A Unified Computation Model for Declarative Programming

Michael Hanus

RWTH Aachen

Declarative Programming

- no coding of algorithms
- description of logical relationships
- more expressive power, higher programming level
- reliable and maintainable programs
  - pointer structures ⇒ algebraic data types
  - complex procedures ⇒ comprehensible parts (pattern matching, local definitions)
- *Functional programming:*
  – functions and equations
  – deterministic reduction
- *Logic programming:*
  – predicates and logical formulas
  – non-deterministic search for solutions

⇒ **Functional logic languages:**

  - expressive power of logic languages
  - efficient execution principles of functional languages

Problems with Prolog

- uncontrolled nondeterminism:
  inefficient, infinite loops
- pruning operators: non-declarative (red cuts)
- non-declarative I/O: backtracking?

Integration of functions:

- better efficiency (determinism!)
- new programming techniques:
  higher-order functions, laziness, infinite data structures
- declarative I/O
- no burden on the programmer:
  most predicates are functions!

Functional logic languages
can avoid impure features of Prolog

Functional Logic Languages

- currently not widely used
- operational principles unclear
- many different evaluation strategies
- important advances in recent years

Curry: [ILPS'95, Dagstuhl'96]

- standard functional logic language
- combines most important features of existing
  - functional languages
  - logic languages
  - functional logic languages
- combine separated research efforts in functional and logic programming
- combine applications of both fields
- boost declarative programming

Main aspect: computation model of Curry [POPL'97]

Values

Imperative languages:

basic values + pointer structures

Declarative languages:

algebraic data types

```
data Bool    = true | false
data Nat     = 0    | s(Nat)
data List(A) = []   | [A|List(A)]
data Tree(A) = leaf(A)
             | node(tree(A),A,tree(A))

data Int = 0 | 1 | -1 | 2 | -2 | ...
```

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

`s(X)`   `s(s(0))`   `[0,s(0),s(s(0))]`

$\approx$ *Herbrand terms*

Functions

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

  unconditional: $f(t_1, \ldots, t_n) \; = \; r$

  conditional: $f(t_1, \ldots, t_n) \mid \{c\} \; = \; r$

  f: defined operation

  $t_1, \ldots, t_n$: data terms

  r: expression (right-hand side)

  c: constraint (conjunction of equations)

```
            0 + Y = Y
         s(X) + Y = s(X+Y)

append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]

member(E,L) | {append(_,[E|_])=L} = true
```

Predicates: Boolean functions: P :- $Q_1, \ldots, Q_n \rightsquigarrow$

P `|` $\{Q_1$`=true,...,`Q_n`=true`$\}$ `= true`

but: functions have more information (determinism)

Evaluation

- 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)

```
0 + Y = Y
s(X) + Y = s(X+Y)
```

```
s(s(0))+s(0)  →  s(s(0)+s(0))
              →  s(s(0+s(0)))
              →  s(s(s(0)))
```

Evaluation strategy: selection function for redexes

Normalizing strategy: always computes value if it exists

Eager strategy: select innermost redex, not normalizing

Lazy strategy: select an outermost redex

Lazy Evaluation

1. Apply evaluation steps at outermost positions:

```
   0 + Y = Y                0 ≤ Y    = true
s(X) + Y = s(X+Y)        s(X) ≤ 0    = false
                         s(X) ≤ s(Y) = X≤Y
```

$0 \le s(0)+s(0) \quad \to \quad true$

2. Evaluate arguments if necessary:

$s(0) \le 0+0 \quad \to \quad s(0) \le 0 \quad \to \quad false$

```
f = f
```

$0+0 \le f$: two outermost redexes:

$0+0 \le f \quad \to \quad 0 \le f \quad \to \quad true$

$0+0 \le f \quad \to \quad 0+0 \le f \quad \to \quad 0+0 \le f \quad \to \quad \cdots$

