INSTITUT FÜR INFORMATIK

Adding Plural Arguments to Curry Programs

Michael Hanus

Bericht Nr. 1304
Juni 2013
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Adding Plural Arguments to Curry Programs

Michael Hanus

Bericht Nr. 1304
Juni 2013
ISSN 2192-6247

e-mail: mh@informatik.uni-kiel.de

Adding Plural Arguments to Curry Programs

Michael Hanus
Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Technical Report 1304, June 2013

Abstract

Functional logic languages combine lazy (demand-driven) evaluation strategies from functional programming with non-deterministic computations from logic programming. To provide a strategy-independent semantics, most languages are based on the call-time choice semantics where parameters are passed as values. From an implementation point of view, the call-time choice semantics fits well with sharing performed by lazy languages. On the other hand, there are also situations where it is intended to pass non-deterministic arguments as sets of values in order to exploit the power of non-deterministic programming. This alternative parameter passing model is known under the name "plural" arguments. In this paper, we show how both mechanisms can be integrated in a single language. In particular, we present a novel technique to implement plural arguments in a call-time choice language so that existing implementations of contemporary functional logic languages can be easily re-used to implement plural parameter passing.

1 Motivation

Functional logic languages support the most important features of functional and logic programming in a single language (see [7, 23] for recent surveys). They provide higher-order functions and demand-driven evaluation from functional programming as well as logic programming features like non-deterministic search and computing with partial information (logic variables). This combination led to new design patterns [5, 8], better abstractions for application programming (e.g., programming with databases [10, 16], GUI programming [20], web programming [21, 22, 25], string parsing [12]), and new techniques to implement programming tools, like partial evaluators [1] or test case generators [17, 36].

The execution model of contemporary functional logic languages, like Curry [26] or TOY [28], is based on (some variant of) needed narrowing [4] which subsumes demand-driven term rewriting, used to evaluate functional programs, and unification and resolution applied in logic programming. Needed narrowing is an optimal evaluation strategy for

large classes of programs. Moreover, operations in functional logic programs can be also non-deterministic, i.e., deliver more than one result on a given (ground) input, like the predefined *choice* operation, denoted by the infix operator "`?`":

```
x ? _ = x
_ ? y = y
```

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the Curry program[1]

(1)

Here, `C` is a data constructor so that the expression "`f (C 0)`" evaluates to the pair `(0,0)`. However, the intended semantics becomes less clear when non-deterministic operations occur as arguments. For instance, what should be the intended results of "`f (C (0?1))`"? Hussmann [27] proposed two options:

Call-time choice semantics: The value of each argument is fixed before parameter passing. In our case, the parameter `(C (0?1))` has the two values `(C 0)` and `(C 1)` so that the call to `f` has also two results: `(0,0)` and `(1,1)`.

Run-time choice semantics: Values are computed when they are needed. Hence, the parameter `(C (0?1))` is not evaluated before parameter passing but copied into the right-hand side so that the call to `f` reduces to the expression `(0?1,0?1)` which subsequently evaluates to four results: `(0,0)`, `(1,0)`, `(0,1)`, and `(1,1)`.

Since the computed results of a run-time choice semantics might depend on the evaluation strategy (e.g., the previous example call would not produce the result `(1,0)` if it is evaluated with an innermost reduction strategy), contemporary functional logic languages, like Curry or TOY, are based on the call-time choice semantics. Note that this semantics does not exclude the demand-driven evaluation of arguments. Actually, it fits well with a lazy evaluation strategy where actual arguments are shared instead of duplicated. A logical (execution- and strategy-independent) foundation for the call-time choice semantics where programs contain non-strict and non-deterministic operations is defined in [18] by the rewriting logic CRWL.

Beyond this operational view of parameter passing, there is also denotational view of parameters [35]:

Singular semantics: Parameter variables denote single values. This is equivalent to call-time choice.

Plural semantics: A parameter variable denotes a set of values, i.e., the set of all results when the parameter is evaluated. Although one might have the impression that this

[1]The syntax of Curry is close to Haskell [31], i.e., variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

corresponds to run-time choice, Rodríguez-Hortalá [35] showed that this is not the case when pattern matching is taken into account. For instance, consider the expression "`f (C 0 ? C 1)`". Since an application of the defining rule for `f` demands for the constructor `C`, the argument `(C 0 ? C 1)` must always be evaluated before applying the `f`-rule. Hence, run-time choice cannot yield the result "`(0,1)`" for this expression. However, a plural semantics specifies that the value of the argument is the set {`C 0, C 1`} so that the parameter variable `x` denotes the set {`0,1`}. As a consequence, "`(0,1)`" is a possible value of the initial expression.

[35] proposed a strategy-independent definition of the plural semantics for non-strict and non-deterministic operations in the form of a "plural rewriting logic" πCRWL. He also showed that there is actually a semantical hierarchy w.r.t. the sets of computed results: all results of a call-time choice semantics are contained in the results of a term-rewriting semantics (which corresponds to run-time choice) which are again contained in the results of a plural semantics. Due to its strategy-independent definition, the plural semantics is an interesting model for programming, in particular, if singular and plural functions or arguments are combined [34]. Such a combination is interesting since it has already been argued in [30] that there are situations in practice where there is no clear preference to either of these options for treating non-determinism.

Since implementations of functional logic languages are based on lazy evaluation and sharing, which fits well with the call-time choice semantics, the implementation of plural arguments or their combination with singular arguments is less clear. [33] developed an implementation of plural arguments by transforming functional logic programs into rewrite rules implementing πCRWL with the Maude system [13].

In this paper, we present a novel implementation technique for plural arguments by transforming them in such a way that their execution with call-time choice produces the intended results. Thus, we can re-use existing implementations of functional logic languages. This does not only ease the implementation efforts but also leads to much more efficient and comprehensive implementations.

In the next section, we sketch the relevant foundations of functional logic programming and Curry. Section 3 reviews the plural semantics and shows some programming examples. Section 4 presents our transformation to implement plural functions with a call-time choice semantics. The correctness of this transformation is shown in Section 5. We sketch an implementation and show its superiority by some benchmarks in Section 6 before we conclude in Section 7.

2 Functional Logic Programming and Curry

The declarative multi-paradigm language Curry [26] combines features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, constraints). A Curry program consists of definitions of data types enumerating their *constructors* and of *operations* or *defined functions* on these types. A functional logic com-

putation reduces an expression to some value, if possible, where a *value* is an expression without defined operations. For instance, `0` and `1` are the values obtained by evaluating the expression `(0?1)`.

The concrete syntax of Curry is close to Haskell but, in addition, allows non-deterministic operations (like "`?`") and free (logic) variables in conditions and right-hand sides of defining rules. Actually, non-deterministic operations and logic variables have the same expressive power [6, 14]. For instance, a Boolean logic variable can be replaced by the non-deterministic *generator* operation for Booleans defined by

```
aBool = True ? False
```

Exploiting this equivalence, one can implement Curry by translation into Haskell augmented with a mechanism to handle non-deterministic computations, as shown recently with the KiCS2 system [11]. Note that call-time choice and sharing is important for this equivalence since different occurrences of the same logic variable should denote the same value. Although the source language Curry allows the explicit introduction (by "`where x,y free`") and use of logic variables, we assume in the theoretical part of this paper that they are replaced by generator operations.

