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 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 $\leadsto$ 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
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

끄http://www.curry-language.org
Basic Concept: Functional Computation

**Functional program**: set of functions defined by equations/rules

\[
\text{double } x = x + x
\]

**Functional computation**: replace subterms by equal subterms

\[
\begin{align*}
\text{double } (1+2) & \Rightarrow (1+2) + (1+2) \\
 & \Rightarrow 3 + (1+2) \\
 & \Rightarrow 3 + 3 \\
 & \Rightarrow 6
\end{align*}
\]

Another computation:

\[
\begin{align*}
\text{double } (1+2) & \Rightarrow (1+2) + (1+2) \\
 & \Rightarrow (1+2) + 3 \\
 & \Rightarrow 3 + 3 \\
 & \Rightarrow 6
\end{align*}
\]

And another computation:

\[
\begin{align*}
\text{double } (1+2) & \Rightarrow \text{double } 3 \\
 & \Rightarrow 3 + 3 \\
 & \Rightarrow 6
\end{align*}
\]
double \( x = x + x \)

\[
\begin{align*}
double \ (1+2) & \Rightarrow (1+2)+(1+2) & \Rightarrow 3+(1+2) & \Rightarrow 3+3 & \Rightarrow 6 \\
double \ (1+2) & \Rightarrow (1+2)+(1+2) & \Rightarrow (1+2)+3 & \Rightarrow 3+3 & \Rightarrow 6 \\
double \ (1+2) & \Rightarrow \text{double } 3 & \Rightarrow 3+3 & \Rightarrow 6
\end{align*}
\]

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?
Values in declarative languages: terms

\[
\text{data Bool } = \quad \text{True} \mid \text{False}
\]

Definition by pattern matching:

\[
\begin{align*}
\text{not True } &= \text{False} \\
\text{not False } &= \text{True}
\end{align*}
\]

Replacing equals by equals still valid:

\[
\text{not (not False) } \Rightarrow \text{not True } \Rightarrow \text{False}
\]
Algebraic Data Types: Lists

List of elements of type \( a \)

\[
\text{data List } a = \text{[] | } a : \text{List } a
\]

Some notation:

\[
\begin{align*}
[a] & \approx \text{List } a \\
[e_1, e_2, \ldots, e_n] & \approx e_1 : e_2 : \ldots : e_n : \text{[]}
\end{align*}
\]

List concatenation “++”

\[
(++) :: [a] \to [a] \to [a]
\]

\[
\begin{align*}
\text{[]} & \quad +++ y s = y s \\
(x:x s) & \quad +++ y s = x : x s++y s
\end{align*}
\]

\[
[1,2,3] \quad +++ \quad [4] \Rightarrow [1,2,3,4]
\]
List concatenation “++”

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

Use “++” to specify other list functions:

**Last element of a list:** \( \text{last } xs = e \) iff \( \exists ys: ys ++ [e] = xs \)

Direct implementation in a functional logic language:
- search for solutions w.r.t. existentially quantified variables
- solve equations over nested functional expressions

**Definition of \text{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

\[ \text{last } xs \mid ys++[e] =:= xs \\
= e \quad \text{where } ys,e \text{ free} \]

Non-constructive, forbidden to provide efficient evaluation strategy

allowed in contrast to traditional rewrite systems
Rewriting not sufficient in the presence of logic variables ⇝

**Narrowing** = variable instantiation + rewriting

**Narrowing step:** \( \widetilde{t \rightarrow_p l \rightarrow_r \sigma t'} \)

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

Why not most general unifiers?
Narrowing with mgu’s is not optimal

```
data Nat = Z | S Nat

add Z y = y
add (S x) y = S(add x y)

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

Another narrowing computation:

```
leq v (add w Z) leq v (add w Z) ⇝ \{v↦Z\} True
```

And another narrowing computation:

```
leq v (add w Z) ⇝ \{w↦Z\} leq v Z leq v Z ⇝ \{v↦S\} False
```

Avoid last derivation by non-mgu in first step:

