Programming with Singular and Plural Non-deterministic Functions

Adrián Riesco
Juan Rodríguez Hortalá

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Final EuroProofNet Symposium - Orly, France
September 2025

The Ingredients

Functional programming

+

Laziness

+

Non-determinism

Programs

$$heads(x:y:xs) \to (x,y)$$
$$repeat(x) \to x:repeat(x)$$
$$coin \to 0 \quad coin \to 1$$

Expressions & Values

$$heads(0:1:repeat(2)) \rightharpoonup (0,1)$$
$$coin \rightharpoonup 0$$
$$coin \rightharpoonup 1$$

The Ingredients

Functional programming

$+$

Laziness

$+$

Non-determinism

Programs

$$heads(x:y:xs) \to (x, y)$$
$$repeat(x) \to x:repeat(x)$$
$$coin \to 0 \quad coin \to 1$$

Expressions & Values

$$heads(0:1:repeat(2)) \twoheadrightarrow (0,1)$$
$$coin \twoheadrightarrow 0$$
$$coin \twoheadrightarrow 1$$

Functional Logic Programming (FLP) - Toy, Curry

(Constructor-based) Term Rewriting Systems (TRS) - Maude

The Decision

Laziness
+
Non-determinism

$\Longrightarrow$ Semantic alternatives

The Decision : Laziness + Non-determinism $\Longrightarrow$ Semantic alternatives

“Operational” perspective

When is it time to fix a
(partial) value for each argument?

$heads(x:y:xs) \rightarrow (x,y), repeat(x) \rightarrow x:repeat(x), coin \rightarrow 0, coin \rightarrow 1$

The Decision : Laziness + Non-determinism $\Longrightarrow$ Semantic alternatives

"Operational" perspective

When is it time to fix a (partial) value for each argument?

$heads(x:y:xs) \to (x,y), repeat(x) \to x:repeat(x), coin \to 0, coin \to 1$

Call-time choice $\Longleftarrow$ FLP

On parameter passing

$heads(repeat(\underline{coin})) \to$

$heads(\underline{repeat(0)}) \to^*$

$\underline{heads(0:0:\perp)}\ \to (0,0)$

$\not\to^* (0,1)$

Rewriting + Sharing

vs.

Run-time choice $\Longleftarrow$ TRS

As they are used

$heads(\underline{repeat(coin)}) \to^*$

$\underline{heads(coin:coin:repeat(coin))}$

$\to (\underline{coin}, \underline{coin}) \to^* (0,0)$

$\to^* (0,1)$

Rewriting

Previously. . .

Grammars as term rewriting systems

Two standard grammar operators

Alternative: $X \mid Y \to X \qquad X \mid Y \to Y$

Kleene's star: $star(X) \to \epsilon \mid X \; ++ \; star(X)$

With them

$letter \to a \mid b \mid \ldots \mid z$

$word \to star(letter)$

Previously. . .

Grammars as term rewriting systems

Two standard grammar operators

Alternative: $X \mid Y \to X \qquad X \mid Y \to Y$

Kleene's star: $star(X) \to \epsilon \mid X \;{+}{+}\; star(X)$

With them

$letter \to a \mid b \mid \ldots \mid z$

$word \to star(letter)$

word only works if *star* is evaluated under run-time choice:

$$word \to star(letter) \to letter \;{+}{+}\; star(letter) \to^* aaa$$
$$\to^* abc$$

Previously. . .

Grammars as term rewriting systems

Two standard grammar operators

Alternative: $X \mid Y \to X \qquad X \mid Y \to Y$

Kleene's star: $star(X) \to \epsilon \mid X \ ++\ star(X)$

With them

$letter \to a \mid b \mid \ldots \mid z$

$word \to star(letter)$

word only works if *star* is evaluated under run-time choice:

$$word \to star(letter) \to letter\ ++\ star(letter) \to^{*} aaa$$
$$\to^{*} abc$$

Palindromes (even length)

$palindrome \to palAux(word) \quad palAux(X) \to X\ ++\ reverse(X)$

Previously...

