Benelog'98

Multi-Paradigm Declarative Programming in Curry

Michael Hanus

RWTH Aachen

Declarative Programming

Common idea:

- description of logical relationships
- powerful abstractions, higher programming level
- reliable and maintainable programs
  - pointer structures ⇒ algebraic data types
  - complex procedures ⇒ comprehensible parts (pattern matching, local definitions)

Different paradigms:

- *Functional programming:*
  functions, equations, λ-calculus
  (lazy) deterministic reduction
- *Logic programming:*
  predicates, logical formulas, predicate logic
  constraint solving, search

⇒ Functional logic languages:

- efficient deterministic reduction (if possible)
- flexibility of logic languages
- avoid non-declarative features of Prolog (arithmetic, I/O, cut)
- combine best of both worlds in a single model

Curry: A Truly Integrated Functional Logic Language

[Dagstuhl'96, POPL'97]

- multi-paradigm language, combines
  - functional programming
  - logic programming
  - concurrent programming
- based on an optimal evaluation strategy
- conservative extension of lazy functional and (concurrent) logic programming
- conditional (constrained) rules
- higher-order, non-deterministic functions
- equational constraints
- encapsulated search, committed choice
- polymorphic type system, modules
- declarative (monadic) I/O
- external functions and constraint solvers

Curry Programs

Values: *data terms* containing *constructors* and *variables* (≈ *Herbrand terms*): `(S x)` `[0,(S 0)]`

```
data Bool   = True | False
data Nat    = 0    | S Nat
data List a = []   | a : List a
```

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

$$f \;\; t_1 \; \ldots \; t_n \;\; | \; c \; = \; r$$

- f: defined operation
- $t_1 \ldots t_n$: data terms
- c: **constraint** (conjunction of equations)
- r: expression

```
    0 + y = y              0 ≤ y       = True
(S x) + y = S(x+y)     (S x) ≤ 0       = False
                       (S x) ≤ (S y)   = x≤y


append []     ys = ys
append (x:xs) ys = x : append xs ys


sub m n   | n + d =:= m    = d   where d free
```

Evaluation: Computing Values

- reduce expressions to their values
- replace equals by equals
- apply reduction step to a subterm (redex)
  (rule's left-hand side must *match* the subterm)

```
    0 + y = y                 0 ≤ y       = True
(S x) + y = S(x+y)        (S x) ≤ 0       = False
                          (S x) ≤ (S y)   = x≤y
```

`(S 0)+(S 0)` → `S (0+(S 0))` → `S (S 0)`

Lazy strategy: select an outermost redex

```
0+0 ≤ (S 0)+(S 0)
→  0 ≤ (S 0)+(S 0)
→  True
```

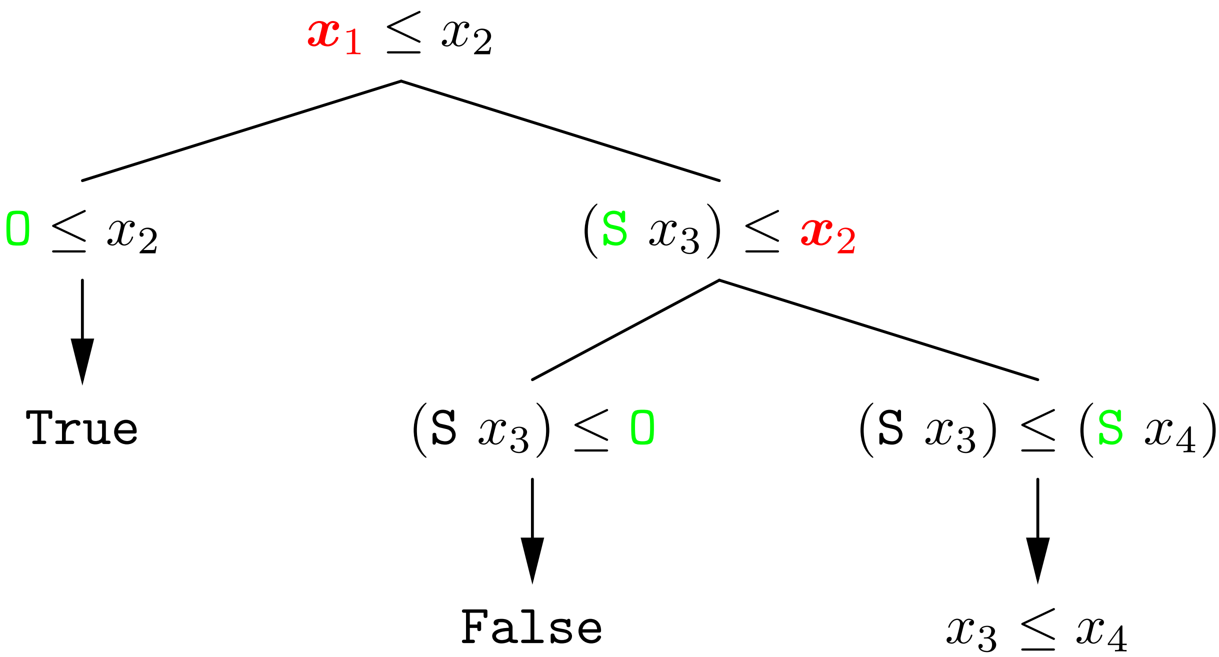
⇝ evaluate only needed redexes
(efficiently computable with definitional trees)

