Declarative Multi-Paradigm Programming in

\[ \text{Curry} \]

Michael Hanus

Christian-Albrechts-Universität Kiel
DECLARATIVE PROGRAMMING

General idea:

• no coding of algorithms

• description of 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)
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
• multi-paradigm language
  (higher-order concurrent functional logic language,
  features for high-level distributed programming)

• extension of Haskell (non-strict functional language)

• developed by an international initiative

• provide a standard for functional logic languages
  (research, teaching, application)

• several implementations available

• PAKCS (Portland Aachen Kiel Curry System):
  ➔ freely available implementation of Curry
  ➔ many libraries (GUI, HTML, XML, meta-programming,...)
  ➔ various tools (CurryDoc, CurryTest, Debuggers, Analyzers,...)
Values in imperative languages: basic types + pointer structures

Declarative languages: **algebraic data types** (Haskell-like syntax)

```
data Bool  = True  | False
data Nat   = Z     | S Nat
data List a = []   | a : List a       -- [a]
data Tree a = Leaf a | Node [Tree a]
data Int   = 0     | 1     | -1    | 2     | -2    | ...
```

**Value  ≈ data term, constructor term:**
well-formed expression containing variables and data type constructors

```
(S Z)   1:(2:[])  [1,2]  Node [Leaf 3, Node [Leaf 4, Leaf 5]]
```
# Functional (Curry) Programs

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

\[ f \ t_1 \ldots t_n \ | \ c = r \]

- **defined operation**
- **data terms**
- **condition** (optional)
- **expression**

<table>
<thead>
<tr>
<th>Equation</th>
<th>LaTeX</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 + y = y )</td>
<td>( 0 \leq y = \text{True} )</td>
</tr>
<tr>
<td>( (S \ x) + y = S(x+y) )</td>
<td>( (S \ x) \leq 0 = \text{False} )</td>
</tr>
<tr>
<td>( (S \ x) \leq (S \ y) = x \leq y )</td>
<td></td>
</tr>
<tr>
<td>( [] ++ ys = ys )</td>
<td></td>
</tr>
<tr>
<td>( (x:xs) ++ ys = x : (xs ++ ys) )</td>
<td></td>
</tr>
<tr>
<td>depth (Leaf _) = 1</td>
<td></td>
</tr>
<tr>
<td>depth (Node []) = 1</td>
<td></td>
</tr>
<tr>
<td>depth (Node (t:ts)) = max (1+depth t) (depth (Node ts))</td>
<td></td>
</tr>
</tbody>
</table>
EVALUATION: COMPUTING VALUES

Reduce expressions to their values

Replace equals by equals

Apply reduction step to a subterm (redex, reducible expression):

variables in rule’s left-hand side are universally quantified

匹配 lhs against subterm (instantiate these variables)

\[
\begin{align*}
0 + y &= y \\
(S \ x) + y &= S(x+y) \\
(S \ x) &= 0 \\
(S \ x) &\leq (S \ y) = x \leq y
\end{align*}
\]

\[(S \ 0) + (S \ 0) \to S (0 + (S \ 0)) \to S (S \ 0)\]
Expressions with several redexes: which evaluate first?

**Strict evaluation:** select an innermost redex (≈ call-by-value)

**Lazy evaluation:** select an outermost redex

\[
\begin{align*}
0 + y &= y & 0 \leq y &= \text{True} \\
(S \; x) + y &= S(x+y) & (S \; x) \leq 0 &= \text{False} \\
(S \; x) \leq (S \; y) &= x \leq y
\end{align*}
\]

**Strict evaluation:**
\[
0 \leq (S \; 0) + (S \; 0) \rightarrow 0 \leq (S \; (0 + (S \; 0))) \rightarrow 0 \leq (S \; (S \; 0)) \rightarrow \text{True}
\]

**Lazy evaluation:**
\[
0 \leq (S \; 0) + (S \; 0) \rightarrow \text{True}
\]
Strict evaluation might need more steps, but it can be even worse...

\[
\begin{align*}
0 + y &= y & 0 \leq y &= \text{True} \\
(S\ x) + y &= S(x+y) & (S\ x) \leq 0 &= \text{False} \\
(S\ x) \leq (S\ y) &= x \leq y
\end{align*}
\]

\[\begin{array}{c}
f = f
\end{array}\]

Lazy evaluation:
\[
0+0 \leq f \rightarrow 0 \leq f \rightarrow \text{True}
\]

Strict evaluation:
\[
0+0 \leq f \rightarrow 0+0 \leq f \rightarrow 0+0 \leq f \rightarrow \ldots
\]

Ideal strategy: evaluate only needed redexes
(i.e., redexes necessary to compute a value)

