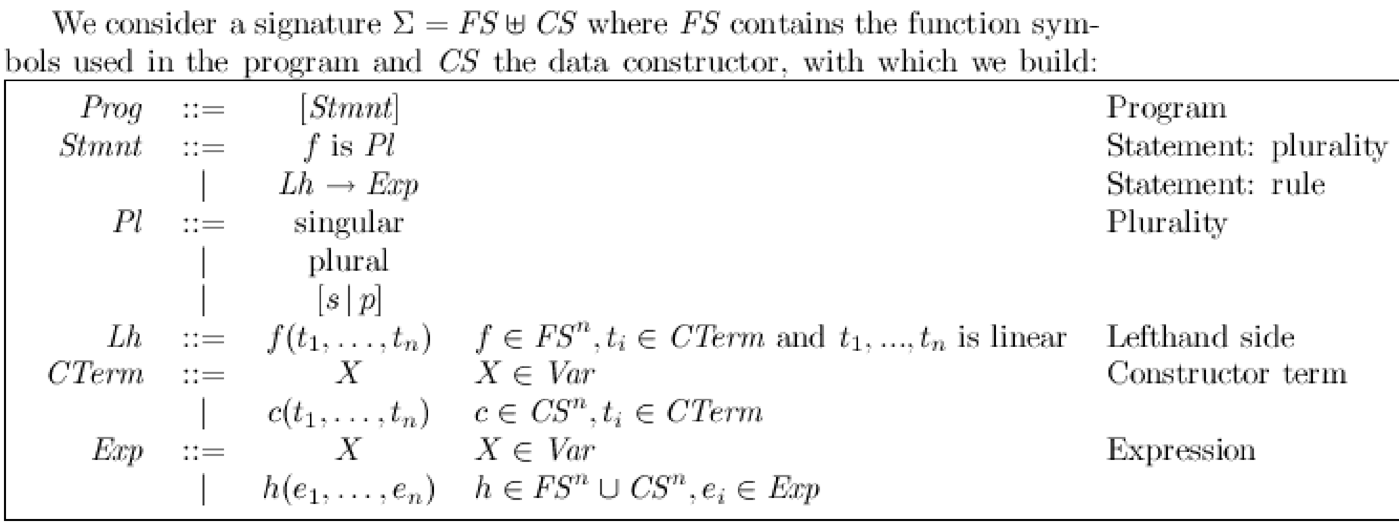
Nevertheless, this traditional identification of run-time choice with a plural semantics is wrong, as we can see evaluating the expression `f(c(0)?c(1))`:

- Under the run-time choice point of view, that is, using term rewriting, the evaluation of the subexpression `c(0)?c(1)` is needed in order to get an expression that matches the left hand side `f(c(X))`. Hence the derivations `f(c(0)?c(1)) --> f(c(0)) --> d(0,0)` and `f(c(0)?c(1)) --> f(c(1)) --> d(1,1)` are sound and compute the values `d(0,0)` and `d(1,1)`, but neither `d(0,1)` nor `d(1,0)` are correct values for `f(c(0)?c(1))`.
- Under the plural semantics point of view, we 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)`.

Which of these is the more suitable perspective for FLP?. Choosing the run-time choice perspective of term rewriting has some unpleasant consequences. First of all the expression `f(c(0?1))` has more values than the expression `f(c(0)?c(1))`, even when the only difference between them is the subexpressions `c(0?1)` and `c(0)?c(1)`, which have the same values both in call-time choice, run-time choice and plural semantics. This is pretty incompatible with the value-based semantic view we are looking for in FLP. And this has to do with the loss of some desirable properties, present in \crwl, when switching to run-time choice. Plural semantics recovers those properties, which are very useful for reasoning about computations, and besides it allows natural encodings of some programs that need to do some collecting work. Hence we claim that the plural semantics perspective is more suitable for a value-based programming language.

