Linköping, 11.2003

Declarative Multi-Paradigm Programming in

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)

DECLARATIVE MULTI-PARADIGM LANGUAGES

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

CURRY

[Dagstuhl'96, POPL'97]

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

- 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

Values in imperative languages: basic types + pointer structures

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

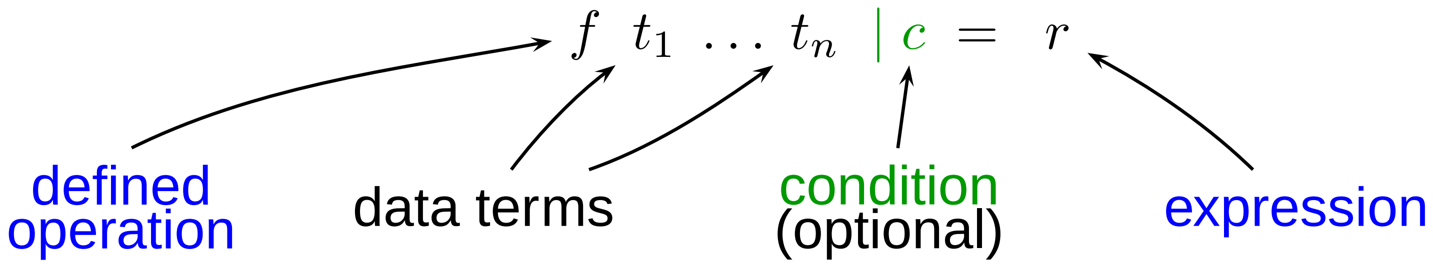
```
data Bool   = True   | False
data Nat    = Z      | S Nat
data List a = []     | a : List a      -- [a]
data Tree a = Leaf a | Node [Tree a]

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)



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

defined operation

data terms

condition (optional)

expression

```
    0 + y = y                  0 ≤ y     = True
(S x) + y = S(x+y)         (S x) ≤ 0     = False
                           (S x) ≤ (S y) = x≤y

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

depth (Leaf _)      = 1
depth (Node [])     = 1
depth (Node (t:ts)) = max (1+depth t) (depth (Node ts))
```

EVALUATION: COMPUTING VALUES

Reduce expressions to their values

Replace equals by equals

Apply reduction step to a subterm (redex, *red*ucible *ex*pression):

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

$\leadsto$ match lhs against subterm (instantiate these variables)

```
    0 + y = y                    0 ≤ y      = True
(S x) + y = S(x+y)           (S x) ≤ 0      = False
                             (S x) ≤ (S y)  = x≤y
```

`(S 0)+(S 0)` → `S (0+(S 0))` → `S (S 0)`

EVALUATION STRATEGIES

Expressions with several redexes: which evaluate first?

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

Lazy evaluation: select an outermost redex

```
    0 + y = y                   0 ≤ y       = True
(S x) + y = S(x+y)          (S x) ≤ 0       = False
                            (S x) ≤ (S y)   = x≤y
```

Strict evaluation:

$\texttt{0} \leq \texttt{(S 0)+(S 0)} \rightarrow \texttt{0} \leq \texttt{(S (0+(S 0))}) \rightarrow \texttt{0} \leq \texttt{(S (S 0))} \rightarrow \texttt{True}$

Lazy evaluation:

$\texttt{0} \leq \texttt{(S 0)+(S 0)} \rightarrow \texttt{True}$

Strict evaluation might need more steps, but it can be even worse. . .

```
    0 + y = y                     0 ≤ y       = True
(S x) + y = S(x+y)            (S x) ≤ 0       = False
                              (S x) ≤ (S y)   = x≤y