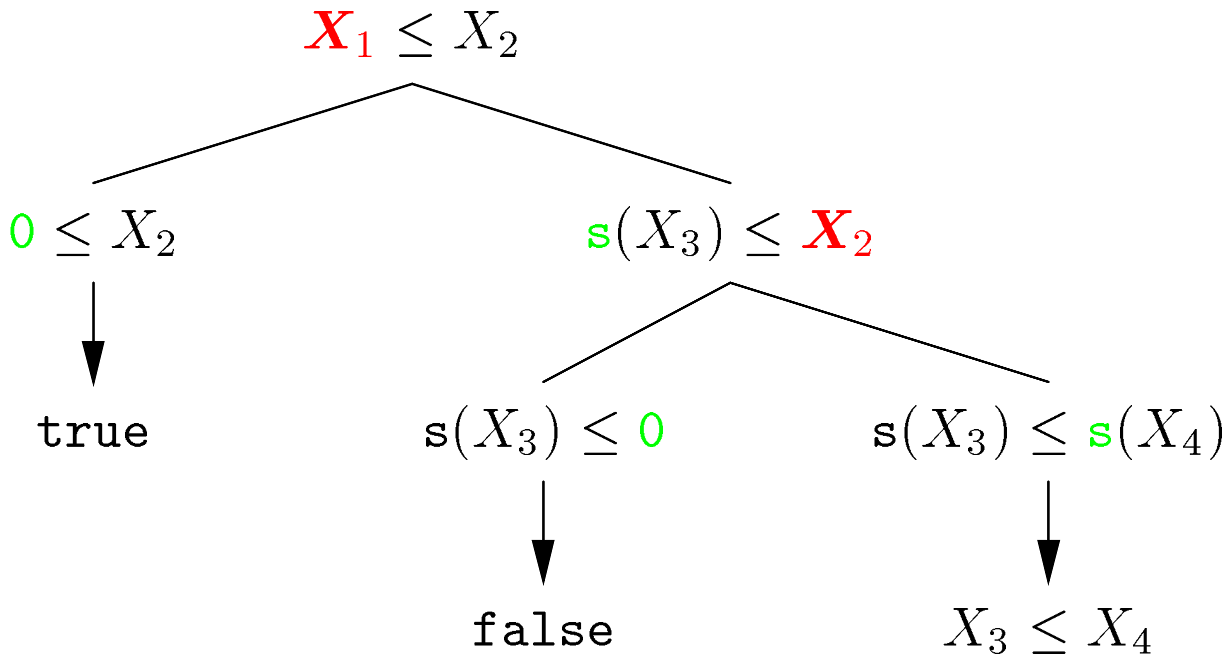
Ideal strategy: evaluate only needed redexes

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

$$
\begin{array}{lcll}
\mathtt{0} & \leq & \mathtt{Y} & = \mathtt{true} \\
\mathtt{s(X)} & \leq & \mathtt{0} & = \mathtt{false} \\
\mathtt{s(X)} & \leq & \mathtt{s(Y)} & = \mathtt{X} \leq \mathtt{Y}
\end{array}
$$



Function call: $t_1 \leq t_2$

1. Reduce t_1 to head normal form
2. If $t_1 = 0$: apply rule
3. If $t_1 = s(\cdots)$: reduce t_2 to head normal form
4. If t_1 variable: not reducible

Properties of Reduction with Definitional Trees

- Normalizing strategy ($\approx$ sound and complete)

- Independent on the order of rules

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

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

- Easily extensible to more general classes

Overlapping Rules: Non-deterministic Rewriting

```
true ∨ X      = true
   X ∨ true   = true
false ∨ false = false
```

No unique argument for a branch (case distinction):

$\boxed{e_1 \vee e_2}$ 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 (*don't know*) e_1 or e_2

Extension to definitional trees:
Introduce *or*-nodes to describe non-deterministic selection of redexes

From Functional Programming to Logic Programming

Functional programming:
initial expression without free variables

Logic programming:
initial expression or rule conditions may contain free variables

Operational extension:

instantiate free variables, if necessary

```
f(a) = c
f(b) = d
```

Evaluate `f(X)`:

- Bind `X` to `a` and reduce `f(a)` to `c`, or:
- Bind `X` to `b` and reduce `f(b)` to `d`

Disjunctive Expressions

Functional programming: *values* of expressions

Logic programming: *computed answers*

Answer expression: $\sigma \,[]\, e$ (bindings + expression)

$\sigma \,[]\, e$ solved: e does not contain defined functions

Disjunctive expression: $\sigma_1 \,[]\, e_1 \mid \ldots \mid \sigma_n \,[]\, e_n$
(omit identity substitutions)

Computation step: reduce an unsolved expression

Deterministic step: replace expression by new expression

Non-deterministic step: replace expression by disjunction of new expressions

```
f(a) = c
f(b) = d
```

