Can Logic Programming Be Liberated from Predicates and Backtracking?

Michael Hanus

Kiel University

Programming Languages and Compiler Construction

Bad Honnef 4/2025

Logic Programming

The ideal view

- write problem specification with Horn clauses
- use SLD-resolution to compute problem solutions

The practice: Prolog

- use backtracking (due to memory limitations in the '70s)
- loss of completeness

Prolog program

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

app3(Xs,Ys,Zs,Ts) :- app(Xs,Ys,Rs), app(Rs,Zs,Ts)

?- app3(Xs,Ys,Zs,[]).   ⇝ Xs=[], Ys=[], Zs=[]  ; no termination!
?- app3(Xs,[1],Zs,[]).  ⇝ no termination!
```

Predicates vs. Functions

Functional logic programming

- program: set of functions defined by equations
- nice compact notation *and* exploit functional dependencies
- Curry (`www.curry-lang.org`, extension of Haskell):
  - demand-driven reduction of function calls ($\leadsto$ FP)
  - non-deterministic rule application ($\leadsto$ LP)

Curry program

```
app []     ys = ys
app (x:xs) ys = x : app xs ys

app3 xs ys zs = app (app xs ys) zs
```

```
> app3 xs ys zs =:= []  ⇝ xs=[], ys=[], zs=[]   (finite evaluation!)
> app3 xs [1] zs =:= [] ⇝ no result             (finite evaluation!)
```

From Logic to Functional Logic Programs

Approach: Functional transformation [TPLP 2022]

- fix some predicate argument(s) as result(s) (default: last argument)
- map n-ary predicates into m-ary functions ($m \leq n$)
- source and target programs are operationally equivalent

Prolog program

```
app([],Ys,Ys).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

Curry program

```
app []     ys                        = ys
app (x:xs) ys | zs =:= app xs ys = x : zs  where zs free
```

From Logic to Functional Logic Programs

Approach: Demand functional transformation

- as before, but use binding (`let`/`where`) instead of unification
- inline bindings for compact notation

Curry program

```
app []     ys = ys
app (x:xs) ys = let zs = app xs ys in x : zs where zs free
```

Evaluation strategy becomes relevant:

- strict (call-by-value): source/target computations are equivalent
- non-strict (lazy): some bindings not demanded $\rightsquigarrow$ fewer steps

Demand Functional Transformation

```
app []     ys = ys                    siglist []       = Zero
app (x:xs) ys = x : app xs ys         siglist [_]      = One
                                      siglist (_:_:_)  = Many
> siglist (app xs ys)
```

strict: $(\mathit{length}(\mathit{xs}) + 2)$ steps vs. lazy: ≤ 3 steps

Problem: lazy strategy ignores non-demanded failures

```
tail([_|Xs],Xs).
sigtail(S) :- tail([],Xs), app([0,1],Xs,Ys), siglist(Ys,S).
```

`?- sigtail(S).` $\rightsquigarrow$ No solution

Demand functional transformation:

```
tail (_:xs) = xs
sigtail = siglist (app [0,1] (tail []))
```

`> sigtail` $\rightsquigarrow$ `Many` !!!

Fix the Demand Functional Transformation

Demand functional transformation correct if all operations are non-failing (totally defined)

$\rightsquigarrow$ force evaluation of possibly failing expressions:

(f \$! e) $\approx$ strict evaluation of e

```
tail (_:xs) = xs  -- not totally defined!
sigtail = siglist (app [0,1] (tail []))
```

$\rightsquigarrow$

```
sigtail = siglist $! (app [0,1] $! (tail []))
```

Fail-sensitive functional transformation:
ensures semantic equivalence of logic and functional logic programs
(laziness $\rightsquigarrow$ possible more general solutions computed)

Abandon Predicates: From LP to FLP

Systematic transformation method

- Details: [TPLP 2022, LOPSTR 2024]
- Automatic transformation tool: `pl2curry`
  - *Input:* (almost pure) Prolog program
  - *Output:* Curry program