A precise definition of call-time choice is proposed in [18] by the rewriting logic CRWL. In order to present this logic, we briefly recall some notions and notations of term rewriting [9, 15].[2] All symbols used in a program must be either *variables* from a set $\mathcal{V}$ or symbols from a *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. The set Exp of *expressions* consists of variables or signature symbols applied to a list of expressions (also called *application*). $\mathcal{V}ar(e)$ denotes the set of variables in an expression e. An expression e is called *ground* if $\mathcal{V}ar(e) = \varnothing$. A *value* belongs to the set $CTerm$ of *constructor terms*, i.e., expressions without defined function symbols. A *program* $\mathcal{P}$ is a set of *rules* of the form $f(t_1, \ldots, t_n) \to e$ where $f \in \mathcal{F}$, $t_1, \ldots, t_n \in CTerm$, $e \in Exp$, and the patterns $t_1, \ldots, t_n$ must not contain multiple occurrences of a same variable. We ignore conditions in the rules since a conditional rule l `|` c `=` r can be translated into the unconditional rule l `=` `cond` c r where the predefined operation `cond` reduces to its second argument if the first one is true [3], e.g., `cond` could be defined by the rule "`cond True x = x`". Moreover, we omit other constructs of source programs, like extra variables or let expressions, and assume that they are eliminated by some program transformation (although we use them in concrete example programs).

A *substitution* $\sigma \in Sub$ is a finite mapping $\sigma : \mathcal{V} \to Exp$ which is homomorphically extended to a mapping $\sigma : Exp \to Exp$. The *domain* of a substitution σ is defined by $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. If $\mathcal{D}om(\sigma_1) \cap \mathcal{D}om(\sigma_2) = \varnothing$, then their *disjoint union* $\sigma_1 \uplus \sigma_2$ is defined by $(\sigma_1 \uplus \sigma_2)(x) = \sigma_i(x)$, if $x \in \mathcal{D}om(\sigma_i)$ for some $i \in \{1, 2\}$, and $(\sigma_1 \uplus \sigma_2)(x) = x$, otherwise. A *C-substitution* $\sigma \in CSub$ satisfies $\sigma(x) \in CTerm$ for all $x \in \mathcal{D}om(\sigma)$.

A *position* p in an expression e could be represented by a sequence of natural numbers. Positions are used to identify specific subterms. Thus, $e|_p$ denotes the *subterm* of e at

[2] Although the theoretical part uses notations from term rewriting, its mapping into the concrete syntax of Curry should be obvious.

$$\text{RR}\frac{}{x \twoheadrightarrow x} \quad x \in \mathcal{V} \qquad \text{DC}\frac{e_1 \twoheadrightarrow t_1 \ \cdots \ e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)} \quad c \in \mathcal{C}$$

$$\text{B}\frac{}{e \twoheadrightarrow \bot} \qquad \text{OR}\frac{e_1 \twoheadrightarrow \sigma(t_1) \ \cdots \ e_n \twoheadrightarrow \sigma(t_n) \quad \sigma(r) \twoheadrightarrow t}{f(e_1,\ldots,e_n) \twoheadrightarrow t} \ \begin{array}{l} f(t_1,\ldots,t_n) \to r \in \mathcal{P} \\ \sigma \in CSub_\bot \end{array}$$

Figure 1: The call-time choice semantics CRWL

$$\text{RR}\frac{}{x \twoheadrightarrow x} \quad x \in \mathcal{V} \qquad \text{DC}\frac{e_1 \twoheadrightarrow t_1 \ \cdots \ e_n \twoheadrightarrow t_n}{c(e_1,\ldots,e_n) \twoheadrightarrow c(t_1,\ldots,t_n)} \quad c \in \mathcal{C}$$

$$\text{B}\frac{}{e \twoheadrightarrow \bot} \qquad \text{POR}\frac{\begin{array}{cccc} e_1 & \twoheadrightarrow & e_n & \twoheadrightarrow \\ \sigma_{11}(t_1) & & \sigma_{n1}(t_n) & \\ \vdots & \cdots & \vdots & \sigma(r) \twoheadrightarrow t \\ e_1 & \twoheadrightarrow & e_n & \twoheadrightarrow \\ \sigma_{1m_1}(t_1) & & \sigma_{nm_n}(t_n) & \end{array}}{f(e_1,\ldots,e_n) \twoheadrightarrow t}$$

$f(t_1,\ldots,t_n) \to r \in \mathcal{P}$, $\sigma_{ij} \in CSub_\bot$, $dom(\sigma_{ij}) = \mathcal{V}ar(t_i)$
$\sigma = ?\{\sigma_{11},\ldots,\sigma_{1m_1}\} \uplus \ldots \uplus ?\{\sigma_{n1},\ldots,\sigma_{nm_n}\}$, $m_i > 0$

Figure 2: The plural semantics πCRWL

position p, and $e[s]_p$ denotes the result of *replacing the subterm* $e|_p$ with the expression s (see [15] for details). The set of all positions of an expression e is denoted by $\mathcal{P}os(e)$.

If $\mathcal{P}$ is a program, then a *rewrite step* $e \to_\mathcal{P} e'$ is defined if there are a position p in e, a rule $l \to r \in \mathcal{P}$, and a substitution σ with $e|_p = \sigma(l)$ such that $e' = e[\sigma(r)]_p$. We denote by $\overset{*}{\to}_\mathcal{P}$ the reflexive and transitive closure of $\to_\mathcal{P}$, and we write $\mathcal{P} \vdash e \overset{*}{\to} t$ if $e \overset{*}{\to}_\mathcal{P} t$.

In order to define the meaning of call-time choice by the rewriting logic CRWL, we extend the standard signature with the new constructor symbol $\bot$ to represent *undefined or unevaluated values*. The set $Exp_\bot$ of *partial expressions* consists of all expressions that might contain occurrences of $\bot$. The sets $CTerm_\bot$ and $CSub_\bot$ are similarly defined. CRWL defines the deduction of *approximation statements* $e \twoheadrightarrow t$ with the intended meaning "the partial constructor term t approximates the value of e." The inference rules defining such statements are summarized in Fig. 1. Rule B specifies that $\bot$ approximates any expression to get a non-strict semantics. Rule DC decomposes constructor-rooted expressions in order to process their argument expressions. Rule OR expresses call-time choice by passing only partial constructor terms as parameters (by the substitution σ). We write $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ if $e \twoheadrightarrow t$ is derivable with the CRWL inference rules.

3 Plural Semantics and Plural Arguments

In this section we review the plural semantics and discuss our proposed extension to support plural arguments in Curry. The formal definition of the approximation relation of the plural semantics πCRWL [35] is shown in Fig. 2. The only difference to the calculus CRWL is the replacement of rule OR by POR (Plural Outer Reduction). In contrast to rule OR used to specify call-time choice, rule POR passes all non-deterministic values of an argument e_i into the right-hand side r via the substitution σ. In order to avoid the explicit introduction of sets of values, the πCRWL calculus allows that variables are mapped into disjunctive values and $?\{\theta_1, \ldots, \theta_n\}$ denotes the substitution which combines the different substitutions $\theta_1, \ldots, \theta_n$ for the same variable into one substitution with disjunctive values (see [35] for detailed definitions). For instance, if $\theta_1(x) = 1$ and $\theta_2(x) = 2$, then $(?\{\theta_1, \theta_2\})(x) = 1?2$. By this mechanism, all non-deterministic values of a parameter variable are available in each occurrence of this variable in the right-hand side. We write $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ if $e \twoheadrightarrow t$ is derivable with the πCRWL inference rules.