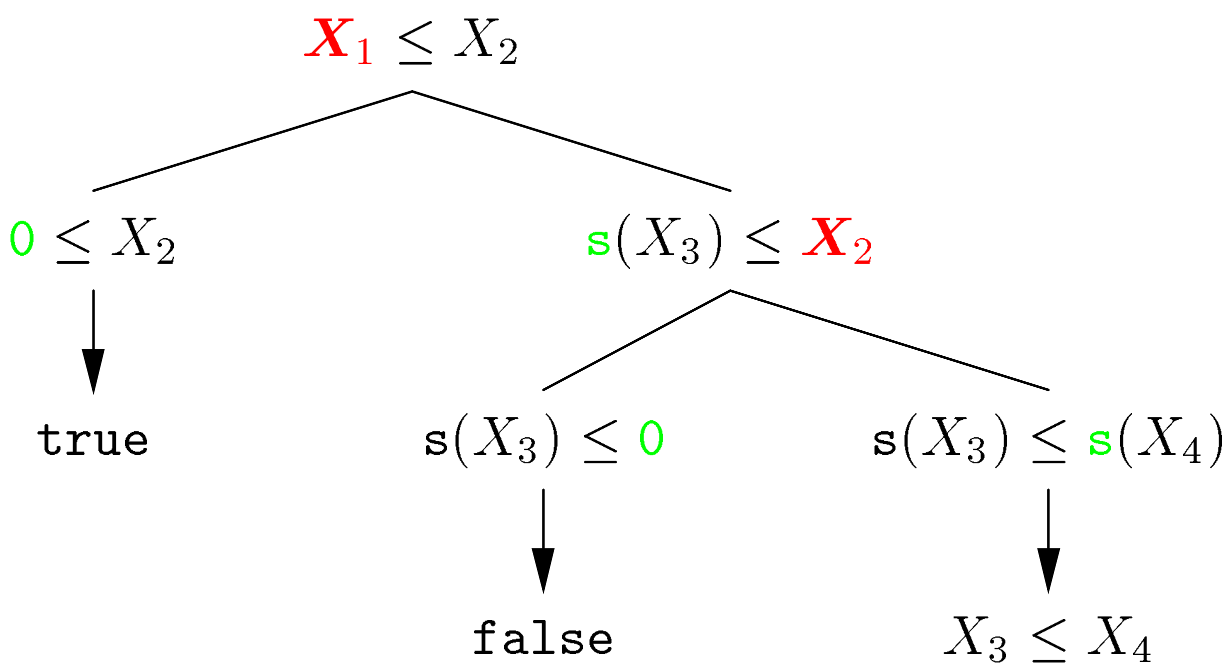
```
f(a)  ⇝  c
f(X)  ⇝  {X=a} c | {X=b} d
```

Compute necessary bindings with definitional trees
$\leadsto$ *needed narrowing*

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

```
   0  ≤ Y    = true
s(X)  ≤ 0    = false
s(X)  ≤ s(Y) = X≤Y
```



Function call: $t_1 \leq t_2$

1. Reduce t_1 to head normal form
2. If $t_1 = 0$: apply rule
3. If $t_1 = s(\cdots)$: reduce t_2 to head normal form
4. If t_1 variable: bind t_1 to 0 or $s(X)$ and proceed

Needed Narrowing

[Antoy/Echahed/Hanus POPL'94]

Inductively sequential systems: Each operation has a definitional tree (without *or* nodes)

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

Optimal strategy:

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 σ and σ' computed by two distinct derivations are independent (i.e., $\sigma(x)$ and $\sigma'(x)$ are not unifiable for some variable x).

Advantages of integrating functions into logic programs:

Better evaluation strategies
without loosing search facilities

Suspending Function Calls

Disadvantage of narrowing:

- function inversion not reasonable for all functions
- 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:

Residuation (Escher, Le Fun, Life, NUE, Oz,...)

