Curry

A Truly Functional Logic Language

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

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
- boost declarative programming

Main aspect: operational principles of Curry

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

$$
\begin{array}{llll}
0 + X & \to X & \qquad & nat(0) \\
s(X) + Y & \to s(X + Y) & \qquad & nat(s(X)) \Leftarrow nat(X)
\end{array}
$$

$$
\begin{array}{ll}
& Z + 0 \approx s(0),\ nat(Z) \\
\rightsquigarrow_{\{Z \mapsto s(X)\}} & s(X) + 0 \approx s(0),\ nat(X) \\
\rightsquigarrow_{\{\}} & s(X + 0) \approx s(0),\ nat(X) \\
\rightsquigarrow_{\{X \mapsto 0\}} & s(0 + 0) \approx s(0) \\
\rightsquigarrow_{\{\}} & s(0) \approx s(0)
\end{array}
$$

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

Advantages:
- deterministic evaluation of functions
- supports concurrent programming

Evaluation Principles of Functional Logic Languages

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

Instantiate goal variables and apply a reduction step

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

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

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

- apply narrowing steps at outermost positions
- evaluate arguments if necessary

$$0 * X \quad \to \quad 0$$

$$0 * (s(0) + s(Z)) \approx 0 \quad \leadsto_{\{\}} \quad 0 \approx 0$$

Permits computations with infinite data structures:

$$\begin{array}{rcl} from(N) & \to & [N|from(s(N))] \\ first(0, L) & \to & [] \\ first(s(N), [E|L]) & \to & [E|first(N, L)] \end{array}$$

$$\begin{array}{ll} first(X, from(Y)) \approx [0] & \\ \leadsto_{\{\}} & first(X, [Y|from(s(Y))]) \approx [0] \\ \leadsto_{\{X \mapsto s(X_1)\}} & [Y|first(X_1, from(s(Y)))] \approx [0] \\ \leadsto_{\{Y \mapsto 0\}} & first(X_1, from(s(Y))) \approx [] \\ \leadsto_{\{X_1 \mapsto 0\}} & [] \approx [] \end{array}$$

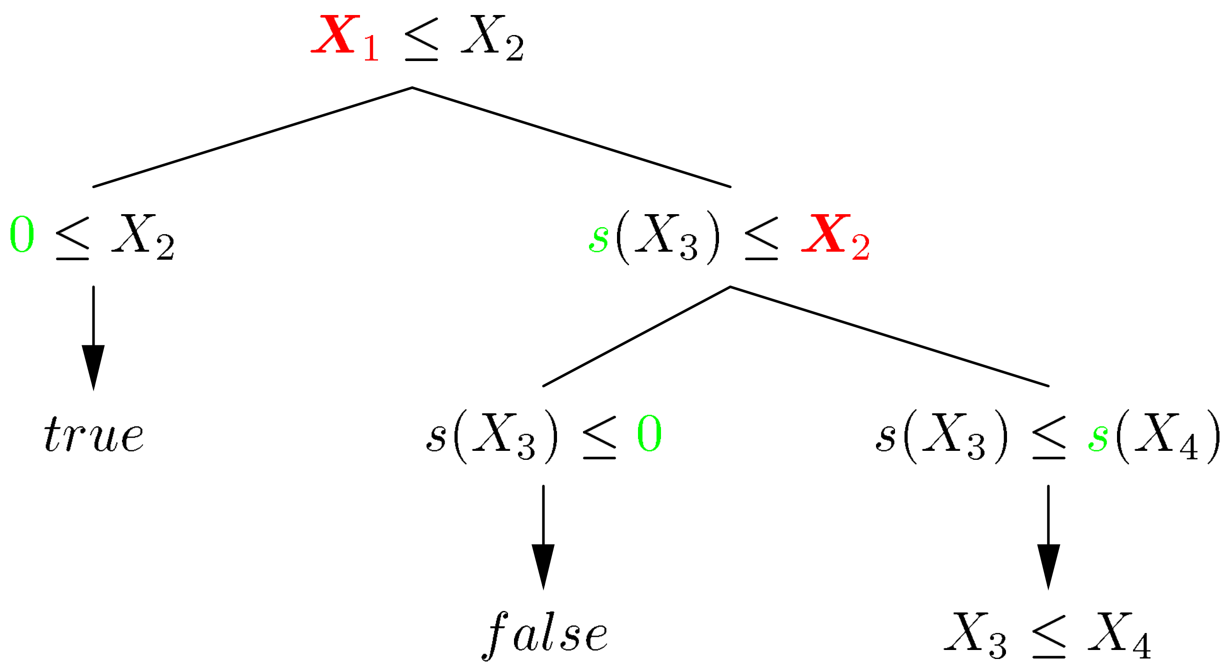
Computed answer: $\{X \mapsto s(0), Y \mapsto 0\}$

Eager strategies run into infinite loop!

