# A Unified Computation Model for Functional and Logic Programming

*Michael Hanus*

*RWTH Aachen*

# Declarative Programming

- no coding of algorithms
- description of logical relationships
- more expressive power, higher programming level
- readable and reliable programs
- two main fields:
  - *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
  - combine applications of both fields
  - many different evaluation strategies

# Evaluation Principles of Functional Logic Languages

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

Delay functions calls until deterministic reduction is possible, non-determinism by predicates/disjunctions

$$\begin{array}{ll} 0 + X \to X & nat(0) \to true \\ s(X) + Y \to s(X+Y) & nat(s(X)) \to nat(X) \end{array}$$

$$\begin{array}{ll} & Z + 0 \approx s(0) \wedge nat(Z) \\ \leadsto_{\{Z \mapsto s(X)\}} & s(X) + 0 \approx s(0) \wedge nat(X) \\ \leadsto_{\{\}} & s(X+0) \approx s(0) \wedge nat(X) \\ \leadsto_{\{X \mapsto 0\}} & s(0+0) \approx s(0) \wedge true \\ \leadsto_{\{\}} & s(0) \approx s(0) \wedge true \quad \leadsto^{*} \quad true \end{array}$$

Computed answer: $\{Z \mapsto s(0)\}$

**Advantages:**

- deterministic evaluation of functions
- supports concurrent programming (synchronization on logical variables)

# Evaluation Principles of Functional Logic Languages

**Narrowing:** (ALF, Babel, K-Leaf, LPG, SLOG,...)

Instantiate goal variables and apply a reduction step

$$\begin{array}{rcl} 0 + X & \rightarrow & X \\ s(X) + Y & \rightarrow & s(X + Y) \end{array}$$

$$\begin{array}{rll} Z + 0 \approx s(0) & \leadsto_{\{Z \mapsto s(X)\}} & s(X + 0) \approx s(0) \\ & \leadsto_{\{X \mapsto 0\}} & s(0) \approx s(0) \end{array}$$

Computed answer: $\{Z \mapsto s(0)\}$

**Advantage:** Ability to solve equations

**Disadvantage:** huge search space
(non-deterministically select a position *and* a rule)

$\Longrightarrow$ **narrowing strategies** [Hanus JLP'94]

# Lazy Narrowing

$$\begin{array}{rcllcl} 0 & \leq & X & & \rightarrow & true \\ s(X) & \leq & 0 & & \rightarrow & false \\ s(X) & \leq & s(Y) & & \rightarrow & X \leq Y \end{array}$$

1. Apply narrowing steps at outermost positions:

$$0 \leq s(0) + s(0) \quad \leadsto_{\{\}} \quad true$$

$$X \leq Y + Y \quad \leadsto_{\{X \mapsto 0\}} \quad true$$

2. Evaluate arguments if necessary:

$$\begin{array}{lll} X \leq Y + Y & \leadsto_{\{Y \mapsto 0\}} & X \leq 0 \\ & \leadsto_{\{X \mapsto s(X_1)\}} & false \end{array}$$

Also possible:

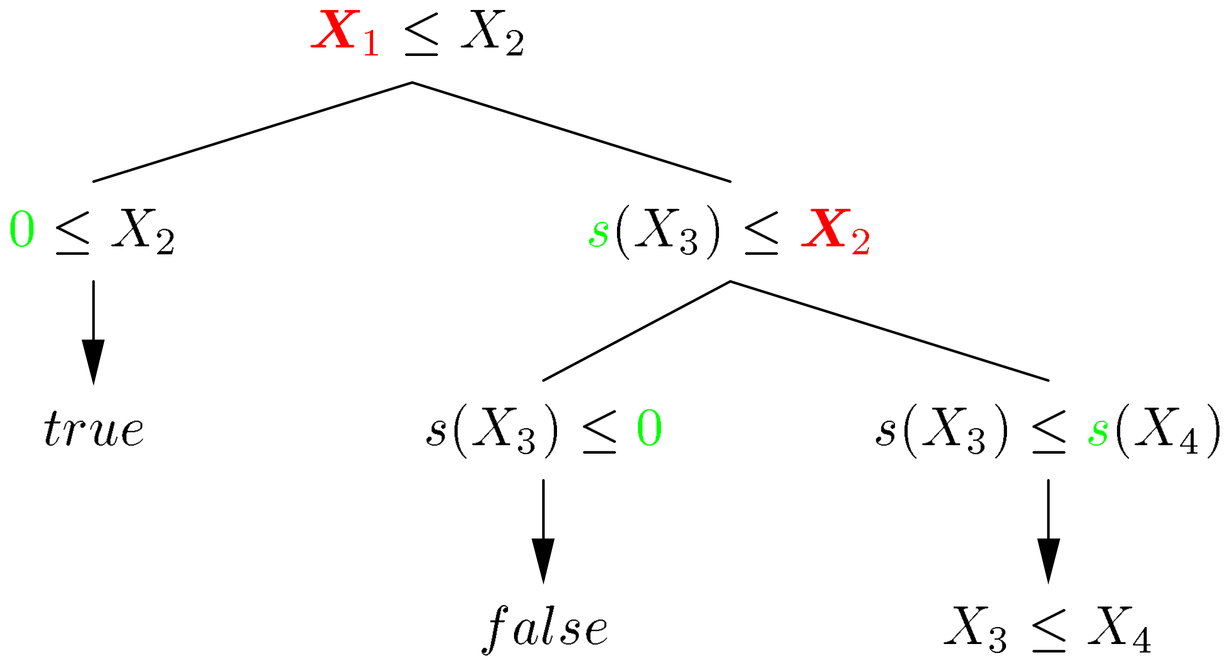
$$\begin{array}{lll} X \leq Y + Y & \leadsto_{\{Y \mapsto 0\}} & X \leq 0 \\ & \leadsto_{\{X \mapsto 0\}} & true \end{array}$$

