Multi-paradigm Declarative Languages

Michael Hanus

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

ICLP 2007

Declarative Programming: The General Idea

Do not no code algorithms and stepwise execution

Describe logical relationships

⇝ powerful abstractions
  - domain specific languages

⇝ higher programming level

⇝ reliable and maintainable programs
  - pointer structures ⇒ algebraic data types
  - complex procedures ⇒ comprehensible parts (pattern matching, local definitions)

Declarative Languages: Current Situation

Declarative languages based on different formalisms, e.g.,

Functional Languages

- lambda calculus
- functions
- directed equations
- reduction of expressions

Logic Languages

- predicate logic
- predicates
- definite clauses
- goal solving by resolution

Constraint Languages

- constraint structures
- constraints
- specific constraint solvers

Declarative Languages: Features

Functional Languages

- higher-order functions
- expressive type systems
- demand-driven evaluation
- optimality, modularity

Logic Languages

- compute with partial information
- nondeterministic search
- unification

Constraint Languages

- specific domains
- efficient constraint solving

All features are useful $\rightsquigarrow$ multi-paradigm declarative languages

Multi-paradigm Declarative 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)
- application-domains of constraint languages
  (constraint solvers for specific domains)

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

Multi-paradigm Declarative Languages: Approaches

Extend logic languages

- add functional notation as syntactic sugar
  (Ciao-Prolog, Mercury, HAL, Oz,. . .)
- defining equations, nested functional expressions
- translation into logic kernel
- don't exploit functional information for execution

Extend functional languages

- add logic features (logic variables, nondeterminism)
  (Escher, TOY, Curry,. . .)
- functional syntax, logic programming use
- retain efficient (demand-driven) evaluation whenever possible
- additional mechanism for logic-oriented computations

Curry

As a language for concrete examples, we use

Curry [POPL'97,...]

- multi-paradigm declarative language
- extension of Haskell (non-strict functional language)
- developed by an international initiative
- provide a standard for functional logic languages (research, teaching, application)
- several implementations and various tools available

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

Basic Concept: Functional Computation

Functional program: set of functions defined by equations/rules

```
double x  =  x + x
```

Functional computation: replace subterms by equal subterms

$$\underline{\texttt{double (1+2)}} \Rightarrow \underline{\texttt{(1+2)}}\texttt{+(1+2)} \Rightarrow \texttt{3+}\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$$

Another computation:

$$\underline{\texttt{double (1+2)}} \Rightarrow \texttt{(1+2)+}\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{(1+2)}}\texttt{+3} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$$

And another computation:

$$\texttt{double }\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{double 3}} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$$

Functional Computation

```
double x  =  x + x
```

$\underline{\texttt{double (1+2)}} \Rightarrow \underline{\texttt{(1+2)}}\texttt{+(1+2)} \Rightarrow \texttt{3+}\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$

$\underline{\texttt{double (1+2)}} \Rightarrow \texttt{(1+2)+}\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{(1+2)}}\texttt{+3} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$

$\texttt{double }\underline{\texttt{(1+2)}} \Rightarrow \underline{\texttt{double 3}} \Rightarrow \underline{\texttt{3+3}} \Rightarrow \texttt{6}$

All derivations $\rightsquigarrow$ same result: referential transparency

- computed result independent of evaluation order
- no side effects
- simplifies reasoning and maintenance

Several strategies: what are good strategies?

Basic Concept: Algebraic Data Types

Values in declarative languages: terms

```
data Bool  =  True | False
```

Definition by pattern matching:

```
not True   =  False
not False  =  True
```

Replacing equals by equals still valid:

$\texttt{not }\underline{\texttt{(not False)}} \Rightarrow \underline{\texttt{not True}} \Rightarrow \texttt{False}$

Algebraic Data Types: Lists

List of elements of type `a`

```
data List a  =  [] | a : List a
```

Some notation: `[a]` $\approx$ `List a`

$[e_1, e_2, \ldots, e_n] \approx e_1 : e_2 : \ldots : e_n : []$

List concatenation “++”

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

`[1,2,3] ++ [4]` $\Rightarrow^*$ `[1,2,3,4]`

From Functional to Functional Logic Programming

List concatenation “++”