For instance, consider again program rule (1) of Section 1. Then rule POR states that `f (C (0?1))` $\twoheadrightarrow t$ holds if `(0?1,0?1)` $\twoheadrightarrow t$ holds (with $\sigma($`x`$) =$ `0?1`). Using the rules for "`?`", we can further deduce that the latter approximation statement holds for the values $t \in \{$`(0,0)`, `(1,0)`, `(0,1)`, `(1,1)`$\}$.

In the following we will discuss how we support plural arguments in a Curry program. It has been argued in [34] that there should not be a choice between a plural or singular *program* but it is more adequate to support a choice for individual *arguments* of operations (since plurality causes an increase of the search space which is intended only in specific situations). Conceptually, the semantics of individual plural arguments can be specified by a combined OR/POR rule where disjunctive values are only passed for the plural arguments. We follow this reasonable design decision and *explicitly mark plural arguments*, i.e., as the default all arguments are singular. For instance, consider the example of Section 1 but now extended with its type definition:

```
data C = C Int

f :: C  → (Int,Int)
f (C x) = (x,x)
```

This is a valid Curry program. Since the call-time choice semantics is the default, the expression "`f (C (0?1))`" evaluates only to the two values `(0,0)` and `(1,1)`. If the programmer wants to change this intended semantics and use plural parameter passing for the argument of `f`, the argument has to be marked as plural. In order to avoid the introduction of specific syntactic constructs for this case (as done in [34]) and to make our implementation available for standard Curry implementations, we mark a plural argument by simply wrapping its type with the type constructor `Plural`:

```
f :: Plural C  → (Int,Int)
f (C x) = (x,x)
```

No other change is necessary and this is again a valid Curry program (after importing

the library `Plural` which contains the definition of the new type constructor). As we will discuss in Section 4, the plural semantics can be implemented by a transformation of the source program (which could be attached as a preprocessor to the compiler). Hence, if we transform and compile the latter program and evaluate the expression "`f (C (0?1))`", we obtain the results `(0,0)`, `(1,0)`, `(0,1)`, and `(1,1)`.

To see another example, consider the parsing of strings, a classical example for both functional and logic languages. It has been shown [12] that functional logic programming provides new opportunities to construct parsers in a natural way. Functional programming is useful to define a parser as a function that consumes some tokens from the list of input tokens and returns the list of remaining tokens:

```
type Parser token  =  [token]  →  [token]
```

Hence, the `empty` parser does not consume a token and the `terminal` parser consumes only a token when it is identical to the token given as an argument:[3]

```
empty :: Parser t
empty xs = xs

terminal :: t  → Parser t
terminal sym (token:tokens) | sym=:=token = tokens
```

Furthermore, we need operations to combine two parsers as alternatives ("`<|>`") or sequentially ("`<*>`"). The alternative combinator can be easily defined using non-determinism:

```
(<|>)  :: Parser t  → Parser t  → Parser t
p <|> q = \xs  → p xs ? q xs
```

For the sequence combinator, we have to ensure that the second parser is applied to the evaluated output of the first parser. This can be obtained by a condition with an equational constraint:[4]

```
(<*>)    :: Parser t  → Parser t  → Parser t
p1 <*> p2 = \xs  → cond (p1 xs =:= ys) (p2 ys)  where ys free
```

Using such combinators, it is easy to define the parsing of palindromes. Since the notion of a palindrome is independent of the underlying sets of tokens, we parameterize the palindrome parser by this set so that it could have the type

```
pali :: a  → Parser a
```

The type variable `a` should denote a *set* of tokens, e.g., specified by a non-deterministic operation. In order to ensure that each element of this set can be used inside the parser, this argument must be a plural one. Thus, we define our parser as follows:

```
pali :: Plural a  → Parser a
```

[3] "`=:=`" denotes an *equational constraint* which is satisfied if its arguments are reducible to unifiable values.

[4] As usual, the lambda abstraction $\backslash x \to e$ denotes an anonymous function which maps x into e.

```
pali t = empty
     <|> terminal t
     <|> let someT = terminal t
          in someT <*> pali t <*> someT
```

Thus, a palindrome is either empty or a single token, or an inner palindrome enclosed with identical tokens. For instance,

```
pali ('a' ? 'b')
```

recognizes palindromes over the letters `a` and `b`, and

```
pali (0 ? 1 ? 2 ? 3 ? 4 ? 5 ? 6 ? 7 ? 8 ? 9)
```

recognizes palindromes over digits. Note that the plural argument is required here. Otherwise, the parameter variable `t` would always denote the same token in the entire palindrome.

We have not discussed the `let` construct of Curry, since it is the same as in functional languages, i.e., `let` x`=`e `in` e' is the same as the application `(\`$x \to e'$`)` e. Since the standard parameter passing is singular, the two occurrences of `someT` denote the same value, as intended for a palindrome. Thus, the combination of singular and plural arguments supports this generic and concise definition.

Our final example is also related to parsing. In this case, we want to provide a generic definition of numbers w.r.t. different digit domains, e.g., octal, decimal, or hexadecimal numbers. Since the syntax of a number should be defined as a non-empty sequence of digits without leading zeros, the following parser combinator for sequences is useful:

```
star :: Plural (Parser t) → Parser t
star p = empty <|> (p <*> star p)
```

This combinator constructs from a given parser `p` a new parser that accepts (possibly empty) sequences of items accepted by `p`. Note that the argument of `star` is marked as plural since the different occurrences of `p` in the right-hand side could nondeterministically accept different items, as already noted in [30]. Similarly to our previous palindrome parser, a parser for numbers is parameterized over the possible leading digits so that we obtain the following definition:

```
number :: Plural Char → Parser Char
number d = terminal d <*> star (terminal (d ? '0'))
```

Note that the digit `'0'` is added as a further choice for the non-leading digits. To use this number parser, we define the choices of non-zero digits for various numeral systems:

```
octDigit = '1' ? '2' ? '3' ? '4' ? '5' ? '6' ? '7'
decDigit = octDigit ? '8' ? '9'
hexDigit = decDigit ? 'A' ? 'B' ? 'C' ? 'D' ? 'E' ? 'F'
```

Then "`number octDigit`", "`number decDigit`", and "`number hexDigit`" are parsers for octal, decimal, and hexadecimal numbers, respectively. Further examples for program-

ming with plural arguments can be found in [34].

4 Transforming Plural Arguments

In this section we present a source-to-source transformation for plural arguments so that the transformed program can be executed under a call-time choice semantics but produces the results intended by the plural semantics.

As already discussed above, a difference between the plural semantics and run-time choice, i.e., term rewriting, occurs when pattern matching is involved. Therefore, [35] already proposed a program transformation to eliminate this difference in order to use term rewriting to implement plural functions. Since pattern matching usually enforces evaluation before function application, which is not appropriate for plural arguments (compare Section 1), the idea of this transformation is to replace pattern matching by explicit match operations and access occurrences of parameters in the right-hand side by projection functions. Consider again our example rule

```
f (C x) = (x,x)
```

This rule is transformed into the definition

```
f y | match y = (project y, project y)
match (C x) = True
project (C x) = x
```

Thus, non-variable patterns in left-hand sides are replaced by fresh variables and a "`match`" condition corresponding to this pattern, and, for each variable occurring in such a pattern, a new "`project`" operation is introduced so that each variable occurrence in the right-hand side of the original rule is replaced by a call to this "`project`" operation. Now it is easy to see that the example expression "`f (C 0 ? C 1)`" of Section 1 can be reduced to `(0,1)` by rewriting with the transformed program.

This transformation is denoted by pST. Its subsequent definition is adapted from [35]. Let $f(t_1, \ldots, t_n) \to r$ be a program rule with $f \notin \{\texttt{?}, \texttt{cond}\}$. This rule is transformed by

$$pST(f(t_1, \ldots, t_n) \to r) \quad = \quad f(y_1, \ldots, y_n) \to \texttt{cond}(match(y_1, \ldots, y_n), \theta(r))$$

where $y_1, \ldots, y_n$ are fresh variables, $\{x_{i1}, \ldots, x_{ik_i}\} = \mathcal{V}ar(t_i) \cap \mathcal{V}ar(r)$ for each $i \in \{1, \ldots, n\}$, and $match$ and $project_{ij}$ are fresh function names where the rules

$match(t_1, \ldots, t_n) \to \texttt{True}$
$project_{ij}(t_i) \to x_{ij}$

are added to the transformed program. Furthermore, the substitution θ used in the transformation pST is defined by

$$\theta = \{x_{ij} \mapsto project_{ij}(y_i) \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}\}$$

This transformation can be improved by transforming only non-variable non-ground pattern arguments Further details about this optimization can be found in [35].

The following theorem states the equivalence of the plural semantics and term rewriting on the transformed programs:

Theorem 1 ([35]) *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in CTerm$. Then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$ holds if and only if $pST(\mathcal{P}) \vdash e \xrightarrow{*} t$ holds.*

This equivalence is exploited in [33] where an implementation of the plural semantics via term rewriting is developed with the Maude system. In the following, we present an alternative implementation that can be used in existing functional logic language implementations based on call-time choice. This implementation is based on the idea to pass plural arguments unevaluated into the right-hand side of a rule and evaluate them (possibly multiple times) when their values are actually required. The evaluation of an expression can be delayed by moving the expression into the body of a new operation and applying the operation when its value is actually needed (since, even in a call-by-value language, the body of an operation is not evaluated when this operation is passed around as an argument). In functional programming, this technique is known as "thunkification" and used for a different purpose, namely to implement a call-by-name semantics in a call-by-value language, e.g., [2].

In a higher-order language, like Curry, this idea can be easily implemented via lambda abstractions. For instance, consider the rules

```
dup x = (x,x)
main = dup (0?1)
```

In order to pass the argument `(0?1)` unevaluated into the right-hand side of the `dup` rule, we wrap the argument into a lambda abstraction and unwrap it in the right-hand side by applying this lambda abstraction to some value (the unit value `()` chosen here could be replaced by any other constant):

```
dup x = (x (), x ())
main = dup (\_ → (0?1))
```

Since partial applications like lambda abstractions are values in a higher-order language, they are not further evaluated w.r.t. a call-time choice semantics [19]. Hence, there exists the following call-time choice derivation:

```
main → dup (\_ →(0?1)) → ((\_ →(0?1)) (), (\_ →(0?1)) ())
     →* ((0?1), (0?1)) →* (0,1)
```

Note that the result `(0,1)` is intended w.r.t. the plural semantics but could not be computed w.r.t. the call-time choice semantics for the original program.

In order to provide a precise definition of this transformation, we define a mapping *pp* on expressions, rules, and programs. In the following, we denote by $\mathcal{F}$ the set of user-defined functions (i.e., without the `match`/`project` operations introduced by pST and the

predefined operations "?" and `cond`). Any expression is transformed by pp as follows:

$$\begin{array}{rcll} pp(x) & = & x\ \texttt{()} & \text{if } x \in \mathcal{V} \\ pp(f(e_1,\ldots,e_n)) & = & f(\texttt{\_} \to pp(e_1),\ldots,\texttt{\_} \to pp(e_n)) & \text{if } f \in \mathcal{F} \\ pp(g(e_1,\ldots,e_n)) & = & g(pp(e_1),\ldots,pp(e_n)) & \text{if } g \notin \mathcal{F} \cup \mathcal{V} \end{array}$$

Hence, parameter variables are replaced by applications (to the "void" value `()`) and parameters in applications of defined functions are replaced by lambda abstractions. All other applications (e.g., constructors and auxiliary operations) are not modified.

A program rule is transformed by pp as follows:

$$pp(l \to r) \ = \ l \to pp(r)$$

Finally, pp transforms a program by applying pp to each rule defining some function belonging to $\mathcal{F}$, i.e., the auxiliary `match`/`project` operations introduced by pST are not modified by pp.

The complete transformation of a source program with plural semantics into a target program executable with call-time choice consists of applying first the transformation pST followed by the transformation pp. For instance, the example program

```
f (C x) = (x,x)

main = f (C (0?1))
```

is transformed by pST/pp into the final program

```
f y | match (y ()) = (project (y ()), project (y ()))
match (C x) = True
project (C x) = x

main = f (\_ → (C (0?1)))
```

The careful reader might have noticed that pp-transformed programs are not programs as defined above since they contain higher-order constructs like lambda abstractions and higher-order applications. This is only a syntactic problem since these higher-order constructs can be eliminated by "defunctionalization" [32], i.e., mapping higher-order features into first-order definitions [37]. For instance, the transformed higher-order program

```
dup x = (x (), x ())
main = dup (\_ → (0?1))
```

can be considered as syntactic sugar or further transformed into a first-order program by naming all anonymous operations and introducing an explicit `apply` operation:

```
dup x = (apply x (), apply x ())
main = dup CoinFunc

coinFunc _ = (0?1)
apply CoinFunc x = coinFunc x
```

Note that a new constructor (`CoinFunc`) is introduced to represent the lambda abstraction passed as an argument. Thus, substitutions that map variables to lambda abstractions are actually constructor substitutions. This property is important to support the passing of lambda abstractions as parameters with the call-time choice semantics. Thus, we assume that this higher-order elimination is implicitly applied to the transformed programs.

5 Correctness of the Transformation

Our transformation presented in the previous section is correct if the original and the transformed program always compute the same results. Thus, if $\mathcal{P}$ is a program, $e \in Exp$ an expression, and $t \in CTerm$ a value (constructor term), then the following two properties should hold:

Soundness: If $pp(pST(\mathcal{P})) \vdash_{CRWL} pp(e) \twoheadrightarrow t$, then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$.

Completeness: If $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$, then $pp(pST(\mathcal{P})) \vdash_{CRWL} pp(e) \twoheadrightarrow t$.