`https://cpm.curry-lang.org/pkgs/prolog2curry.html` (source)
`https://hub.docker.com/r/currylang/prolog2curry` (docker)
`https://cpm.curry-lang.org/webapps/pl2curry/` (webapp)

Advantages [LOPSTR 2024]

- semantic equivalence
- worst case: same number of evaluation steps
- general cases:
  - less evaluation steps (if some subexpressions not demanded)
  - reduce infinite search spaces to finite ones

Evaluation: Benchmarks

Benchmarks (run-time in seconds)

Language: System:	Prolog SWI 9.0.4	Prolog SICStus 4.9.0	Curry KiCS2 3.1.0
`rev_4096`	0.23	0.22	0.10
`tak_27_16_8`	6.97	3.23	0.74
`ackermann_3_9`	2.13	8.72	0.07
`pali_[]`	∞	∞	0.01
`siglist_app_0`	∞	∞	0.01
`numleaves_7`	∞	∞	0.01
`sublist_1_2`	∞	∞	0.01
`permsort_10`	1.43	0.28	0.03
`permsort_11`	16.16	1.38	0.08
`permsort_12`	206.34	15.23	0.28

- `rev_4096`, `tak_27_16_8`, `ackermann_3_9`: same number of steps
- `permsort_`*n*: demand-driven exploration of search space

Abandon Backtracking: Complete Search Strategies

Strategies for non-deterministic search

Prolog:

- backtracking (due to limited hardware resources)
- not easy to change: many non-logical features rely on backtracking

Curry:

- no fixed search strategy
- PAKCS ($\rightsquigarrow$ Prolog): backtracking
- KiCS2 ($\rightsquigarrow$ Haskell): depth-first (DFS), breadth-first (BFS)
- Curry2Go ($\rightsquigarrow$ Go): DFS, BFS, fair search (FS) via *goroutines*

Non-deterministic identity

```
idND n = loop ? n ? loop   -- "?": non-deterministic choice
> idND True
```

DFS, BFS: loops (no choice in `loop`), FS: returns `True`

Evaluating Search Strategies

Example	PAKCS SWIPL	KiCS2 DFS	KiCS2 BFS	Curry2Go DFS	Curry2Go BFS	Curry2Go FS
`nrev_4096`	6.29	0.10	0.10	0.85	0.85	0.85
`takPeano_24_16_8`	56.78	0.12	0.12	8.05	7.98	7.76
`primesHO_1000`	29.46	0.04	0.04	3.51	3.58	3.55
`psort_13`	18.92	0.35	2.32	7.11	7.25	9.51
`addNum_2`	0.18	0.24	0.57	0.28	0.29	0.28
`addNum_5`	0.20	2.01	4.36	0.67	0.67	0.35
`addNum_10`	0.24	11.83	16.84	1.53	1.54	0.54
`select_50`	0.09	0.19	0.27	0.02	0.02	0.02
`select_100`	0.27	4.13	4.80	0.06	0.06	0.03
`select_150`	0.56	25.10	32.42	0.13	0.13	0.06
`isort_primes4`	9.56	0.02	0.02	1.15	1.14	1.11
`psort_primes4`	112.38	0.02	0.02	1.11	1.11	0.71

`primes4` $\hat{=}$ `[primes!!303, primes!!302, primes!!301, primes!!300]`

$\Rightarrow$ non-deterministic algorithm faster due to concurrency

Conclusions

Can LP Be Liberated from Predicates and Backtracking?

YES!

- functions: only advantages, no disadvantages!
  - transformed programs compute same or more general answers
  - worst case: same number of evaluation steps
  - general: reduced number of steps, infinite $\rightsquigarrow$ finite search spaces
  - optimal evaluation for inductively sequential programs
- complete strategies are not slow, we have enough memory/processors!

Advantages

- modern language concepts: functions, nested expressions
- avoid incompleteness: close theory/practice gap of LP
- easier teaching of *declarative programming*
- avoid non-declarative features (cut, `is`, I/O side effects,...)

$\Rightarrow$ **keep LP ideas in future programming systems!**