The logic

We work with first order left-linear constructor-based term rewrite systems as programs. Regarding syntax:



We consider a signature $\Sigma = FS \uplus CS$ where FS contains the function symbols used in the program and CS the data constructor, with which we build:

$Prog$	$::=$	$[Stmnt]$		Program
$Stmnt$	$::=$	f is Pl		Statement: plurality
	$\mid$	$Lh \to Exp$		Statement: rule
Pl	$::=$	singular		Plurality
	$\mid$	plural		
	$\mid$	$[s \mid p]$		
Lh	$::=$	$f(t_1, \ldots, t_n)$	$f \in FS^n, t_i \in CTerm$ and $t_1, ..., t_n$ is linear	Lefthand side
$CTerm$	$::=$	X	$X \in Var$	Constructor term
	$\mid$	$c(t_1, \ldots, t_n)$	$c \in CS^n, t_i \in CTerm$	
Exp	$::=$	X	$X \in Var$	Expression
	$\mid$	$h(e_1, \ldots, e_n)$	$h \in FS^n \cup CS^n, e_i \in Exp$	

The following logic defines the meaning of an expression as the set of values (constructed terms) corresponding to that expression:

$$\textbf{(RR)}\quad \frac{}{X \rightarrow X}\quad X \in \mathcal{V} \qquad\qquad \textbf{(B)}\quad \frac{}{e \rightarrow \perp}$$

$$\textbf{(DC)}\quad \frac{e_1 \rightarrow t_1 \ \ldots\ e_n \rightarrow t_n}{c(e_1, \ldots, e_n) \rightarrow c(t_1, \ldots, t_n)}\quad c \in CS^n$$

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

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

The denotation of an expression $e \in Exp_\perp$ under a program $\mathcal{P}$ is defined as $[\![e]\!]^{sap}_{\mathcal{P}} = \{t \in CTerm_\perp \mid \mathcal{P} \vdash_{CRWL^{\sigma}_{\pi^{\alpha}}} e \rightarrow t\}$

The system

The current prototype of plural semantics is based on a source-to-source transformation of term rewriting systems. The transformation is implemented on the Maude system [8], and the transformed program is also interpreted by Maude too, using the natural rewriting on-demand strategy [7].

Installation

Linux users

1. Download the file plural-maude-linux.tgz.
2. Uncompress it.
```
tar -xvzf plural-maude-linux.tgz
```
3. Enjoy!
```
cd plural-maude-linux/
./plural.bin
```

Other OS

As our implementation is based on Maude, the first step would be getting it running on your computer:

1. The download page for Maude and the installation instructions can be found here.
2. Download the file plural.maude.
3. Execute (assuming that the command to start Maude is `maude`):
```
maude plural.maude
```

Once you have started the system you will see the following promt:

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
```

```
                     /||||||||||||||||||\
            Maude 2.3 built: Feb 14 2007 17:53:50
            Copyright 1997-2007 SRI International
                   Mon Oct 20 16:54:42 2008

        Executable Plural Semantics (October 13th, 2008)

Maude>
```

Usage

Programs

Programs are sequences of plurality annotations for function symbols and constructor-based left-linear rewrite rules, written in the following format:

```
(plural <PROGRAM-NAME> is
  <op 1> is <plurality 1> .
         ...
  <op N> is <plurality N> .
  <lefthand side 1> -> <righthand side 1> .
                     ...
  <lefthand side M> -> <righthand side M> .
endp)
```

For example the following is a valid plural semantics program:

```
(plural CLERKS is

  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) .

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

Remarks:

- The *variable names* must start with an upper-case letter
- You can use Maude *comments* in the programs, that is, `---` or `***` for single line comments and `***(` ended by `)` for multiline comments.
- If the plurality of an function symbol is not indicated, it is considered singular by default.