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

Use “++” to specify other list functions:

Last element of a list: `last xs = e` iff `∃ys: ys ++ [e] = xs`

Direct implementation in a functional logic language:

- search for solutions w.r.t. existentially quantified variables
- solve equations over nested functional expressions

Definition of `last` in Curry

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

Functional Logic Programs

Set of functions defined by equations (or rules)

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

f : function name
$t_1 \ldots t_n$: data terms (constructors, variables)
c : condition (optional)
r : expression

Constructor-based term rewriting system

Rules with extra variables

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

allowed in contrast to traditional rewrite systems

non-constructive, forbidden to provide efficient evaluation strategy

Functional Logic Computations: Narrowing

Rewriting not sufficient in the presence of logic variables $\leadsto$

Narrowing = variable instantiation $+$ rewriting

Narrowing step: $t \leadsto_{p, l \rightarrow r, \sigma} t'$

p : non-variable position in t
$l \rightarrow r$: program rule (variant)
σ : unifier for $t|_p$ and l
t' : $\sigma(t[r]_p)$

Why not most general unifiers?

Functional Logic Computations: Narrowing

Narrowing with mgu's is not optimal

```
data Nat = O | S Nat          leq O     _     = True
add O     y = y               leq (S _) O     = False
add (S x) y = S(add x y)      leq (S x) (S y) = leq x y
```

$\texttt{leq v (add w O)}\underline{\texttt{leq v (add w O)}} \rightsquigarrow_{\{v\mapsto O\}} \texttt{True}$

Another narrowing computation:

$\texttt{leq v }\underline{\texttt{(add w O)}} \rightsquigarrow_{\{w\mapsto O\}} \texttt{leq v O}\underline{\texttt{leq v O}} \rightsquigarrow_{\{v\mapsto S\,z\}} \texttt{False}$

And another narrowing computation:

$\texttt{leq v }\underline{\texttt{(add w O)}} \rightsquigarrow_{\{w\mapsto O\}} \underline{\texttt{leq v O}} \rightsquigarrow_{\{v\mapsto O\}} \texttt{True}$ superfluous!

Avoid last derivation by non-mgu in first step:

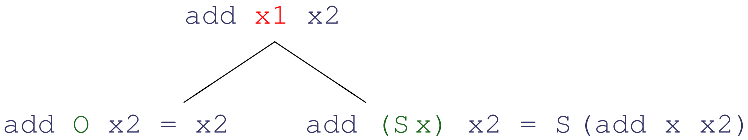
$\texttt{leq v }\underline{\texttt{(add w O)}} \rightsquigarrow_{\{v\mapsto S\,z,\,w\mapsto O\}} \texttt{leq (S z) O}$

Needed Narrowing [JACM'00]

- constructive method to compute positions and unifiers
- defined on inductively sequential rewrite systems
- basic idea: organize all rules in a definitional tree: branch nodes (case distinction), rule nodes

Definitional tree of

```
add O     y  =  y
add (S x) y  =  S(add x y)
```



add x1 x2

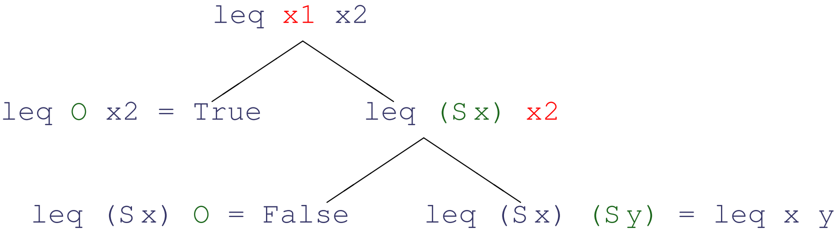
add O x2 = x2

add (S x) x2 = S (add x x2)

Definitional Trees

```
leq O     _     = True
leq (S _) O     = False
leq (S x) (S y) = leq x y
```