Superfluous step + specialized answer!

# 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

$$\begin{array}{rcll} 0 & \leq & X & \to \mathit{true} \\ s(X) & \leq & 0 & \to \mathit{false} \\ s(X) & \leq & s(Y) & \to X \leq Y \end{array}$$




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: bind $t_1$ to 0 or $s(X)$ and proceed

rigid:

[redacted]

do not evaluate function call

# Needed Narrowing

[Antoy/Echahed/Hanus POPL'94]

**Inductively sequential systems:** The rules of each operation can be organized in a *definitional tree* (similar to functional programs)

**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 (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**

# Residuation vs. Narrowing

## Residuation:

+ deterministic evaluation of functions
+ provide concurrent computations
+ simple connection of external functions (arithmetic,...)

− incomplete (no solution if functions not evaluable)

• non-determinism in predicates or disjunctions

## Narrowing:

+ completeness w.r.t. functional and logic programming
+ optimal evaluation (for typical programs)

− expensive function inversion (non-determinism)

− combination with external functions unclear

• implicit non-determinism ($\sim$ logic programming)

$\Longrightarrow$ **Combine residuation and narrowing**

# 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:** $(\sigma_1 \,[]\, e_1) \vee \ldots \vee (\sigma_n \,[]\, e_n)$

Computation step: reduce an unsolved expression

Deterministic step: replace expression by new expression

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

$$\begin{array}{lcl} p(a) & \to & true \\ p(b) & \to & true \end{array}$$

$$p(a) \leadsto true$$

$$p(X) \leadsto (X = a \,[]\, true) \vee (X = b \,[]\, true)$$

$\Rightarrow$ Specification by (extended) definitional trees

## Specify Concurrent Evaluation: *and* nodes

If more than one argument needed for evaluation:

$\boxed{and(\mathcal{T}_1, \mathcal{T}_2)}$ $\mathcal{T}_1, \mathcal{T}_2$ definitional trees with pattern $\pi$

Strategy: Try $\mathcal{T}_1$; if suspended: try $\mathcal{T}_2$

$$true \wedge true \;\rightarrow\; true$$

$$\begin{array}{rcl} and(branch(X_1 \wedge X_2 & \stackrel{rigid}{\Longrightarrow} & branch(true \wedge X_2 \\ & \stackrel{rigid}{\Longrightarrow} & \boxed{true \wedge true \rightarrow true})), \\ branch(X_1 \wedge X_2 & \stackrel{rigid}{\Longrightarrow} & branch(X_1 \wedge true \\ & \stackrel{rigid}{\Longrightarrow} & \boxed{true \wedge true \rightarrow true})) \end{array}$$

Strategy: To evaluate $t_1 \wedge t_2$, evaluate $t_1$

If evaluation of $t_1$ suspends, evaluate $t_2$

# Overlapping Rules: *or* nodes

No unique argument for a branch (case distinction):

$\boxed{or(\mathcal{T}_1, \mathcal{T}_2)}$ $\mathcal{T}_1, \mathcal{T}_2$ definitional trees with pattern $\pi$

