Declarative Multi-paradigm Programming

Michael Hanus

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

WFLP/WLP 2014

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
- non-deterministic search
- unification

Constraint Languages

- specific domains
- efficient constraint solving

All features are useful ⇝ declarative multi-paradigm languages

Declarative Multi-paradigm 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)

Declarative Multi-paradigm Languages: Approaches

Extend logic languages

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

Extend functional languages

- add logic features (logic variables, non-determinism) (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.curry-language.org`

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
```

$$\begin{array}{lclclclclc}
\underline{\texttt{double (1+2)}} & \Rightarrow & \underline{\texttt{(1+2)}}\texttt{+(1+2)} & \Rightarrow & \texttt{3+}\underline{\texttt{(1+2)}} & \Rightarrow & \underline{\texttt{3+3}} & \Rightarrow & \texttt{6} \\
\underline{\texttt{double (1+2)}} & \Rightarrow & \texttt{(1+2)+}\underline{\texttt{(1+2)}} & \Rightarrow & \underline{\texttt{(1+2)}}\texttt{+3} & \Rightarrow & \underline{\texttt{3+3}} & \Rightarrow & \texttt{6} \\
\texttt{double }\underline{\texttt{(1+2)}} & \Rightarrow & \underline{\texttt{double 3}} & \Rightarrow & \underline{\texttt{3+3}} & \Rightarrow & \texttt{6} & &
\end{array}$$

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`,...,`e_n`]` $\approx$ e_1`:`e_2`:...:`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

Non-constructor-based rules

Rules with extra variables

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

non-constructive, forbidden to provide efficient evaluation strategy
allowed in contrast to traditional rewrite systems

Functional Logic Computations: Narrowing

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

Narrowing = variable instantiation $+$ rewriting

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

p : non-variable position in t

$l \to r$: 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 = Z | S Nat                 leq Z     _     = True
add Z     y = y                      leq (S _) Z     = False
add (S x) y = S(add x y)             leq (S x) (S y) = leq x y
```

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

Another narrowing computation:

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

And another narrowing computation:

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

Avoid last derivation by non-mgu in first step:

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

Needed Narrowing [JACM'00]

- constructive method to compute positions and unifiers
- defined on inductively sequential rewrite systems:
  there is always a discriminating argument
- formal definition: organize rules in *definitional trees* [Antoy'92]
- here: transform rules into `case` expressions

```
add Z     y  =  y                 add x y = case x of
add (S x) y  =  S(add x y)   ⇒                 Z   → y
                                               S z → S(add z y)
```

```
leq Z     _     = True      ⇒   leq x y = case x of
leq (S _) Z     = False                Z   → True
leq (S x) (S y) = leq x y              S a → case y of
                                                 Z   → False
                                                 S b → leq a b
```

Needed Narrowing

`case` expressions

- standard compile-time transformation to implement pattern matching
- guide *lazy* evaluation strategy

```
leq x y = case x of Z   → True
                    S a → case y of Z   → False
                                    S b → leq a b
```

Evaluate function call `leq` t_1 t_2

1. Evaluate t_1 to head normal form h_1
2. If $h_1 =$ `Z`: return `True`
3. If $h_1 =$ (`S` ...): evaluate t_2 to head normal form
4. If h_1 variable: bind h_1 to `Z` or `(S _)` and proceed

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

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 non-deterministic step during evaluation of ground expressions
   (≈ functional programming)

Note: similar results unknown for purely logic programming!

Non-Deterministic Operations

Non-deterministic choice

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

- `0 ? 1` (don't know) evaluates to `0` or `1`
- `case` expressions not sufficient (no discriminating argument)
- *weakly needed narrowing* = needed narrowing + choice

Non-deterministic operations/functions

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

Programming with Non-Deterministic Operations

Non-deterministic list insertion

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

Permutations of a list

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

Permutation sort

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

psort xs = sorted (permute xs)
```

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

Programming with Non-Deterministic Operations

Advantages of non-deterministic operations as generators:

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

$$\texttt{psort [5,4,3,2,1]} \leadsto \texttt{sorted (permute [5,4,3,2,1])}$$

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

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

Permutation sort for $\texttt{[}n,n{-}1,\texttt{...,2,1]}$: #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 non-deterministic operations: treatment of arguments

```
coin = 0 ? 1                  double x = 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 [González-Moreno et al., 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 non-determinism in predicates or disjunctions
- concurrency primitive required:
  “`c1 & c2`” evaluates constraints `c1` and `c2` concurrently