Determine needed redexes with definitional trees
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

\[
x_1 \leq x_2
\]

\[
0 \leq x_2 = \text{True} \quad (S \ x_3) \leq x_2
\]

\[
(S \ x_3) \leq 0 = \text{False} \quad (S \ x_3) \leq (S \ x_4) = x_3 \leq x_4
\]
Evaluating function call $t_1 \leq t_2$:

1. Reduce $t_1$ to head normal form (constructor-rooted expression)
2. If $t_1 = 0$: apply rule
3. If $t_1 = (S \ldots)$: reduce $t_2$ to head normal form
Properties of Reduction with Definitional Trees

- **Normalizing strategy**
  i.e., always computes value if it exists \( \approx \) sound and complete

- Independent on the order of rules

- Definitional trees can be automatically generated
  \( \rightarrow \) pattern matching compiler

- Identical to lazy functional languages (e.g., Miranda, Haskell) for the subclass of uniform programs
  (i.e., programs with strong left-to-right pattern matching)

- **Optimal strategy:** each reduction step is needed

- Easily extensible to more general classes
Previous functions: inductively defined on data structures

Sometimes overlapping rules more natural:

\[
\begin{align*}
\text{True} \lor x &= \text{True} \\
x \lor \text{True} &= \text{True} \\
\text{False} \lor \text{False} &= \text{False}
\end{align*}
\]

First two rules overlap on \( \text{True} \lor \text{True} \)

\[ \leadsto \text{Problem: no needed argument: } e_1 \lor e_2 \text{ evaluate } e_1 \text{ or } e_2? \]

Functional languages: backtracking: Evaluate \( e_1 \), if not successful: \( e_2 \)

Disadvantage: not normalizing (\( e_1 \) may not terminate)
**NON-DETERMINISTIC EVALUATION**

\[
\begin{align*}
\text{True} \lor x &= \text{True} \\
    x \lor \text{True} &= \text{True} \\
\text{False} \lor \text{False} &= \text{False}
\end{align*}
\]

Evaluation of \( e_1 \lor e_2 \)?

1. **Parallel reduction of** \( e_1 \) **and** \( e_2 \) [[Sekar/Ramakrishnan 93]]

2. **Non-deterministic reduction:** try \( \text{don’t know} \) \( e_1 \) or \( e_2 \)

Extension to definitional trees / pattern matching:

Introduce **or-nodes** to describe non-deterministic selection of redexes

\[ \leadsto \text{non-deterministic evaluation: } e \rightarrow e_1 \mid \cdots \mid e_n \]

\[ \text{disjunctive expression} \]

\[ \leadsto \text{non-deterministic functions} \]
NON-DETERMINISTIC / SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

more than one result on a given input

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

x ! y = x
x ! y = y
```
Rules must be constructor-based but not confluent:

→ more than one result on a given input

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

x ! y = x
x ! y = y

insert e [] = [e]
insert e (x:xs) = e : x : xs ! x : insert e xs
```
Rules must be constructor-based but not confluent:
\[ \leadsto \text{more than one result on a given input} \]

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

x ! y = x
x ! y = y

insert e [] = [e]
insert e (x:xs) = e : x : xs ! x : insert e xs

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

\[ \text{perm [1,2,3]} \leadsto [1,2,3] \mid [1,3,2] \mid [2,1,3] \mid \ldots \]
NON-DETERMINISTIC / SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

→ more than one result on a given input

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

x ! y = x
x ! y = y

insert e [] = [e]
insert e (x:xs) = e : x : xs ! x : insert e xs

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

perm [1,2,3] → [1,2,3] | [1,3,2] | [2,1,3] | ...

Demand-driven search (search space reduction): sorted (perm xs)
Logic Programming

Distinguished features:

- compute with partial information (constraints)
- deal with free variables in expressions
- compute solutions to free variables
- built-in search
- non-deterministic evaluation

Functional programming: values, no free variables

Logic programming: computed answers for free variables

Operational extension: instantiate free variables, if necessary
From Functional Programming to Logic Programming

\[
\begin{align*}
  f\ 0 &= 2 \\
  f\ 1 &= 3
\end{align*}
\]

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: \(\text{bind}\) and \(\text{reduce}\):

\[
\begin{align*}
  e &\leadsto \{\sigma_1\} e_1 | \cdots | \{\sigma_n\} e_n \\
\end{align*}
\]

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/JACM’00]
\[ x_1 \leq x_2 \]

\[ 0 \leq x_2 = \text{True} \quad (S \ x_3) \leq x_2 \]

\[ (S \ x_3)\leq 0 = \text{False} \quad (S \ x_3) \leq (S \ x_4) = x_3 \leq x_4 \]