Definitional tree:



leq x1 x2

leq O x2 = True

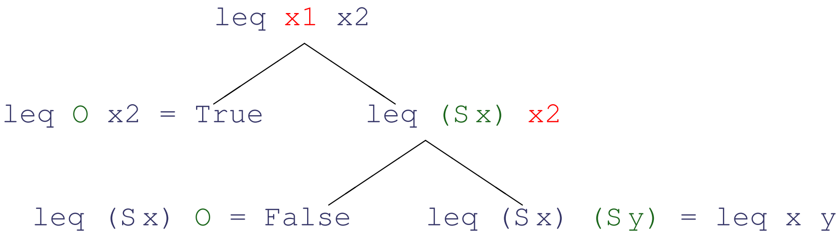
leq (S x) x2

leq (S x) O = False

leq (S x) (S y) = leq x y

- contains all rules of a function
- can be computed at compile time
- guides the narrowing strategy

Needed Narrowing with Definitional Trees



`leq x1 x2`

`leq O x2 = True`

`leq (S x) x2`

`leq (S x) O = False`

`leq (S x) (S y) = leq x y`

Evaluate function call `leq` t_1 t_2

1. Reduce t_1 to head normal form
2. If $t_1 = \texttt{O}$: apply rule
3. If $t_1 = (\texttt{S} \ldots)$: reduce t_2 to head normal form
4. If t_1 variable: bind t_1 to `O` or `(S _)` and proceed

$$\texttt{leq v } \underline{\texttt{(add w O)}} \rightsquigarrow_{\{\texttt{v}\mapsto\texttt{S z},\, \texttt{w}\mapsto\texttt{O}\}} \texttt{leq (S z) O}$$

Strict Equality

Needed narrowing solves equations t_1 `=:=` t_2

Interpretation of "`=:=`":

- strict equality on terms
- t_1 `=:=` t_2 satisfied if both sides reducible to same value (finite data term)
- undefined on infinite terms

```
f = 0 : f
g = 0 : g
```

⇝ `f =:= g` does not hold

- constructive form of equality (definable by standard rewrite rules)
- used in current functional and logic languages

Needed Narrowing: Properties

Sound and complete (w.r.t. strict equality)

Optimal strategy:

1. No unnecessary steps:
   Each step is needed, i.e., unavoidable to compute a solution.
2. Shortest derivations:
   If common subterms are shared, derivations have minimal length.
3. Minimal set of computed solutions:
   Solutions computed by two distinct derivations are independent.
4. Determinism:
   No nondeterministic step during evaluation of ground expressions
   (≈ functional programming)

Extension: Weakly Needed Narrowing

Overlapping rules: *parallel-or*

```
or True  _     = True
or _     True  = True
or False False = False
```

`or s t`: reduce `s` or `t`?

Solution of current functional logic languages:

- nondeterministically select one of the arguments
- extend definitional trees with *or* nodes
- extend needed narrowing to weakly needed narrowing

Theoretically better, practically more costly:

- parallel evaluation of both arguments

Nondeterministic Operations

Functional languages: each function call has at most one value

Functional logic languages can handle more:

Nondeterministic choice

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

`0 ? 1` (don't know) evaluates to `0` or `1`

Nondeterministic operations/functions

- interpretation: mapping from values into sets of values
- declarative semantics [JLP'99]
- supported in modern functional logic languages
- advantage compared to predicates: demand-driven evaluation

Programming with Nondeterministic Operations

Nondeterministic list insertion

```
insert e []     = [e]
insert e (x:xs) = (e : x : xs) ? (x : insert e xs)
```

Permutations of a list

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

Permutation sort

```
sorted []                      = []
sorted [x]                     = [x]
sorted (x1:x2:xs) | x1 ≤ x2 = x1 : sorted (x2:xs)

psort xs = sorted (perm xs)
```

Reduced search space due to demand-driven evaluation of `(perm xs)`

Programming with Nondeterministic Operations

Advantages of nondeterministic operations as generators:

- demand-driven generation of solutions
- modular program structure, no floundering

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

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

Permutation sort for `[`n`,`$n$$-$`1,...,2,1]`: #or-branches/disjunctions

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

Call-Time vs. Need-Time Choice

Subtle aspect of nondeterministic operations: treatment as arguments

```
   coin = 0 ? 1                    double = x+x
```

```
double coin
 ⇝ coin+coin     ⇝*    0 | 1 | 1 | 2   need-time choice
 ⇝ double 0 | double 1  ⇝*  0 | 2   call-time choice
```

Call-time choice

- semantics with “least astonishment”
- declarative foundation: CRWL calculus [JLP’99]
- implementation: demand-driven + sharing
- used in current functional logic languages

Residuation

Narrowing

- resolution extended to functional logic programming
- sound, complete
- efficient (optimal) by exploiting functional information

Alternative principle: Residuation (Escher, Life, NUE-Prolog, Oz,...)