Grammars as term rewriting systems

Two standard grammar operators

Alternative: $X \mid Y \to X \quad X \mid Y \to Y$

Kleene's star: $star(X) \to \epsilon \mid X ++ star(X)$

With them

$letter \to a \mid b \mid \ldots \mid z$

$word \to star(letter)$

word only works if *star* is evaluated under run-time choice:

$$word \to star(letter) \to letter ++ star(letter) \to^* aaa$$
$$\to^* abc$$

Palindromes (even length)

$palindrome \to palAux(word) \quad palAux(X) \to X ++ reverse(X)$

palindrome only works if *palAux* is evaluated under call-time choice:

$$palindrome \to palAux(word) \to word ++ reverse(word) \to^* abba$$
$$\not\to^* oops$$

Previously...

Moral

No single semantics
for non-determinism is adequate
for all cases

The Decision : Laziness + Non-determinism $\Longrightarrow$ Semantic alternatives

Denotational perspective

Which domain is used to
instantiate the program rules?

$heads(x:y:xs) \rightarrow (x,y), repeat(x) \rightarrow x:repeat(x), coin \rightarrow 0, coin \rightarrow 1$

The Decision : Laziness + Non-determinism $\Longrightarrow$ Semantic alternatives

Denotational perspective

Which domain is used to instantiate the program rules?

$heads(x : y : xs) \rightarrow (x, y), repeat(x) \rightarrow x : repeat(x), coin \rightarrow 0, coin \rightarrow 1$

Singular semantics $\Longleftarrow$ FLP

Variables go to values

$heads(repeat(\underline{coin})) \rightarrow$

$heads(\underline{repeat(0)}) \rightarrow^*$

$\underline{heads(0 : 0 : \perp)} \rightarrow (0, 0)$

$\not\rightarrow^* (0, 1)$

vs.

Plural semantics $\Longleftarrow$??? TRS

Variables go to sets of values

$heads(repeat(\underline{coin})) \rightarrow$

$heads(\underline{repeat(\{0, 1\})}) \rightarrow^*$

$\underline{heads(\{0 : 1 : \perp, 1 : 0 : \perp,}$

$\underline{0 : 0 : \perp, 1 : 1 : \perp\})}$

$\rightarrow \{(0, 0), (0, 1), (1, 0), (1, 1)\}$

The Mistake

The Folklore $\left\{ \text{Call-time choice} \equiv \text{Singular semantics} \quad \checkmark \right.$

The Mistake

The Folklore $\begin{cases} \text{Call-time choice} \equiv \text{Singular semantics} & \checkmark \\ \text{Run-time choice} \equiv \text{Plural semantics} & ✘ \end{cases}$

$$f(c(x)) \to (x, x), \;\; x \mathbin{?} y \to x, \;\; x \mathbin{?} y \to y$$

The Mistake

The Folklore $\begin{cases} \text{Call-time choice} \equiv \text{Singular semantics} & \checkmark \\ \text{Run-time choice} \equiv \text{Plural semantics} & \text{✘} \end{cases}$

$$f(c(x)) \to (x,x),\ \ x\ ?\ y \to x,\ \ x\ ?\ y \to y$$

Run-time choice

Argument values are fixed as they are used

$$\begin{aligned} f(\underline{c(0)?c(1)}) &\to \underline{f(c(0))} \to (0,0) \\ &\to \underline{f(c(1))} \to (1,1) \end{aligned}$$

vs.