**Evaluating function call** \( t_1 \leq t_2 \):

1. Reduce \( t_1 \) to head normal form
2. If \( t_1 = 0 \): apply rule
3. If \( t_1 = (S \ldots) \): reduce \( t_2 \) to head normal form
Evaluating function call $t_1 \leq t_2$:

1. Reduce $t_1$ to head normal form
2. If $t_1 = 0$: apply rule
3. If $t_1 = (S \ldots)$: reduce $t_2$ to head normal form
4. If $t_1$ variable: bind $t_1$ to 0 or $(S x)$
Properties of Needed Narrowing

Sound and complete (w.r.t. strict equality, no termination requirement)

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 Minimal set of computed solutions:
Two solutions \( \sigma \) and \( \sigma' \) computed by two distinct derivations are independent.
Properties of Needed Narrowing

Determinism:
No non-deterministic step during the evaluation of ground expressions
(≈ functional programming)

Restriction: inductively sequential rules
(i.e., no overlapping left-hand sides)

Extensible to

→ conditional rules [Hanus ICLP’95, Antoy/Braßel/Hanus PPDP’03]
→ overlapping left-hand sides [Antoy/Echahed/Hanus ICLP’97]
→ multiple right-hand sides [Antoy ALP’97]
→ higher-order rules [Hanus/Prehofer JFP’99]
→ concurrent evaluation [Hanus POPL’97]
Logic programming: solve goals, compute solutions

Functional logic programming: solve equations

**Strict equality:** identity on *finite* objects

(only reasonable notion of equality in the presence of non-terminating functions)
EQUATIONAL CONSTRAINTS

Logic programming: solve goals, compute solutions

Functional logic programming: solve equations

Strict equality: identity on finite objects
(only reasonable notion of equality in the presence of non-terminating functions)

Equational constraint \( e_1 \equiv e_2 \)
successful if both sides evaluable to unifiable data terms

\[ \Rightarrow e_1 \equiv e_2 \text{ does not hold if } e_1 \text{ or } e_2 \text{ undefined or infinite} \]

\[ \Rightarrow e_1 \equiv e_2 \text{ and } e_1, e_2 \text{ data terms } \approx \text{ unification in logic programming} \]
**FUNCTIONAL LOGIC PROGRAMMING: EXAMPLES**

List concatenation:

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

\[ \[] \quad ++ \quad ys \quad = \quad ys \]

\[ (x:xs) \quad ++ \quad ys \quad = \quad x \quad : \quad (xs \quad ++ \quad ys) \]

Functional programming:

\[ [1,2] \quad ++ \quad [3,4] \quad \leadsto \quad [1,2,3,4] \]

Logic programming:

\[ x \quad ++ \quad y \quad =:= \quad [1,2] \quad \leadsto \quad \{ x=[] \space , \space y=[1,2] \} \quad | \quad \{ x=[1] \space , \space y=[2] \} \quad | \quad \{ x=[1,2] \space , \space y=[] \} \]
List concatenation:

\[
(\text{++}) \::\ [a] \rightarrow [a] \rightarrow [a]
\]

\[
[] \rightarrow \text{++} \rightarrow \text{ys} = \text{ys}
\]

\[
(x:xs) \rightarrow \text{++} \rightarrow \text{ys} = x : (xs \rightarrow \text{++} \rightarrow \text{ys})
\]

Functional programming:

\[
[1,2] \rightarrow \text{++} \rightarrow \text{[3,4]} \rightarrow \sim \rightarrow \text{[1,2,3,4]}
\]

Logic programming:

\[
\text{x ++ y =:= [1,2]} \rightarrow \sim \rightarrow
\]

\[
\{x=[],y=[1,2]\} \mid \{x=[1],y=[2]\} \mid \{x=[1,2],y=[]\}
\]

Last list element:

\[
\text{last xs | ys} \rightarrow \text{++ [x]} =:= xs = x
\]
Non-deterministic functions for generating permutations:

\[
\begin{align*}
\text{insert } e \; [] &= [e] \\
\text{insert } e \; (x:xs) &= e:x:xs ! y:\text{insert } e \; xs \\
\text{perm } [] &= [] \\
\text{perm } (x:xs) &= \text{insert } x \; (\text{perm } xs)
\end{align*}
\]
Non-deterministic functions for generating permutations:

\[
\begin{align*}
\text{insert } e \; [] &= [e] \\
\text{insert } e \; (x:xs) &= e:x:xs \quad \text{!} \quad y:\text{insert } e \; xs \\
\text{perm } [] &= [] \\
\text{perm } (x:xs) &= \text{insert } x \; (\text{perm } xs)
\end{align*}
\]

