Can Logic Programming Be Liberated from Backtracking?

– Extended Abstract –

Michael Hanus
mh@informatik.uni-kiel.de
Institut für Informatik, CAU Kiel, 24098 Kiel, Germany

Abstract
Logic programming is historically tight with Prolog and its backtracking search strategy. The use of backtracking was justified by efficiency reasons when Prolog was invented and is still present, although the incompleteness of backtracking destroys the elegant connection of logic programming and the underlying Horn clause logic. Moreover, it causes difficulties to teach logic programming. In this paper we argue that this is no longer necessary if new implementation approaches are taken into account.

1 Introduction

Logic programming was developed as a restriction of the general resolution principle [Robinson, 1965] to Horn clauses so that efficient linear (SLD-resolution) proofs can be constructed (see also Cohen [1988] for some historical background). It became popular when concrete implementations in the form of interpreters (and later compilers) for the programming language Prolog were available. Horn clauses and SLD-resolution are tightly connected to mathematical logic. The soundness and completeness of SLD-resolution establish the foundation of logic programming [Lloyd, 1987]. Unfortunately, the memory restrictions of computers at that time caused a gap between these theoretical foundations and the practice of logic programming in Prolog: non-deterministic computations are implemented by backtracking so that the theoretical completeness of SLD-resolution is lost. For instance, consider the definition of a Prolog predicate relating a list and its last element:

```
last([H|T],E) :- last(T,E).
last([E],E).
```

This definition works when the list is known:

```
?- last([1,2,3],E).
E = 3
```

One of the advantages of logic programming is the absence of fixed input and output parameters. Instead of providing a known value for an argument of a predicate, one can also call the predicate a free variable for this argument (as `E` above) so that a result is computed by binding this variable to some value. In practice, this advantage is often lost when non-deterministic search is implemented by backtracking, since infinite branches in a search tree might preclude the computation of valid answers. For instance, Prolog does not compute any result for the definition of `last`, as shown above, when the list is unknown, e.g., for the goal `last(L,3)`: the backtracking strategy causes an infinite chain of applications of the first rule.

Generally, the use of backtracking in logic programming has several disadvantages:

- The theoretical completeness of SLD-resolution is lost.
- It hinders the teaching of logic programming since beginners are often faced with the influence of the search strategy.
- Programmers have to think about the influence of backtracking to the success of computations—a contradiction to the idea of *declarative programming*.

These problems can be solved if backtracking is replaced by a complete search strategy. Thus, abandoning backtracking as a default for logic programming is similar to the removal of the von Neumann bottleneck by functional programming [Backus, 1978]: one obtains a higher, declarative programming style which frees the programmer from thinking about low-level control details.

Unfortunately, many aspects of Prolog, in particular, the connection to the external world (e.g., file system, networks, graphics) heavily depends on the backtracking strategy. Thus, in order to get a better logic programming language, we have to switch from Prolog to a paradigm supporting a clean and declarative connection to external resources, as developed in functional programming [Wadler, 1997]. Therefore, we consider in the following *functional logic* languages.

2 Functional Logic Programming

Functional logic languages [Antoy and Hanus, 2010] combine the main features of functional and logic languages in a single programming model. In particular, demand-driven evaluation of expressions is amalgamated with non-deterministic

search for values. In functional logic programs, operations are defined by rewrite rules, as in functional languages, but rules can be overlapping, as in logic languages. The archetype of an operation defined by overlapping rules is the non-deterministic *choice*, defined in the functional logic language Curry [Hanus (ed.), 2016] as the infix operator "`?`" as follows:

```
x ? _ = x
_ ? y = y
```

Hence, an expression like "`0 ? 1`" yields two values: `0` and `1`. In contrast to Prolog, the concrete strategy to compute these values, i.e., the search strategy, is not fixed in Curry. Implementations of Curry can support various search strategies. For instance, the Curry system KiCS2 [Braßel, Hanus, et al., 2011] has options to select different search strategies, like depth-first, breadth-first, iterative deepening, or parallel search.

Early implementations of functional logic languages, like PAKCS [Antoy and Hanus, 2000] or TOY [López-Fraguas and Sánchez-Hernández, 1999], used Prolog as a target language due to its built-in support for non-determinism. A drawback of this approach is that they inherit the incompleteness of Prolog's backtracking strategy. In order to get rid of this fixed search strategy, subsequent implementations are based on the idea to represent non-deterministic choices as data. Thus, instead of directly evaluating non-deterministic branches, the alternatives are returned as a tree structure so that search strategies can be defined as tree traversals, which supports an easy switch between different strategies.

We omit the details of actual techniques to evaluate expressions to such tree structures. It is sufficient to keep in mind that the operational semantics is based on graph transformations, like pull-tabbing [Antoy, 2011], but the details are not relevant for the programmer as long as a complete search strategy is used. In general, breadth-first search could be used. However, if the search space is finite, depth-first search is also reasonable.