⇝ *functional programming*

Definitional Trees [Antoy 92]

- data structure to organize the rules of an operation
- each node has a distinct *pattern*
- *branch* nodes (case distinction), *rule* nodes

```
    O   ≤ y     = True
(S x)   ≤ O     = False
(S x)   ≤ (S y) = x≤y
```



Function call: $t_1 \leq t_2$

1. Reduce t_1 to head normal form
2. If $t_1 =$ `O`: apply rule
3. If $t_1 =$ `S`...: reduce t_2 to head normal form
4. If t_1 variable: not reducible or bind t_1 to `O` or (`S` x)

Overlapping Rules: Non-deterministic Rewriting

```
True  ∨ x     = True
    x ∨ True  = True
False ∨ False = False
```

Problem: no needed argument:

$e_1 \vee e_2$ evaluate e_1 or e_2?

Functional languages: Evaluate e_1, if not successful: e_2

Disadvantage: not normalizing (e_1 may not terminate)

Solutions:

1. Parallel reduction of e_1 and e_2
   [Sekar/Ramakrishnan 93]

2. **Non-deterministic reduction:**
   try (*don't know*) e_1 or e_2

Extension to definitional trees:
Introduce *or*-nodes to describe non-deterministic selection of redexes

From Functional Programming to Logic Programming

Functional programming: *values*, no free variables

Logic programming: *computed answers* for free variables

Operational extension:

instantiate free variables, if necessary

```
f 0 = 2
f 1 = 3
```

Evaluate `(f x)`: – bind `x` to `0` and reduce `(f 0)` to `2`, or:

– bind `x` to `1` and reduce `(f 1)` to `3`

Computation step: $\underbrace{\textbf{bind}}_{\textit{logic}}$ and $\underbrace{\textbf{reduce}}_{\textit{functional}}$

$$e \leadsto \underbrace{\{\sigma_1\}\ e_1 \mid \cdots \mid \{\sigma_n\}\ e_n}_{\text{disjunctive expression}}$$

Reduce: `(f 0)` $\leadsto$ `2`

Bind and reduce: `(f x)` $\leadsto$ `{x=0} 2 | {x=1} 3`

Compute necessary bindings with *needed* strategy

$\leadsto$ *needed narrowing* [Antoy/Echahed/Hanus POPL'94]

Properties of Needed Narrowing

[Antoy/Echahed/Hanus POPL'94]