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

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



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 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

Residuation vs. Narrowing

Residuation:

+ deterministic evaluation of functions
+ provide concurrent computations
+ simple connection of external functions (arithmetic,...)
– incomplete (no solution if functions not evaluable)
– unclear relation to Prolog's resolution strategy (infinite residuation derivations [Hanus JLP'95])

Narrowing:

+ completeness w.r.t. functional and logic programming
+ clear relation to Prolog's resolution strategy (always better or at least equal)
+ optimal evaluation (for typical programs)
– expensive function inversion (non-determinism)
– infinite loops
– combination with external functions unclear

Curry's Operational Semantics: Combining Residuation and Narrowing

Problem: **Non-determinism**

Residuation:

- only in predicates or disjunctions
- explicit representation by disjunctions (Escher, Oz)

Disadvantage:
loss of information by coding functions as predicates

Narrowing:

- also in nested function calls
- implicit (as in logic programming)

Our approach:

- residuation and narrowing in a single framework
- represent non-determinism by disjunctions
- narrowing steps may generate new disjunctions

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: $\{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$
also written as: $(\sigma_1 \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \,[\!]\, e_n)$

Computation step: reduce an unsolved expression

Deterministic step: replace expression by new expression

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

$$
\boxed{
\begin{array}{rcl}
p(a) & \to & \mathit{true} \\
p(b) & \to & \mathit{true}
\end{array}
}
$$

$$p(a) \;\rightsquigarrow\; \mathit{true}$$

$$p(X) \;\rightsquigarrow\; (X = a \,[\!]\, \mathit{true}) \vee (X = b \,[\!]\, \mathit{true})$$

Definitional Trees

- data structure to specify computation strategy
- each node has a distinct *pattern* π
- kinds of nodes with pattern π:

  $\boxed{rule(\pi \to r)}$ 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 π,
  $r \in \{rigid, flex\}$,
  $c_1, \ldots, c_k$ 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 π

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

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

Examples

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

No unique argument to distinguish the rules:

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

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

$$\begin{array}{rcl} true \wedge X & \to & X \\ X \wedge true & \to & X \end{array}$$

$$\begin{array}{r} and(branch(X_1 \wedge X_2 \overset{rigid}{\Longrightarrow} \boxed{true \wedge X_2 \to X_2}), \\ branch(X_1 \wedge X_2 \overset{rigid}{\Longrightarrow} \boxed{X_1 \wedge true \to X_1})) \end{array}$$

Strategy: To evaluate $t_1 \wedge t_2$, evaluate t_1
If evaluation of t_1 suspends, evaluate t_2

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)

Curry's Computation Strategy: Residuation + Narrowing

Specify exact evaluation strategy by
evaluation annotations ($\sim$ definitional trees):

(can be omitted by using a default strategy)

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

`eval` $\leq$ `1:(s => 2)`

Evaluate 1. argument to head normal form,
if headed by `s`: evaluate 2. argument

`eval` $\leq$ `1:rigid(s => 2)`

Evaluate 1. argument to head normal form,
proceed only if $\neq$ free variable

`eval` $\leq$ `1:rigid(s => 2:rigid)`

Residuation strategy

Programming in Curry

```
function append: list(T), list(T) -> list(T)
append([]    ,Ys) = Ys
append([X|Xs],Ys) = [X|append(Xs,Ys)]
```

Functional programming:

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

Logic programming (branch is *flexible*):