Sorting lists with test-of-generate principle:

\[
\begin{align*}
\text{sorted } [] &= [] \\
\text{sorted } [x] &= [x] \\
\text{sorted } (x:y:ys) \mid x \leq y &= x : \text{sorted } (y:ys) \\
\text{psort } xs &= \text{sorted } (\text{perm } xs)
\end{align*}
\]
Advantages of non-deterministic functions as generators:

→ demand-driven generation of solutions (due to laziness)
→ modular program structure

\[ \text{psort } [5,4,3,2,1] \mapsto \text{sorted (perm } [5,4,3,2,1]) \]

\[ \mapsto^* \text{sorted (5:4:perm } [3,2,1]) \]

\[ \text{undefined: discard this alternative} \]

\[ \]

**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>
Logic Programming:

- compute with partial information \textit{(constraints)}
- data structures (constraint domain): constructor terms
- basic constraint: (strict) equality
- constraint solver: \textit{unification}

Constraint Programming: generalizes logic programming by

- new specific \textit{constraint domains} (e.g., reals, finite sets)
- new \textit{basic constraints} over these domains
- sophisticated \textit{constraint solvers} for these constraints
Constraint Programming over Reals

Constraint domain: real numbers

Basic constraints: equations / inequations over real arithmetic expressions

Constraint solvers: Gaussian elimination, simplex method

Examples:

5.1 =:= x + 3.5  \leadsto \{x=1.6\}

x <= 1.5 & x+1.3 >= 2.8  \leadsto \{x=1.5\}
EXAMPLE: CIRCUIT ANALYSIS

Define relation \( cvi \) between electrical circuit, voltage, and current

Circuits are defined by the data type

\[
data \text{ Circuit} = \text{ Resistor Float} \\
| \text{ Series Circuit Circuit} \\
| \text{ Parallel Circuit Circuit}
\]

Rules for relation \( cvi \):

\[
cvi (\text{Resistor } r) v i = v =:= i \times r \quad -- \text{Ohm’s law}
\]

\[
cvi (\text{Series } c1 c2) v i = v =:= v1 + v2 \quad -- \text{Kirchhoff’s law}
\]

\[
cvi (\text{Parallel } c1 c2) v i = i =:= i1 + i2 \quad -- \text{Kirchhoff’s law}
\]
Querying the circuit specification:

Current in a sequence of resistors:
\[ \text{cvi (Series (Resistor 180.0) (Resistor 470.0)) 5.0 i} \]
\[ \sim \{ i = 0.007692307692307693 \} \]

Relation between resistance and voltage in a circuit:
\[ \text{cvi (Series (Series (Resistor r) (Resistor r)) (Resistor r)) v 5.0} \]
\[ \sim \{ v=15.0*r \} \]

Also synthesis of circuits possible
Constraint domain: finite set of values

Basic constraints: equality / disequality / membership / . . .

Constraint solvers: OR methods (e.g., arc consistency)

Application areas: combinatorial problems
(job scheduling, timetabling, routing, . . .)

General method:

① define the domain of the variables (possible values)
② define the constraints between all variables
③ “labeling”, i.e., non-deterministic instantiation of the variables

constraint solver reduces the domain of the variables by sophisticated pruning techniques using the given constraints

Usually: finite domain \(\approx\) finite subset of integers
Example: A Crypto-Arithmetic Puzzle

Assign a different digit to each different letter such that the following calculation is valid:

\[
\begin{array}{c}
s
e
n
d \\
+ \\
m
o
r
e \\
\hline \\
m
o
n
\end{array}
\]

puzzle send more =

domain \{s,e,n,d,m,o,r,y\} 0 9 & \quad \text{-- define domain}

s > 0 & m > 0 &

all_different \{s,e,n,d,m,o,r,y\} &

1000 \times s + 100 \times e + 10 \times n + d \\
+ \\
1000 \times m + 100 \times o + 10 \times r + e \\
= 10000 \times m + 1000 \times o + 100 \times n + 10 \times e + y & \quad \text{-- define constraints}

labeling \{s,e,n,d,m,o,r,y\} \quad \text{-- instantiate variables}

puzzle send more \sim \{s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2\}
Disadvantage of narrowing:

- functions on recursive data structures ↝ narrowing may not terminate
- all rules must be explicitly known ↝ combination with external functions?
Disadvantage of narrowing:

➔ functions on recursive data structures ⇨ narrowing may not terminate
➔ all rules must be explicitly known ⇨ combination with external functions?

Solution: Delay function calls if a needed argument is free

⇝ residuation principle [Aït-Kaci et al. 87]
(used in Escher, Le Fun, Life, NUE-Prolog, Oz, . . .)
Disadvantage of narrowing:

- functions on recursive data structures $\leadsto$ narrowing may not terminate
- all rules must be explicitly known $\leadsto$ combination with external functions?