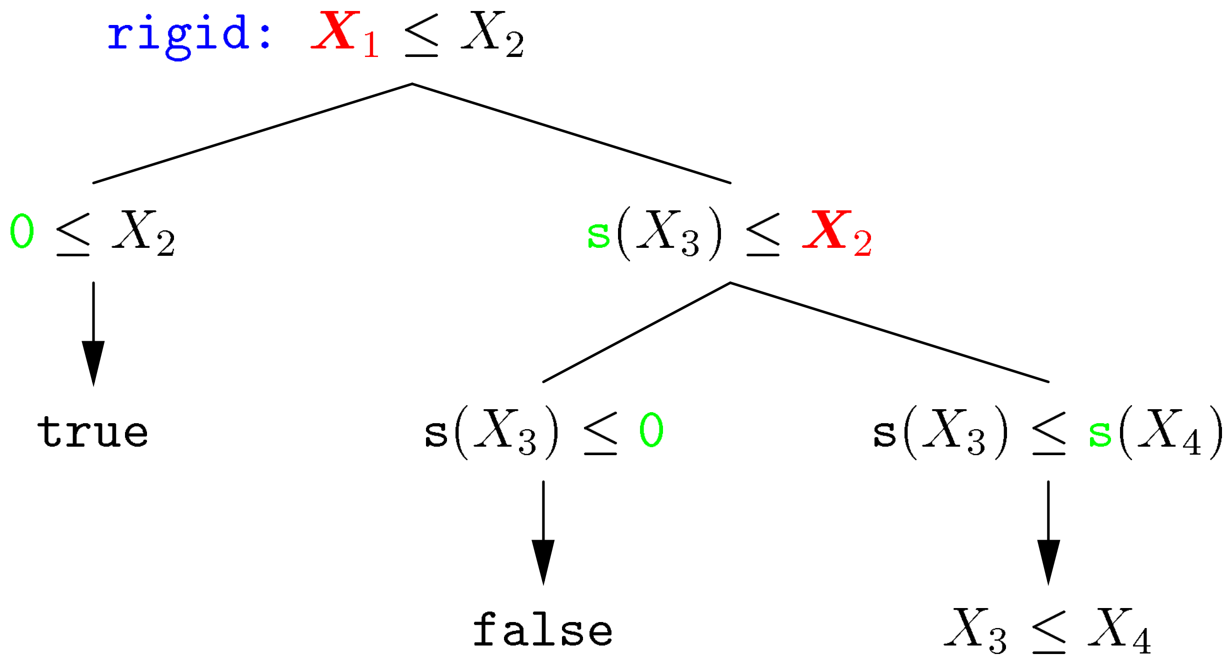
`X+X=Y, X=2`	$\leadsto$	`{X=2} 2+2=Y`	(suspend `X+X`)
	$\leadsto$	`{X=2} 4=Y`	(evaluate `2+2`)
	$\leadsto$	`{X=2, Y=4} valid`	

Extension to definitional trees:
Mark *branch* nodes as *rigid* or *flexible*

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

```
   0  ≤ Y    = true
s(X)  ≤ 0    = false
s(X)  ≤ s(Y) = X≤Y
```



Function call: $t_1 \leq t_2$

1. Reduce t_1 to head normal form
2. If $t_1 = 0$: apply rule
3. If $t_1 = s(\cdots)$: reduce t_2 to head normal form
4. If t_1 variable: do not evaluate function call

Equality and Constraints

Equational constraint e_1=e_2:

satisfied if both sides reducible to unifiable data terms (*strict equality*)

$\Rightarrow$ e_1=e_2 does not hold if e_1 or e_2 undefined

Constraints

- occur in conditions of rules: $l \,|\, \{c\} \;=\; r$
- are solved when this rule is applied
- are connected by concurrent conjunction $\wedge$:
  To evaluate $c_1 \wedge c_2$, evaluate c_1
  If evaluation of c_1 suspends, evaluate c_2

  Notation: $\{c_1, \ldots, c_n\}$ instead of $\{c_1 \wedge \ldots \wedge c_n\}$
- are first-class objects:

Transform a list of constraints into a single constraint:

```
conj([])     = {}
conj([C|Cs]) = {C, conj Cs}
```

$\leadsto$ provide functions to generate flexible process architectures

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

$$e \quad \leadsto^* \quad (\sigma_1 \,[\!]\, e_1) \mid \ldots \mid (\sigma_n \,[\!]\, e_n)$$

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

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

$$e \quad \leadsto^* \quad (\sigma_1 \,[\!]\, e_1) \mid \ldots \mid (\sigma_n \,[\!]\, e_n)$$