- **Sound** and **complete** (w.r.t. strict equality)
- **Optimality:**
  1. **No unnecessary steps:**
     Each narrowing step is needed, i.e., it cannot be avoided if a solution should be computed.
  2. **Shortest derivations:**
     If common subterms are shared, needed narrowing derivations have minimal length.
  3. **Independence of solutions:**
     Two solutions σ and σ' computed by two distinct derivations are independent.
- **Determinism:**
  No non-deterministic step during the evaluation of ground expressions ($\approx$ functional programming)
- **Restriction: inductively sequential rules**
  (i.e., no overlapping left-hand sides)
- Extensible to
  - conditional rules [Hanus ICLP'95]
  - overlapping lhs [Antoy/Echahed/Hanus ICLP'97]
  - multiple rhs [Antoy ALP'97]
  - concurrent evaluation [Hanus POPL'97]

Strict Equality and Equational Constraints

Problems with equality in the presence of non-terminating rules:

1. Equality on infinite objects undecidable:

```
f = [0|f]                g = [0|g]
```

   Is `f = g` valid?

2. Semantics of non-terminating functions:

```
f x = f (x+1)            g x = g (x+1)
```

   Is `f 0 = g 0` valid?

Avoided by **strict equality**: identity on *finite* objects (both sides reducible to same ground data term)

Equational constraint e_1 `=:=` e_2:

satisfied if both sides evaluable to unifiable data terms

$\Rightarrow$ e_1 `=:=` e_2 does not hold if e_1 or e_2 undefined

$\Rightarrow$ e_1 `=:=` e_2 and e_1, e_2 data terms $\approx$ unification in LP

Non-deterministic Functions

Functions can have more than one result value:

```
choose x y = x
choose x y = y
```

`choose 1 2` $\leadsto$ `1 | 2`

Non-deterministic list insertion and *permutations*:

```
insert x []     = [x]
insert x (y:ys) = choose (x:y:ys)
                         (y:insert x ys)

permute []     = []
permute (x:xs) = insert x (permute xs)
```

```
permute [1,2,3]  ⇝
    [1,2,3] | [2,1,3] | [2,3,1] |
    [1,3,2] | [3,1,2] | [3,2,1]
```

Programming Demand-driven Search

Prolog: generate-and-test:

```
psort(Xs,Ys) :- permute(Xs,Ys), ordered(Ys).
```

Functional programming: list comprehensions:

```
psort xs = [ys | ys<-perms xs, sorted ys]
```

Prolog with coroutining: test-and-generate

```
psort(Xs,Ys) :- ordered(Ys), permute(Xs,Ys).
```

(Problem: floundering, heuristics)

Functional logic programming: test-of-generate:

```
sorted []  = []
sorted [x] = [x]
sorted (x:y:ys) | x<=y  = x : sorted (y:ys)

psort xs = sorted (permute xs)
```

Advantages:

- demand-driven generation of solutions (due to laziness)
- same efficiency as coroutining
- no floundering
- modular program structure

Example: Demand-driven Search

```
sorted []    = []
sorted [x]   = [x]
sorted (x:y:ys) | x<=y
                = x : sorted (y:ys)

psort xs = sorted (permute xs)
```

`psort [5,4,3,2,1]`

$\leadsto$ `sorted (permute [5,4,3,2,1])`

$\leadsto^*$ $\underbrace{\texttt{sorted (5 : 4 : permute [3,2,1])}}_{\text{undefined: discard this alternative}}$ $\mid$ $\cdots$

$\leadsto$ $\ldots$

Effect: Permutations of `[3,2,1]` are not enumerated!

Permutation sort for `[`n`,`n−1`,`…`,2,1]`: #or-branches

Length of the list:	4	5	6	8	10
generate-and-test	24	120	720	40320	3628800
test-of-generate	19	59	180	1637	14758

Encapsulated Search

[Hanus/Steiner PLILP'98]

Technique to avoid global search (backtracking)
(non-backtrackable I/O, efficiency control,. . .)

