Curry

A Truly Functional Logic Language

Michael Hanus

Herbert Kuchen

RWTH Aachen

Juan José Moreno Navarro

Universidad Politénica Madrid

Motivation

Problems of current logic languages:

- 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:

- standard functional logic language
- combines most important feature of existing
  - functional languages
  - logic languages
  - functional logic languages
- combine separated research efforts in functional and logic programming
- boost declarative programming
- avoid many impure features of Prolog

Evaluation Principles of Functional Logic Languages

Residuation:

(Escher, Le Fun, Life, NUE-Prolog, Oz,...)
Delay functions calls until deterministic reduction is possible

```
  0+X → X              nat(0)
s(X)+Y → s(X+Y)        nat(s(X)) <= nat(X)
```

$$
\begin{array}{ll}
 & \texttt{X+0 = s(0), nat(X)} \\
\leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{s(X1)+0 = s(0), nat(X1)} \\
\leadsto_{\{\}} & \texttt{s(X1+0) = s(0), nat(X1)} \\
\leadsto_{\{\texttt{X1}\mapsto\texttt{0}\}} & \texttt{s(0+0) = s(0)} \\
\leadsto_{\{\}} & \texttt{s(0) = s(0)}
\end{array}
$$

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

Advantage: Deterministic evaluation of function calls

Evaluation Principles of Functional Logic Languages

Narrowing:

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

Instantiate goal variables and apply a reduction step

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

$$\begin{array}{llll} \texttt{Z+s(0) = s(s(0))} & \leadsto_{\{\texttt{Z}\mapsto\texttt{s(X)}\}} & \texttt{s(X+s(0)) = s(s(0))} \\ & \leadsto_{\{\texttt{X}\mapsto\texttt{0}\}} & \texttt{s(s(0)) = s(s(0))} \end{array}$$

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

Advantage: Ability to solve equations

Disadvantage: huge search space

(nondeterministically select a position *and* a rule)

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

Lazy Narrowing

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

$$\texttt{0*X} \quad \to \quad \texttt{0}$$

$$\texttt{0*(s(0)+s(z)) = 0} \quad \leadsto_{\{\}} \quad \texttt{0 = 0}$$

Permits computations with infinite data structures:

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

$$\begin{array}{ll} \multicolumn{2}{l}{\texttt{first(X,from(Y)) = [0]}} \\ \leadsto_{\{\}} & \texttt{first(X,[Y|from(s(Y))]) = [0]} \\ \leadsto_{\{\texttt{X}\mapsto\texttt{s(X1)}\}} & \texttt{[Y|first(X1,from(s(Y)))] = [0]} \\ \leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}} & \texttt{first(X1,from(s(Y))) = []} \\ \leadsto_{\{\texttt{X1}\mapsto\texttt{0}\}} & \texttt{[] = []} \end{array}$$

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

Eager narrowing strategies run into infinite loop!

Needed Narrowing

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)
– expensive function inversion (nondeterminism)
– infinite loops
– combination with external functions unclear

Curry's Operational Semantics: Residuation + Narrowing

Default strategy satisfies:

1. Existing solutions are computed
2. Existing values are computed

Necessary: sophisticated lazy narrowing strategy

Alternative strategies can be specified by **evaluation restrictions**:

```
function leq: nat -> nat -> bool
leq 0     N     = true
leq (s M) 0     = false
leq (s M) (s N) = leq M N
```

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

Evaluate 1. arg. to hnf, if headed by `s`: evaluate 2. arg.

`eval leq 1:rigid`

Evaluate 1. arg. to hnf, proceed only if $\neq$ free variable

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

Residuation strategy

Sophisticated Lazy Narrowing Strategy

```
one 0     = s 0               f 0 X = 0
one (s X) = one X             f X 0 = 0
```

`f (one X) 0` $\leadsto$ `0`

– lazy narrowing with simplification [Hanus PLILP'95]
– parallel narrowing [Antoy et al. 96]

Equivalent Prolog program has infinite search space:

```
one(0,s(0)).                  f(0,X,0).
one(s(X),Y) :- one(X,Y).      f(X,0,0).
```

`one(X,Y), f(Y,0,Z)` $\leadsto$ infinitely many solutions

Type and Function Declarations

Type system 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]
```

Conditional equations:

- predicates as Boolean functions
- strict equations l==r
  (l, r unifiable constructor terms)
- extra variables in conditions

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

Higher-Order Features

Higher-order functions:

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

Curry allows logical variables for functional values:

`map F [1 2] == [2 3]` $\leadsto$ `F=inc`

Operational principles:

1. higher-order unification
   (restricted since lhs are patterns)
2. quantification over all defined functions
   $\Rightarrow$ translation into first-order logic [Warren 82]
3. delay applications with unknown functions

Lambda abstractions:

capture notion of bound variables

```
diff λX.F = λX.0
diff λX.X = λX.1
diff λX.(sin (F X)) = λX.(cos F X)
                              * diff (λY.F Y) X
```

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:

- find solutions to some goal (Boolean expression)
- goal contains search variable
- bind search variable to different solutions
  $\Rightarrow$ abstract search variable: $\lambda x.g$
- result of search steps:

```
datatype searchspace A =
  failed                   % failed search
| solved (A -> bool)       % successful search
| distributed [A -> bool]  % disjunction
```

Encapsulated Search: Example

```
function solve: (A -> bool) -> searchspace A
```

`solve` $\lambda x.g$: eval. goal, stop after nondeterministic step

Depth-first search: compute leftmost solution

```
function dfs: [A -> bool] -> searchspace A
dfs []      =  failed
dfs [X|Xs] =
     case  solve X  of
        failed:      dfs Xs
        solved Y:    solved Y
        distributed [Z|Zs]:
           case  dfs [Z]  of
              solved V:  solved V
              failed:    dfs (append Zs Xs)
```

dfs [λL.(append L [1] == [0,1])]
$\leadsto$ solved λL.(L==[0])

Further search strategies:

- breadth-first search
- collecting all solutions in a list
- $\cdots$

Further Features

- **Monadic I/O**
  - I/O: transformation on the outside world
  - only one version accessible at each point
  - no I/O in nondeterministic parts:
    encapsulate search
- **Constraints**
  - necessary feature (CLP applications)
  - combination with narrowing strategies unclear
  - currently only disequality constraints
    [Kuchen et al. JICSLP'92]
- **Implication and Quantifiers**
  - permit implications $rules$ `=>` e and quantifiers
  - declarative alternative to `assert` and `retract`
  - provide for scoping constructs
- **Modules**
  - standard module system (ALF, Gödel,...)

Implementation

- No implementation available

- Design influenced by current implementation techniques:

  – constructor-based rules

  – higher-order patterns

  – evaluation restrictions

- Efficient implementation possible by combining known implementation techniques

  $\rightarrow$ ALF, Babel, λProlog, …

Conclusions

Curry: a truly integration of declarative paradigms

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

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

Logic programms $\hookrightarrow$ Curry:

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

Evaluation restriction of ",": `1:(true => 2)`

Curry avoids impure Prolog constructs:

- *cut*: deterministic evaluation of functions
- `call`: higher-order features
- `assert/retract`: implications
- `read/write`: monadic I/O

Curry: a declarative successor of Prolog?