Solution: Delay function calls if a needed argument is free

$\leadsto$ residuation principle [Aït-Kaci et al. 87]
(used in Escher, Le Fun, Life, NUE-Prolog, Oz, . . .)

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
FLEXIBLE VS. RIGID FUNCTIONS

\[
\begin{align*}
  f \ 0 &= 2 \\
  f \ 1 &= 3 \\
\end{align*}
\]

rigid/flexible status not relevant for ground calls:

\[
  f \ 1 \leadsto 3
\]

\(f\) flexible:

\[
  f \ x =:= y \leadsto \{x=0,y=2\} \mid \{x=1,y=3\}
\]

\(f\) rigid:

\[
  f \ x =:= y \leadsto \text{suspend}
\]
**Flexible vs. Rigid Functions**

\[
\begin{align*}
  f \ 0 &= 2 \\
  f \ 1 &= 3
\end{align*}
\]

Rigid/flexible status not relevant for ground calls:

\[
f \ \ 1 \ \ \mapsto \ \ 3
\]

**f** flexible:

\[
f \ x \ =:= \ y \ \mapsto \ \begin{cases} x=0, y=2 & | & x=1, y=3 \end{cases}
\]

**f** rigid:

\[
f \ x \ =:= \ y \ \mapsto \ \text{suspend}
\]

\[
f \ x \ =:= \ y \ \& \ x =:= 1
\]
**Flexible vs. Rigid Functions**

\[
\begin{align*}
  f \ 0 &= 2 \\
  f \ 1 &= 3
\end{align*}
\]

rigid/flexible status not relevant for ground calls:

\[
f \ 1 \ \rightarrow \ 3
\]

\(f\) flexible:

\[
f \ x =:= y \ \rightarrow \ \{x=0,y=2\} \ \mid \ \{x=1,y=3\}
\]

\(f\) rigid:

\[
f \ x =:= y \ \rightarrow \ \text{suspend}
\]

\[
f \ x =:= y \ \& \ x =:= 1 \ \rightarrow \ \{x=1\} \ f \ 1 =:= y \quad (\text{suspend } f \ x)
\]
rigid/flexible status not relevant for ground calls:

\[ f \ 1 \rightarrow \ 3 \]

\( f \) flexible:

\[ f \ x =:= y \rightarrow \ \{x=0,y=2\} \mid \{x=1,y=3\} \]

\( f \) rigid:

\[ f \ x =:= y \rightarrow \ suspend \]

\[ f \ x =:= y \land x =:= 1 \rightarrow \ \{x=1\} \ f \ 1 =:= y \quad \text{(suspend } f \ x) \]

\[ \rightarrow \ \{x=1\} \ 3 =:= y \quad \text{(evaluate } f \ 1) \]
flexible vs. rigid functions

\[
f 0 = 2 \\
f 1 = 3
\]

rigid/flexible status not relevant for ground calls:

\[
f 1 \quad \rightsquigarrow \quad 3
\]

\( f \) flexible:

\[
f \ x \ =:= \ y \quad \rightsquigarrow \quad \{x=0,y=2\} \mid \{x=1,y=3\}
\]

\( f \) rigid:

\[
f \ x \ =:= \ y \quad \rightsquigarrow \quad suspend
\]

\[
f \ x \ =:= \ y \ \& \ x =:= 1 \quad \rightsquigarrow \quad \{x=1\} \quad f \ 1 \ =:= \ y \quad (suspend \ f \ x)
\]

\[
\quad \rightsquigarrow \quad \{x=1\} \ 3 \ =:= \ y \quad (evaluate \ f \ 1)
\]

\[
\quad \rightsquigarrow \quad \{x=1,y=3\}
\]

Default in Curry: flexible (except for predefined and I/O functions)
## Unification of Declarative Computation Models

<table>
<thead>
<tr>
<th>Computation model</th>
<th>Restrictions on programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needed narrowing</td>
<td>inductively sequential rules; optimal strategy</td>
</tr>
<tr>
<td>Weakly needed narrowing (∼Babel)</td>
<td>only flexible functions</td>
</tr>
<tr>
<td>Resolution (∼Prolog)</td>
<td>only (flexible) predicates (∼ constraints)</td>
</tr>
<tr>
<td>Lazy functional languages (∼Haskell)</td>
<td>no free variables in expressions</td>
</tr>
<tr>
<td>Parallel functional langs. (∼Goffin, Eden)</td>
<td>only rigid functions, concurrent conjunction</td>
</tr>
<tr>
<td>Residuation (∼Life, Oz)</td>
<td>constraints are flexible; all others are rigid</td>
</tr>
</tbody>
</table>
**SUMMARY: CURRY PROGRAMS**