Plural semantics

Variables go to sets of values

$$f(\underline{c(0)?c(1)}) \to f(\{c(0),c(1)\})$$

$$\begin{aligned} \to (\{0,1\},\{0,1\}) &\to^* (0,0) \\ &\to^* (0,1) \\ &\to^* (1,0) \\ &\to^* (1,1) \end{aligned}$$

The Mistake

The Folklore
- Call-time choice ≡ Singular semantics ✓
- Run-time choice ≡ Plural semantics ✗

$$f(c(x)) \to (x,x),\ \ x \mathbin{?} y \to x,\ \ x \mathbin{?} y \to y$$

Run-time choice ⟸ TRS

Argument values are fixed as they are used

$$f(\underline{c(0)?c(1)}) \to \underline{f(c(0))} \to (0,0)$$
$$\to \underline{f(c(1))} \to (1,1)$$

vs.

Plural semantics

Variables go to sets of values

$$f(\underline{c(0)?c(1)}) \to f(\{c(0), c(1)\})$$
$$\to (\{0,1\},\{0,1\}) \to^* (0,0)$$
$$\to^* (0,1)$$
$$\to^* (1,0)$$
$$\to^* (1,1)$$

The Mistake

The Folklore
- Call-time choice ≡ Singular semantics ✓
- Run-time choice ≡ Plural semantics ✘

$$f(c(x)) \to (x,x),\ \ x \ ?\ y \to x,\ \ x \ ?\ y \to y$$

Run-time choice $\Longleftarrow$ TRS

Argument values are fixed as they are used

$$f(\underline{c(0)?c(1)}) \to \underline{f(c(0))} \to (0,0)$$
$$\to \underline{f(c(1))} \to (1,1)$$

vs.

Plural semantics $\not\Longleftarrow$ TRS

Variables go to sets of values

$$f(\underline{c(0)?c(1)}) \to f(\overline{\{c(0),c(1)\}})$$
$$\to (\{0,1\},\{0,1\}) \to^* (0,0)$$
$$\to^* (0,1)$$
$$\to^* (1,0)$$
$$\to^* (1,1)$$

Run-time choice ≠ Plural semantics

Compositionality

A desirable property ...

Compositionality : Exps with the same values are interchangeable

$$[\![e]\!] = [\![e']\!] \Leftrightarrow [\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$$

Compositionality

A desirable property ...

Compositionality : Exps with the same values are interchangeable

$$[\![e]\!] = [\![e']\!] \Leftrightarrow [\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$$

$$[\![c(0?1)]\!] = [\![c(0)?c(1)]\!] = \{c(0), c(1)\}$$

but with run-time choice, under $\{f(c(x)) \rightarrow (x, x),\ x \ ?\ y \rightarrow x,\ x \ ?\ y \rightarrow y\}$

$$f(c(0?1)) \rightarrow (0?1, 0?1) \rightarrow^* (0, 1) \not\leftarrow^* f(c(0)?c(1))$$

... becomes fundamental in a value-based language

Compositionality

A desirable property ...

Compositionality : Exps with the same values are interchangeable

$$[\![e]\!] = [\![e']\!] \Leftrightarrow [\![\mathcal{C}[e]]\!] = [\![\mathcal{C}[e']]\!]$$

$$[\![c(0?1)]\!] = [\![c(0)?c(1)]\!] = \{c(0), c(1)\}$$

but with run-time choice, under $\{f(c(x)) \to (x, x),\ x \mathbin{?} y \to x,\ x \mathbin{?} y \to y\}$

$$f(c(0?1)) \to (0?1, 0?1) \to^* (0, 1) \not\leftarrow^* f(c(0)?c(1))$$

... becomes fundamental in a value-based language

Philosophy : *"All I know about an expression is its set of values"*

- plural and singular are compositional $\Rightarrow$ good for value-based langs
- run-time choice $\Rightarrow$ good for other langs and purposes

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural
    - a function is plural or singular if each of its arguments is
    - in the previous program: `star is plural`
      `palAux is singular`

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural
    - a function is plural or singular if each of its arguments is
  - a logic calculus formalizes the intended semantics

resulting framework generalizes both alternatives
preserves compositionality

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural
    - a function is plural or singular if each of its arguments is
  - a logic calculus formalizes the intended semantics

    resulting framework generalizes both alternatives

    preserves compositionality
  - programs transformed to a core language according to annotations

