Declarative Multi-Paradigm Programming in

\[ \lambda u \rightarrow \text{Curry} \]

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)

```haskell
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** \( \approx \) data term, constructor term:
well-formed expression containing variables and data type constructors

\[(S \ Z) \ 1:(2::[])) \ [1,2] \ \text{Node} \ [\text{Leaf} \ 3, \text{Node} \ [\text{Leaf} \ 4, \text{Leaf} \ 5]]\]
FUNCTIONAL CURRY PROGRAMS

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

\[ f(t_1 \ldots t_n) \mid c = r \]

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

Examples:

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

\[
\begin{align*}
[] \star ys &= ys \\
(x:xs) \star ys &= x : (xs \star ys)
\end{align*}
\]

\[
\begin{align*}
depth(Leaf \_\_) &= 1 \\
depth(Node []) &= 1 \\
depth(Node (t:ts)) &= \max (1 + depth t) (depth (Node ts))
\end{align*}
\]
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

\[ (S \ x) + y = S(x+y) \]
\[ (S \ x) \leq 0 \quad = \text{False} \]
\[ (S \ x) \leq (S \ y) = x \leq y \]

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

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

**Lazy evaluation:** select an outermost redex

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

**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...

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

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

\[
\begin{align*}
0 \leq x_2 &= \text{True} \\
(S x_{3}) \leq x_2 &= x_{3} \leq x_4
\end{align*}
\]

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

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
NON-DETERMINISTIC EVALUATION

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 \(\text{True} \lor \text{True}\)

\(\leadsto\) Problem: no needed argument: \(\text{\color{blue}{e_1 \lor e_2}}\) evaluate \(e_1\) 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**

True \( \lor x \) = True  
\( x \lor \text{True} \) = True  
False \( \lor \) False = False

Evaluation of \( e_1 \lor e_2 \) ?

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

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

Extension to definitional trees / pattern matching:
Introduce \( or\)-nodes to describe non-deterministic selection of redexes

\( \leadsto \) non-deterministic evaluation:  
\[ e \rightarrow e_1 | \cdots | e_n \]

\( \leadsto \) 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:

\[ \sim \text{ more than one result on a given input} \]

```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
```

Demand-driven search (search space reduction):

\[ \text{sorted (perm xs)} \]
NON-DETERMINISTIC / SET-VALUED FUNCTIONS

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

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

perm [1,2,3] ~ [1,2,3] | [1,3,2] | [2,1,3] | ...
```
NON-DETERMINISTIC / SET-VALUED FUNCTIONS

Rules must be constructor-based but not confluent:

\[ \leadsto \text{more than one result on a given input} \]

```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

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

perm [1,2,3] \leadsto [1,2,3] \mid [1,3,2] \mid [2,1,3] \mid ... 
```

Demand-driven search (search space reduction): sorted (perm xs)
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: bind and reduce:
\[
\begin{array}{c}
\text{logic} \\
\text{functional}
\end{array}
\]
\[
e \sim \{\sigma_1\} e_1 \mid \cdots \mid \{\sigma_n\} e_n
\]

Reduce:
\[
(f\ 0) \sim 2
\]

Bind and reduce:
\[
(f\ x) \sim \{x=0\} 2 \mid \{x=1\} 3
\]

Compute necessary bindings with needed strategy
\sim \textit{needed narrowing} \ [\text{Antoy/Echahed/Hanus POPL'94/JACM'00}]
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
(\(\approx\) 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)
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** \( e_1 =:= e_2 \)

successful if both sides evaluable to unifiable data terms

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

\[ \Rightarrow e_1 =:= e_2 \text{ and } e_1, e_2 \text{ data terms } \approx \text{ unification in logic programming} \]
List concatenation:

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

\[
[] ++ ys = ys
\]

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

Functional programming:

\[\begin{align*}
[1,2] ++ [3,4] & \leadsto [1,2,3,4] \\
\end{align*}\]

Logic programming:

\[\begin{align*}
x ++ y &=:= [1,2] \\
\{x=[],y=[1,2]\} & \lor \{x=[1],y=[2]\} & \lor \{x=[1,2],y=[]\}
\end{align*}\]
List concatenation:

\[
(++) :: [a] -> [a] -> [a]
\]

\[
[] ++ ys = ys
\]