Idea:
Compute until a non-deterministic step occurs,
then give programmer control over this situation
(generalization of Oz's operator [Schulte/Smolka 94])

Search:

- solve constraint containing search variable
- evaluate until *failure*, *success*, or *non-determinism*
- return result in a list
- bind search variable to different solutions
  ⇒ abstract search variable: `\`x`->`c $(\approx \lambda x.c)$

Primitive **search operator:**

```
try :: (a -> Constraint) -> [a -> Constraint]
```

```
try \x-> 1=:=2        ~> []                          failure
try \x-> [x]=:=[0]    ~> [\x-> x=:=0]                success
try \x-> f x =:= 3    ~> [\x-> x=:=0 & f 0 =:= 3,
                          \x-> x=:=1 & f 1 =:= 3]
                                                 disjunction
```

Encapsulated Search: Search Strategies

`try \x->c`: eval. c, stop after non-deterministic step

Depth-first search: collect all solutions

```
all :: (a -> Constraint) -> [a -> Constraint]
all g = collect (try g)
where
 collect []           = []
 collect [g]          = [g]
 collect (g1:g2:gs) =
                 concat (map all (g1:g2:gs))
```

```
all \l -> append l [1] =:= [0,1]
⇝  [\l -> l =:= [0]]
```

Further search strategies:

- compute only first solution:

  ```
  once g = head (all g)
  ```

- `findall`, best solution search, parallel search, ...
- negation as failure:

  ```
  naf c = (all \_->c) =:= []
  ```

  ⇝ control failures

Handling solutions

Extract value of the search variable by application:

```
(\x->x=:=1) freevar
⇒ freevar=:=1
⇒ {freevar=1} success
```

Prolog's findall:

```
unpack :: [a -> Constraint] -> [a]

unpack [] = []
unpack (g:gs) | g v  = v : unpack gs
                where v free


findall g = unpack (all g)
```

```
findall (\(x,y) -> append x y =:= [1,2])
```

$\stackrel{*}{\Rightarrow}$ `[([],[1,2]),([1],[2]),([1,2],[])]`

Exploiting laziness

Demand-driven encapsulated search easily obtained by laziness:

```
prolog g = printloop (all g)

printloop []     = putStr("no") >> nl
printloop (a:as) = browse a >> putStr "? " >>
                   getChar >>= evalAnswer as

evalAnswer as ';'  = nl >> printloop as
evalAnswer as '\n' = nl >> putStr "yes" >> nl
```

```
prolog \(x,y) -> append x y =:= [1,2]
```

$\stackrel{*}{\Rightarrow}$ `([],[1,2]) ?  ;`

`([1],[2]) ?  <-`

`yes`

`prolog \x -> 1 =:= 2` $\stackrel{*}{\Rightarrow}$ `no`

⇝ **Separation of Logic and Control**

⇝ **Modularity:**

- Prolog with breadth-first search:

  `prolog_bfs g = printloop (bfs g)`

- Prolog with depth-bounded search:

  `prolog_bound g b = printloop (bound g b)`

From Function Logic Programming to Concurrent Programming

Disadvantage of narrowing:

- functions on recursive data structures
  $\leadsto$ narrowing may not terminate

- all rules must be explicitly known
  $\leadsto$ combination with external functions unclear (basic arithmetic,...)

Solution:
Delay function calls if a particular argument is free

Distinguish:
rigid (consumer) and *flexible* (generator) functions

Necessary:
Concurrent conjunction of constraints: $c_1 \,\&\, c_2$
Meaning: evaluate c_1 and c_2 concurrently, if possible

```
x+x=:=y  &  x=:=2
~>  {x=2}  2+2=:=y      (suspend x+x)
~>  {x=2}  4=:=y        (evaluate 2+2)
~>  {x=2, y=4}
```

Parallel Functional Programming

[Goffin,Eden]

Parallel evaluation of arguments:

```
f t1 t2 = letpar  x = g t1
                  y = h t2  in  k x y
```

with concurrent conjunction of equations:

```
f t1 t2 | x =:= g t1 & y = h t2  = k x y
        where x,y free
```

Skeleton-based parallel programming:

Applying a function to all list elements (sequentially):

```
map f []       = []
map f (x:xs)   = f x : map f xs
```

`farm`: parallel version of `map`

```
farm f []      = []
farm f (x:xs) | r =:= f x & rs =:= farm f xs
              = r : rs       where r,rs free
```

Concurrent Objects with State

Modelling objects with state as a constraint function:

- first parameter: stream of messages (wait for input)
- second parameter: current state