Strategy: Try non-deterministically both $\mathcal{T}_1$ and $\mathcal{T}_2$

$$\begin{array}{rcl} 0 * X & \to & 0 \\ X * 0 & \to & 0 \end{array}$$

No unique argument to distinguish the rules:

$$\begin{array}{l} or(branch(X_1 * X_2 \overset{rigid}{\Longrightarrow} \boxed{0 * X_2 \to 0}), \\ \quad branch(X_1 * X_2 \overset{rigid}{\Longrightarrow} \boxed{X_1 * 0 \to 0})) \end{array}$$

Strategy: reduce $t_1 * t_2$ if one argument is 0,
otherwise suspend (e.g., for $X * Y$)

# Definitional Trees

- data structure to specify computation strategy

- each node has a distinct *pattern* $\pi$

- kinds of nodes with pattern $\pi$:

  $\boxed{rule(\pi \to r)}$ if $\pi \to r$ variant of a rule

  $\boxed{branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)}$
  $o$ occurrence of variable in $\pi$,
  $r \in \{rigid, flex\}$,
  $c_1, \ldots, c_k$ different constructors,
  $\mathcal{T}_i$ tree with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$

  $\boxed{or(\mathcal{T}_1, \mathcal{T}_2)}$ $\mathcal{T}_1, \mathcal{T}_2$ definitional trees with pattern $\pi$

  $\boxed{and(\mathcal{T}_1, \mathcal{T}_2)}$ $\mathcal{T}_1, \mathcal{T}_2$ definitional trees with pattern $\pi$

- definitional tree [Antoy 92]: without *or*/*and*-nodes

# Computation Strategy

To evaluate $t$, consider tree of outermost function in $t$:

$rule(\cdots)$: apply this rule

$branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$: consider subterm $t|_o$:

- $f(\cdots)$: evaluate $t|_o$ to head normal form
- $c(\cdots)$: select appropriate $\mathcal{T}_j$ and proceed with $\mathcal{T}_j$
- $X$ and $r = rigid$: suspend evaluation
- $X$ and $r = flex$: unify $X$ with (pattern of $\mathcal{T}_j$)$|_o$ (non-deterministic step)

$or(\mathcal{T}_1, \mathcal{T}_2)$: try non-deterministically both $\mathcal{T}_1$ and $\mathcal{T}_2$ (no unique argument to branch)

$and(\mathcal{T}_1, \mathcal{T}_2)$: try $\mathcal{T}_1$; if suspended: try $\mathcal{T}_2$ (more than one argument to evaluate)

# Soundness and Completeness

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

**Soundness:** If

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

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

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

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

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

**Completeness w.r.t. flexible functions:**
All *branch* nodes *flexible*: If $\sigma(e) \to^*_{\mathcal{R}} c$ , then $\exists$

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

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

# 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 can be omitted:
  automatic generation:
  - left-to-right pattern matching
  - *rigid* branches for functions
  - *flexible* branches for predicates
- conditional rules
- higher-order functions

$\leadsto$ POPL'97 Proceedings

First prototype implementation available
(Prolog interpreter)

# Functions vs. Predicates

*rigid* branches for functions not always reasonable:

```
append []    L = L
append [E|R] L = [E|append R L]
```

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)` $\wedge$ `append(L,Z,[])`
infinite search space

# Conditional Rules

```
function member: A -> [A] -> bool
member E L = true  if  append L1 [E|L2] == L
```

Conditional rules $l \to r$ `if` $c$:

- Boolean expression as condition $c$
- strict equations $l$==$r$ in condition
  ($l, r$ unifiable constructor terms)
- extra variables in conditions
- implement conditional rule as "$l \to (c \Rightarrow r)$"
- semantics of *conditional expression* "$c \Rightarrow r$":

$$(\texttt{true} \Rightarrow X) \;\to\; X$$

`member X [a,b,c]`

$\leadsto$ $\{$`X=a`$\}$ `true` $\vee$ $\{$`X=b`$\}$ `true` $\vee$ $\{$`X=c`$\}$ `true`

Logic programs $\hookrightarrow$ Curry:

`p :-` $\mathtt{p}_1, \ldots, \mathtt{p}_n$ $\mapsto$ `p = true if` $\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_n$

# Higher-Order Features

Higher-order functions:

```
function 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
- application function $apply(F, X) \to F(X)$
- first argument of *apply* is *rigid*

  $\leadsto$ delay applications with unknown functions
- future extension: higher-order unification

# Conclusions

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

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

**A unified computation model:**

- evaluation strategy specified by definitional trees
- lazy reduction of expressions
- narrowing: instantiate variable arguments
- residuation: suspend on variable arguments
- narrowing $\leftrightarrow$ residuation: one bit in branch nodes
- soundness and completeness results
- default strategy: automatic generation of definitional trees (left-to-right pattern matching)

$\Rightarrow$ **Curry**

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