In the following subsections, we prove these main results of this paper, i.e., a soundness (Theorem 5) and a completeness (Theorem 7) result for the combined pST/pp transformation. Since πCRWL-derivations of the original program and CRWL-derivations of the transformed program have quite different shapes (due to the points where arguments are evaluated), it is unclear how to construct a direct mapping between these kinds of derivations. Therefore, the proof exploits let-rewriting [29] to link the different derivations. Let-rewriting is similar to ordinary rewriting but uses let-expressions to express sharing which is necessary for call-time choice. Thus, soundness is proved by exploiting the completeness of let-rewriting w.r.t. CRWL to construct a let-rewrite derivation from $pp(e)$ to t. This implies the existence of an ordinary rewrite derivation which can be mapped (by induction on the derivation steps) into a rewrite derivation on pST-transformed programs. Then the soundness of pST w.r.t. term rewriting (Theorem 1) ensures the existence of a πCRWL-derivation from e to t. Similarly, the completeness of our transformation can be proved by completeness of pST w.r.t. term rewriting, mapping term rewriting into let-rewriting, and applying a soundness result for let rewriting.

5.1 Soundness

In order to prove the soundness of our transformation, we will show that a rewriting derivation on a pp-transformed program implies the existence of a rewriting derivation on a non-transformed program. A difficulty in proving this correspondence is the fact that a pp-transformed program performs intermediate steps that are not directly related to non-transformed programs. For instance, consider the program $\mathcal{P}$ consisting of the rule

$f(x) \to (x, x)$

This program is transformed by pp into the program $\mathcal{P}' = pp(\mathcal{P})$:

$f(x) \to (x\ (), x\ ())$

Consider the expression $e = f(0?1)$ which is transformed into $pp(e) = f(\backslash\_ \to 0?1)$. Then there are the rewrite steps

$$f(\backslash\_ \to 0?1) \quad \to_{\mathcal{P}'} \quad ((\backslash\_ \to 0?1)\ (), (\backslash\_ \to 0?1)\ ())$$

and

$$f(0?1) \quad \to_{\mathcal{P}} \quad (0?1, 0?1)$$

The results of these steps are not syntactically equal but they are equal if we evaluate the applications of the lambda abstractions that remain after the $\to_{\mathcal{P}'}$-step.

As we already remarked in Section 4, we consider the higher-order features used in the transformation *pp* as syntactic sugar for their first-order equivalent obtained by defunctionalization [32] or Warren's translation [37]. We also call this translation from higher-order into first-order programs (sketched at the end of Section 4) also *HO-FO translation*. Since the generated lambda abstractions and applications are quite simple (only one argument, no pattern matching, no partial application), we do not explicitly introduce this HO-FO translation but implicitly assume that applications are always reduced, i.e., the expressions $((\backslash\_ \to e)\ ())$ and e are equivalent so that there is no explicit reduction step leading from $((\backslash\_ \to e)\ ())$ to e. Thus, when we talk about "rewrite steps" in the following, we consider only steps that are not related to such applications of lambda abstractions. Moreover, it should be noted that the transformation into first-order programs ensures that rewrite steps are never applied inside the body of lambda abstractions.

Formally, we define the following normalization function $norm_\beta$ on expressions that might occur in *pp*-transformed programs:

$$\begin{array}{rcll} norm_\beta((\backslash\_ \to e)\ ()) & = & e & \\ norm_\beta(x) & = & x & \text{if } x \in \mathcal{V} \\ norm_\beta(f(e_1, \ldots, e_n)) & = & f(norm_\beta(e_1), \ldots, norm_\beta(e_n)) & \text{if } f \in \mathcal{F} \cup \mathcal{C} \\ norm_\beta(\backslash\_ \to e) & = & \backslash\_ \to norm_\beta(e) & \end{array}$$

The following proposition states that $norm_\beta$ does not change the number of rewrite steps, which is obvious since $norm_\beta$ does not delete subterms that are relevant for rewriting.

Proposition 2 *Let $\mathcal{P}$ be a program obtained by the transformation pp. If $e \stackrel{*}{\to}_{\mathcal{P}} t$ is a derivation with n rewrite steps, then there is a corresponding derivation $norm_\beta(e) \stackrel{*}{\to}_{\mathcal{P}} t$ with n rewrite steps.*

The next lemma states a correspondence between rewrite rule applications in the original and transformed program.

Lemma 3 *Let $\mathcal{P}$ be a program, $\mathcal{P}' = pp(\mathcal{P})$, $f(x_1, \ldots, x_n) \to r \in \mathcal{P}$, and $\sigma(x_i) = e_i$ for $i = 1, \ldots, n$. Then $norm_\beta(\sigma'(pp(r))) = pp(\sigma(r))$ if $\sigma'(x_i) = \backslash\_ \to pp(e_i)$ for $i = 1, \ldots, n$.*

Proof: Let σ and σ' be given as in the lemma. We prove the slightly generalized claim

$$norm_\beta(\sigma'(pp(e))) = pp(\sigma(e))$$

for any expression e containing symbols of $\mathcal{P}$ and variables from $\{x_1, \ldots, x_n\}$ by induction on the size of e.
If e is a constant (0-ary constructor) c, then

$$norm_\beta(\sigma'(pp(c))) = norm_\beta(\sigma'(c)) = norm_\beta(c) = c = pp(c) = pp(\sigma(c))$$

If $e = x_i$ for some $i \in \{1, \ldots, n\}$, then

$$norm_\beta(\sigma'(pp(x_i))) = norm_\beta(\sigma'(x_i\ \texttt{()})) = norm_\beta((\texttt{\_} \to pp(e_i))\ \texttt{()}) = pp(e_i) = pp(\sigma(x_i))$$

If $e = c(e_1, \ldots, e_n)$ for some constructor or predefined operation $c \notin \mathcal{F}$, then

$$\begin{array}{lll} & norm_\beta(\sigma'(pp(c(e_1, \ldots, e_n)))) & \\ = & norm_\beta(\sigma'(c(pp(e_1), \ldots, pp(e_n)))) & \\ = & norm_\beta(c(\sigma'(pp(e_1)), \ldots, \sigma'(pp(e_n)))) & \\ = & c(norm_\beta(\sigma'(pp(e_1))), \ldots, norm_\beta(\sigma'(pp(e_n)))) & \\ = & c(pp(\sigma(e_1)), \ldots, pp(\sigma(e_n))) & \text{(by ind. hypothesis)} \\ = & pp(c(\sigma(e_1), \ldots, \sigma(e_n))) & \\ = & pp(\sigma(c(e_1, \ldots, e_n))) & \end{array}$$

If $e = g(e_1, \ldots, e_n)$ for some defined function $g \in \mathcal{F}$, then

$$\begin{array}{lll} & norm_\beta(\sigma'(pp(g(e_1, \ldots, e_n)))) & \\ = & norm_\beta(\sigma'(g(\texttt{\_} \to pp(e_1), \ldots, \texttt{\_} \to pp(e_n)))) & \\ = & norm_\beta(g(\texttt{\_} \to \sigma'(pp(e_1)), \ldots, \texttt{\_} \to \sigma'(pp(e_n)))) & \\ = & g(\texttt{\_} \to norm_\beta(\sigma'(pp(e_1))), \ldots, \texttt{\_} \to norm_\beta(\sigma'(pp(e_n)))) & \\ = & g(\texttt{\_} \to pp(\sigma(e_1)), \ldots, \texttt{\_} \to pp(\sigma(e_n))) & \text{(by ind. hypothesis)} \\ = & pp(g(\sigma(e_1), \ldots, \sigma(e_n))) & \\ = & pp(\sigma(g(e_1, \ldots, e_n))) & \end{array}$$

■

With these preparations we can prove the following important lemma which states the soundness of the transformation pp w.r.t. standard rewriting.

