Adding Plural Arguments to Curry Programs

Michael Hanus

Christian-Albrechts-University of Kiel
Programming Languages and Compiler Construction

ICLP 2013
Functional Logic Languages

Goal: combine best of declarative paradigms in a single model

- **efficient execution** principles of functional languages (determinism, laziness)

- **flexibility** of logic languages (computation with partial information, built-in search)

- avoid non-declarative features of Prolog (arithmetic, cut, I/O, side-effects)

Curry [POPL’97,…] ⇝ http://www.curry-language.org/

- declarative multi-paradigm language (higher-order concurrent functional logic language)

- extension of Haskell (non-strict functional language)

- SWE advantage: better (high-level) APIs for various application domains (GUI programming, web programming, database programming,…)

Michael Hanus (CAU Kiel)  Adding Plural Arguments to Curry Programs  ICLP 2013
Curry Programs

Datatypes (values): enumerate all constructors

```
data Bool   = True  | False
data List a = []    | a : List a  -- [a]
```

Program rules: \( f t_1 \ldots t_n \mid c = r \)

- \( f \): function name
- \( t_1 \ldots t_n \): data terms
- \( c \): condition of type \textit{Success} (optional)
- \( r \): expression

Example

```
(++) :: [a] → [a] → [a]  last :: [a] → a
[]   ++ ys = ys          last xs | ys ++ [x] =:= xs
(x:xs) ++ ys = x : xs ++ ys = x where ys, x free
```
Non-deterministic Operations

**Choice operation**

\[
\begin{align*}
  x \ ? \ _ &= x \\
  _ \ ? \ y &= y
\end{align*}
\]

\[
\text{coin} = 0 \ ? \ 1
\]

\[
f(C \ x) = (x, x)
\]

Values of \[f(C \ \text{coin}) \]?

**Call-time choice (⇝ Curry, TOY)**

Argument values fixed before function call:

\[
\begin{align*}
  &\leftrightarrow (0, 0) (1, 1)
\end{align*}
\]

Implementation: call-by-value or call-by-need (sharing!)

**Run-time choice**

Argument values fixed when they are used:

\[
\begin{align*}
  &\leftrightarrow (0, 0) (0, 1) (1, 0) (1, 1)
\end{align*}
\]

Implementation: term rewriting

Problem: result might depend on strategy
## Non-deterministic Operations

### A denotational view

\[
\begin{align*}
    x \ ? \ _ &= x \\
    _ \ ? \ y &= y \\
    \text{coin} &= 0 \ ? \ 1 \\
    f(C \ x) &= (x, x)
\end{align*}
\]

Domain of parameters?

### Singular semantics

Parameters are single values \(\approx\) call-time choice

\[
f(C \ \text{coin}) \rightsquigarrow \text{Parameter: } (C \ 0) \text{ or } (C \ 1), \text{i.e., } x=0 \text{ or } x=1
\]

### Plural semantics

Parameters are sets of values \(\neq\) run-time choice!

\[
f(C \ \text{coin}) \rightsquigarrow \text{Parameter: } \{ (C \ 0), (C \ 1) \}, \text{i.e., } x=\{0, 1\}\]
Plural Semantics

Juan Rodríguez-Hortalá et al. ’08/’10/’12

- hierarchy of semantics (singular ⊂ run-time choice ⊂ plural)
- programming examples (passing sets / non-deterministic values as arguments)
- transformation ↦ execute plural programs by term rewriting
- implementation in Maude

Our contribution

- new transformation to implement plural arguments
- combine plural and singular arguments
- execute target programs with call-time choice semantics

↝ reuse existing Curry implementations!
Programming with Plural Arguments

**Default: call-time choice / singular semantics**

```haskell
data C = C Int

f :: C → (Int,Int)
f (C x) = (x,x)

main = f (C (0 ? 1))  \sim\ (0,0) (1,1)
```

**Mark arguments: plural semantics**

```haskell
data C = C Int

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

main = f (C (0 ? 1))  \sim\ (0,0) (0,1) (1,0) (1,1)
```
Assume standard parser combinators:

- `empty`  terminal `t`  `<*>` (sequence)  `<|>` (alternative)

Palindromes parameterized over terminal alphabet (represented by non-deterministic value):

```haskell
pali :: Plural a -> Parser a
pali t = empty
  <|> terminal t
  <|> let someT = terminal t
      in someT <*> pali t <*> someT
```

Palindromes over letters 'a' and 'b':

```haskell
abPali s = pali ('a' ? 'b') s =:= []
abPali "abaaba"  ~ success
abPali "ac0ca"  ~ failure
```

Palindromes over digits:

```haskell
```
Implementation and Benchmarks

Implementation

Combined library/preprocessor approach:

- import library **Plural** to mark plural arguments by type declarations
- apply preprocessor to perform program transformations:
  - replace pattern matching by explicit match operations
  - wrap actual parameters into $\lambda$-abstractions
  - unwrap access to formal parameters

Benchmarks

- Maude implementation of plurality [Riesco/Rodríguez-Hortalá ENTCS’10]
- our transformation executed by PAKCS
- results in msecs (Ubuntu 12.04, Intel Core i5 (2.53GHz), 4GB mem)

<table>
<thead>
<tr>
<th></th>
<th>nrev8</th>
<th>nrev16</th>
<th>nrev32</th>
<th>nrev256</th>
<th>pali10</th>
<th>pali14</th>
<th>pali16</th>
<th>pali514</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maude</td>
<td>120</td>
<td>1180</td>
<td>error</td>
<td>error</td>
<td>36</td>
<td>260</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>PAKCS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>
Conclusions

This talk:
- plural arguments useful for particular applications
- typically only a few plural arguments in larger programs
- exploit existing efficient implementations of call-time choice with a simple transformation
- reuse strategies, language features, libraries, …

In the paper:
- precise definition of program transformation
- soundness and completeness results
Example transformation

\[
f :: \text{Plural C} \rightarrow (\text{Int, Int})
f (\text{C x}) = (x, x)
\]

\[
\text{main} = f (\text{C (0 ? 1)})
\]

Apply program transformation:

\[
f y1 \mid \text{match1} (y1 ())
\]
\[
= (\text{project11} (y1 ()), \text{project11} (y1 ()))
\]
\[
\text{where}
\]
\[
\text{match1} (\text{C x}) = \text{success}
\]
\[
\text{project11} (\text{C x}) = x
\]

\[
\text{main} = f (\lambda \_ \rightarrow \text{C (0 ? 1)})
\]
Plural Semantics ≠ Run-time Choice

\begin{align*}
  x \oplus _ = & x \\
  _ \oplus y & = y \\
  \text{coin} = & 0 \oplus 1 \\
  f (C \; x) & = (x, x)
\end{align*}

\begin{align*}
  f (C \; (0 \oplus 1))
  \begin{align*}
  \text{Plural semantics: } x & = \{0, 1\} \implies (0, 0) \; (0, 1) \; (1, 0) \; (1, 1) \\
  \text{Run-time choice: } & \implies (0, 0) \; (0, 1) \; (1, 0) \; (1, 1)
  \end{align*}
\end{align*}

\begin{align*}
  f (C \; 0 \oplus C \; 1)
  \begin{align*}
  \text{Plural semantics: } x & = \{0, 1\} \implies (0, 0) \; (0, 1) \; (1, 0) \; (1, 1) \\
  \text{Run-time choice: } & \implies f (C \; 0) \to (0, 0) \; \text{or} \implies f (C \; 1) \to (1, 1)
  \end{align*}
\end{align*}

Conclusion

In case of pattern matching: plural semantics ≠ run-time choice