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

Functional programming:

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

Logic programming:

\[
x ++ y =:= [1,2] \leadsto
\{x=[],y=[1,2]\} \cup \{x=[1],y=[2]\} \cup \{x=[1,2],y=[]\}
\]

Last list element:

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

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

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

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

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

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

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

**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 (**constraints**)
- data structures (constraint domain): **constructor terms**
- basic constraint: (strict) **equality**
- constraint solver: **unification**

Constraint Programming: generalizes logic programming by
- new specific **constraint domains** (e.g., reals, finite sets)
- new **basic constraints** over these domains
- sophisticated **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:

\[ 5.1 =: x + 3.5 \quad \leadsto \quad \{x=1.6\} \]
\[ x \leq 1.5 \quad \& \quad x + 1.3 \geq 2.8 \quad \leadsto \quad \{x=1.5\} \]
Define relation \( cvi \) between electrical circuit, voltage, and current

Circuits are defined by the data type

```haskell
data Circuit = Resistor Float
  | Series Circuit Circuit
  | Parallel Circuit Circuit
```

Rules for relation \( cvi \):

- \( cvi \) (Resistor \( r \)) \( v \ i = v :== i \times r \) \hspace{1cm} -- Ohm’s law
- \( cvi \) (Series \( c1 \) \( c2 \)) \( v \ i = \\
  v :== v1 + v2 \hspace{0.5cm} \& \hspace{0.5cm} cvi \ c1 \ v1 \ i \hspace{0.5cm} \& \hspace{0.5cm} cvi \ c2 \ v2 \ i \hspace{1cm} -- Kirchhoff’s law
- \( cvi \) (Parallel \( c1 \) \( c2 \)) \( v \ i = \\
  i :== i1 + i2 \hspace{0.5cm} \& \hspace{0.5cm} cvi \ c1 \ v \ i1 \hspace{0.5cm} \& \hspace{0.5cm} cvi \ c2 \ v \ i2 \hspace{1cm} -- 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:

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

puzzle s e n d m o r y =

\[
\text{domain } [s,e,n,d,m,o,r,y] 0 9 \ & \ -- \ \text{define domain}
\]
\[
s > 0 \ & \ m > 0 \ & \ -- \ \text{define constraints}
\]
\[
\text{all_different } [s,e,n,d,m,o,r,y] \ &
\]
\[
1000 * s + 100 * e + 10 * n + d
\]
\[
+ 1000 * m + 100 * o + 10 * r + e
\]
\[
= 10000 * m + 1000 * o + 100 * n + 10 * e + y \ &
\]
\[
\text{labeling } [s,e,n,d,m,o,r,y] \ & \ -- \ \text{instantiate variables}
\]

puzzle s e n d m o r y \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 $\rightarrow$ narrowing may not terminate
- all rules must be explicitly known $\rightarrow$ 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
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:
\[ f \ x =:= y \ \leadsto \ suspend \]
**FLEXIBLE VS. RIGID FUNCTIONS**

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

rigid/flexible status not relevant for ground calls:

\[
\begin{align*}
& \text{\( f\ 1 \mapsto 3 \)}
\end{align*}
\]

\( f \) flexible:

\[
\begin{align*}
& \text{\( f\ x =:= y \mapsto \{x=0, y=2\} \mid \{x=1, y=3\} \)}
\end{align*}
\]

\( f \) rigid:

\[
\begin{align*}
& \text{\( f\ x =:= y \mapsto \text{suspend} \)}
\end{align*}
\]

\[
\begin{align*}
& \text{\( f\ x =:= y \& x =:= 1 \)}
\end{align*}
\]
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:

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

\[
f x =:= y \land x =:= 1 \leadsto \{x=1\} f 1 =:= y \ (suspend \ f \ x)
\]
FLEXIBLE VS. RIGID FUNCTIONS

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

rigid/flexible status not relevant for ground calls:
\[
f\ 1 \quad \leadsto \quad 3
\]

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

\( f \) rigid:
\[
f\ x =:= y \quad \leadsto \quad \text{suspend}
\]
\[
f\ x =:= y \ & \ x =:= 1 \quad \leadsto \quad \{ x=1 \} \ f\ 1 =:= y \quad (\text{suspend } f\ x)
\]
\[
\leadsto \quad \{ x=1 \} \ 3 =:= y \quad (\text{evaluate } f\ 1)
\]
**FLEXIBLE VS. RIGID FUNCTIONS**