Lemma 4 *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in CTerm$. If $\mathcal{P}' = pST(\mathcal{P})$ and $pp(\mathcal{P}') \vdash pp(e) \xrightarrow{*} t$, then $\mathcal{P}' \vdash e \xrightarrow{*} t$.*

Proof: Let $\bar{\mathcal{P}} = pp(\mathcal{P}')$ and assume that $\bar{\mathcal{P}} \vdash pp(e) \xrightarrow{*} t$. We prove the lemma by induction on the number k of rewrite steps in this derivation.

Base case: $k = 0$: Since there is no rewrite step, $pp(e) = t$. Since t is a constructor term, by definition of pp, e does not contain defined functions so that $pp(e) = e$ which implies $e \xrightarrow{*}_{\mathcal{P}'} t$.

Inductive case: $k > 0$: Thus, there exists a derivation

$$e_0 \ \to_{\bar{\mathcal{P}}}\ e_1 \ \to_{\bar{\mathcal{P}}}\ \cdots \ \to_{\bar{\mathcal{P}}}\ e_k$$

with $e_0 = pp(e)$ and $e_k = t$. Consider the first rewrite step $e_0 \ \to_{\bar{\mathcal{P}}} \ e_1$, i.e., there is a position p, a rule $l \to r \in \bar{\mathcal{P}}$, and a substitution σ such that $e_0|_p = \sigma(l)$ and $e_1 = e_0[\sigma(r)]_p$. We distinguish the kind of the applied rule $l \to r \in \bar{\mathcal{P}}$.

1. $l \to r$ is a match rule, i.e., $l \to r = match(t_1, \ldots, t_n) \to \texttt{True}$. Since match rules are not modified by pp, $l \to r \in \mathcal{P}'$. Consider the expression $e'_1 = e[\texttt{True}]_p$. Clearly, $e \to_{\mathcal{P}'} e'_1$ and $pp(e'_1) = e_1$. Since the derivation

   $$pp(e'_1) \ \to_{\bar{\mathcal{P}}} \ e_2 \cdots \ \to_{\bar{\mathcal{P}}} \ e_k$$

   has $k - 1$ steps, we can apply the induction hypothesis so that $e'_1 \stackrel{*}{\to}_{\mathcal{P}'} t$ holds. Hence, $e \stackrel{*}{\to}_{\mathcal{P}'} t$.

2. $l \to r$ is a project rule, i.e., $l \to r = project(t) \to x$ for some $x \in \mathcal{V}ar(t)$. This rule is not modified by pp so that $l \to r \in \mathcal{P}'$. Since $x \in \mathcal{V}ar(t)$ and $\sigma(project(t)) = e_0|_p$, there is a position p' below p (i.e., $p \leq p'$) with $e_0|_{p'} = \sigma(x)$. Since $pp(e) = e_0$ and $e_1 = e_0[\sigma(x)]_p$, the same rule is applicable to e (note that p cannot be a position below a lambda abstraction introduced by pp), i.e., there exists a substitution σ' with $e|_p = \sigma'(project(t))$, $e'_1 = e[\sigma'(x)]_p$, and $pp(e'_1) = e_1$. Since the derivation

   $$pp(e'_1) \ \to_{\bar{\mathcal{P}}} \ e_2 \cdots \ \to_{\bar{\mathcal{P}}} \ e_k$$

   has $k - 1$ steps, we can apply the induction hypothesis so that $e'_1 \stackrel{*}{\to}_{\mathcal{P}'} t$ holds. Hence, $e \to_{\mathcal{P}'} e'_1 \stackrel{*}{\to}_{\mathcal{P}'} t$.

3. $l \to r$ is a rule defining the choice operator "`?`" or the conditional function `cond`. Since these kind of rules are not changed by pp and its structure is similar to a project rule, this case can be treated as the previous one.

4. $l \to r$ is a program rule transformed by pp, i.e., there exists a rule $l \to r' \in \mathcal{P}'$ with $l = f(x_1, \ldots, x_n)$ (note that all non-variable patterns in the left-hand side have been eliminated by the transformation pST), and $r = pp(r')$. By definition of the first rewrite step, $e_0|_p = f(\sigma(x_1), \ldots, \sigma(x_n))$ and $e_1 = e_0[\sigma(pp(r'))]$. By definition of pp (note that p cannot be a position below a lambda abstraction introduced by pp), $\sigma(x_i) = \backslash\_ \to pp(t_i)$ where $e|_p = f(t_1, \ldots, t_n)$. Let σ' be a substitution defined by $\sigma'(x_i) = t_i$ for $i = 1, \ldots, n$. Then $e \to_{\mathcal{P}'} e[\sigma'(r')]_p$ and

   $$e_0[norm_\beta(\sigma(pp(r')))]_p = pp(e[\sigma'(r')]_p) \tag{2}$$

   by Lemma 3. Since the derivation

   $$e_0[\sigma(pp(r'))]_p = e_1 \ \to_{\bar{\mathcal{P}}} \ e_2 \cdots \ \to_{\bar{\mathcal{P}}} \ e_k$$

   has $k - 1$ steps, there is a corresponding derivation

   $$e_0[norm_\beta(\sigma(pp(r')))]_p \ \stackrel{*}{\to}_{\bar{\mathcal{P}}} \ e_k$$

   with $k-1$ steps by Proposition 2. Thus, by the induction hypothesis and equality (2), $e[\sigma'(r')]_p \stackrel{*}{\to}_{\mathcal{P}'} t$ holds. Hence, $e \to_{\mathcal{P}'} e[\sigma'(r')]_p \stackrel{*}{\to}_{\mathcal{P}'} t$.

■

Now we can prove the soundness of our transformation.

Theorem 5 (Soundness of pST/pp) *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in CTerm$. If $pp(pST(\mathcal{P})) \vdash_{CRWL} pp(e) \twoheadrightarrow t$, then $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$.*

Proof: Let $\mathcal{P}' = pp(pST(\mathcal{P}))$ and assume that $\mathcal{P}' \vdash_{CRWL} pp(e) \twoheadrightarrow t$. By completeness of let-rewriting [29, Theorem 6], there exists a let-rewriting derivation $\mathcal{P}' \vdash pp(e) \xrightarrow{*}_l t$. Since let-rewriting is a sub-relation of standard rewriting when applied to expressions without *let*-occurrences [29, Theorem 8], there is also a rewrite derivation $\mathcal{P}' \vdash pp(e) \xrightarrow{*} t$. By Lemma 4, there exists a rewriting sequence $pST(\mathcal{P}) \vdash e \xrightarrow{*} t$. Hence, the correctness of the transformation pST w.r.t. the plural semantics (Theorem 1) implies $\mathcal{P} \vdash_{\pi CRWL} pp(e) \twoheadrightarrow t$. ■

5.2 Completeness

In order to prove the completeness of our transformation, we prove a lemma which states the completeness of the transformation pp w.r.t. let-rewriting as defined in [29]. In the following we denote by $e \to_l e'$ a let-rewriting step from e to e' and by $\xrightarrow{*}_l$ the reflexive and transitive closure of $\to_l$. To make the considered program explicit, we write $\mathcal{P} \vdash e \xrightarrow{*}_l t$ if $e \xrightarrow{*}_l t$ is a let-rewriting sequence w.r.t. program $\mathcal{P}$.