3 Comparing Search Strategies

These theoretical considerations are useless if they are not supported by practical implementations. Instead of compiling functional logic languages into Prolog, one can compile them into a deterministic target language and add explicit support for non-determinism, as done with KiCS2 [Braßel, Hanus, et al., 2011] which compiles Curry programs into Haskell programs. The intermediate language ICurry [Antoy, Hanus, et al., 2020] is intended to compile Curry into imperative

target languages. It has been used to translate Curry to LLVM code [Antoy and Jost, 2016], to C or Python programs [Wittorf, 2018], and to Julia programs [Hanus and Teegen, 2021]. A recent implementation, called Curry2Go[1], compiles ICurry programs into Go[2] programs. Go is a statically typed language with garbage collection and direct support for CSP-like concurrency [Hoare, 1978] and lightweight threads (*goroutines*). The latter feature is used by Curry2Go to provide a fair search strategy which avoids the limitations of backtracking-based logic programming languages discussed above.

Due to the explicit handling of non-deterministic computations, Curry2Go supports various search strategies. The run-time system works with a queue or set of tasks (depending on the search strategy) where each task evaluates some non-deterministic branch. The difference between depth-first (DFS) and breadth-first (BFS) search amounts to a different strategy to add new tasks to the queue: DFS adds new tasks at the front and BFS adds them at the tail of the queue.

Each task evaluates an expression to some value (to be more precise, a head normal form) or is split into two new tasks if some non-deterministic choice occurs. If the evaluation of an expression does not terminate and non-deterministic choices do not occur, even a breadth-first search strategy might not compute existing values. For instance, consider the following contrived example:

```
idND :: a → a
idND n = loop ? n ? loop
```

where `loop` is non-terminating (e.g., defined by `loop = loop`). Semantically, `idND` is the identity function but, operationally, it is non-deterministically defined with looping alternatives. Although `0` is a value of `idND 0`, both DFS and BFS do not return any value but loop. To avoid such kind of incompleteness, Curry2Go also implements a *fair search* (FS) strategy. FS evaluates each task concurrently as a goroutine and collects the computed result in a channel where these goroutines write their computed results. More details about this implementation can be found in Böhm, Hanus, and Teegen [2021].

Table 1 shows the run times[3] (in seconds as the average of three runs) of some examples executed with different Curry systems and search strategies. PAKCS [Hanus, Antoy, et al., 2020], which is part of Debian and Ubuntu Linux distributions, compiles to Prolog (SWI-Prolog 8.0) and is based on backtrack-

[1] The source code is available at https://github.com/curry-language/curry2go. A distribution can be downloaded at https://www-ps.informatik.uni-kiel.de/curry2go/.

[2] https://golang.org/

[3] All benchmarks were executed on a Linux machine running running Debian 10 with an Intel Core i7-7700K (4.2GHz) processor with eight cores.

Table 1. Comparing Curry system with search strategies

	PAKCS	KiCS2		Curry2Go		
Example		DFS	BFS	DFS	BFS	FS
`nrev_4096`	8.28	0.43	0.44	1.16	1.17	1.16
`takPeano_24_16_8`	54.75	0.30	0.30	5.08	5.09	5.08
`primesHO_1000`	38.88	0.43	0.44	4.08	4.09	4.08
`psort_13`	16.46	0.77	2.88	5.20	5.27	5.43
`addNum_2`	0.19	0.98	1.77	0.44	0.43	0.41
`addNum_5`	0.22	3.21	5.18	1.06	1.06	0.45
`addNum_10`	0.29	10.03	15.55	2.48	2.48	0.69
`select_50`	0.14	0.61	0.67	0.11	0.11	0.08
`select_100`	0.45	4.97	5.25	0.14	0.14	0.10
`select_150`	1.08	21.25	26.14	0.23	0.23	0.12
`isort_primes4`	15.63	0.42	0.42	1.74	1.74	1.72
`psort_primes4`	155.95	0.40	0.42	1.72	1.72	0.94

ing. KICS2 [Braßel, Hanus, et al., 2011] compiles to Haskell (GHC 8.4) where non-determinism is implemented by lazily generating search trees which are explored by various search strategies. Curry2Go compiles to Go (Version 1.16) and manages a queue of tasks to implement DFS and BFS or use goroutines communicating via channels to implement FS.

The first three benchmarks are typical purely functional programs. `nrev_4096` is the quadratic naive reverse algorithm applied to a list with 4096 elements, `takPeano` is a highly recursive function on naturals [Partain, 1993] applied to arguments (24,16,8) in Peano representation, and `primesHO_1000` computes the 1000th prime number by constructing an infinite list of all primes via the sieve of Eratosthenes (using higher-order functions). These benchmarks indicate that, for purely functional programs, Curry2Go is much faster than PAKCS but less efficient than KiCS2. The latter is not surprising since Haskell/GHC is highly optimized for these kinds of programs.