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 *branch* nodes *flexible*: If $\sigma(e) \rightarrow^*_{\mathcal{R}} c$, then $\exists$

$$e \quad \leadsto^* \quad (\sigma_1 \,[\!]\, e_1) \mid \ldots \mid (\sigma_n \,[\!]\, e_n)$$

with $e_i = c$ and $\sigma = \varphi \circ \sigma_i$ for some i and φ

Definitional Trees: Unification of Computation Models

Strategy	Restrictions on definitional trees
Needed narrowing [POPL'94]	only *rule* and flexible *branch* nodes; optimal w.r.t. length of derivations and number of computed solutions
Weakly needed narrowing (∼Babel)	only *rule*, flexible *branch*, and *or* nodes
Simple lazy narrowing and resolution (∼Prolog)	particular trees with flexible *branch* nodes (one *branch*/*rule* tree for each left-hand side, all rules connected by *or* nodes)
Lazy functional languages (∼Haskell)	trees with left-to-right pattern matching; initial expression has no free variable
Residuation (∼Life)	rigid branches for non-Boolean functions; flexible branches for predicates

Curry: A Truly Integrated Functional Logic Language

- combines most important features of existing
  - functional languages
  - logic languages
  - functional logic languages
- definitional trees are automatically generated:
  - left-to-right pattern matching
  - *rigid* branches for functions
  - *flexible* branches for predicates
- conditional rules
- higher-order functions
- polymorphic type system, modules
- declarative (monadic) I/O
- committed choice, encapsulated search
- equational constraints
- external functions and constraint solvers

Programming in Curry

```
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
```

Functional programming:

`append([1,2],[3,4])` $\leadsto$ `[1,2,3,4]`

Logic programming (branch is *flexible*):

`append(L,M)=[1,2]`

$\leadsto$ `{L=[],M=[1,2]} valid`

| `{L=[1],M=[2]} valid`

| `{L=[1,2],M=[]} valid`

```
           from(N) = [N|from(s(N))]
first(0,L)         = []
first(s(N),[E|L])  = [E|first(N,L)]
```

Lazy functional programming:

`first(s(s(0)),from(0))` $\leadsto$ `[0,s(0)]`

Lazy functional logic programming:

`first(X,from(Y))=[0]` $\leadsto$ `{X=s(0),Y=0} valid`

Conditional Rules

```
member(E,L) | {append(_,[E|_])=L} = true
```

Conditional rules l `|` $\{c\}$ `=` r

- condition c: conjunction of equational constraints
- l `|` b `=` r (b`::Bool`) $\approx$ l `|` $\{b$`=true`$\}$ `=` r
- extra variables in conditions
- implement conditional rule as "l `=` `(`$c \Rightarrow r$`)`"
- semantics of *guarded expression* "$c \Rightarrow r$":

$$\texttt{(valid} \Rightarrow \texttt{X)} \;=\; \texttt{X}$$

`member(X,[a,b,c])`

$\rightsquigarrow$ `{X=a} valid | {X=b} valid | {X=c} valid`

Translation: Logic programs $\hookrightarrow$ Curry:

$\texttt{p :- p}_1\texttt{,...,p}_n$

$\mapsto$ $\texttt{p | p}_1\texttt{ \&\&...\&\& p}_n \texttt{ = true}$ (*sequential*)

$\mapsto$ $\texttt{p | \{p}_1\texttt{=true,...,p}_n\texttt{=true\} = true}$ (*concurrent*)

Better: define functions instead of predicates

Functions vs. Predicates

rigid branches for functions not always reasonable:

```
append([]     ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
```

Concatenate known lists:

`append([a,b],[c,d])` $\rightsquigarrow$ `[a,b,c,d]`

Splitting a list:

`append(L,[b]) = [a,b]` $\rightsquigarrow$ *not reducible*

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

Prolog: define `append` always as a predicate
$\Rightarrow$ worse operational behavior than a function:

Curry: `append(append(X,Y),Z) = []`
finite search space (with *flexible* branch node)

Prolog: `append(X,Y,L), append(L,Z,[])`
infinite search space

Functional Logic Programming vs. (Concurrent) Logic Programming

Implementation of functions by flattening
$\rightsquigarrow$ loss of functional dependencies:

```
               from(N) = [N|from(s(N))]
    first(0,L)         = []
    first(s(N),[E|L])  = [E|first(N,L)]
```

`first(X,from(X)) = []`
$\rightsquigarrow$ `{X=0} [] = []  |  {X=s(N)}` ...*failure*...
$\rightsquigarrow$ `{X=0} valid`