```
leq v (add w Z) ⇝ \{v↦S\}, \{w↦Z\} 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

\[
\begin{align*}
\text{add } Z & \quad y = y \\
\text{add } (S \ x) \ y & = S(\text{add } x \ y) \\
\text{add } x \ y & = \text{case } x \ of \\
& \quad Z \rightarrow y \\
& \quad S \ z \rightarrow S(\text{add } z \ y)
\end{align*}
\]

\[
\begin{align*}
\text{leq } Z & \quad _ = \text{True} \\
\text{leq } (S \ _) \ Z & = \text{False} \\
\text{leq } (S \ x) \ (S \ y) & = \text{leq } x \ y \\
\text{leq } x \ y & = \text{case } x \ of \\
& \quad Z \rightarrow \text{True} \\
& \quad S \ a \rightarrow \text{case } y \ of \\
& \quad \quad Z \rightarrow \text{False} \\
& \quad \quad S \ b \rightarrow \text{leq } a \ b
\end{align*}
\]
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 \( \text{leq } t_1 \ t_2 \)

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

\[
\text{leq } v \ (\text{add } w \ Z) \ \sim \{v \mapsto S \ a, w \mapsto Z\} \ \text{leq } (S \ a) \ Z
\]
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

\[
\begin{align*}
  f &= 0 : f \\
g &= 0 : g
\end{align*}
\]

\( \leadsto f =:= g \) 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
   \((\approx\) functional programming)

Note: similar results unknown for purely logic programming!
Non-Deterministic Operations

### Non-deterministic choice

\[
\begin{align*}
x \ ? \ y &= x \\
x \ ? \ y &= y
\end{align*}
\]

- \(0 \ ? \ 1\) (don’t know) evaluates to 0 or 1
- \textit{case} expressions not sufficient (no discriminating argument)
- \textit{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

\[
\text{insert } e \; [] = [e] \\
\text{insert } e \; (x:xs) = (e : x : xs) \; ? \; (x : \text{insert } e \; xs)
\]

Permutations of a list

\[
\text{permute } [] = [] \\
\text{permute } (x:xs) = \text{insert } x \; (\text{permute } xs)
\]

Permutation sort

\[
\text{sorted } [] = [] \\
\text{sorted } [x] = [x] \\
\text{sorted } (x1:x2:xs) \mid x1 \leq x2 = x1 : \text{sorted } (x2:xs)
\]

\[
\text{psort } xs = \text{sorted } (\text{permute } xs)
\]

Reduced search space due to demand-driven evaluation of \((\text{permute } xs)\)
Advantages of non-deterministic operations as generators:

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

```
psort [5,4,3,2,1] → sorted (permute [5,4,3,2,1])
→* sorted (5 : 4 : permute [3,2,1])
```

 Effect: Permutations of \([3,2,1]\) are not enumerated!

Permutation sort for \([n, n-1, \ldots, 2, 1]\): #or-branches/disjunctions

<table>
<thead>
<tr>
<th>Length of the list:</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate-and-test</td>
<td>24</td>
<td>120</td>
<td>720</td>
<td>40320</td>
<td>3628800</td>
</tr>
<tr>
<td>test-of-generate</td>
<td>19</td>
<td>59</td>
<td>180</td>
<td>1637</td>
<td>14758</td>
</tr>
</tbody>
</table>
Call-Time vs. Need-Time Choice

Subtle aspect of non-deterministic operations: treatment of arguments

\[
\text{coin} = 0 \ ? \ 1 \quad \text{double} \ x = x + x
\]

\[
\text{double} \ \text{coin}
\]

\[
\leadsto \text{coin} + \text{coin} \quad \leadsto^* \ 0 \ | \ 1 \ | \ 1 \ | \ 2 \quad \text{need-time choice}
\]

\[
\leadsto \text{double} \ 0 \ | \ \text{double} \ 1 \quad \leadsto^* \ 0 \ | \ 2 \quad \text{call-time choice}
\]

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

\[
\text{add } Z \quad y = y \\
\text{add } (S \ x) \ y = S(\text{add } x \ y) \\
\text{nat } Z \quad = \text{ success}
\]

\[
\text{add } (S \ x) \ Z =:= S Z \ & \ \text{nat } x
\]