Residuation: Example

```
add Z     y = y                  nat Z     = success
add (S x) y = S(add x y)         nat (S x) = nat x
```

Evaluate function `add` by residuation:

$$\begin{array}{ll}
 & \texttt{add y Z =:= S Z \& nat }\underline{\texttt{ynat y}}\\
\to_{\{y\mapsto S\,x\}} & \underline{\texttt{add (S x) Z}}\texttt{ =:= S Z \& nat x}\\
\to_{\{\}} & \underline{\texttt{S (add x Z) =:= S Z}}\texttt{ \& nat x}\\
\to_{\{\}} & \texttt{add x Z =:= Z \& }\underline{\texttt{nat x}}\\
\to_{\{x\mapsto Z\}} & \underline{\texttt{add Z Z}}\texttt{ =:= Z \& success}\\
\to_{\{\}} & \underline{\texttt{Z =:= Z}}\texttt{ \& success}\\
\to_{\{\}} & \underline{\texttt{success \& success}}\\
\to_{\{\}} & \texttt{success}
\end{array}$$

Narrowing vs. Residuation

Narrowing

- sound and complete
- possible non-deterministic 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 `case` expressions

- flexible `case`: instantiate free argument variable (narrowing)
- rigid `case`: suspend on free argument variable (residuation)

External Operations

Narrowing not applicable (no explicit defining rules available)

Appropriate model: residuation

Declarative interpretation: defined by infinite set of rules

External arithmetic operations

$$\begin{array}{ll} 0 + 0 = 0 & 0 * 0 = 0 \\ 0 + 1 = 1 & 1 * 1 = 1 \\ 1 + 1 = 2 & 2 * 2 = 4 \\ \vdots & \vdots \end{array}$$

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)
```

Functional Patterns

Requirement on programs: constructor-based rules

Last element of a list

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

Eliminate non-constructor pattern with extra-variables:

```
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 functional patterns (patterns with defined functions)

Possible due to functional logic kernel!

Functional Patterns: Transformational Semantics

Functional 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
...
```

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

Application: XML Processing

```
<contacts>
  <entry>
    <name>Hanus</name>
    <first>Michael</first>
    <phone>0431/8807271</phone>
    <email>mh@informatik.uni-kiel.de</email>
    <email>hanus@acm.org</email>
  </entry>
  <entry>
    <name>Smith</name>
    <first>William</first>
    <nickname>Bill</nickname>
    <phone>+1-987-742-9388</phone>
  </entry>
</contacts>
```

- processing: matching, querying, transformation
- basically term structures, declarative languages seem appropriate
- problems: structure often incompletely specified, evolves over time
- specialized languages: XPath, XQuery, XSLT, Xcerpt [Bry et al. '02]

An eDSL for XML Processing

XML documents are term structures:

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

Useful abstractions

```
xml t c = XElem t [] c
xtxt s  = XText s
```

```
xml "entry" [xml "name"  [xtxt "Hanus"],
             xml "first" [xtxt "Michael"],
             xml "phone" [xtxt "0431/8807271"]]
```

pretty printing ⇒

```
<entry>
  <name>Hanus</name>
  <first>Michael</first>
  <phone>0431/8807271</phone>
</entry>
```

Matching in XML Documents

Extract name and phone number by pattern matching:

```
getNamePhone
   (xml "entry"
         [xml "name"   [xtxt name],
          _,
          xml "phone" [xtxt phone]]) = name++": "++phone
```

Functional patterns improves readability, but still problematic:

- exact XML structure must be known
- many details of large structures often irrelevant
- change in structure ⇝ update all patterns

Better: define appropriate abstractions and use them in functional patterns

Feature: Partial Patterns

- do no enumerate all children of a structure
- provide flexibility for future structure extensions

```
getNamePhone
 (xml "entry"
   (with [xml "name" [xtxt name],
          xml "phone" [xtxt phone]])) = name++": "++phone
```

```
with :: [a]  → [a]   -- return some list containing elements
with []      = _
with (x:xs) = _ ++ x : with xs
```

Example: `with [1,2]` ⇝ $x_1 : \ldots : x_m : 1 : y_1 : \ldots : y_n : 2 : zs$

Feature: Unordered Patterns

- order of children unspecified
- provide flexibility for future structural changes

```
getNamePhone
 (xml "entry"
  (with
   (anyorder [xml "phone" [xtxt phone],
              xml "name"  [xtxt name]]))) = name++": "++phone
```