Example: **Bank account**

```
data Messages = Deposit Int | Withdraw Int
              | Balance Int

account eval rigid  -- declare a rigid func.

account []  _                = success
account (Deposit  a : ms) n = account ms (n+a)
account (Withdraw a : ms) n = account ms (n-a)
account (Balance  b : ms) n = b=:=n & account ms n

make_account s = account s 0
```

```
make_account s,  -- create account object
 s = [Deposit 200, Withdraw 50, Balance b]
```

$\leadsto$ `{b=150, s=...}`

Soundness and Completeness

Relate derivations to standard rewriting $\to_{\mathcal{R}}$ ($\to_{\mathcal{R}}$ sound and complete w.r.t. model-theoretic semantics)

Soundness: If

$$e \quad \rightsquigarrow^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

then $\sigma_i(e) \to^*_{\mathcal{R}} e_i$ for $i = 1, \ldots, n$

Completeness: If $\sigma(e) \to^*_{\mathcal{R}} c$ and

$$e \quad \rightsquigarrow^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

then $\exists \varphi, i$ with $\sigma = \varphi \circ \sigma_i$ and $\varphi(e_i) \to^*_{\mathcal{R}} c$

Completeness w.r.t. flexible functions:
All functions are *flexible*: If $\sigma(e) \to^*_{\mathcal{R}} c$, then

$$\exists \quad e \quad \rightsquigarrow^* \quad \{\sigma_1\}\, e_1 \mid \ldots \mid \{\sigma_n\}\, e_n$$

with $e_i = c$ and $\sigma = \varphi \circ \sigma_i$ for some i and φ

Curry: Unification of Computation Models

Computation model	Restrictions on programs
Needed narrowing [POPL'94]	inductively sequential rules; optimal w.r.t. length of derivations and number of computed solutions
Weakly needed narrowing (∼Babel)	only flexible functions
Resolution (∼Prolog)	only (flexible) predicates (∼ constraints)
Lazy functional languages (∼Haskell)	no free variables in expressions
parallel functional languages (∼Goffin, Eden)	only rigid functions, concurrent conjunction
Residuation (∼Life, Oz)	constraints are flexible; all other functions are rigid (default in Curry)

Programming in Curry

```
append :: [a] -> [a] -> [a]
append eval flex  -- append is flexible

append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Functional programming:

`append [1,2] [3,4]` $\leadsto$ `[1,2,3,4]`

Logic programming (append is *flexible*):

`append x y =:= [1,2]` $\leadsto$

`{x=[],y=[1,2]} | {x=[1],y=[2]} | {x=[1,2],y=[]}`

```
               from n = n : from (S n)
first 0     xs     = []
first (S n) (x:xs) = x : first n xs
```

Lazy functional programming:

`first (S (S 0)) (from 0)` $\leadsto$ `[0,(S 0)]`

Lazy functional logic programming:

`first x (from y) =:= [0]` $\leadsto$ `{x=(S 0),y=0}`

Functions vs. Predicates

rigid functions not always reasonable:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Concatenate known lists:

`append [1,2] [3,4]` $\leadsto$ `[1,2,3,4]`

Splitting a list:

`append x [2] =:= [1,2]` $\leadsto$ *not reducible* (*delay*)

Escher [Lloyd 94]: provide additional split predicate (superfluous from a declarative point of view)

Prolog: define `append` always as a predicate
$\Rightarrow$ worse operational behavior than a function:

Curry: `append (append x y) z =:= []`
finite search space (if append is flexible)

Prolog: `append(X,Y,L), append(L,Z,[])`
infinite search space

Functional Logic Programming vs. (Concurrent) Logic Programming

Implementation of functions by flattening
$\leadsto$ loss of functional dependencies:

```
              from n = n : from (S n)
first 0       xs     = []
first (S n) (x:xs)   = x : first n xs
```

```
first x (from x) =:= []
⤳ {x=0} [] =:= []   |   {x=(S n)} ...failure...
⤳ {x=0}
```

Translation of functions into predicates by flattening:

```
from(N,[N|R]) :- from(s(N),R).
first(0,L,[]).
first(s(N),[E|L],[E|R]) :- first(N,L,R).
```