Evaluate function \text{add} by residuation:

\[
\begin{align*}
\text{add } y \ Z &=:= S Z \ & \ \text{nat } y
\rightarrow \{y \mapsto S x\} \quad \text{add } (S \ x) \ Z &=:= S Z \ & \ \text{nat } x \\
\rightarrow \{} \quad S (\text{add } x \ Z) &=:= S Z \ & \ \text{nat } x \\
\rightarrow \{} \quad \text{add } x \ Z &=:= Z \ & \ \text{nat } x \\
\rightarrow \{x \mapsto Z\} \quad \text{add } Z \ Z &=:= Z \ & \ \text{success} \\
\rightarrow \{} \quad Z &=:= Z \ & \ \text{success} \\
\rightarrow \{} \quad \text{success} \ & \ \text{success} \\
\rightarrow \{} \quad \text{success}
\end{align*}
\]
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 $\Rightarrow$ 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0</td>
<td>0</td>
</tr>
<tr>
<td>0 + 1</td>
<td>1</td>
</tr>
<tr>
<td>1 + 1</td>
<td>2</td>
</tr>
</tbody>
</table>

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 \rightarrow b) \rightarrow [a] \rightarrow [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

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

Implementation:

- **primitive operation** `apply`: `apply f e \rightarrow 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 \( \approx \) expression of type \( \text{Success} \)

Basic constraints

- **strict equality**
  \[ (=:=) :: a \to a \to \text{Success} \]

- **concurrent conjunction**
  \[ (&) :: \text{Success} \to \text{Success} \to \text{Success} \]

- **always satisfied**
  \[ \text{success} :: \text{Success} \]

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

Constraints are ordinary expressions \(\rightsquigarrow\) pass as arguments or results

Constraint combinator

```haskell
allValid :: [Success] → Success
allValid [] = success
allValid (c:cs) = c & allValid cs
```

Constraint programming: add constraints to deal with specific domains

Finite domain constraints

```haskell
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
SuDoku puzzle: $9 \times 9$ matrix of digits

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

SuDoku Solver with FD Constraints

```haskell
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

\[
\text{last } (xs++[e]) = e \quad -- \text{not allowed}
\]

Eliminate non-constructor pattern with extra-variables:

\[
\text{last } xs \mid ys++[e]=:=xs = e \quad \text{where } ys, e \text{ free}
\]

Disadvantage: strict equality evaluates all arguments

\[
\text{last } [\text{failed}, 3] \leadsto* \text{failure (instead of 3)}
\]

Solution: allow functional patterns (patterns with defined functions)
Possible due to functional logic kernel!
### Functional Pattern

A functional pattern is a set of patterns where functions are evaluated. Evaluation of `xs++[e]` can be represented as:

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xs++[e]</code></td>
<td><code>last (xs++[e]) = e</code></td>
</tr>
<tr>
<td><code>xs++[e]</code></td>
<td><code>last [e] = e</code></td>
</tr>
<tr>
<td><code>xs++[e]</code></td>
<td><code>last [x1,e] = e</code></td>
</tr>
<tr>
<td><code>xs++[e]</code></td>
<td><code>last [x1,x2,e] = e</code></td>
</tr>
</tbody>
</table>

...
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:

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

Useful abstractions:

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

```haskell
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>
```
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 \(\leadsto\) 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] ↦ x₁::...::xₘ:1::y₁::...::yₙ:2::zs
```
Feature: Unordered Patterns

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

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

`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

getchPhone (<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"
Transformation of Documents

- transform XML documents into other XML or HTML documents
- transformation task almost trivial in pattern-based languages, e.g.:
  
  \[\text{transform pattern} = \text{newdoc}\]

Accumulate Results

- accumulation of global or intermediate results
- requires "findall" (\textit{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
Idea

Associate to any operation $f$ a new operation $f_S$ (set function)

- $f_S$ computes set of all values computed by $f$
- $(f_S \ e) \approx$ 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

```plaintext
coin = 0 ? 1  \rightarrow coins = \{0,1\}

id x = x  \rightarrow id_S v = \{v\}  \text{ for all values } v

bigCoin = 2 \ ? 4
f x = coin + x

f_S bigCoin  \rightarrow  \{2,3\} \text{ or } \{4,5\}
```
**n-queens puzzle**

Place \( n \) queens on an \( n \times n \) 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:

\[
\text{unsafe} \ (\_++[x]++y++[z]++\_) \ = \ \text{abs} \,(x-z) \ := \ \text{length} \ y + 1
\]

\[
\text{queens} \ n \ | \ \text{isEmpty} \ (\text{unsafe} \ p) = p \\
\text{where} \ p = \text{permute} \ [1..n]
\]

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 $\leadsto$ algebraic data type
- logical structure: dependencies in structure $\leadsto$ logic variables
- event handlers $\leadsto$ 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 $\Rightarrow$ search space reduction
- residuation $\Rightarrow$ 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