```
-- Return a permutation of the input list:
anyorder :: [a] → [a]
anyorder []             = []
anyorder (xs++[x]++ys) = x : anyorder (xs++ys)
```

Feature: Patterns at Arbitrary Depth

Deep pattern

- structure at the root or at a descendant (at arbitrary depth) of the root
- ease queries in complex structures
- provide flexibility for future structural changes

```
getNamePhone
 (deepXml "entry"
   (with [xml "name"  [xtxt name],
          xml "phone" [xtxt phone]])) = name++": "++phone
```

```
deepXml :: String → [XmlExp] → XmlExp
deepXml tag elems = xml tag elems
deepXml tag elems = xml _ (_ ++ [deepXml tag elems] ++ _)
```

Example: XML Pattern Matching at Arbitrary Depth

```
getPhone (deepXml "phone" [xtxt num]) = num
```

```
getPhone (<contacts>
            <entry>
              <name>Hanus</name>
              <first>Michael</first>
              <phone>0431/8807271</phone>
              <email>mh@informatik.uni-kiel.de</email>
              <email>hanus@acm.org</email>
            </entry>
            <entry>
              <name>Smith</name>
              <first>William</first>
              <nickname>Bill</nickname>
              <phone>+1-987-742-9388</phone>
            </entry>
          </contacts>)
⇝ "0431-8807271"
⇝ "+1-987-742-9388"
```

More XML Processing

Transformation of Documents

- transform XML documents into other XML or HTML documents
- transformation task almost trivial in pattern-based languages, e.g.:
  transform pattern = *newdoc*

```
transPhone (deepXml "entry" (with [xml "name"  [xtxt n],
                                   xml "first" [xtxt f],
                                   xml "phone" phone])) =
  xml "phonename" [xml "phone" phone,
                   xml "fullname" [xtxt (f++' ':n)]]
```

Accumulate Results

- accumulation of global or intermediate results
- requires “`findall`” (*encapsulated search*)

Encapsulated Search

Encapsulating non-deterministic search is important

- accumulate intermediate results
- select optimal/best solutions
- non-deterministic search between I/O must be encapsulated
- complication: demand-driven evaluation + sharing + “`findall`”

```
let y=0?1 in findall (...y...)
```

- evaluate “`0?1`” inside or outside the capsule?
- order of solutions might depend on evaluation time

Declarative capsule: set functions

Set Functions [PPDP'09]

Idea

Associate to any operation f a new operation $f_{\mathcal{S}}$ (set function)

- $f_{\mathcal{S}}$ computes set of all values computed by f
- $(f_{\mathcal{S}}\ \ e) \approx$ sets of all non-deterministic values of $(f\ \ v)$ if v is a value of e
- capture non-determinism of f
- exclude non-determinism originating from arguments
- order-independent encapsulation of non-determinism

```
coin = 0 ? 1      ⇝ coinS ={0,1}
id x = x          ⇝ idS v ={v}   for all values v

bigCoin = 2 ? 4
f x = coin + x

fS bigCoin  ⇝   {2,3} or {4,5}
```

Programming with Set Functions

n-queens puzzle

Place *n* queens on an *n* × *n* board without capturing:

- represent placement by a permutation (row of each queen)
- choose a *safe* permutation

A permutation is not safe if some queens are in the same diagonal:

```
unsafe (_++[x]++y++[z]++_)  =  abs (x-z) =:= length y + 1

queens n | isEmpty (unsafe_S p) = p
  where p = permute [1..n]
```

Note: use of set function is important here
(all occurrences of `p` must denote the *same* permutation!)

Applications

Application areas: areas of individual paradigms +

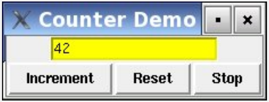
Functional logic design patterns [FLOPS'02, WFLP'11]

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

High-level interfaces for application libraries

- GUIs
- (type-safe) web programming
- databases
- string parsing
- testing
- ...

Applications: GUI Programming



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
```

Implementations

MCC (Münster Curry Compiler)

- compiles to C
- supports programmable search, real arithmetic constraints

PAKCS (Portland Aachen Kiel Curry System)

- compiles to Prolog
- non-determinism by backtracking, various constraint solvers

KiCS2 (Kiel Curry Compiler Vers. 2)

- compiles to Haskell (fastest for deterministic programs)
- various search strategies
  (depth-first, breadth-first, iterative deepening, parallel)
- programmable encapsulated (demand-driven) search

... (or try `http://www-ps.informatik.uni-kiel.de/smap/`)

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 $\leadsto$ search space reduction
- residuation $\leadsto$ 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**