LOPSTR 2005

Declarative Programming
with
Function Patterns

Michael Hanus

Christian-Albrechts-Universität Kiel

(joint work with Sergio Antoy, Portland State University)

FUNCTIONAL LOGIC LANGUAGES

Approach to amalgamate ideas of declarative programming

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

- flexibility of logic languages
  (constraints, built-in search)

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

- combine best of both worlds in a single model
  (higher-order functions, declarative I/O, concurrent constraints)

- Advantages:
  - ➜ optimal evaluation strategies [JACM'00,ALP'97]
  - ➜ new design patterns [FLOPS'02]
  - ➜ better abstractions for application programming
    (GUI programming [PADL'00], web programming [PADL'01])

FUNCTIONAL LOGIC PROGRAMS: CURRY [POPL'97]

FUNCTIONAL LOGIC PROGRAMS: CURRY [POPL'97]

Datatypes (≈ admissible values): enumerate all data constructors

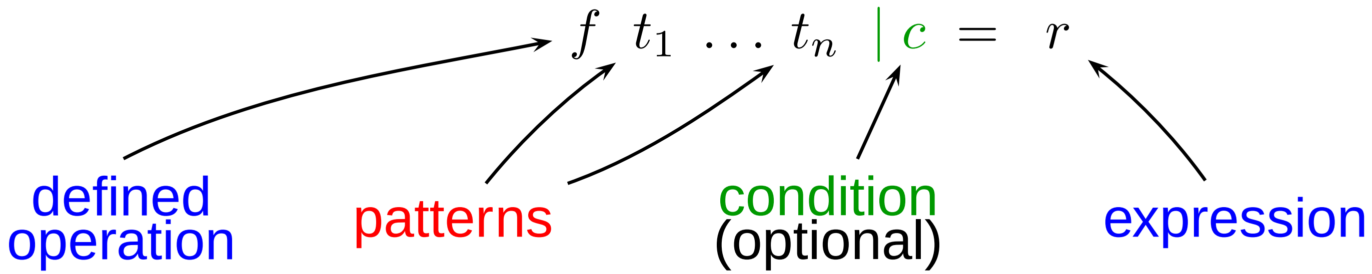
```
data Bool     =  True       |  False
data List a   =  []         |  a : List a        -- [a]
```

FUNCTIONAL LOGIC PROGRAMS: CURRY [POPL'97]

Datatypes (≈ admissible values): enumerate all data constructors

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

Functions: operations on values defined by equations (or rules)



$$f\ t_1\ \ldots\ t_n\ \mid c\ =\ r$$

Pattern: linear data term

```
(++) :: [a] -> [a] -> [a]             head :: [a] -> a
[]     ++ ys = ys                     head (x:xs) = x
(x:xs) ++ ys = x : xs ++ ys
```

FUNCTIONAL LOGIC PROGRAMS

Functional evaluation: (lazy) rewriting

`[1,2]++[3]` → `1:([2]++[3])` → `1:(2:([]++[3]))` → `[1,2,3]`

Functional logic evaluation: equation solving, guess values for unknowns

`xs++[x] =:= [1,2,3]` ⇝ {`xs`↦`[1,2]`, `x`↦`3`}

FUNCTIONAL LOGIC PROGRAMS

Functional evaluation: (lazy) rewriting

`[1,2]++[3]` → `1:([2]++[3])` → `1:(2:([]++[3]))` → `[1,2,3]`

Functional logic evaluation: equation solving, guess values for unknowns

`xs++[x] =:= [1,2,3]` $\rightsquigarrow$ {`xs`↦`[1,2]`, `x`↦`3`}

Define functions by **conditional equations**:

```
last :: [a] -> a
last xs | ys ++ [x] =:= xs  = x      where x,ys free
```

`last [1,2]` $\rightsquigarrow$ `2`

MEANING OF “=:=”

Modern functional logic languages (Curry, Toy): non-strict semantics

- ➜ lazy evaluation
- ➜ computing with infinite structures
- ➜ comparison of arbitrary infinite objects?

MEANING OF “=:=”

Modern functional logic languages (Curry, Toy): non-strict semantics

- ➜ lazy evaluation
- ➜ computing with infinite structures
- ➜ comparison of arbitrary infinite objects?