`append(L,M)=[1,2]`
$\rightsquigarrow$ {`L=[],M=[1,2]`} `true`
$\lor$ {`L=[1],M=[2]`} `true`
$\lor$ {`L=[1,2],M=[]`} `true`

```
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))` $\rightsquigarrow$ `[0,s(0)]`

Lazy functional logic programming:

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

Soundness and Completeness

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

Soundness: If

$$e \quad \leadsto^* \quad \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$$

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

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

$$e \quad \leadsto^* \quad \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$$

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 $\exists$

$$e \quad \leadsto^* \quad \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$$

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

Relation to Narrowing

Inductively sequential functions:

Definitional trees without *or*/*and* nodes

All branch nodes *flexible* $\approx$ needed narrowing

$\Longrightarrow$ conservative extension of a complete and optimal evaluation strategy

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

Definitional tree: $branch(X_1 + X_2, 1, flex, \ldots)$

$$
\begin{array}{ll}
& Z + 0 \approx s(0) \\
\rightsquigarrow & (Z = 0 \, [] \, \underbrace{0 \approx s(0)}_{fail}) \vee (Z = s(X) \, [] \, s(X+0) \approx s(0)) \\
\rightsquigarrow & (Z = s(X) \, [] \, s(X+0) \approx s(0)) \\
\rightsquigarrow & (Z = s(X) \, [] \, X + 0 \approx 0) \\
\rightsquigarrow & (Z = s(0) \wedge X = 0 \, [] \, 0 \approx 0) \\
& \vee \; (Z = s(s(Y)) \wedge X = s(Y) \, [] \, \underbrace{s(Y+0) \approx 0}_{fail}) \\
\rightsquigarrow & (Z = s(0) \wedge X = 0 \, [] \, true)
\end{array}
$$

Relation to Residuation

Boolean functions (predicates):
all branch nodes *flexible*

Non-Boolean functions:
all branch nodes *rigid*

$\leadsto$ suspend function calls if variable argument needed

$\Longrightarrow$ operational semantics of Le Fun, Life, Escher

(note: definitional trees more flexible than `NONVAR`)

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

Definitional trees:

$branch(X_1 + X_2, 1, rigid, \ldots)$

$branch(nat(X), 1, flex, \ldots)$

Functional Logic Programming vs. (Concurrent) Logic Programming

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

$$\begin{array}{rcl} from(N) & \rightarrow & [N|from(s(N))] \\ first(0,L) & \rightarrow & [] \\ first(s(N),[E|L]) & \rightarrow & [E|first(N,L)] \end{array}$$

$$\begin{array}{ll} & first(X, from(X)) \approx [] \\ \leadsto & (X = 0 \,[\!]\, [] \approx []) \vee (X = s(0) \,[\!]\, \ldots fail \ldots) \\ \leadsto & (X = 0 \,[\!]\, true) \end{array}$$

$$\begin{array}{l} from(N,[N|L]) \Leftarrow from(s(N),L) \\ first(0,L,[]) \\ first(s(N),[E|L],[E|R]) \Leftarrow first(N,L,R) \end{array}$$

$$first(X,L,[]), from(X,L)$$
$$\leadsto_{\{X \mapsto 0\}} \quad from(0,L) \quad \leadsto \quad from(s(0),L_1) \quad \leadsto \quad \cdots$$

Type and Function Declarations

Curry: strongly typed with parametric polymorphism

Data type declarations:

```
datatype bool = true | false
datatype nat = 0 | s nat
datatype tree A = leaf A
                | node (tree A) A (tree A)
```

Function declarations (types in curried form):

```
function append: [A] -> [A] -> [A]
append []    L = L
append [E|R] L = [E|append R L]
```

- types of functions can be omitted (type inference)
- definitional trees can be omitted:
  automatic generation:
  - left-to-right pattern matching
  - *rigid* branches for functions
  - *flexible* branches for predicates

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  <=  append L1 [E|L2] == L
```

Conditional rules $l \to r \Leftarrow c$:

- 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 :- p`$_1$`,...,p`$_n$ $\mapsto$ `p = true <= p`$_1 \wedge \ldots \wedge$`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

Further Features

- **Monadic I/O**
  - declarative I/O concept
  - I/O: transformation on the outside world
  - only one version accessible at each point
  - no I/O in non-deterministic parts: encapsulate search
- **Encapsulated Search**
  - technique to avoid global search (backtracking) (non-backtrackable I/O, efficiency control,...)
  - select one solution in a disjunction
  - implementation by deep guards and comitted choice
- **Constraints**
  - important feature (CLP applications)
  - interface to constraint solvers
  - later: built-in constraints

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 (without *and* nodes) into Prolog

- Implementation of *and* nodes using coroutining features of Prolog

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

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
- Narrowing: instantiate variable arguments
- Residuation: suspend on variable arguments
- Narrowing $\leftrightarrow$ Residuation: one bit in branch nodes