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

\[
\begin{align*}
& f \ t_1 \ldots \ t_n \mid c = r \\
& \text{defined operation} \quad \text{data terms} \quad \text{constraint (optional)} \quad \text{expression}
\end{align*}
\]

\[
\begin{align*}
\text{conc} \ [\] & \quad y = y \\
\text{conc} \ (x:xs) & \quad y = x : \text{conc} \ xs \ y \\
\text{last} \ xs \mid \text{conc} \ y [x] & \quad =: x \\
& \quad = x \quad \text{where } x,y \text{ free}
\end{align*}
\]
SUMMARY: EXPRESSIONS

\[
e ::= e_0 \ldots e_n (constants) \\
x (variables x) \\
(e_0 \ldots e_n) (application) \\
\langle x \rightarrow e \rangle (abstraction) \\
\text{if } b \text{ then } e_1 \text{ else } e_2 (conditional)
\]
SUMMARY: EXPRESSIONS

$$e ::=$$

- $c$ (constants)
- $x$ (variables $x$)
- $(e_0 \ e_1 \ldots e_n)$ (application)
- $\lambda x \to e$ (abstraction)
- if $b$ then $e_1$ else $e_2$ (conditional)
- $e_1=:=e_2$ (equational constraint)
- $e_1 \ & \ e_2$ (concurrent conjunction)
- let $x_1,\ldots,x_n$ free in $e$ (existential quantification)
SUMMARY: EXPRESSIONS

\[ e ::= \]
\[ e \in \{ \text{constants} \} \]
\[ x \in \{ \text{variables } x \} \]
\[ (e_0 \ e_1 \ldots e_n) \in \{ \text{application} \} \]
\[ \lambda x \to e \in \{ \text{abstraction} \} \]
\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \in \{ \text{conditional} \} \]
\[ e_1 =:= e_2 \in \{ \text{equational constraint} \} \]
\[ e_1 \& e_2 \in \{ \text{concurrent conjunction} \} \]
\[ \text{let } x_1, \ldots, x_n \text{ free in } e \in \{ \text{existential quantification} \} \]

Equational constraints over functional expressions:
\[ \text{conc } ys \ [x] =:= [1,2] \leadsto \{ys=[1], x=2\} \]

Further constraints: real arithmetic, finite domain, ports (\(\leadsto\) OOP)
FEATURES OF CURRY

Curry’s basic operational model:

- conservative extension of lazy functional and (concurrent) logic programming
- generalization of concurrent constraint programming with lazy (optimal) strategy [POPL’97, WFLP’02, WRS’02, ENTCS76]
Curry’s basic operational model:

- conservative extension of lazy functional and (concurrent) logic programming
- generalization of concurrent constraint programming with lazy (optimal) strategy [POPL’97, WFLP’02, WRS’02, ENTCS76]

Features for application programming:

- types, higher-order functions, modules
- monadic I/O
- encapsulated search [PLILP’98]
- ports for distributed programming [PPDP’99]
- libraries for
  - constraint programming
  - GUI programming [PADL’00]
  - HTML programming [PADL’01]
  - XML programming
  - meta-programming
  - persistent terms
  - ...

CAU Kiel  Michael Hanus 36
Integration of different programming paradigms is possible

Functional programming is a good starting point:

- lazy evaluation → modularity, optimal evaluation
- higher-order functions → code reuse, design patterns
- polymorphism → type safety, static checking

Stepwise extensible in a conservative manner to cover

- logic programming: non-determinism, free variables
- constraint programming: specific constraint structures
- concurrent programming: suspending function calls, synchronization on logical variables
- object-oriented programming: constraint functions, ports [IFL 2000]
- imperative programming: monadic I/O, sequential composition (∼ Haskell)
- distributed programming: external ports [PPDP‘99]
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

• do not teach two paradigms, but one: declarative programming

[PLILP’97]

• choose the most appropriate features for application programming
APPLICATION: HTML/CGI PROGRAMMING

Early days of the World Wide Web: web pages with static contents
Common Gateway Interface (CGI): web pages with dynamic contents

Retrieval of a dynamic page:
- server executes a program
- program computes an HTML string, writes it to stdout
- server sends result back to client

HTML with input elements (forms):
- client fills out input elements
- input values are sent to server
- server program decodes input values for computing its answer
Traditional CGI Programming

CGI programs on the server can be written in any programming language

- access to environment variables (for input values)
- writes a string to stdout

Scripting languages: (Perl, Tk, ...)

- simple programming of single pages
- error-prone: correctness of HTML result not ensured
- difficult programming of interaction sequences

Specialized languages: (MAWL, DynDoc, ...)