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

rigid/flexible status not relevant for ground calls:

\[f 1 \leadsto 3\]

\(f\) flexible:

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

\(f\) rigid:

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

\[f \ x = := \ y \land x = := 1 \leadsto \{x=1\} \ f 1 = := \ y \ (suspend \ f \ x)\]

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

\[\leadsto \{x=1,y=3\}\]

Default in Curry: flexible (except for predefined and I/O functions)
<table>
<thead>
<tr>
<th>Computation model</th>
<th>Restrictions on programs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Needed narrowing</strong></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*)

- Defined operation
- Data terms
- Constraint (optional)
- Expression

\[
\begin{align*}
& f \ t_1 \ldots t_n \mid c = r \\
& \text{conc} \ [\ldots] = \text{ys} = \text{ys} \\
& \text{conc} \ (x:xs) \ ys = x : \text{conc} \ xs \ ys \\
& \text{last} \ xs \mid \text{conc} \ ys \ [x] =:= xs \\
& \quad = x \\
\end{align*}
\]

where \( x, ys \) free
\textbf{SUMMARY: Expressions}

\[ e \ ::= \]
\[
c\] \hspace{1cm} \text{(constants)}
\[
x\] \hspace{1cm} \text{(variables $x$)}
\[
(e_0 \ e_1 \ldots e_n) \] \hspace{1cm} \text{(application)}
\[
\backslash x \rightarrow e\] \hspace{1cm} \text{(abstraction)}
\[
\text{if } b \text{ then } e_1 \text{ else } e_2 \] \hspace{1cm} \text{(conditional)}

Equational constraints over functional expressions:
\[\text{conc } y_1 \ [x] := [1,2];\]
\[\{y_1=[1], x=2\}\]

Further constraints:
real arithmetic, finite domain, ports (OOP)
SUMMARY: EXPRESSIONS

\[ e ::= \]

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

\[ e ::= \]

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

Equational constraints over functional expressions:

\[ \text{conc } ys [x] =:= [1,2] \quad \leadsto \quad \{ ys=[1], x=2 \} \]

**Further constraints:** real arithmetic, finite domain, **ports** (\( \leadsto \) OOP)
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
  - HTML programming [PADL '00]
  - XML programming [PADL '01]
  - meta-programming
  - persistent terms
  - . . .
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
  - ...
Integration of different programming paradigms is possible

Functional programming is a good starting point:
- lazy evaluation \(\leadsto\) modularity, optimal evaluation
- higher-order functions \(\leadsto\) code reuse, design patterns
- polymorphism \(\leadsto\) 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 (\(\sim\) 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**
- choose the most appropriate features for application programming

[PLILP’97]
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
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

...
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!"]]
```

\[ \rightarrow 1. \text{Hello World} \]

\[ Hello \text{ world!} \]
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!"]]
```

\[ 1. \textit{Hello World} \]
\[ \textbf{Hello world!} \]

**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)

```haskell
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
```

**Useful abbreviation:**
```haskell
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)
Web Pages with User Interaction

General concept: submit form with input elements $\mapsto$ 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.:

\[
\text{textfield ref "initial contents" :: HtmlExp}
\]

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

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

\[
\text{type EventHandler = (CgiRef -> String) -> IO HtmlForm}
\]

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

**CGI reference:**

\[
\rightarrow \text{identifies input element of HTML form}
\]

\[
\rightarrow \text{abstract data type (instead of strings as in raw CGI, Perl, PHP,...)}
\]

\[
\rightarrow \text{logical variable in HTML forms}
\]
**EXAMPLE: FORM TO REVERSE/DUPLICATE A STRING**

```haskell
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)]]
```

CAU Kiel

Michael Hanus
**Example: Retrieving Files from a Web Server**

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

```haskell
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 $\sim$ 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

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

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

Curry supports appropriate abstractions for software development; functional logic design patterns [FLOPS’02]

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 $\Rightarrow$ 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: 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 supports appropriate abstractions for software development

$\rightsquigarrow$ functional logic design patterns [FLOPS’02]

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