Strict equality (K-LEAF [Giovannetti et al. ’91])

- ➜ identity on finite data terms ($\leadsto$ not reflexive)
- ➜ e_1 `=:=` e_2 satisfied iff e_1 and e_2 reducible to same (unifiable) constructor term
- ➜ “`x =:= head []`” does not hold

MEANING OF "=:="

Modern functional logic languages (Curry, Toy): non-strict semantics

- ➜ lazy evaluation
- ➜ computing with infinite structures
- ➜ comparison of arbitrary infinite objects?

Strict equality (K-LEAF [Giovannetti et al. '91])

- ➜ identity on finite data terms ($\leadsto$ not reflexive)
- ➜ e_1 `=:=` e_2 satisfied iff e_1 and e_2 reducible to same (unifiable) constructor term
- ➜ "`x =:= head []`" does not hold

Disadvantage: strict equality evaluates more than necessary

`last [failed,2]` $\leadsto$ no result!

STRICT EQUALITY: MOTIVATION

Difficulty: comparison of infinite structures

```
from x = x : from (x+1)      ⇒     from 0 ⇝ 0:1:2:3:4:5:...

rtail (x:xs) = rtail xs
```

`rtail (from 0) =:= rtail (from 5)` : should hold with reflexivity

STRICT EQUALITY: MOTIVATION

Difficulty: comparison of infinite structures

```
from x = x : from (x+1)      ⇒     from 0 ⇝ 0:1:2:3:4:5:...

rtail (x:xs) = rtail xs
```

`rtail (from 0) =:= rtail (from 5) :` should hold with reflexivity

```
from2 x = x : x+1 : from2 (x+2)
```

`from 0 =:= from2 0 :` should hold with reflexivity, generally undecidable

STRICT EQUALITY: MOTIVATION

Difficulty: comparison of infinite structures

```
from x = x : from (x+1)     ⇒    from 0 ⇝ 0:1:2:3:4:5:...

rtail (x:xs) = rtail xs
```

`rtail (from 0) =:= rtail (from 5)` : should hold with reflexivity

```
from2 x = x : x+1 : from2 (x+2)
```

`from 0 =:= from2 0` : should hold with reflexivity, generally undecidable

⟹ strict equality is not reflexive

`head [] =:= head []` ⇝ no solution

(not specific to FLP, e.g., Haskell, Java,. . .)

RELAXING STRICT EQUALITY?

Is evaluation always necessary?

`x =:= head []` $\leadsto$ no solution

Why not: solve x `=:=` t by binding x to t (without evaluating t)?

RELAXING STRICT EQUALITY?

Is evaluation always necessary?

`x =:= head []` ⇝ no solution

Why not: solve x =:= t by binding x to t (without evaluating t)?

Desirable in some cases (e.g., `last`), non-intuitive in other cases:

```
f x | x =:= from 0  = 99
```

```
f x             ⇝    99
(f x, 99)       ⇝    (99, 99)
(f x, f x)      ⇝    no termination
```

RELAXING STRICT EQUALITY?

Is evaluation always necessary?

`x =:= head []` $\leadsto$ no solution

Why not: solve $x\,\texttt{=:=}\,t$ by binding x to t (without evaluating t)?

Desirable in some cases (e.g., `last`), non-intuitive in other cases:

```
f x | x =:= from 0  = 99
```

`f x` $\leadsto$ `99`

`(f x, 99)` $\leadsto$ `(99, 99)`

`(f x, f x)` $\leadsto$ no termination

Solution: Distinguish between

- ➜ logic variables: bind only to finite constructor terms
- ➜ pattern variables: bind to arbitrary (unevaluated) terms

$\leadsto$ **function patterns**

FUNCTION PATTERNS: SYNTAX

Function pattern: pattern containing

- ➜ variables
- ➜ constructors
- ➜ defined operation symbols

```
last :: [a] -> a
last (xs ++ [x]) = x
```

Advantages:

- ➜ concise definition
- ➜ `xs` and `x` pattern variables $\leadsto$ can be bound to unevaluated expressions
- ➜ `last [failed,2]` $\leadsto$ `2` (with $\{$`xs`$\mapsto$`[failed]`, `x`$\mapsto$`2`$\}$)