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural
    - a function is plural or singular if each of its arguments is
  - a logic calculus formalizes the intended semantics

    resulting framework generalizes both alternatives
    preserves compositionality
  - programs transformed to a core language according to annotations
- Main goal: exploring the expressive capabilities of this combination

This Work

Combining singular + plural non-determinism

- In the same language
  - function arguments annotated as singular or plural
    - a function is plural or singular if each of its arguments is
  - a logic calculus formalizes the intended semantics

    resulting framework generalizes both alternatives

    preserves compositionality
  - programs transformed to a core language according to annotations
- Main goal: exploring the expressive capabilities of this combination
- Prototype: `https://github.com/ariesco/Plural-semantics`

The Semantics

The Semantics: $CRWL_\pi^\sigma$

B $\dfrac{}{e \rightarrow \perp}$ **RR** $\dfrac{}{x \rightarrow x}$ $x \in \mathcal{V}$

DC $\dfrac{e_1 \rightarrow t_1 \ \ \ldots \ \ e_n \rightarrow t_n}{c(e_1, \ldots, e_n) \rightarrow c(t_1, \ldots, t_n)}$ $c \in CS^n, \ t_i \in CTerm_\perp$

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

$$(f(\overline{p}) \rightarrow r) \in \mathcal{P}, \ \theta = ?\{\theta_{11}, \ldots, \theta_{1m_1}\} \uplus \ldots \uplus \ ?\{\theta_{n1}, \ldots, \theta_{nm_n}\}$$
$$\forall i, j. \ \theta_{ij} \in CSubst_\perp \wedge dom(\theta_{ij}) = var(p_i)$$
$$\forall i. \ m_i > 0, \ \forall i \in sgArgs(f). \ m_i = 1$$

The Semantics: $CRWL_\pi^\sigma$

Theorem (Compositionality)

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

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

"all I know about an expression is its set of values"

The Semantics: $CRWL_\pi^\sigma$

Theorem (Compositionality)

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

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

"all I know about an expression is its set of values"

Theorem (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* $[\![e]\!]^{\mathcal{P}}_{CRWL_\pi^\sigma} = [\![e]\!]^{\mathcal{P}}_{\pi CRWL}$.

$[\![e]\!]^{\mathcal{P}}_{CRWL_\pi^\sigma}$, $[\![e]\!]^{\mathcal{P}}_{CRWL}$ *and* $[\![e]\!]^{\mathcal{P}}_{\pi CRWL}$: *denotations for e under* $\mathcal{P}$ *given by* $CRWL_\pi^\sigma$, *singular and plural semantics*

Using $CRWL^{\sigma}_{\pi}$

Clerks

Performing a search in the database of a company

- System predefined functions: `tt` is "true" and `ff` is "false"

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

`_?_` and `if_then_` are plural for flexibility $\Leftarrow$ singularity is sticky : *once you fix a value it remains fixed*

- Different branches defined using `?` $\sim$ set union operator

```
branches -> madrid ? vigo ? badajoz .
employees(madrid) -> e(john, men, clerk) ? e(larry, men, boss) .
employees(vigo) -> ...
...
```

  - plurarity doesn't matter for ground arguments or no arguments
  - functions are singular by default

Clerks

- Enumerating the employees

```
Maude> (eval employees(branches) .)
Result: e(john,men,clerk)
Maude> (more .)
Result: e(larry,men,boss)
...
```

- Looking for two clerks

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

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

  - Ok, but we want
    - two different clerks
    - generalize it to any number of clerks

Clerks

- Adding an element to a list ensuring that the remaining elements are different

```
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 .
...
```

  - No disequality constraints $\rightsquigarrow$ ground version with program rules
  - Tests like `newIns`, `diffL`, `neq` $\Longrightarrow$ singularity