- evaluate functions only deterministically
- suspend function calls if necessary
- encode nondeterminism in predicates or disjunctions
- concurrency primitive required:
  "`c1 & c2`" evaluates constraints `c1` and `c2` concurrently

Residuation: Example

```
add O     y = y                  nat O     = Success
add (S x) y = S(add x y)         nat (S x) = nat x
```

Evaluate function `add` by residuation:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{add y O =:= S O \& nat }\underline{\texttt{ynat y}}} \\
\to_{\{y\mapsto S\,x\}} & \underline{\texttt{add (S x) O}}\texttt{ =:= S O \& nat x} \\
\to_{\{\}} & \underline{\texttt{S (add x O) =:= S O}}\texttt{ \& nat x} \\
\to_{\{\}} & \texttt{add x O =:= O \& }\underline{\texttt{nat x}} \\
\to_{\{x\mapsto O\}} & \underline{\texttt{add O O}}\texttt{ =:= O \& Success} \\
\to_{\{\}} & \underline{\texttt{O =:= O}}\texttt{ \& Success} \\
\to_{\{\}} & \underline{\texttt{Success \& Success}} \\
\to_{\{\}} & \texttt{Success}
\end{array}
$$

Narrowing vs. Residuation

Narrowing

- sound and complete
- possible nondeterministic evaluation of functions
- optimal for particular classes of programs

Residuation

- incomplete (floundering)
- deterministic evaluation of functions
- supports concurrency (declarative concurrency)
- method to connect external functions

No clear winner ⇝ combine narrowing + residuation

Possible by adding *flexible*/*rigid* tags in definitional trees

- flexible function: evaluated by narrowing
- rigid function: suspends on free argument variable

External Operations

Narrowing not applicable (no explicit defining rules available)

Appropriate model: residuation

Declarative interpretation: defined by infinite set of rules

External arithmetic operations

```
0 + 0 = 0          0 * 0 = 0
0 + 1 = 1          1 * 1 = 1
1 + 1 = 2          2 * 2 = 4
...                ...
```

Implemented in some other language:

- rules not accessible
- can't deal with unevaluated/free arguments
- reduce arguments to ground values before the call
- suspend in case of free variable (residuation)

Higher-order Operations

Important technique for generic programming and code reuse

Map a function on all list elements

```
map :: (a->b) -> [a] -> [b]
map _ []       = []
map f (x:xs) = f x : map f xs

map double [1,2,3]      ⇝*  [2,4,6]
map (\x->x*x) [2,3,4]  ⇝*  [4,9,16]
```

Implementation:

- primitive operation `apply`: `apply f e` ⇝ `f e`
- sufficient to support higher-order functional programming

Problem: application of unknown functions?

- instantiate function variable: costly
- pragmatic solution: function application is rigid (i.e., no guessing)

Constraints

- occur in conditions of conditional rules
- restrict applicability: solve constraints before applying rule
- no syntactic extension necessary:
  constraint ≈ expression of type `Success`

Basic constraints

```
-- strict equality
(=:=) :: a -> a -> Success

-- concurrenct conjunction
(&)    :: Success -> Success -> Success

-- always satisfied
success :: Success
```

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

Constraints

Constraints are ordinary expressions ⇝ pass as arguments or results

Constraint combinator

```
allValid :: [Success] -> Success
allValid []       = success
allValid (c:cs) = c & allValid cs
```

Constraint programming: add constraints to deal with specific domains

Finite domain constraints

```
domain       :: [Int] -> Int -> Int -> Success
allDifferent :: [Int] -> Success
labeling     :: [LabelingOption] -> [Int] -> Success
```