FUNCTION PATTERNS: TRANSFORMATIONAL SEMANTICS

- ➜ Reuse existing semantics and models of functional logic programs
- ➜ Transform programs with function patterns into standard programs

FUNCTION PATTERNS: TRANSFORMATIONAL SEMANTICS

- ➜ Reuse existing semantics and models of functional logic programs
- ➜ Transform programs with function patterns into standard programs

Basic idea: rule with function patterns ↦ set of rules where each function pattern is replaced by its evaluation to some data term

Example: Evaluations of `xs++[x]`:

$$
\begin{array}{lll}
\texttt{xs++[x]} & \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[]}} & \texttt{[x]} \\
\texttt{xs++[x]} & \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[x1]}} & \texttt{[x1,x]} \\
\texttt{xs++[x]} & \overset{*}{\leadsto}_{\texttt{xs}\mapsto\texttt{[x1,x2]}} & \texttt{[x1,x2,x]} \\
\ldots & &
\end{array}
$$

⇒ `last (xs ++ [x]) = x` abbreviates the set of rules

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

SEMANTICS OF FUNCTION PATTERNS

Potential problems of this approach:

SEMANTICS OF FUNCTION PATTERNS

Potential problems of this approach:

1. infinite set of transformed rules ⇝ perform transformation at run time

SEMANTICS OF FUNCTION PATTERNS

Potential problems of this approach:

1. infinite set of transformed rules ⇝ perform transformation at run time
2. circular definition, e.g.,

   ```
   (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
   ```

   ⇝ avoid circular definitions by restriction to stratified programs

SEMANTICS OF FUNCTION PATTERNS

Potential problems of this approach:

1. infinite set of transformed rules ⇝ perform transformation at run time

2. circular definition, e.g.,

   ```
   (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
   ```

   ⇝ avoid circular definitions by restriction to stratified programs

3. non-left-linear transformed rules

   ```
   idpair x = (x,x)
   ```

   ```
   f (idpair x) = 0
   ```

   Transformation into: `f (x,x) = 0`

   Not allowed in standard FLP ⇝ linearization of left-hand sides:

   ```
   f (x,y) | x=:=y = 0
   ```

Details ⇝ paper

EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra'76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra'76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

```
data Color = Red | White | Blue
```

EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra'76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

```
data Color = Red | White | Blue

solve (x++[White]++y++[ Red ]++z) = solve (x++[ Red ]++y++[White]++z)
```

EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra'76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

```
data Color = Red | White | Blue

solve (x++[White]++y++[ Red ]++z) = solve (x++[ Red ]++y++[White]++z)

solve (x++[Blue ]++y++[ Red ]++z) = solve (x++[ Red ]++y++[Blue ]++z)

solve (x++[Blue ]++y++[White]++z) = solve (x++[White]++y++[Blue ]++z)
```

EXAMPLE: PROBLEM SOLVING

Dutch National Flag (Dijkstra'76): arrange a sequence of objects colored by red, white or blue so that they appear in the order of the Dutch flag

```
data Color = Red | White | Blue

solve (x++[White]++y++[ Red ]++z) = solve (x++[ Red ]++y++[White]++z)

solve (x++[Blue ]++y++[ Red ]++z) = solve (x++[ Red ]++y++[Blue ]++z)

solve (x++[Blue ]++y++[White]++z) = solve (x++[White]++y++[Blue ]++z)

solve flag | isDutchFlag flag = flag
 where isDutchFlag (uni Red ++ uni White ++ uni Blue) = success

       uni color = []
       uni color = color : uni color
```

EXAMPLE: TESTING INFINITE STRUCTURES

Task: compute length of a stream up to the first repeated element
(part of an ACM programming contest)

EXAMPLE: TESTING INFINITE STRUCTURES

Task: compute length of a stream up to the first repeated element
(part of an ACM programming contest)

Implementation with function patterns:

```
lengthUpToRepeat (p++[r]++q)
      | nub p == p && elem r p
      = length p + 1
```