- Generating lists of different values for an expression

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

  - Combination of plural (`vals`) and singular (`newIns` $\Rightarrow$ tests)

Clerks

- Generating a number of different values for an expression

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

  - sp $\Longrightarrow$ { `N` is singular : fixed number; `E` is plural : different values }
  - Simulation of meta primitives of call-time choice: `collect`, `findall`

- Looking for a number of different clerks

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

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

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

Rule of thumb

singular arguments fix their the values
plural arguments represent sets of values

Dungeon

Ulysses has been captured, he wants to cheat his guardians using a bottomless bag of gold $\rightsquigarrow$ he interchanges items and information with his guardians in order to obtain the key of its jail

- Interchanging items and information

```
ask is sp .
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 .
```

  - sp $\Longrightarrow$ fix a guardian, offer several items

- Next step in Ulysses' path to freedom: several items and their single provider

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

Dungeon

- Finding the path for freedom

```
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 .
```

`discoverHow`

- returns what I had: `T`
- or performs an interchange and iterates the process

last ? T allows to use
items obtained in different recursive calls
for the same interchange

Dungeon

- Starting the search

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

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

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

Implementation

Two Transformations

From plural to run-time

- Neither run-time can simulate call-time nor vice versa

Two Transformations

From plural to run-time

- Neither run-time can simulate call-time nor vice versa
- But run-time simulates plural easily: just postpone pattern matching

Example

$f(c(x)) \rightarrow (x, x) \quad \Longrightarrow \quad f(y) \rightarrow if\ \ match(y)\ then\ (project(y), project(y)),$
$match(c(x)) \rightarrow true,\ \ project(c(x)) \rightarrow x$

Two Transformations

Putting singular/call-time inside run-time

Main idea

start from a run-time choice environment (pure rewriting)

$+$

add a *let* primitive for sharing

$$LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$$

Two Transformations

Putting singular/call-time inside run-time

Main idea

start from a run-time choice environment (pure rewriting)

$+$

add a *let* primitive for sharing

$$LExp \ni e ::= X \mid h(e_1, \ldots, e_n) \mid let\ X = e_1\ in\ e_2$$

Intended meaning

In the reduction of $let\ X = e_1\ in\ e_2$ all the occurrences of X in e_2 share the value produced by e_1

Example

$$\begin{array}{ll} let\ X = 0\ ?\ 1\ in\ (X, X) & \to^* (0, 0) \\ & \not\to^* (0, 1) \end{array}$$

Transformation

Introduce a *let* binding for each variable in a singular argument

The Maude System

- Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation
  - Maude modules correspond to specifications in rewriting logic
  - In particular it can be used to implement term rewriting systems ≡ run-time choice
- A key distinguishing feature of Maude is its systematic and efficient use of reflection
  - It allows many advanced metaprogramming and metalanguage applications
  - Maude also provides modules to specify input/output interactions with the user

Our program transformations, its execution—including the implementation of natural rewriting and the operational semantics—, and the user interactions are implemented in Maude itself

Conclusions

The Contributions

- A formal framework for programming with non-deterministic functions
  - Allows the combination of singular and plural non-determinism
  - A safe extension of both options
  - Preserves compositionality
- Have explored the expressive capabilities of this combination
  - Several examples have been presented (more in the paper)
  - A Maude based prototype has been developed
- Use of plural
  - Mainstream approaches to FLP only support singular/call-time
  - Previous mixes employed run-time choice $\neq$ plural

The Future

- Extensions
  - Equality and disequality constraints
  - Higher order capabilities
    - Generic `discover` function
    - Face the challenges implementing type classes in FLP
  - Matching modulo

- Understand programs better
  - Equivalence of annotations $\rightsquigarrow$ determinism analysis
  - Equational laws for non-determinism

- Some kind of sharing of sets of values is needed to improve efficiency

Try it!!!

https://github.com/ariesco/Plural-semantics