- HTML support (structure checking)
- interaction support (partially)
- restricted or connection to existing languages
Library implemented in Curry

Exploit functional and logic features for

- HTML support (data type for HTML structures)
- simple access to input values (free variables and environments)
- simple programming of interactions (event handlers)
- wrapper for hiding details

Exploit imperative features for

- environment access (files, data bases, ...)

Domain-specific language for HTML/CGI programming
Data type for representing HTML expressions:

```haskell
data HtmlExp = HtmlText String
  | HtmlStruct String [(String,String)] [HtmlExp]
```

Some useful abbreviations:

- `htxt s = HtmlText (htmlQuote s)`
  
  -- plain string

- `bold hexps = HtmlStruct "B" [] hexps`
  
  -- bold font

- `italic hexps = HtmlStruct "I" [] hexps`
  
  -- italic font

- `h1 hexps = HtmlStruct "H1" [] hexps`
  
  -- main header

Example:

```
[h1 [htxt "1. Hello World"],
  italic [htxt "Hello"],
  bold [htxt "world!"],
]
```

Advantage:

- static checking of HTML structure
MODELING HTML

Data type for representing HTML expressions:

```haskell
data HtmlExp = HtmlText String
             | HtmlStruct String [(String,String)] [HtmlExp]
```

Some useful abbreviations:

- `htxt s = HtmlText (htmlQuote s)` -- plain string
- `bold hexps = HtmlStruct "B" [] hexps` -- bold font
- `italic hexps = HtmlStruct "I" [] hexps` -- italic font
- `h1 hexps = HtmlStruct "H1" [] hexps` -- main header

Example:
```
[h1 [htxt "1. Hello World"],
  italic [htxt "Hello"],
  bold [htxt "world!"]]
```

Advantage:
static checking of HTML structure
MODELING HTML

Data type for representing HTML expressions:

```haskell
data HtmlExp = HtmlText String
              | HtmlStruct String [(String,String)] [HtmlExp]
```

Some useful abbreviations:

- `htxt s = HtmlText (htmlQuote s)`  -- plain string
- `bold hexps = HtmlStruct "B" [] hexps`  -- bold font
- `italic hexps = HtmlStruct "I" [] hexps`  -- italic font
- `h1 hexps = HtmlStruct "H1" [] hexps`  -- main header

... Example: `[h1 [htxt "1. Hello World"],
               italic [htxt "Hello"],
               bold [htxt "world!"]]`

1. Hello World

*Hello world!*
Data type for representing HTML expressions:

```
data HtmlExp = HtmlText String
  | HtmlStruct String [(String,String)] [HtmlExp]
```

Some useful abbreviations:

- `htxt s = HtmlText (htmlQuote s)`  -- plain string
- `bold hexps = HtmlStruct "B" [] hexps`  -- bold font
- `italic hexps = HtmlStruct "I" [] hexps`  -- italic font
- `h1 hexps = HtmlStruct "H1" [] hexps`  -- main header

Example:

```
[h1 [htxt "1. Hello World"],
  italic [htxt "Hello"],
  bold [htxt "world!"]]
```

1. Hello World

```
Hello world!
```

Advantage: static checking of HTML structure
Dynamic Web Pages

- Web pages with dynamic contents and interaction
- Content is computed at the page request time

Data type to represent complete HTML documents:
(title, optional parameters (cookies, style sheets), contents)

data HtmlForm = HtmlForm String [FormParam] [HtmlExp]

Useful abbreviation:
form title hexps = HtmlForm title [] hexps
Dynamic Web Pages

- Web pages with dynamic contents and interaction
- Content is computed at the page request time

Data type to represent complete HTML documents:
(title, optional parameters (cookies, style sheets), contents)

data HtmlForm = HtmlForm String [FormParam] [HtmlExp]

Useful abbreviation:
form title hexps = HtmlForm title [] hexps

Type of dynamic web page: IO HtmlForm
(I/O action that computes a page depending on current environment)

helloPage = return (form "Hello" hello)
General concept: submit form with input elements $\leadsto$ answer form

Specific HTML elements for dealing with user input, e.g.:

textfield ref "initial contents" :: HtmlExp
General concept: submit form with input elements $\leadsto$ answer form

Specific HTML elements for dealing with user input, e.g.:

```
textfield ref "initial contents" :: HtmlExp
```