- `(nub xs)`: list without duplicates
- function pattern + condition: break input list into part without repeated elements and first repeated element
- with strict equality (i.e., `xs =:= p++[r]++q`): works only for finite lists and evaluates also elements after first repeated element

EXAMPLE: TRANSFORMATION OF EXPRESSIONS

Task: simplify symbolic arithmetic expressions, e.g., $1 * (x + 0) \leadsto x$

```
data Exp = Lit Int | Var [Char] | Add Exp Exp | Mul Exp Exp
```

EXAMPLE: TRANSFORMATION OF EXPRESSIONS

Task: simplify symbolic arithmetic expressions, e.g., $1 * (x + 0) \leadsto x$

```
data Exp = Lit Int | Var [Char] | Add Exp Exp | Mul Exp Exp
```

`(evalTo e)`: expressions that simplify to `e`

```
evalTo e = Add (Lit 0) e          evalTo e = Mul (Lit 1) e
evalTo e = Add e (Lit 0)          evalTo e = Mul e (Lit 1)
...
```

EXAMPLE: TRANSFORMATION OF EXPRESSIONS

Task: simplify symbolic arithmetic expressions, e.g., $1 * (x + 0) \leadsto x$

```
data Exp = Lit Int | Var [Char] | Add Exp Exp | Mul Exp Exp
```

`(evalTo e)`: expressions that simplify to `e`

```
evalTo e = Add (Lit 0) e          evalTo e = Mul (Lit 1) e
evalTo e = Add e (Lit 0)          evalTo e = Mul e (Lit 1)
...
```

`(replace` $c\ p\ e$`)`: term replacement $c[e]_p$

```
replace _ [] x = x
replace (Add l r) (1:p) x = Add (replace l p x) r
replace (Add l r) (2:p) x = Add l (replace r p x)
...
```

EXAMPLE: TRANSFORMATION OF EXPRESSIONS

Task: simplify symbolic arithmetic expressions, e.g., $1 * (x + 0) \leadsto x$

```
data Exp = Lit Int | Var [Char] | Add Exp Exp | Mul Exp Exp
```

`(evalTo e)`: expressions that simplify to `e`

```
evalTo e = Add (Lit 0) e          evalTo e = Mul (Lit 1) e
evalTo e = Add e (Lit 0)          evalTo e = Mul e (Lit 1)
...
```

`(replace` c p e`)`: term replacement $c[e]_p$

```
replace _ [] x = x
replace (Add l r) (1:p) x = Add (replace l p x) r
replace (Add l r) (2:p) x = Add l (replace r p x)
...
simplify (replace c p (evalTo x)) = replace c p x
```

Two applications of function patterns:

- ➜ define abstractions for complex collections of patterns (`evalTo`)
- ➜ specify transformations at arbitrary positions inside an argument (`replace`)
  e.g., variable in expression: `varInExp (replace c p (Var v)) = v`

EXAMPLE: XML QUERIES AND TRANSFORMATIONS

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

EXAMPLE: XML QUERIES AND TRANSFORMATIONS

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Some useful abstractions:

```
xtxt s   = XText s         -- basic text element
xml t c = XElem t [] c   -- XML element without attributes
```

$\leadsto$ `xml "program" [xml "language" [xtxt "Curry"],...]`

EXAMPLE: XML QUERIES AND TRANSFORMATIONS

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Some useful abstractions:

```
xtxt s   = XText s         -- basic text element
xml t c = XElem t [] c   -- XML element without attributes
```

$\leadsto$ `xml "program" [xml "language" [xtxt "Curry"],...]`

(`replace` $xe\ c\ s$): XML term replacement $xs[s]_p$

```
replace _ [] s = s
replace (XElem tag atts xes) (i:p) s =
           XElem tag atts (replaceElem i (\x -> replace x p s) xes)
  where
   replaceElem O     f (x:xs) = f x : xs
   replaceElem (S n) f (x:xs) = x : replaceElem n f xs
```

EXAMPLE: XML QUERIES AND TRANSFORMATIONS

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

Some useful abstractions:

```
xtxt s  = XText s        -- basic text element
xml t c = XElem t [] c   -- XML element without attributes
```

$\rightsquigarrow$ `xml "program" [xml "language" [xtxt "Curry"],...]`

(`replace` $xe\ c\ s$): XML term replacement $xs[s]_p$

```
replace _ [] s = s
replace (XElem tag atts xes) (i:p) s =
         XElem tag atts (replaceElem i (\x -> replace x p s) xes)
  where
   replaceElem 0     f (x:xs) = f x : xs
   replaceElem (S n) f (x:xs) = x : replaceElem n f xs
```

Example: Find element `<city>...</city>` in XML expression:

```
cityOf (replace _ _ (xml "city" [xtxt c])) = c
```

IMPLEMENTATION

Basic idea: perform transformation of rules containing function patterns demand-driven at run time

Integration of function patterns into existing implementations:

① Preprocessor eliminates function patterns:
replace by new variable and introduce specific unification “`=:<=`” in condition

② Provide implementation of “`=:<=`”

Example:

```
last (xs++[x]) = x
```

is transformed into

```
last ys | xs++[x] =:<= ys = x     where xs,x free
```

FUNCTION PATTERN UNIFICATION

To evaluate e_1 `=:<=` e_2: (e_1: function pattern)

① Evaluate e_1 to a head normal form h_1

② If h_1 is a variable: bind it to e_2

③ If $h_1 = c\ t_1 \ldots t_n$ (where c is a constructor):

(a) Evaluate e_2 to a head normal form h_2

(b) If h_2 is a variable: instantiate h_2 to $c\ x_1 \ldots x_n$ ($x_1, \ldots, x_n$ are fresh variables) and evaluate t_1 `=:<=` x_1 `&` ... `&` t_n `=:<=` x_n

(c) If $h_2 = c\ s_1 \ldots s_n$: evaluate t_1 `=:<=` s_1 `&` ... `&` t_n `=:<=` s_n

(d) Otherwise: fail

FUNCTION PATTERN UNIFICATION

To evaluate e_1 `=:<=` e_2:   (e_1: function pattern)

① Evaluate e_1 to a head normal form h_1

② If h_1 is a variable: bind it to e_2

③ If $h_1 = c\ t_1 \ldots t_n$ (where c is a constructor):

- (a) Evaluate e_2 to a head normal form h_2
- (b) If h_2 is a variable: instantiate h_2 to $c\ x_1 \ldots x_n$ ($x_1, \ldots, x_n$ are fresh variables) and evaluate t_1 `=:<=` x_1 `&` ... `&` t_n `=:<=` x_n
- (c) If $h_2 = c\ s_1 \ldots s_n$: evaluate t_1 `=:<=` s_1 `&` ... `&` t_n `=:<=` s_n
- (d) Otherwise: fail

➜ finite search space for `xs++[x] =:<= [failed,2]`

➜ missing: linearization of transformed left-hand sides
$\leadsto$ mark pattern variables that occur in step (2) and generate strict equalities for multiple marked variables

➜ useful: more efficient function pattern unification "`=:<<=`" for *linear* function patterns ($\leadsto$ compiler optimization)

BENCHMARKS

Implementation of function patterns provided in Curry programming environment PAKCS

Function pattern increases expressiveness, but they can also increase efficiency in comparison to strict equality:

Expression:	`=:=`	`=:<=`	`=:<<=`
`last (take 10000 (repeat failed) ++ [1])`	no solution	380	250
`last (map (inc 0) [1..2000])`	20900	90	60
`lengthUpToRepeat ([1..50]++[1]++[51..])`	∞	200	200
`simplify*`	1200	1080	690
`varsInExp`	2240	1040	100

Further optimization:
compile-time specialization of function patterns ($\leadsto$ paper)

CONCLUSIONS

Declarative programs with function patterns:

- concise definitions, problems with strict equality avoided
- executable high-level definitions of complex transformation tasks and queries on tree-like structures
- semantics defined by transformation into standard programs
- implementation by specific function pattern unification
- **extension specific to integrated functional logic languages**
  (LP: lack of evaluable functions, FP: lack of nondeterminism)
- functional logic languages:
  ideal environments for building high-level abstractions

Prototype implementation available in recent releases of PAKCS:

`http://www.informatik.uni-kiel.de/~pakcs/`