        f = f
```

Lazy evaluation:

`0+0 ≤ f   →   0 ≤ f   →   True`

Strict evaluation:

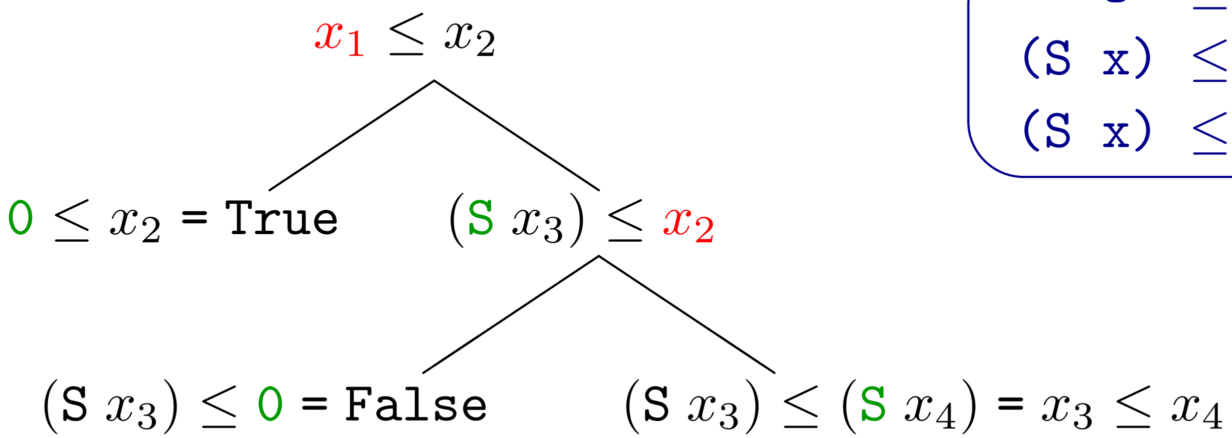
`0+0 ≤ f   →   0+0 ≤ f   →   0+0 ≤ f   →   ···`

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



$x_1 \leq x_2$

$\mathtt{0} \leq x_2$ = `True`

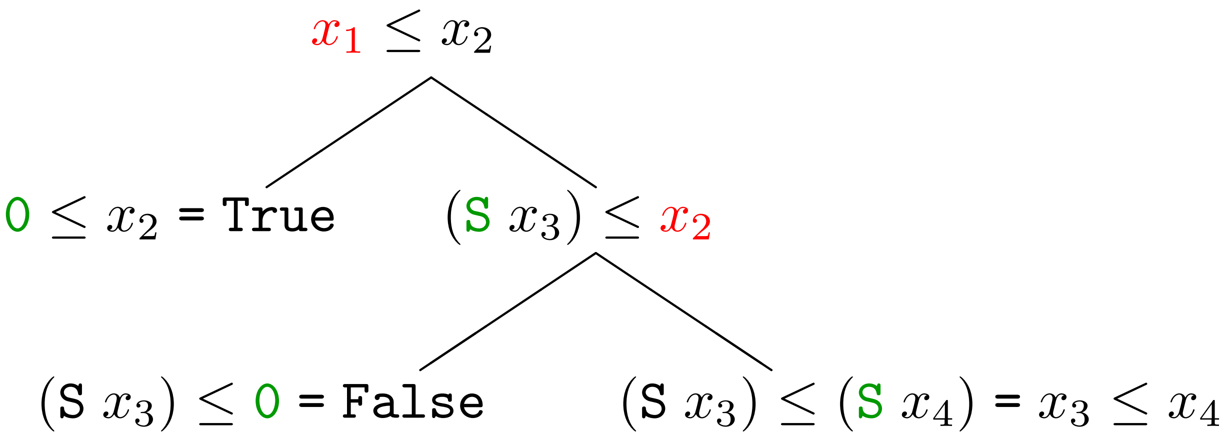
$(\mathtt{S}\ x_3) \leq x_2$

$(\mathtt{S}\ x_3) \leq \mathtt{0}$ = `False`

$(\mathtt{S}\ x_3) \leq (\mathtt{S}\ x_4)$ = $x_3 \leq x_4$

```
    0  ≤  y      =  True
(S x)  ≤  0      =  False
(S x)  ≤  (S y)  =  x≤y
```

EVALUATION WITH DEFINITIONAL TREES



$x_1 \leq x_2$

$\texttt{0} \leq x_2$ `= True`

$(\texttt{S}\ x_3) \leq x_2$

$(\texttt{S}\ x_3) \leq \texttt{0}$ `= False`

$(\texttt{S}\ x_3) \leq (\texttt{S}\ x_4)$ `=` $x_3 \leq x_4$

Evaluating function call $t_1 \leq t_2$:

① Reduce t_1 to head normal form (constructor-rooted expression)

② If $t_1 = \texttt{0}$: apply rule

③ If $t_1 = (\texttt{S} \ldots)$: 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
  $\longrightarrow$ 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:

```
True ∨ x     = True
   x ∨ True  = True
False ∨ False = False
```

First two rules overlap on `True ∨ True`

⇝ Problem: no needed argument: $\boxed{e_1 \lor e_2}$ 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 ∨ x     = True
    x ∨ True  = True
False ∨ False = False
```

Evaluation of $\boxed{e_1 \vee e_2}$?

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: $e \quad \rightarrow \quad \underbrace{e_1 \mid \cdots \mid e_n}_{\text{disjunctive expression}}$

$\leadsto$ 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
```

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
```

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] | ...

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

```
f 0 = 2
f 1 = 3
```

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: $\underbrace{\textbf{bind}}_{\textit{logic}}$ and $\underbrace{\textbf{reduce}}_{\textit{functional}}$: $e \rightsquigarrow \underbrace{\{\sigma_1\}\ e_1 \mid \cdots \mid \{\sigma_n\}\ e_n}_{\text{disjunctive expression}}$

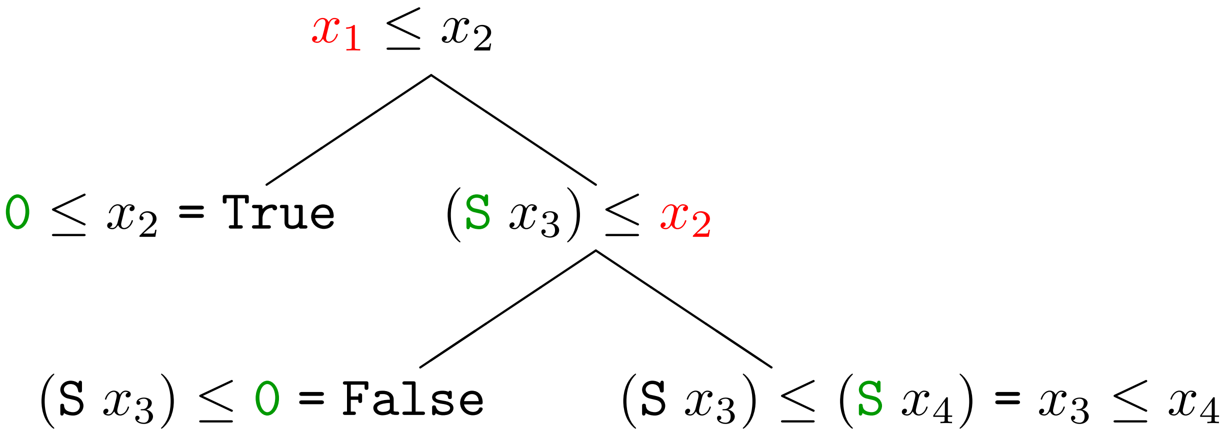
Reduce: `(f 0)` $\rightsquigarrow$ `2`

Bind and reduce: `(f x)` $\rightsquigarrow$ $\{$`x=0`$\}$ `2 |` $\{$`x=1`$\}$ `3`

Compute necessary bindings with needed strategy

$\rightsquigarrow$ **needed narrowing** [Antoy/Echahed/Hanus POPL’94/JACM’00]

NEEDED NARROWING



$x_1 \leq x_2$

$\texttt{0} \leq x_2$ `= True`

$(\texttt{S}\ x_3) \leq x_2$

$(\texttt{S}\ x_3) \leq \texttt{0}$ `= False`

$(\texttt{S}\ x_3) \leq (\texttt{S}\ x_4)$ `=` $x_3 \leq x_4$

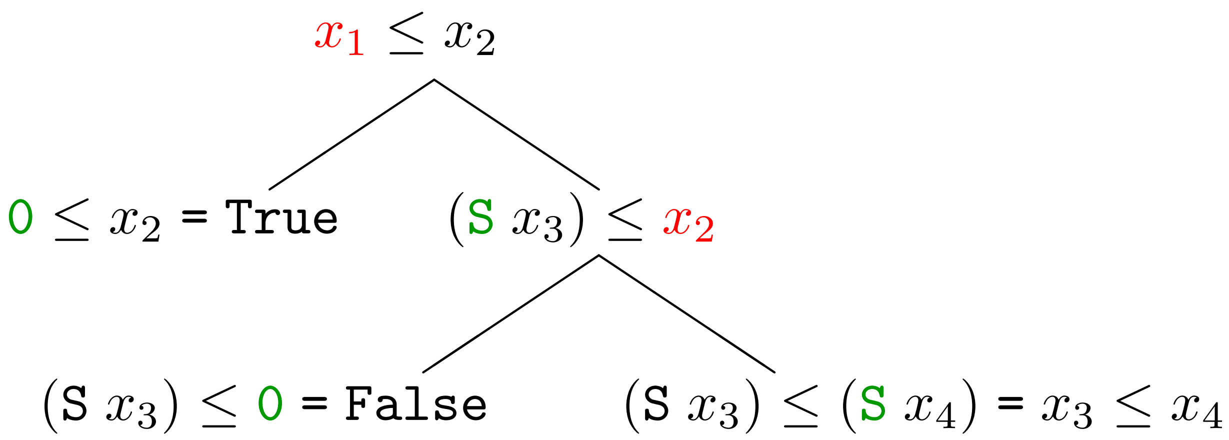
Evaluating function call $t_1 \leq t_2$:

① Reduce t_1 to head normal form

② If $t_1 = \texttt{0}$: apply rule

③ If $t_1 = (\texttt{S} \ldots)$: reduce t_2 to head normal form

NEEDED NARROWING



$x_1 \leq x_2$

$\texttt{0} \leq x_2$ = `True`

$(\texttt{S}\ x_3) \leq x_2$

$(\texttt{S}\ x_3) \leq \texttt{0}$ = `False`

$(\texttt{S}\ x_3) \leq (\texttt{S}\ x_4)$ = $x_3 \leq x_4$

Evaluating function call $t_1 \leq t_2$:

① Reduce t_1 to head normal form

② If $t_1 = \texttt{0}$: apply rule

③ If $t_1 = (\texttt{S} \ldots)$: reduce t_2 to head normal form

④ If t_1 variable: bind t_1 to `0` or $(\texttt{S}\ x)$

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 σ and σ' 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]

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 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 $\boxed{e_1 \texttt{=:=} e_2}$

successful if both sides evaluable to unifiable data terms

$\Rightarrow$ $e_1 \texttt{=:=} e_2$ does not hold if e_1 or e_2 undefined or infinite

$\Rightarrow$ $e_1 \texttt{=:=} e_2$ and e_1, e_2 data terms $\approx$ unification in logic programming

FUNCTIONAL LOGIC PROGRAMMING: EXAMPLES

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]`$\}$ `|` $\{$`x=[1],y=[2]`$\}$ `|` $\{$`x=[1,2],y=[]`$\}$

FUNCTIONAL LOGIC PROGRAMMING: EXAMPLES

List concatenation:

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

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

Functional programming:

`[1,2] ++ [3,4]` ⇝ `[1,2,3,4]`

Logic programming:

`x ++ y =:= [1,2]` ⇝

`{x=[],y=[1,2]} | {x=[1],y=[2]} | {x=[1,2],y=[]}`

Last list element:

```
last xs | ys ++ [x] =:= xs  = x
```

PROGRAMMING DEMAND-DRIVEN SEARCH

Non-deterministic functions for generating permutations:

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

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

PROGRAMMING DEMAND-DRIVEN SEARCH

Non-deterministic functions for generating permutations:

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

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

Sorting lists with test-of-generate principle:

```
sorted []  = []
sorted [x] = [x]
sorted (x:y:ys) | x<=y  = x : sorted (y:ys)

psort xs = sorted (perm xs)
```

Advantages of non-deterministic functions as generators:

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

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

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

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

Permutation sort for $\texttt{[}n\texttt{,}n{-}1\texttt{,}\ldots\texttt{,2,1]}$: #or-branches/disjunctions

Length of the list:	4	5	6	8	10
generate-and-test	24	120	720	40320	3628800
test-of-generate	19	59	180	1637	14758

CONSTRAINT PROGRAMMING

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` $\leadsto$ `{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 Circuit = Resistor Float
             | Series   Circuit Circuit
             | Parallel Circuit Circuit
             ⋮
```

Rules for relation `cvi`:

```
cvi (Resistor r) v i =  v =:= i * r          -- Ohm's law

cvi (Series   c1 c2) v i =                   -- Kirchhoff's law
    v=:=v1+v2  &  cvi c1 v1 i  &  cvi c2 v2 i

cvi (Parallel c1 c2) v i =                   -- Kirchhoff's law
    i=:=i1+i2  &  cvi c1 v i1  &  cvi c2 v i2
```

Querying the circuit specification:

Current in a sequence of resistors:

```
cvi (Series (Resistor 180.0) (Resistor 470.0)) 5.0 i
⇝ {i = 0.007692307692307693}
```

Relation between resistance and voltage in a circuit:

```
cvi (Series (Series (Resistor r) (Resistor r)) (Resistor r)) v 5.0
⇝ {v=15.0*r}
```

Also synthesis of circuits possible

CONSTRAINT PROGRAMMING WITH FINITE DOMAINS

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:

$$\begin{array}{ccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

```
puzzle s e n d m o r y =
  domain [s,e,n,d,m,o,r,y] 0 9  &       -- define domain
  s > 0  &  m > 0  &                    -- define constraints
  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  &
  labeling [s,e,n,d,m,o,r,y]            -- instantiate variables
```

`puzzle s e n d m o r y` $\leadsto$ `{s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2}`

FROM FUNCTIONAL LOGIC TO CONCURRENT PROGRAMMING

Disadvantage of narrowing:

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

FROM FUNCTIONAL LOGIC TO CONCURRENT PROGRAMMING

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,. . .)

FROM FUNCTIONAL LOGIC TO CONCURRENT PROGRAMMING

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

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

`f 1` $\leadsto$ `3`

`f` flexible:

`f x =:= y` $\leadsto$ `{x=0,y=2} | {x=1,y=3}`

`f` rigid:

`f x =:= y` $\leadsto$ *suspend*

FLEXIBLE VS. RIGID FUNCTIONS

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

`f 1` $\leadsto$ `3`

`f` flexible:

`f x =:= y` $\leadsto$ `{x=0,y=2} | {x=1,y=3}`

`f` rigid:

`f x =:= y` $\leadsto$ *suspend*

`f x =:= y & x=:=1`

FLEXIBLE VS. RIGID FUNCTIONS

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

`f 1` $\leadsto$ `3`

`f` flexible:

`f x =:= y` $\leadsto$ $\{$`x=0,y=2`$\}$ `|` $\{$`x=1,y=3`$\}$

`f` rigid:

`f x =:= y` $\leadsto$ *suspend*

`f x =:= y & x=:=1` $\leadsto$ $\{$`x=1`$\}$ `f 1 =:= y` (suspend `f x`)

FLEXIBLE VS. RIGID FUNCTIONS

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

`f 1` $\leadsto$ `3`

`f` flexible:

`f x =:= y` $\leadsto$ {`x=0,y=2`} `|` {`x=1,y=3`}

`f` rigid:

`f x =:= y` $\leadsto$ *suspend*

`f x =:= y & x=:=1` $\leadsto$ {`x=1`} `f 1 =:= y` (suspend `f x`)

$\leadsto$ {`x=1`} `3 =:= y` (evaluate `f 1`)

FLEXIBLE VS. RIGID FUNCTIONS

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

`f 1` ⇝ `3`

`f` flexible:

`f x =:= y` ⇝ `{x=0,y=2} | {x=1,y=3}`

`f` rigid:

`f x =:= y` ⇝ *suspend*

`f x =:= y & x=:=1` ⇝ `{x=1} f 1 =:= y` (suspend `f x`)
⇝ `{x=1} 3 =:= y` (evaluate `f 1`)
⇝ `{x=1,y=3}`

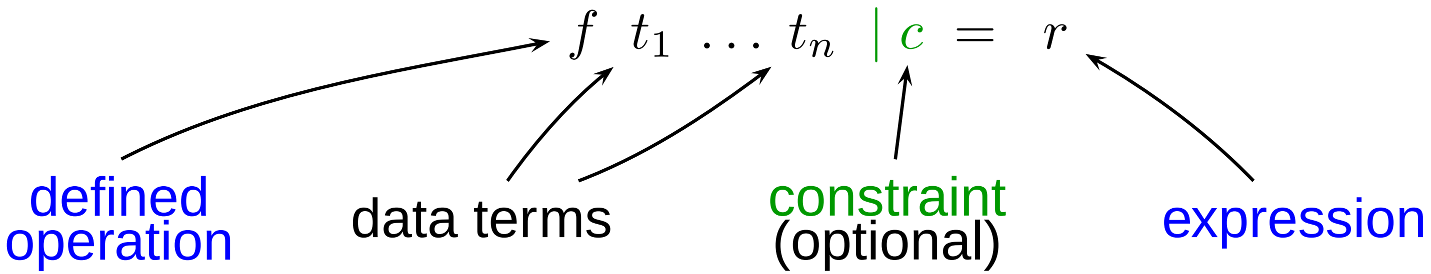
Default in Curry: flexible (except for predefined and I/O functions)

UNIFICATION OF DECLARATIVE COMPUTATION MODELS

Computation model	Restrictions on programs
Needed narrowing	inductively sequential rules; optimal strategy
Weakly needed narrowing (∼Babel)	only flexible functions
Resolution (∼Prolog)	only (flexible) predicates (∼ constraints)
Lazy functional languages (∼Haskell)	no free variables in expressions
Parallel functional langs. (∼Goffin, Eden)	only rigid functions, concurrent conjunction
Residuation (∼Life, Oz)	constraints are flexible; all others are rigid

SUMMARY: CURRY PROGRAMS

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



$$f \;\; t_1 \;\ldots\; t_n \;\; | \; c \; = \; r$$

```
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs
        = x                        where x,ys free
```

SUMMARY: EXPRESSIONS

$e\ ::=$

c	(constants)
x	(variables x)
$(e_0\ \ e_1 \ldots e_n)$	(application)
$\backslash x\ \texttt{->}\ e$	(abstraction)
$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	(conditional)

SUMMARY: EXPRESSIONS

e ::=

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

SUMMARY: EXPRESSIONS

e ::=

c	(constants)
x	(variables x)
$(e_0\ e_1 \ldots e_n)$	(application)
$\backslash x$ `->` 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)

Equational constraints over functional expressions:

`conc ys [x] =:= [1,2]` ⇝ $\{$`ys=[1],x=2`$\}$

Further constraints: real arithmetic, finite domain, **ports** (⇝ 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]

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]

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
  - ...

CURRY: A MULTI-PARADIGM PROGRAMMING LANGUAGE

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**
  [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

HTML/CGI PROGRAMMING WITH CURRY [PADL'01]

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

MODELING HTML

Data type for representing HTML expressions:

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

MODELING HTML

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
...
```

MODELING HTML

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!**

MODELING HTML

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)
```

WEB PAGES WITH USER INTERACTION

General concept: submit form with input elements ⇝ answer form

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

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

WEB PAGES WITH USER INTERACTION

General concept: submit form with input elements ⇝ 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

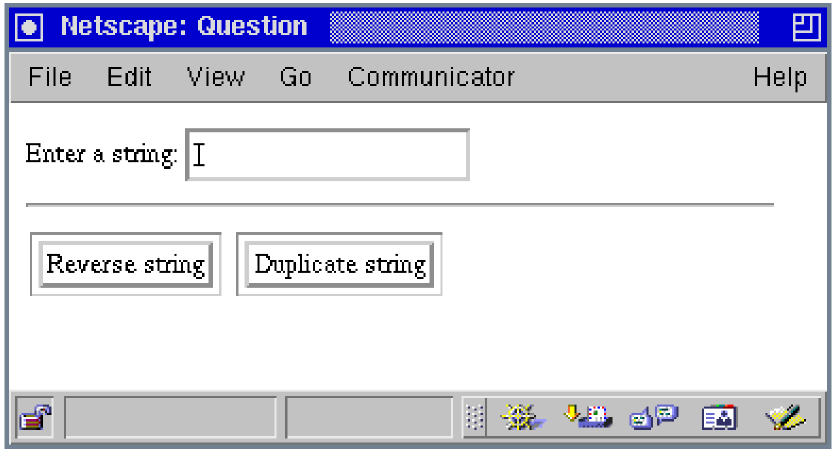
```
type EventHandler = (CgiRef -> String) -> IO HtmlForm
```

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

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

CURRYWEB: MAIN INTERFACE

http://www-ps.informatik.uni-kiel.de/~mh/DocCurryWeb/

Author: Michael Hanus

The Idea of the CurryWeb

last modified: *02/11/2003*

Abstract

This document describes the basic idea of the CurryWeb.

prerequisites: 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 rankend yet.

FURTHER WEB APPLICATIONS

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

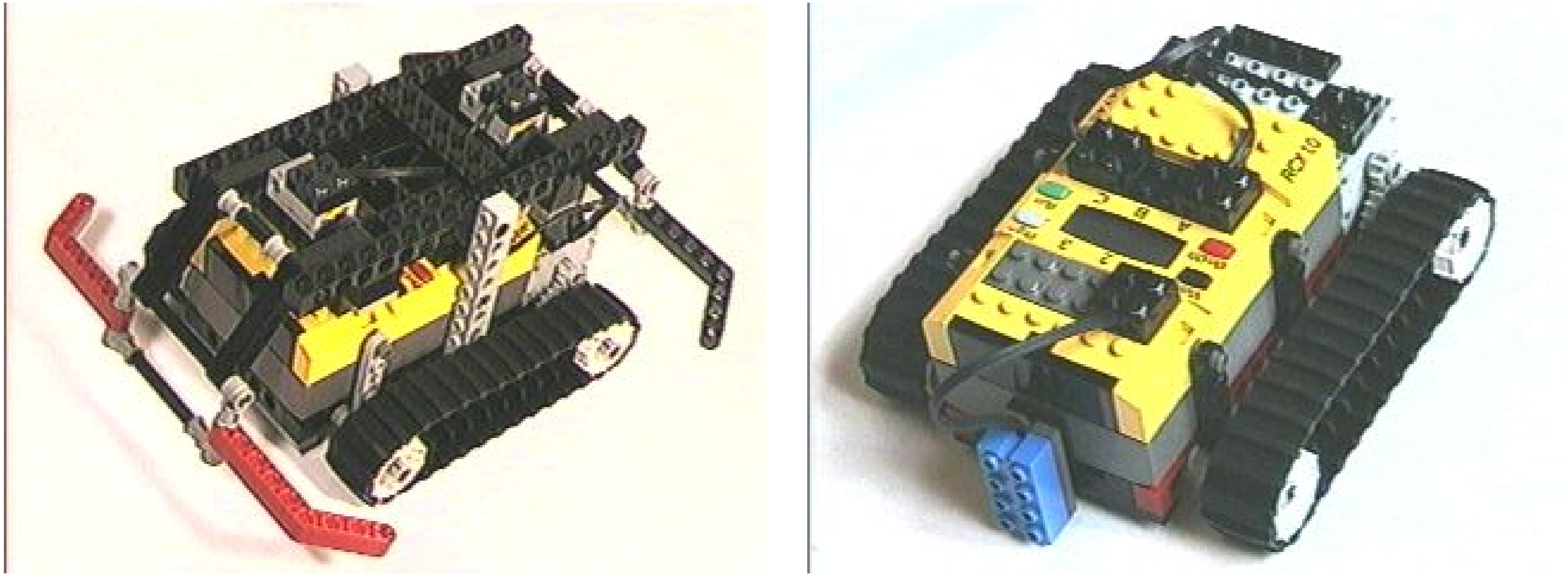
Module Directory: a web-based system to administrate module descriptions in our CS department

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

```
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 $\Longrightarrow$ extension of Haskell

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

CURRY: A TRUE INTEGRATION OF DECLARATIVE PARADIGMS

Functional programming: lazy evaluation, deterministic evaluation of ground expressions, higher-order functions, polymorphic types, monadic I/O $\Longrightarrow$ 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 ⟹ 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
⇝ functional logic design patterns [FLOPS’02]

More infos on Curry:

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