```
first(X,L,[]), from(X,L)
⤳{X↦0} from(0,L) ⤳ from(s(0),L1) ⤳ ···
```

Higher-Order Features

Higher-order functions:

```
map :: (a -> b) -> [a] -> [b]

map f []       = []
map f (x:xs) = (f x) : map f xs
```

`map (append [1]) [[2],[3]]` $\leadsto$ `[[1,2],[1,3]]`

- higher-order features of functional languages (partial applications, λ-abstractions)
- first-order definition of application function (as in [Warren 82])
- application function is *rigid*

  $\leadsto$ delay applications with unknown functions
- future extension(?): higher-order unification

Monadic Input/Output

- declarative I/O concept
- I/O: transformation on the outside world
- interactive program: compute **actions**
  (transformation on the *world*)
- type of actions: `IO t ≈ World -> (t,World)`

  ```
  getChar :: IO Char
  getLine :: IO String
  putLine :: String -> IO ()
  ```

  `getChar` applied to a world
  $\leadsto$ character + new (transformed) world
- compose actions:
  `(>>=) ::  IO a -> (a -> IO b) -> IO b`

  `getLine >>= putLine`:
  copies a line from input to output
- no I/O in disjunctions ("cannot copy the world"):
  encapsulate search between I/O actions

External Functions

- infinite set of defining equations

```
0+0 = 0
0+1 = 1
0+2 = 2
...
2+1 = 3
...
```

- definition not accessible
- external implementation (without side effects)
- suspend external function calls until arguments are fully known, i.e., ground
  [Bonnier/Maluszynski 88, Boye 91]
- external function interface
- implementation of basic arithmetic
  (+, -, *,...: external functions)

Not possible in narrowing-based languages!

Arithmetic

0, 1, 2, ...: constructors

+, -, *, ...: external functions

`x =:= 2+3*4` ⇝ `{x=14}`

`x =:= 2*3+y` ⇝ `{} x =:= 6+y` (*suspend*)

`x+x =:= y & x =:= 2`
⇝ `{x=2} 2+2 =:= y` (suspend `x+x`)
⇝ `{x=2} 4 =:= y` (evaluate `2+2`)
⇝ `{x=2, y=4}`

⇒ Functions as passive constraints (Life)

```
digit 0 = success
...
digit 9 = success
```

`x+x =:= y & x*x =:= y & digit x`

⇝ `{x=0, y=0} | {x=2, y=4}`

Implementations of Curry

- First prototypical implementations available

- Interpreter in Prolog: TasteCurry-System
  (RWTH Aachen, Portland State University)
  `http://www-i2.informatik.rwth-aachen.de/`
  `~hanus/tastecurry`

- [Hanus LOPSTR'95]: Efficient implementation of needed narrowing by transformation into Prolog
  $\leadsto$ Sloth-System [Mariño/Rey WFLP'98]

- Compiler Curry→Java [Hanus/Sadre ILPS'97]
  (Java threads for concurrency and non-determinism)
  - portable
  - simplified implementation
    (garbage collection, threads)
  - slow but (hopefully!) better Java
    implementations in the future

- abstract Curry machine [Lux WFLP'98]

Why Integration of Declarative Paradigms?

- more expressive than pure functional languages (compute with partial information/constraints)
- more structural information than in pure logic programs (functional dependencies)
- more efficient than logic programs (determinism, laziness)
- functions: declarative notion to improve control in logic programming
- avoid impure features of Prolog (arithmetic, I/O)
- combine research efforts in FP and LP

$\leadsto$ Do not teach two paradigms, but one:

Declarative Programming

[Hanus PLILP'97]

Curry: A True Integration of Declarative Paradigms

Functional programming: lazy evaluation, deterministic evaluation of ground expressions, higher-order functions, polymorphic types, monadic I/O

$\Longrightarrow$ extension of Haskell

Logic programming: logical variables, partial data structures, search facilities, concurrent constraint solving

Curry:

- efficiency (functional programming) + expressivity (search, concurrency)
- possible with "good" evaluation strategies
- one paradigm: **declarative programming**

More infos on Curry:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry`