Multi-Paradigm

Declarative Programming

in Curry

Michael Hanus

RWTH Aachen
Declarative Programming

Common idea:

- description of logical relationships
- powerful abstractions, higher programming level
- reliable and maintainable programs
  - pointer structures \( \Rightarrow \) algebraic data types
  - complex procedures \( \Rightarrow \) comprehensible parts
    (pattern matching, local definitions)

Different paradigms:

- **Functional programming:**
  functions, equations, \( \lambda \)-calculus
  (lazy) deterministic reduction

- **Logic programming:**
  predicates, logical formulas, predicate logic
  constraint solving, search

\( \Rightarrow \) **Functional logic languages:**

- efficient deterministic reduction (if possible)
- flexibility of logic languages
- avoid non-declarative features of Prolog
  (arithmetic, I/O, cut)
- combine best of both worlds in a single model
Curry: A Truly Integrated Functional Logic Language

[Dagstuhl’96, POPL’97]

- multi-paradigm language, combines
  - functional programming
  - logic programming
  - concurrent programming

- based on an **optimal evaluation strategy**

- conservative extension of lazy functional and (concurrent) logic programming

- conditional (constrained) rules

- higher-order, **non-deterministic functions**

- equational constraints

- **encapsulated search**, committed choice

- polymorphic type system, modules

- declarative (monadic) I/O

- external functions and constraint solvers
Curry Programs

Values: data terms containing constructors and variables (≈ Herbrand terms): (S x) [0, (S 0)]

| data Bool      | = True | False |
| data Nat       | = 0    | S Nat |
| data List a    | = []   | a : List a |

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

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

- \(0 + y = y\)
- \((S x) + y = S(x+y)\)
- \(0 \leq y = True\)
- \((S x) \leq 0 = False\)
- \((S x) \leq (S y) = x \leq y\)

- append \([]\) ys = ys
- append \((x:xs)\) ys = x : append xs ys

\[\text{sub } m n \mid n + d := m = d \text{ where } d \text{ free}\]
Evaluation: Computing Values

- reduce expressions to their values
- replace equals by equals
- apply reduction step to a subterm (redex) (rule’s left-hand side must match the subterm)

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

\[(S \ 0)+(S \ 0) \rightarrow S \ (O+(S \ 0)) \rightarrow S \ (S \ 0)\]

Lazy strategy: select an outermost redex

\[
\begin{align*}
0+0 &\leq (S \ 0)+(S \ 0) \\
\rightarrow \ 0 &\leq (S \ 0)+(S \ 0) \\
\rightarrow \ \text{True}
\end{align*}
\]

\(\rightarrow\) evaluate only needed redexes (efficiently computable with definitional trees)

\(\rightarrow\) functional programming
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

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

\[
\begin{tikzpicture}
  \node (x1) {$x_1 \leq x_2$};
  \node (x2) [below left of=x1] {$0 \leq x_2$};
  \node (x3) [below right of=x1] {$(S \ x_3) \leq x_2$};
  \node (true) [below of=x2] {True};
  \node (false) [below of=x3] {False};
  \node (x4) [below of=true, anchor=south] {$x_3 \leq x_4$};
  \node (zero) [below of=false, anchor=south] {$(S \ x_3) \leq 0$};
  \node (szero) [below of=false, anchor=south] {$(S \ x_3) \leq (S \ x_4)$};
  \draw (x1) -- (x2); \draw (x1) -- (x3);
  \draw (x2) -- (true); \draw (x3) -- (false);
  \draw (true) -- (zero); \draw (false) -- (szero);
\end{tikzpicture}
\]

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: not reducible or bind $t_1$ to 0 or (S x)
Overlapping Rules: Non-deterministic Rewriting

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

Problem: no needed argument:

\( e_1 \lor e_2 \) evaluate \( e_1 \) or \( e_2 \)?

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

Disadvantage: not normalizing (\( e_1 \) may not terminate)

Solutions:

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:
Introduce \textit{or}-nodes to describe non-deterministic selection of redexes
From Functional Programming to Logic Programming

*Functional programming:* values, no free variables

*Logic programming:* computed answers for free variables

Operational extension:

instantiate free variables, if necessary

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

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**

**logic** and **functional**

\[
e \leadsto \{\sigma_1\} e_1 \mid \cdots \mid \{\sigma_n\} e_n
\]

disjunctive expression

Reduce:

\[
(f 0) \leadsto 2
\]

Bind and reduce:

\[
(f x) \leadsto \{x=0\} 2 \mid \{x=1\} 3
\]

Compute necessary bindings with *needed* strategy

\(\leadsto needed\ narrowing\; \text{[Antoy/Echahed/Hanus POPL’94]}\)
Properties of Needed Narrowing

[Antoy/Echahed/Hanus POPL’94]

- **Sound** and **complete** (w.r.t. strict equality)

- **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. **Independence of solutions:**
     Two solutions $\sigma$ and $\sigma'$ computed by two distinct derivations are independent.

- **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]
  - overlapping lhs [Antoy/Echahed/Hanus ICLP’97]
  - multiple rhs [Antoy ALP’97]
  - concurrent evaluation [Hanus POPL’97]
Problems with equality in the presence of non-terminating rules:

1. Equality on infinite objects undecidable:

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

Is \( f = g \) valid?

2. Semantics of non-terminating functions:

\[
\begin{align*}
f \ x &= f \ (x+1) & g \ x &= g \ (x+1) \\
\end{align*}
\]

Is \( f \ 0 = g \ 0 \) valid?

Avoided by strict equality: identity on finite objects (both sides reducible to same ground data term)

Equational constraint \( e_1 =:= e_2 \):

satisfied if both sides evaluable to unifiable data terms

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

\[ \Rightarrow e_1 =:= e_2 \text{ and } e_1, e_2 \text{ data terms } \approx \text{ unification in LP} \]
Non-deterministic Functions

Functions can have more than one result value:

choose x y = x
choose x y = y

choose 1 2 $\leadsto$ 1 | 2

Non-deterministic list insertion and permutations:

insert x [] = [x]
insert x (y:ys) = choose (x:y:ys)
              (y:insert x ys)

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

permute [1,2,3] $\leadsto$
    [1,2,3] | [2,1,3] | [2,3,1] |
    [1,3,2] | [3,1,2] | [3,2,1]
Programming Demand-driven Search

Prolog: generate-and-test:

\[ \text{psort}(Xs,Ys) :- \text{permute}(Xs,Ys), \text{ordered}(Ys). \]

Functional programming: list comprehensions:

\[ \text{psort } xs = [ys | ys<-\text{perms } xs, \text{sorted } ys] \]

Prolog with coroutining: test-and-generate

\[ \text{psort}(Xs,Ys) :- \text{ordered}(Ys), \text{permute}(Xs,Ys). \]

(Problem: floundering, heuristics)

Functional logic programming: test-of-generate:

\[
\begin{align*}
\text{sorted } [] & = [] \\
\text{sorted } [x] & = [x] \\
\text{sorted } (x:y:ys) & | x<=y = x : \text{sorted } (y:ys) \\
\text{psort } xs & = \text{sorted } (\text{permute } xs)
\end{align*}
\]

Advantages:

- demand-driven generation of solutions (due to laziness)
- same efficiency as coroutining
- no floundering
- modular program structure
Example: Demand-driven Search

\[
\begin{align*}
\text{sorted} \; [\;] &= [\;] \\
\text{sorted} \; [x] &= [x] \\
\text{sorted} \; (x:y:ys) \mid x \leq y &= x : \text{sorted} \; (y:ys) \\
\text{psort} \; xs &= \text{sorted} \; (\text{permute} \; xs)
\end{align*}
\]

\[
\begin{align*}
\text{psort} \; [5,4,3,2,1] \\
\leadsto \text{sorted} \; (\text{permute} \; [5,4,3,2,1]) \\
\leadsto^* \text{sorted} \; (5:4:\text{permute} \; [3,2,1]) &\mid \ldots \\
\text{undefined: discard this alternative} \\
\leadsto \ldots
\end{align*}
\]

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

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

<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>
Encapsulated Search

[Hanus/Steiner PLILP’98]

Technique to avoid global search (backtracking)
(non-backtrackable I/O, efficiency control, . . . )

Idea:
Compute until a non-deterministic step occurs,
then give programmer control over this situation
(generalization of Oz’s operator [Schulte/Smolka 94])

Search:
• solve constraint containing search variable
• evaluate until failure, success, or non-determinism
• return result in a list
• bind search variable to different solutions
  ⇒ abstract search variable: \( x \to c \) \( (\approx \lambda x.c) \)

Primitive search operator:

\[
\text{try} :: (a \to \text{Constraint}) \to [a \to \text{Constraint}]
\]

\[
\begin{align*}
\text{try } \backslash x \to 1\text{:=2} & \leadsto [] & \text{failure} \\
\text{try } \backslash x \to [x]\text{:=}[0] & \leadsto [\backslash x \to x\text{:=0}] & \text{success} \\
\text{try } \backslash x \to f\ x\text{:=}3 & \leadsto [\backslash x \to x\text{:=0} \& f\ 0\text{:=}3, \\
& & \backslash x \to x\text{:=1} \& f\ 1\text{:=}3] & \text{disjunction}
\end{align*}
\]
Encapsulated Search:
Search Strategies

try \(x\rightarrow c\): eval. \(c\), stop after non-deterministic step

**Depth-first search:** collect all solutions

```haskell
all :: (a -> Constraint) -> [a -> Constraint]
all g = collect (try g)
where
  collect [] = []
  collect [g] = [g]
  collect (g1:g2:gs) =
    concat (map all (g1:g2:gs))
```

all \(l\) -> append \(l\) \([1]\) =:= \([0,1]\)

\(\leadsto\) \([l\rightarrow 1=:=[0]]\)

Further search strategies:

- compute only first solution:
  
  once \(g = \text{head} \ (\text{all} \ g)\)

- \texttt{findall}, best solution search, parallel search, ...

- negation as failure:
  
  \(\text{naf} \ c = (\text{all} \ \_\rightarrow c) =:= []\)

  \(\leadsto\) control failures
Extract value of the search variable by application:

\( (\forall x : x =:= 1) \) freevar

\( \Rightarrow \) freevar = := 1

\( \Rightarrow \{\text{freevar}=1\} \) success

**Prolog’s findall:**

\[
\text{unpack} :: [a \rightarrow \text{Constraint}] \rightarrow [a]
\]

\[
\text{unpack} \ [\] = []
\]

\[
\text{unpack} \ (g:gs) \mid g \ v = v : \text{unpack} \ gs
\]

\[
\quad \text{where } v \text{ free}
\]

\[
\text{findall} \ g = \text{unpack} \ (\text{all} \ g)
\]

\[
\text{findall} \ (\forall (x,y) : \text{append} \ x \ y =:= [1,2])
\]

\[
\Rightarrow [([],[1,2]),([1],[2]),([1,2],[])]
\]
Exploiting laziness

Demand-driven encapsulated search easily obtained by laziness:

```prolog
prolog g = printloop (all g)
printloop [] = putStr("no") >> nl
printloop (a:as) = browse a >>= putStr "? " >>
                  getChar >>= evalAnswer as
evalAnswer as ';'; = nl >> printloop as
evalAnswer as '\n' = nl >> putStr "yes" >> nl
```

```
prolog \(x,y\) \rightarrow\ append x y =:= [1,2]
\Rightarrow (\[],[1,2]) ? ;
    ([1],[2]) ? <- yes
prolog \x \rightarrow\ 1=:=2 \Rightarrow no
```

\(\Rightarrow\ Separation\ of\ Logic\ and\ Control\)

\(\Rightarrow\ Modularity:\)

- Prolog with breadth-first search:
  prolog_bfs g = printloop (bfs g)

- Prolog with depth-bounded search:
  prolog_bound g b = printloop (bound g b)
Disadvantage of narrowing:

- functions on recursive data structures
  \( \leadsto \) narrowing may not terminate

- all rules must be explicitly known
  \( \leadsto \) combination with external functions unclear
  (basic arithmetic,...)

Solution:
Delay function calls if a particular argument is free

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

\[
x + x =: y \ & \ x = := 2
\]
\[
\leadsto \ \{ x = 2 \} \ 2 + 2 = := y \quad (\text{suspend } x + x)
\]
\[
\leadsto \ \{ x = 2 \} \ 4 = := y \quad (\text{evaluate } 2 + 2)
\]
\[
\leadsto \ \{ x = 2, y = 4 \}
\]
Parallel Functional Programming

[Goffin, Eden]

Parallel evaluation of arguments:

\[ f \ t_1 \ t_2 = \text{letpar} \ x = g \ t_1 \]
\[ y = h \ t_2 \text{ in } k \ x \ y \]

with concurrent conjunction of equations:

\[ f \ t_1 \ t_2 \mid x =:= g \ t_1 \& y = h \ t_2 = k \ x \ y \]
where \( x, y \) free

Skeleton-based parallel programming:

Applying a function to all list elements (sequentially):

\[ \text{map } f \ [\] = [] \]
\[ \text{map } f \ (x:xs) = f \ x : \text{map } f \ xs \]

\( \text{farm} \): parallel version of \( \text{map} \)

\[ \text{farm } f \ [\] = [] \]
\[ \text{farm } f \ (x:xs) \mid r =:= f \ x \& rs =:= \text{farm } f \ xs \]
\[ = r : rs \text{ where } r, rs \text{ free} \]
Modelling objects with state as a constraint function:

- first parameter: stream of messages (wait for input)
- second parameter: current state

Example: **Bank account**

```haskell
data Messages = Deposit Int | Withdraw Int
               | Balance Int

account eval rigid  -- declare a rigid func.
account [] _        = success
account (Deposit a : ms) n = account ms (n+a)
account (Withdraw a : ms) n = account ms (n-a)
account (Balance b : ms) n = b:=:=n & account ms n

make_account s = account s 0

make_account s,  -- create account object
                s = [Deposit 200, Withdraw 50, Balance b]
                \rightarrow \{b=150, s=...\}
```
Soundness and Completeness

Relate derivations to standard rewriting $\rightarrow_{\mathcal{R}}$ ($\rightarrow_{\mathcal{R}}$ sound and complete w.r.t. model-theoretic semantics)

**Soundness:** If

$$e \rightsquigarrow^* \{\sigma_1\} e_1 | \ldots | \{\sigma_n\} e_n$$

then $\sigma_i(e) \rightarrow_{\mathcal{R}}^* e_i$ for $i = 1, \ldots, n$

**Completeness:** If $\sigma(e) \rightarrow_{\mathcal{R}}^* c$ and

$$e \rightsquigarrow^* \{\sigma_1\} e_1 | \ldots | \{\sigma_n\} e_n$$

then $\exists \varphi, i$ with $\sigma = \varphi \circ \sigma_i$ and $\varphi(e_i) \rightarrow_{\mathcal{R}}^* c$

**Completeness w.r.t. flexible functions:**

All functions are *flexible*: If $\sigma(e) \rightarrow_{\mathcal{R}}^* c$, then

$$\exists e \rightsquigarrow^* \{\sigma_1\} e_1 | \ldots | \{\sigma_n\} e_n$$

with $e_i = c$ and $\sigma = \varphi \circ \sigma_i$ for some $i$ and $\varphi$
## Curry: Unification of Computation Models

<table>
<thead>
<tr>
<th>Computation model</th>
<th>Restrictions on programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Needed narrowing [POPL’94]</td>
<td>inductively sequential rules; optimal w.r.t. length of derivations and number of computed solutions</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 languages (~Goffin, Eden)</td>
<td>only rigid functions, concurrent conjunction</td>
</tr>
<tr>
<td>Residuation (~Life, Oz)</td>
<td>constraints are flexible; all other functions are rigid (default in Curry)</td>
</tr>
</tbody>
</table>
Append:

\[
\begin{align*}
\text{append} & : [a] \to [a] \to [a] \\
\text{append eval flex} & \quad \text{-- append is flexible} \\
\text{append} &[\ ] & ys = ys \\
\text{append} & (x:xs) & ys = x : \text{append} \ xs \ ys
\end{align*}
\]

Functional programming:

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

Logic programming (append is flexible):

\[
\text{append} \ x \ y \ :=: \ [1,2] \ \sim \ \\
\{x=[] , y=[1,2]\} \mid \{x=[1], y=[2]\} \mid \{x=[1,2], y=[]\}
\]

Lazy functional programming:

\[
\begin{align*}
\text{from} \ n & = n : \text{from} \ (S \ n) \\
\text{first} \ 0 \ xs & = [\ ] \\
\text{first} \ (S \ n) \ (x:xs) & = x : \text{first} \ n \ xs
\end{align*}
\]

Lazy functional logic programming:

\[
\text{first} \ x \ (\text{from} \ y) \ :=: \ [0] \ \sim \ \{x=(S \ 0), y=0\}
\]
**Functions vs. Predicates**

*rigid* functions not always reasonable:

\[
\begin{align*}
\text{append } [] & \quad ys = ys \\
\text{append } (x:xs) & \quad ys = x : \text{append } xs \hspace{1pt} ys
\end{align*}
\]

Concatenate known lists:

\[
\text{append } [1,2] \quad [3,4] \quad \mapsto \quad [1,2,3,4]
\]

Splitting a list:

\[
\text{append } x \quad [2] \quad =:= \quad [1,2] \quad \mapsto \quad \text{not reducible (delay)}
\]

Escher [Lloyd 94]: provide additional split predicate (superfluous from a declarative point of view)

Prolog: define \texttt{append} always as a predicate

⇒ worse operational behavior than a function:

Curry: \texttt{append (append x y) z =:= []}

finite search space (if append is flexible)

Prolog: \texttt{append(X,Y,L), append(L,Z,[])}

infinite search space
Functional Logic Programming

vs.

(Concurrent) Logic Programming

Implementation of functions by flattening

→ loss of functional dependencies:

\[
\begin{align*}
\text{from } n &= n : \text{from } (S \ n) \\
\text{first } 0 \ x s &= [] \\
\text{first } (S \ n) \ (x:xs) &= x : \text{first } n \ xs
\end{align*}
\]

\[
\begin{align*}
\text{first } x \ (\text{from } x) &= [: [] \\
\sim \{x=0\} \ [] &= [: [] | \{x=(S \ n)\} ... \text{failure}... \sim \{x=0\}
\end{align*}
\]

Translation of functions into predicates by flattening:

\[
\begin{align*}
\text{from}(N,[N|R]) &:- \text{from}(s(N),R). \\
\text{first}(0,L,[]). \\
\text{first}(s(N),[E|L],[E|R]) &:- \text{first}(N,L,R).
\end{align*}
\]

\[
\begin{align*}
\text{first}(X,L,[]), \text{from}(X,L) \\
\sim\{x \rightarrow 0\} \ \text{from}(0,L) \sim \text{from}(s(0),L1) \sim ...
\end{align*}
\]
Higher-order functions:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
\[
\text{map } f \ [\] \ = \ []
\]
\[
\text{map } f \ (x:xs) = (f \ x) : \text{map } f \ xs
\]

\[
\text{map} \ (\text{append } [1]) \ [[2],[3]] \ \leadsto \ [[1,2],[1,3]]
\]

- higher-order features of functional languages (partial applications, λ-abstractions)
- first-order definition of application function (as in [Warren 82])
- application function is \textit{rigid}
  \[
  \leadsto \text{delay applications with unknown functions}
  \]
- future extension(?): higher-order unification
Monadic Input/Output

- declarative I/O concept
- I/O: transformation on the outside world
- interactive program: compute actions (transformation on the world)
- type of actions: $\text{IO } t \cong \text{World} \to (t,\text{World})$

getChar :: IO Char
getLine :: IO String
putLine :: String \to IO ()

getChar applied to a world
\(\rightsquigarrow\) character + new (transformed) world

- compose actions:
  
  (\(\gg\gg\)) :: IO a \to (a \to IO b) \to IO b

getLine \(\gg\gg\) putLine:
copies a line from input to output

- no I/O in disjunctions ("cannot copy the world"): encapsulate search between I/O actions
External Functions

- infinite set of defining equations
  \[ 0+0 = 0 \]
  \[ 0+1 = 1 \]
  \[ 0+2 = 2 \]
  \[ \ldots \]
  \[ 2+1 = 3 \]
  \[ \ldots \]

- definition not accessible

- external implementation (without side effects)

- suspend external function calls until arguments are fully known, i.e., ground
  [Bonnier/Maluszynski 88, Boye 91]

- external function interface

- implementation of basic arithmetic
  (+, -, *, \ldots: external functions)

Not possible in narrowing-based languages!
Arithmetic

0, 1, 2, …: constructors

+, −, *, …: external functions

\[ x := 2 + 3 \times 4 \leadsto \{ x = 14 \} \]

\[ x := 2 \times 3 + y \leadsto \{ \} \ x := 6 + y \quad (suspend) \]

\[ x + x := y \ & \ x := 2 \]

\[ \leadsto \{ x = 2 \} \ 2 + 2 := y \quad (suspend \ x + x) \]

\[ \leadsto \{ x = 2 \} \ 4 := y \quad (evaluate \ 2 + 2) \]

\[ \leadsto \{ x = 2, \ y = 4 \} \]

⇒ Functions as passive constraints (Life)

\[
\begin{align*}
\text{digit 0} &= \text{success} \\
\ldots \\
\text{digit 9} &= \text{success}
\end{align*}
\]

\[ x + x =:= y \ & \ x \times x =:= y \ & \ \text{digit} \ x \]

\[ \leadsto \{ x = 0, \ y = 0 \} \ | \ \{ x = 2, \ y = 4 \} \]
Implementations of Curry

- First prototypical implementations available

- Interpreter in Prolog: TasteCurry-System (RWTH Aachen, Portland State University)
  http://www-i2.informatik.rwth-aachen.de/~hanus/tastecurry

  [Hanus LOPSTR’95]: Efficient implementation of needed narrowing by transformation into Prolog
  Sloth-System [Mariño/Rey WFLP’98]

- Compiler Curry→Java [Hanus/Sadre ILPS’97]
  (Java threads for concurrency and non-determinism)
  - portable
  - simplified implementation (garbage collection, threads)
  - slow but (hopefully!) better Java implementations in the future

- abstract Curry machine [Lux WFLP’98]
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

[Hanus PLILP’97]
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**

More infos on Curry:

http://www-i2.informatik.rwth-aachen.de/~hanus/curry