HTML library: **programming with call-back functions**

**Event handler:** attached to submit buttons in HTML forms

type EventHandler = (CgiRef -> String) -> IO HtmlForm

**CGI environment:** mapping from CGI references to actual input values

**CGI reference:**

- identifies input element of HTML form
- abstract data type (instead of strings as in raw CGI, Perl, PHP, . . .)
- logical variable in HTML forms
EXAMPLE: FORM TO REVERSE/DUPLICATE A STRING

form "Question" [htxt "Enter a string: ", textfield ref "", hr, button "Reverse string" revhandler, button "Duplicate string" duphandler]

where

ref free

revhandler env = return $ form "Answer" [h1 [htxt ("Reversed input: " ++ rev (env ref))]]

duphandler env = return $ form "Answer" [h1 [htxt ("Duplicated input: " ++ env ref ++ env ref)]]
EXAMPLE: RETRIEVING FILES FROM A WEB SERVER

Form to show the contents of an arbitrary file stored at the server:

getFile = return $ form "Question"
    [htxt "Enter local file name: ",
     textfield fileref "", 
     button "Get file!" handler]

where

    fileref free

    handler env = do contents <- readFile (env fileref)
     return $ form "Answer"
     [h1 [htxt ("Contents of " ++ env fileref)],
      verbatim contents]

Functional + logic features  \(\leadsto\)  simple interaction + retrieval of user input
APPLICATION: E-LEARNING

CurryWeb: a system to support web-based learning

**openness:** no distinction between instructors and students, users can learn or add new material, rank material, write critics, . . .

**self-responsible use:** users are responsible to select right material
**APPLICATION: E-LEARNING**

**CurryWeb**: a system to support web-based learning

**openness**: no distinction between instructors and students, users can learn or add new material, rank material, write critics,...

**self-responsible use**: users are responsible to select right material

**Requirements:**
- provide structure to learning material to support selection process
- management of users

**Implementation:**
- completely implemented in Curry (around 8000 lines of code)
- shows how Curry's features support high-level implementation
- declarative languages are appropriate for implementing complex web-based systems
- done by students without prior knowledge to Curry
**APPLICATION: E-LEARNING**

**CurryWeb**: a system to support web-based learning

**openness**: no distinction between instructors and students, users can learn or add new material, rank material, write critics, ...  

**self-responsible use**: users are responsible to select right material

**Requirements**:  
→ provide structure to learning material to support selection process  
→ management of users

**Implementation**:  
→ completely implemented in Curry (around 8000 lines of code)  
→ shows how Curry’s features support high-level implementation  
→ declarative languages are appropriate for implementing complex web-based systems  
→ done by students without prior knowledge to Curry
The Idea of the CurryWeb

Author: Michael Hanus

Abstract
This document describes the basic idea of the CurryWeb.

prerequisites objectives

Using a Web Browser

Content

Critics for this educational unit:

Ranking of Educational Unit "The Idea of the CurryWeb"

Your ranking: 1 submit your ranking

This educational unit has not been ranked yet.
FURTHER WEB APPLICATIONS

**PASTA**: a web-based system to submit and test exercises in a programming course

**Module Directory**: a web-based system to administrate module descriptions in our CS department

**Questionnaire**: a system for the web-based submission and evaluation of questionnaires

**Conference/Journal Submission**: a system for the web-based submission and administration of papers (used for various workshops/conferences and JFLP)
FURTHER APPLICATIONS: PROGRAMMING EMBEDDED SYSTEMS

[WFLP 2002, WFLP 2003]
APPLICATION: PROGRAMMING AUTONOMOUS ROBOTS

```plaintext

go _ _ =
  [Send (MotorDir Out_A Fwd),
   Send (MotorDir Out_C Fwd)]
|> Proc waitEvent

waitEvent (TouchLeft:_)_ =
  [Deq TouchLeft]  |> Proc (turn TouchLeft)

waitEvent (TouchRight:_)_ =
  [Deq TouchRight]  |> Proc (turn TouchRight)

turn t _ _ =
  [Send (MotorDir Out_A Rev), Send (MotorDir Out_C Rev)]  |> Proc (wait 2) >>>
  atomic
  [Send (MotorDir (if t==TouchLeft then Out_A else Out_C) Fwd)] >>>
  Proc (wait 2) >>> Proc go
```
**Curry: A True Integration of Declarative Paradigms**

**Functional programming:** lazy evaluation, deterministic evaluation of ground expressions, higher-order functions, polymorphic types, monadic I/O $\Rightarrow$ extension of Haskell

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

More info on Curry: [http://www.informatik.uni-kiel.de/~curry](http://www.informatik.uni-kiel.de/~curry)
**CURRY: A TRUE INTEGRATION OF DECLARATIVE PARADIGMS**

**Functional programming:** lazy evaluation, deterministic evaluation of ground expressions, higher-order functions, polymorphic types, monadic I/O $\implies$ 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**
**Curry: A True Integration of Declarative Paradigms**

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

Curry supports appropriate abstractions for software development
\(\sim\) functional logic design patterns [FLOPS’02]

More infos on Curry:
http://www.informatik.uni-kiel.de/~curry