Translation of functions into predicates by flattening:

```
  from(N,[N|R]) :- from(s(N),R).
  first(0,L,[]).
  first(s(N),[E|L],[E|R]) :- first(N,L,R).
```

`first(X,L,[]), from(X,L)`
$\rightsquigarrow_{\{X \mapsto 0\}}$ `from(0,L)` $\rightsquigarrow$ `from(s(0),L1)` $\rightsquigarrow$ $\cdots$

Higher-Order Features

Higher-order functions:

```
map :: (A -> B) -> [A] -> [B]
map(F,[]    ) = []
map(F,[E|L]) = [F@E|map(F,L)]
```

`map(add@1,[1,2])` $\leadsto$ `[2,3]`

- higher-order features of functional languages (partial applications, λ-abstractions)
- application function `F@X` $\approx$ `F(X)` (implemented as in [Warren 82])
- first argument of `@` is *rigid*

  $\leadsto$ 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: `IO(t)` $\approx$ `World -> (t,World)`

  ```
  getChar :: IO(Char)
  getLine :: IO(String)
  putLine :: String -> IO()
  ```

  `getChar` applied to a world
  $\rightsquigarrow$ character + new (transformed) world
- compose actions:

  ```
  (>>=) ::  IO(a) -> (a -> IO(b)) -> IO(b)
  ```

  `getLine >>= putLine`:
  copies a line from input to output
- no I/O in disjunctions ("cannot copy the world"): *encapsulate search*

Encapsulated Search

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

Idea:
Compute until a nondeterministic step occurs,
then give programmer control over this situation
(generalization of Oz's search operator)

Search:

- solve constraint containing search variable
- bind search variable to different solutions
  $\Rightarrow$ abstract search variable: $\lambda x.\{c\}$
- primitive search operator:

```
try :: (A -> Constraint) -> [A -> Constraint]
```

$\texttt{try}(\lambda\texttt{X}.\{\texttt{1=2}\}) \leadsto \texttt{[]}$ *failure*

$\texttt{try}(\lambda\texttt{X}.\{\texttt{[X]=[0]}\}) \leadsto [\lambda\texttt{X}.\{\texttt{X=0}\}]$ *success*

$\texttt{try}(\lambda\texttt{X}.\{\texttt{X}\leq\texttt{0=true}\}) \leadsto [\lambda\texttt{X}.\{\texttt{X=0, 0}\leq\texttt{0=true}\},$
$\lambda\texttt{X}.\{\texttt{X=s(Y), s(Y)}\leq\texttt{0=true}\}]$
disjunction

Encapsulated Search: Search Strategies

`try`$(\lambda x.\{c\})$: evaluate, stop after nondeterministic step

Depth-first search: collect all solutions

```
dfs :: (A -> Constraint) -> [A -> Constraint]
dfs(G) = collect(try(G))
where
 collect([])   = []
 collect([S])  = [S]
 collect([G1,G2|Gs]) =
                  concat(map(dfs,[G1,G2|Gs]))
```

`dfs(`λ`L.{append(L,[1])=[0,1]})` $\rightsquigarrow$ `[`λ`L.{L=[0]}]`

Further search strategies:

- compute only first solution:

  `first(G) = head(dfs(G))`

- best solution search
- $\cdots$

External Functions

- infinite set of defining equations

```
0+0 = 0
0+1 = 1
0+2 = 2
...
2+1 = 3
...
```

- 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
  (+, -, *,. . . : external functions)

Not possible in narrowing-based languages!

Implementation

- First prototype implementation available (Prolog interpreter)

- Efficient implementation possible based on pattern-matching

- [Hanus LOPSTR'95]: Efficient implementation of needed narrowing by transforming definitional trees into Prolog

- Better implementation of deterministic computations by compilation into abstract machines ($\sim$ ALF, Babel, ...)

- Implementation in Java (threads for concurrency) under work

Conclusions

Curry: a truly integration of declarative paradigms

Functional programming: higher-order functions, lazy evaluation, deterministic evaluation of ground expressions

Logic programming: logical variables, partial data structures, search facilities

Evaluation strategy:

- specified by definitional trees
- lazy reduction of expressions
- necessary instantiation of free variables
- suspension on uninstantiated variables
- concurrent (constraint) programming
- higher-order programming

More infos on Curry:

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