Lemma 6 *Let $\mathcal{P}$ be a program generated by pST (i.e., $\mathcal{P} = pST(\bar{\mathcal{P}})$ for some program $\bar{\mathcal{P}}$), $e \in Exp$, and $t \in CTerm$. If $\mathcal{P} \vdash e \xrightarrow{*} t$, then there exists a let-rewriting derivation $pp(\mathcal{P}) \vdash pp(e) \xrightarrow{*}_l t$.*

Proof: Let $\mathcal{P}' = pp(\mathcal{P})$ and assume that $\mathcal{P} \vdash e \xrightarrow{*} t$. We prove the lemma by induction on the number k of rewrite steps in this derivation.

Base case: $k = 0$: This case is trivial since there is no rewrite step, i.e., $e = t$. Since $pp(e) = pp(t) = t$, the claim holds.

Inductive case: $k > 0$: Thus, there exists a derivation

$$e_0 \;\to\; e_1 \;\to\; \cdots \;\to\; e_k$$

with $e_0 = e$ and $e_k = t$. Consider the first rewrite step $e_0 \to e_1$, i.e., there is a position p, a rule $l \to r \in \mathcal{P}$, and a substitution σ such that $e_0|_p = \sigma(l)$ and $e_1 = e_0[\sigma(r)]_p$. First, we assume that $p = \epsilon$ (the root position), i.e., $e_0 = f(e'_1, \ldots, e'_n)$. We distinguish the kind of the applied rule $l \to r \in \mathcal{P}$.

1. $l \to r$ is a program rule (and not a match/project rule), i.e., $l = f(x_1, \ldots, x_n)$ (note that all non-variable patterns in the left-hand side have been eliminated by the transformation pST), $\sigma(x_i) = e'_i$ $(i = 1, \ldots, n)$ and $e_1 = \sigma(r)$. Note that

$f(x_1,\ldots,x_n) \to pp(r) \in \mathcal{P}'$. We define the substitution σ' by $\sigma(x_i) = \backslash\_ \to pp(e'_i)$ for $i = 1,\ldots,n$. Then

$$pp(e_0) = \sigma'(f(x_1,\ldots,x_n)) \;\to_l\; \sigma'(pp(r))$$

is a let-rewriting step (using the rule (Fapp) of let-rewriting, where it should be noted that lambda abstractions correspond to constructor terms in our HO-FO translation, i.e., $\sigma' \in CSub$). By the induction hypothesis, $\mathcal{P}' \vdash pp(e_1) \stackrel{*}{\to}_l t$. By Lemma 3,

$$norm_\beta(\sigma'(pp(r)) = pp(\sigma(r)) = pp(e_1)$$

Hence, $\mathcal{P}' \vdash pp(e_0) \stackrel{*}{\to}_l t$ (since the normalization steps performed by $norm_\beta$ are also rewrite steps in the HO-FO-translated programs).

2. $l \to r$ is a match rule, i.e., $l \to r = match(t_1,\ldots,t_n) \to \texttt{True}$, $\sigma(t_i) = e'_i$, and $e_1 = \texttt{True}$. Since match rules are not modified by pp, $l \to r \in \mathcal{P}'$. Let $\mathcal{V}ar(l) = \{x_1,\ldots,x_m\}$. We define the substitution σ' by $\sigma'(x_i) = pp(\sigma(x_i))$ for $i = 1,\ldots,m$. Since $\sigma(match(t_1,\ldots,t_n)) = e_0$ and pp does not modify constructor terms, $pp(\sigma(t_i)) = \sigma'(t_i)$ for $i = 1,\ldots,n$. Thus, it is

$$\begin{array}{rcl} pp(e_0) & = & pp(match(e'_1,\ldots,e'_n)) \\ & = & match(pp(e'_1),\ldots,pp(e'_n)) \\ & = & match(pp(\sigma(t_1)),\ldots,pp(\sigma(t_n))) \\ & = & match(\sigma'(t_1)),\ldots,\sigma'(t_n))) \end{array}$$

so that $pp(e_0) \to_l \texttt{True}$ holds if $\sigma' \in CSub$ (by the rule (Fapp) of let-rewriting). If there is a subterm of some $\sigma'(x_i)$ which is operation-rooted (so that $\sigma' \notin CSub$), replace all these subterms by fresh variables using the (LetIn) rule of let-rewriting, then apply the (Fapp) rule, and eliminate the introduced variables by subsequent applications of the (Elim) rule. Hence, $pp(e_0) \stackrel{*}{\to}_l \texttt{True}$ holds also in this case.

3. $l \to r$ is a project rule, i.e., $l \to r = project(t) \to x$ for some $x \in \mathcal{V}ar(t)$, $\sigma(t) = e'_1$, and $e_1 = \sigma(x)$. This rule is not modified by pp so that $l \to r \in \mathcal{P}'$. Let $\mathcal{V}ar(l) = \{x_1,\ldots,x_m\}$ and define the substitution σ' by $\sigma'(x_i) = pp(\sigma(x_i))$ for $i = 1,\ldots,m$. We distinguish two cases:

   (a) $\sigma(x)$ is a constructor term: Since $\sigma(project(t)) = e_0$ and pp does not modify constructor applications, $pp(\sigma(t)) = \sigma'(t)$. Hence,

   $$pp(e_0) \;\stackrel{*}{\to}_l\; \sigma'(x)$$

   by an application of the (Fapp) rule (provided that $\sigma \in CSub$; otherwise, we eliminate the superfluous operation-rooted subterms as in the previous case 2). Thus,

   $$pp(e_0) \;\stackrel{*}{\to}_l\; \sigma'(x) = pp(\sigma(x)) = pp(e_1) \;\stackrel{*}{\to}_l\; t$$

   where the last step follows by the induction hypothesis.

(b) $\sigma(x)$ is operation-rooted: Then we apply the (LetIn) rule to obtain

$$pp(e_0) = \sigma'(project(t)) \;\rightarrow_l\; let\ z = pp(\sigma(x))\ in\ \sigma'(project(t'))$$

where $\{z \mapsto pp(\sigma(x))\}(t') = t$. Like in the first case, we have

$$let\ z = pp(\sigma(x))\ in\ \sigma'(project(t')) \;\stackrel{*}{\rightarrow}_l\; let\ z = pp(\sigma(x))\ in\ z$$

Remember that $pp(\sigma(x)) = e_1$. By the induction hypothesis, $\mathcal{P}' \vdash pp(e_1) \stackrel{*}{\rightarrow}_l t$. By context closedness of the let-rewriting relation $\rightarrow_l$, we have

$$let\ z = pp(e_1)\ in\ z \;\stackrel{*}{\rightarrow}_l\; let\ z = t\ in\ z \;\stackrel{*}{\rightarrow}_l\; t$$

where the latter step is an application of the (Bind) rule of let-rewriting. Thus, $\mathcal{P} \vdash pp(e_0) \stackrel{*}{\rightarrow}_l t$ holds.

