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,...] $\leadsto$ `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,...)

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

c : condition of type `Success` (optional)

$t_1 \ldots t_n$: data terms

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

```
x ? _  =  x                coin = 0 ? 1
_ ? y  =  y                f (C x) = (x,x)
```

Values of `f (C coin))` ?

Call-time choice (⇝ Curry, TOY)

Argument values fixed before function call:

⇝ `(0,0)  (1,1)`

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

Run-time choice

Argument values fixed when they are used:

⇝ `(0,0)  (0,1)   (1,0)   (1,1)`

Implementation: term rewriting

Problem: result might depend on strategy

Non-deterministic Operations

A denotational view

```
x ? _  =  x                    coin = 0 ? 1
_ ? y  =  y                    f (C x) = (x,x)
```

Domain of parameters?

Singular semantics

Parameters are single values $\approx$ call-time choice

`f (C coin))` $\rightsquigarrow$ Parameter: `(C 0)` or `(C 1)`, i.e., `x=0` or `x=1`

Plural semantics

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

`f (C coin))` $\rightsquigarrow$ Parameter: $\{$`(C 0)`, `(C 1)`$\}$, i.e., `x=`$\{$`0,1`$\}$

Plural Semantics

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

- hierarchy of semantics (singular $\subset$ run-time choice $\subset$ plural)
- programming examples
  (passing sets / non-deterministic values as arguments)
- transformation $\leadsto$ 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

$\leadsto$ reuse existing Curry implementations!

Programming with Plural Arguments

Default: call-time choice / singular semantics

```
data C = C Int

f :: C  →  (Int,Int)
f (C x) = (x,x)

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

Mark arguments: plural semantics

```
data C = C Int

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

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

Example: Exploiting Plural Arguments

Example: parameterized parsing

Assume standard parser combinators:

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

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

```
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’:

```
abPali s = pali ('a' ? 'b') s =:= []
```

```
abPali "abaaba"  ⇝ success
abPali "ac0ca"   ⇝ failure
```

Palindromes over digits:

```
numPali s = pali (0?1?2?3?4?5?6?7?8?9) s =:= []
```

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 λ-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)

	nrev8	nrev16	nrev32	nrev256	pali10	pali14	pali16	pali514
Maude	120	1180	error	error	36	260	error	error
PAKCS	0	0	0	30	0	0	0	100

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

Plural Semantics → Call-time Choice

Example transformation

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

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

Apply program transformation:

```
f y1 | match1 (y1 ())
     = (project11 (y1 ()), project11 (y1 ()))
  where
   match1 (C x) = success
   project11 (C x) = x

main = f (λ _  → C (0 ? 1))
```

Plural Semantics $\neq$ Run-time Choice

```
x ? _  =  x            coin = 0 ? 1
_ ? y  =  y            f (C x) = (x,x)
```

`f (C (0 ? 1))`

Plural semantics: `x={0,1}` ⇝ `(0,0) (0,1) (1,0) (1,1)`

Run-time choice: ⇝ `(0,0) (0,1) (1,0) (1,1)`

`f (C 0 ? C 1)`

Plural semantics: `x={0,1}` ⇝ `(0,0) (0,1) (1,0) (1,1)`

Run-time choice: ⇝ `f (C 0)` → `(0,0)` or ⇝ `f (C 1)` → `(1,1)`

Conclusion

In case of pattern matching: plural semantics $\neq$ run-time choice