Integration of constraint programming as in CLP

Combined with lazy higher-order programming

Constraints: SuDoku Solver

SuDoku puzzle: 9 × 9 matrix of digits

Representation: matrix `m` (list of lists of FD variables)

9			2			5		
	4			6			3	
		3						6
			9			2		
				5			8	
		7			4			3
7						1		
	5			2			4	
		1			6			9

SuDoku Solver with FD Constraints

```
sudoku :: [[Int]] -> Success
sudoku m =
  domain (concat m) 1 9                           &
  allValid (map allDifferent m)                   &
  allValid (map allDifferent (transpose m))       &
  allValid (map allDifferent (squaresOfNine m))   &
  labeling [FirstFailConstrained] (concat m)
```

Function Patterns

Requirement on programs: constructor-based rules

Last element of a list

```
last (xs++[e]) = e          -- not allowed
```

Eliminate non-constructor pattern:

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

Disadvantage: strict equality evaluates *all* arguments

last [failed,3] $\leadsto^*$ *failure (instead of 3)*

Solution: allow function patterns (patterns with defined functions)

Possible due to functional logic kernel!

Function Patterns: Transformational Semantics

Function pattern ≈ set of patterns where functions are evaluated

Evaluations of `xs++[e]`

$$\texttt{xs++[e]} \leadsto^*_{\texttt{xs}\mapsto\texttt{[e]}} \texttt{[e]}$$
$$\texttt{xs++[e]} \leadsto^*_{\texttt{xs}\mapsto\texttt{[x1]}} \texttt{[x1,e]}$$
$$\texttt{xs++[e]} \leadsto^*_{\texttt{xs}\mapsto\texttt{[x1,x2]}} \texttt{[x1,x2,e]}$$

...

Interpretation of `last (xs++[e]) = e`

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

- $\texttt{last [failed,3]} \leadsto^* \texttt{3}$
- implementation: demand-driven function pattern unification
- powerful concept to express transformation problems

Encapsulated Search

Encapsulating nondeterministic search is important

- declarative I/O ≈ transformation on the outside world
- “can’t clone the outside world”
- nondeterministic search between I/O must be encapsulated
- complication: demand-driven evaluation + sharing + “findall”

```
let y=coin in findall(...y...)
```

- evaluate `coin` inside or outside the capsule?
- order of solutions might depend on evaluation time

Better: encapsulate search on I/O (top) level

Encapsulated Search

Search primitive on I/O level

```
getSearchTree :: a -> IO (SearchTree a)

data SearchTree a = Or [SearchTree a]
                  | Val a
                  | Fail
```

- strong encapsulation (clone search expression):
  avoid sharing problems
- compute search tree demand-driven
- define concrete search strategies as tree traversals

Applications

Application areas: areas of individual paradigms +

Functional logic design patterns

- constraint constructor: generate only valid data (functions, constraints, programming with failure)
- locally defined global identifier: structures with unique references (functions, logic variables)
- …

General advantage: high-level interfaces for application libraries

- GUIs
- web programming
- databases
- distributed programming
- …

Applications: GUI Programming

42

Graphical User Interfaces (GUIs)

- layout structure: hierarchical structure ⇝ algebraic data type
- logical structure: dependencies in structure ⇝ logic variables
- event handlers ⇝ functions associated to layout structures
- advantages: compositional, less error prone

Specification of a counter GUI

```
Col [Entry [WRef val, Text "0", Background "yellow"],
    Row [Button (updateValue incr val) [Text "Increment"],
         Button (setValue val "0")     [Text "Reset"],
         Button exitGUI                [Text "Stop"]  ]]
   where val free
```

Conclusions

Combining declarative paradigms is possible and useful

- functional notation: more than syntactic sugar
- exploit functions: better strategies without loosing generality
- needed narrowing: sound, complete, optimal
- demand-driven search $\rightsquigarrow$ search space reduction
- residuation $\rightsquigarrow$ concurrency, clean connection to external functions
- more declarative style of programming: no cuts, no side effects,...
- appropriate abstractions for high-level software development

One paradigm: **Declarative Programming**