4. The rules defining the choice operator "`?`" or the conditional function `cond` are treated in an analogous manner.

Finally, the case $p \neq \epsilon$ holds by context closedness of the let-rewriting relation $\rightarrow_l$. ■
Now we can prove the completeness of our transformation.

Theorem 7 (Completeness of pST/pp) *Let $\mathcal{P}$ be a program, $e \in Exp$, and $t \in CTerm$. If $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$, then $pp(pST(\mathcal{P})) \vdash_{CRWL} pp(e) \twoheadrightarrow t$.*

Proof: Let $\mathcal{P}' = pp(pST(\mathcal{P}))$ and assume that $\mathcal{P} \vdash_{\pi CRWL} e \twoheadrightarrow t$. By Theorem 1, $pST(\mathcal{P}) \vdash e \stackrel{*}{\rightarrow} t$. By Lemma 6, there exists a let-rewriting sequence $\mathcal{P}' \vdash pp(e) \stackrel{*}{\rightarrow}_l t$. Hence, the soundness of let-rewriting [29, Theorem 4] implies $\mathcal{P}' \vdash_{CRWL} pp(e) \twoheadrightarrow t$. ■

6 Implementation and Benchmarks

The actual implementation of plural arguments in Curry consists of a library `Plural` containing a few definitions to mark plural arguments and support the transformation and the implementation of the transformations pST/pp on Curry programs. To mark plural arguments, the library `Plural` contains the following "identity" type definition:

```
type Plural a = a
```

Hence, marking a plural argument in a type definition of an operation does not change its actual type so that the "marked" Curry program is still valid and can be processed by the front end of each Curry system. Furthermore, the library contains the following definitions used in the transformed programs:

```
data PluralArg a = PluralArg (() → a)

plural :: PluralArg a → a
plural (PluralArg pfun) = pfun ()
```

The type `PluralArg` denotes a plural argument wrapped into a lambda abstraction, and the operation `plural` unwraps such an argument by applying the lambda abstraction.

The program transformation tool replaces each occurrence of the type constructor `Plural` by the type constructor `PluralArg`, wrap plural arguments at call sites into lambda abstractions that are put into the data constructor `PluralArg`, and unwrap occurrences of plural arguments in right-hand sides by calls to `plural`. For instance, the rules

```
dup :: Plural Int → (Int,Int)
dup x = (x,x)

main = dup (0?1)
```

are transformed into

```
dup :: PluralArg Int → (Int,Int)
dup x = (plural x, plural x)

main = dup (PluralArg (\_ → (0?1)))
```

Furthermore, occurrences of non-variables patterns are transformed by *pST* as already shown above.

In order to evaluate our transformational approach, we have performed a few benchmarks comparing our implementation with the Maude implementation of [33]. The transformed programs have been executed by PAKCS [24], an implementation of Curry that compiles into Prolog (executed by SICStus-Prolog). Due to the fact that the Maude implementation is a prototype and does not contain features that are important for application programming (e.g., predefined data types like numbers, characters, or strings, arithmetic operations, data structures, input/output operations, etc), we could only compare quite small programs. The following table contains the result of the naive reverse operation (where plural arguments are not present), the palindrome and decimal number parsers (see Section 3), and an expression parser where the digits and operations occurring in an expression are passed as plural arguments. All operations are applied to lists of different lengths (as specified in the table). The programs have been executed on a PC running Ubuntu 12.04 with an Intel Core i5 (2.53GHz) and 4GB of main memory. The run times are in milliseconds (or "–" if the execution delivers no result, e.g., runs out of memory), where 0 denotes a run-time of less than ten milliseconds:

	`nrev`				`pali`				`number`			`expr`			
Length:	8	16	32	256	6	18	34	514	20	80	320	9	21	93	1533
Maude:	120	1180	–	–	36	260	–	–	210	1410	–	90	280	–	–
PAKCS:	0	0	0	30	0	0	0	100	0	0	50	0	0	0	30

Although these benchmarks are small, they clearly show the superiority of our transformational approach over a new implementation of the plural semantics. Furthermore, our approach has the advantage that all advanced language features required for application programming (predefined operations, application libraries) are immediately available from the host language.

7 Conclusions

In this paper we have shown how plural arguments can be added to existing functional logic languages based on the call-time choice semantics. In practice, plural arguments could be a useful feature. However, executing complete programs with a plural semantics increases the search space considerably and might produce unintended results. Thus, in larger programs only a few arguments should be passed with the plural semantics. We support this idea by a program transformation that changes only the handling of plural arguments so that the entire program can be executed with a call-time choice semantics. This has the advantage that existing implementations can be re-used and all language features, execution strategies, libraries, or programming environments, are immediately available also for this extended language. Beyond its correctness, we have also shown for a widely used implementation of Curry that this approach is much more efficient than a dedicated implementation of the plural semantics.

For future work, it is interesting to explore the use of plural arguments in larger applications since this is now possible with our transformational approach. Furthermore, it could be useful to analyze plural arguments in order to deduce for which occurrences of plural arguments our transformation could be omitted in order to improve the efficiency of the overall implementation.

References

[1] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 4, pp. 768–844, 1998.

[2] T. Amtoft. Minimal Thunkification. In *Proc. Third International Workshop on Static Analysis (WSA'93)*, pp. 218–229. Springer LNCS 724, 1993.

[3] S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.

[4] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[5] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.

[6] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pp. 87–101. Springer LNCS 4079, 2006.

[7] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.

[8] S. Antoy and M. Hanus. New Functional Logic Design Patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 19–34. Springer LNCS 6816, 2011.

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

[10] B. Braßel, M. Hanus, and M. Müller. High-Level Database Programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pp. 316–332. Springer LNCS 4902, 2008.

[11] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pp. 1–18. Springer LNCS 6816, 2011.

[12] R. Caballero and F.J. López-Fraguas. A Functional-Logic Perspective of Parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 85–99. Springer LNCS 1722, 1999.

[13] 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*. Springer LNCS 4350, 2007.

[14] J. de Dios Castro and F.J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science*, Vol. 188, pp. 3–19, 2007.

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.

[16] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.

[17] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 63–74. ACM Press, 2007.

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

[19] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 153–167. MIT Press, 1997.

[20] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.

[21] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.

[22] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.

[23] M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pp. 123–168. Springer LNCS 7797, 2013.

[24] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2013.

[25] M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.

[26] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[27] H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, Vol. 12, pp. 237–255, 1992.

[28] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.

[29] F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 197–208. ACM Press, 2007.

[30] F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Flexible Framework for Programming with Non-deterministic Functions. In *Proc. of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 2009)*, pp. 91–100. ACM Press, 2009.

[31] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[32] J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pp. 717–740. ACM Press, 1972.

[33] A. Riesco and J. Rodríguez-Hortalá. A Natural Implementation of Plural Semantics in Maude. *ENTCS*, Vol. 253, No. 7, pp. 165–175, 2010.

[34] A. Riesco and J. Rodríguez-Hortalá. Programming with singular and plural non-deterministic functions. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010)*, pp. 83–92. ACM Press, 2010.

[35] J. Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, volume 2, pp. 328–339. LIPIcs, 2008.

[36] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pp. 37–48. ACM Press, 2008.

[37] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.