Commands

The following commands are available:

- **load** : Loads a program.
  ```
  load <file>
  ```
  You can also loads programs by directly copying them in the promt:
  ```
  Maude> (plural CLERKS is
  >   branches -> madrid .
  >   branches -> vigo .
  >   employees(madrid) -> e(pepe, men, clerk) .
  >   employees(madrid) -> e(paco, men, boss) .
  ```

```
>   employees(vigo) -> e(maria, women, clerk) .
>   employees(vigo) -> e(jaime, men, boss) .
>   twoclerks -> search(employees(branches)) .
>   search(e(N,S,clerk)) -> p(N,N) .
> endp)

Module introduced.

Maude>
```

- **eval** : Evaluates an expression to a constructor normal form (a value) and prints it, or prints `The term cannot be reduced to a cterm` if it does not exists.
  ```
  (eval <exp> .)
  ```
- **eval [depth]** : Like eval but with a limit in the depth of the exploration of the search space.
  ```
  (eval [depth= <depth>] <exp> .)
  ```
- **more** : Prints the next value for the current evaluated expression or prints `No more results` if there are no more values for that expression.
  ```
  (more .)
  ```
- **depth-first** : Turns depth-first search on. This is the *default.*
  ```
  (depth-first .)
  ```
- **breadth-first** : Turns breadth-first search on.
  ```
  (breadth-first .)
  ```
- **stop** : In case the interpreter has entered in some kind of infinite loop, you can stop the current computation by pressing `Control-C`.
- **reboot** : Reboots the system.
  ```
  (reboot .)
  ```
- **quit** : Quits the system.
  ```
  quit
  ```
- **show timing information** :
  ```
  set show loop stats on .
  set show loop timing on .
  ```

An example session

```
./plural.bin
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude alpha95 built: Oct  6 2010 18:12:57
            Copyright 1997-2010 SRI International
                   Tue Feb  8 15:58:49 2011

        Executable Plural Semantics (November 1st, 2010)

Maude> load clerks-sp.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)

Maude> (more .)

Result: p(pepe,laura)

Maude> (more .)

Result: p(pepe,david)

Maude> (more .)

Result: p(maria,pepe)

Maude> (more .)

Result: p(maria,maria)

Maude> (more .)

Result: p(maria,laura)

...

Maude> (more .)

Result: p(david,laura)

Maude> (more .)

Result: p(david,david)

Maude> (more .)

No more results.

Maude> quit .
Bye.
```

Examples

- Clerks example: clerks.plural.
- Odysseus example: dungeon.plural.
- **NEW**: Examples for the combination of singular and plural arguments, see [6].
  - clerks-sp.plural.
  - dungeon-sp.plural.
  - exams.plural.

Other resources

References

1. ^ Juan Rodríguez-Hortalá, A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems, In Proc. FSTTCS'08, 2008, http://drops.dagstuhl.de/opus/frontdoor.php?source_opus=1764
2. ^ Juan Rodríguez-Hortalá, A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems (Extended version), Technical report SIC-10-08, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2008, http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-10-08.pdf
3. ^ M. Rodríguez- Artalejo, J. C. González-Moreno, T. Hortalá-González, F. Lopez-Fraguas, An approach to declarative programming based on a rewriting logic, J. Log. Program., 40(1), 1999, 40(1), pages 47-87
4. ^ F. Baader and T. Nipkow, United Kingdom, Term Rewriting and All That, Cambridge University Press, 1998
5. ^ Adrián Riesco and Juan Rodríguez-Hortalá, A natural implementation of Plural Semantics in Maude, In Proc. LDTA'09, ENCTS 253(7), 2009, http://dx.doi.org/10.1016/j.entcs.2010.08.039
6. ^ Adrián Riesco and Juan Rodríguez-Hortalá, Programming with singular and plural non-deterministic functions, In Proc. PEPM'10, 2010, http://doi.acm.org/10.1145/1706356.1706373
7. ^ S. Escobar, Re?ning weakly outermost-needed rewriting and narrowing, ACM, In Proc.PPDP'03, 2003, pages 113?123
8. ^ M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. L. Talcott, All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, 4350, Springer, 2007, 4350

Contact

This sofware is developed by Adrián Riesco Rodríguez

Error: Macro Image(adri_mail.png) failed

```
sequence item 0: expected string, NoneType found
```

and Juan Rodríguez Hortalá, at the Department of Computer Systems and Computing, Universidad Complutense de Madrid, Spain.