One might think that the less efficient behavior of Prolog-based PAKCS is the fact that Prolog also supports non-determinism. This hypothesis is refuted by the remaining non-deterministic benchmark programs. `psort_13` is the naive permutation sort applied to a list of 13 elements. `addNum_`n non-deterministically chooses a number (out of 2000) and adds it n times, and `select_`n non-deterministically selects an element in a list of length n and sums up the element and the list without the selected element. The considerable slowdown in KiCS2 with increas-

ing values for n is caused by the duplication of choices in pull-tab steps when non-deterministic expressions are shared, as discussed in Hanus and Teegen [2021]. This is avoided in Curry2Go by adding a sort of memoization for choices, as described in Böhm, Hanus, and Teegen [2021]; Hanus and Teegen [2021].

Apart from the fact that the fair search strategy of Curry2Go is the only operationally complete strategy (e.g., it is able to compute a value for `idND 0`), there are also other interesting differences between the search strategies. For instance, KiCS2 shows some overhead of BFS compared to DFS (possibly due to the additional structures used to implement a breadth-first tree search), whereas there is almost no overhead in Curry2Go (since the difference between BFS and DFS is just a different schedule of tasks). Moreover, the fair search (FS) strategy is sometimes faster than BFS and DFS thanks to the use of goroutines possibly scheduled on different processors. This is also visible in the last two lines of Table 1 which show the time to sort

```
[primes!!303, primes!!302, primes!!301, primes!!300]
```

with the deterministic insertion sort (`isort`) and the non-deterministic permutation sort (`psort`) algorithm, respectively, where `primes` defines the infinite list of all prime numbers. Due to backtracking, identical computations might be repeated if they occur in different non-deterministic branches. Thus, `primes` is re-evaluated by PAKCS several times when the list is passed to the non-deterministic operation `psort`. This is not the case in implementations which represent choices in a graph structure so that the results of deterministic computations are shared across non-deterministic evaluations [Braßel and Huch, 2007]. In Curry2Go, where non-deterministic branches are evaluated by goroutines, it could be even better to use a non-deterministic algorithm since it might map evaluations of common subexpressions to different computation nodes, as shown by the results for `psort_primes4`. This is also demonstrated with the following benchmark, where the timings for `psort_primes4` increased to a list of eight prime numbers and executed with different numbers of processors (by setting the Go variable `GOMAXPROCS`) are shown:

# processors	1	2	4	8
`psort_primes8`	6.57	3.41	1.99	1.55

Hence, the presence of multiple processors can be exploited in a non-deterministic program without requiring specific user annotations.

4 Conclusions

We have compared the implementation of different search strategies for non-deterministic programming, like depth-first, breadth-first and fair (concurrent) search. Due to memory restrictions in early years, depth-first search implemented by backtracking was introduced and is still used in the logic programming language Prolog. Backtracking causes a gap between the theory and practice of logic programming and complicates teaching and the practical use of logic programming techniques. By using recent implementation techniques developed in functional logic programming, operationally complete strategies can compete with backtracking and can even be faster on multi-processor architectures. Hence, logic programming must not be tight to backtracking: with modern implementation technologies, one can use better strategies that avoid the classical drawbacks of backtracking, namely the operational incompleteness of search. This closes the gap between theory and practice of logic programming and could lead to a higher, really declarative programming style.

References

Antoy, S. (2011). "On the correctness of pull-tabbing". In: *Theory and Practice of Logic Programming* 11.4-5, pp. 713–730. DOI: 10.1017/S1471068411000263.

Antoy, S. and M. Hanus (2000). "Compiling multi-paradigm declarative programs into Prolog". In: *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*. Springer LNCS 1794, pp. 171–185. DOI: 10.1007/10720084_12.

– (2010). "Functional logic programming". In: *Communications of the ACM* 53.4, pp. 74–85. DOI: 10.1145/1721654.1721675.

Antoy, S., M. Hanus, et al. (2020). "ICurry". In: *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, pp. 286–307. DOI: 10.1007/978-3-030-46714-2_18.

Antoy, S. and A. Jost (2016). "A new functional-logic compiler for Curry: Sprite". In: *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, pp. 97–113. DOI: 10.1007/978-3-319-63139-4_6.

Backus, J. (1978). "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs". In: *Comm. of the ACM* 21.8, pp. 613–641.

Böhm, J., M. Hanus, and F. Teegen (2021). "From non-determinism to goroutines: a fair implementation of Curry in Go". In: *Proc. of the 23rd International Sympo-*

sium on Principles and Practice of Declarative Programming (PPDP 2021). ACM Press.

Braßel, B., M. Hanus, et al. (2011). "KiCS2: a new compiler from Curry to Haskell". In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, pp. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

Braßel, B. and F. Huch (2007). "On a tighter integration of functional and logic programming". In: *Proc. APLAS 2007*. Springer LNCS 4807, pp. 122–138. DOI: 10.1007/978-3-540-76637-7_9.

Cohen, J. (1988). "A view of the origins and development of Prolog". In: *Communications of the ACM* 31.1, pp. 26–36. DOI: 10.1145/35043.35045.

Hanus, M., S. Antoy, et al. (2020). *PAKCS: the Portland Aachen Kiel Curry System*. Available at http://www.informatik.uni-kiel.de/~pakcs/.

Hanus, M. and F. Teegen (2021). "Memoized pull-tabbing for functional logic programming". In: *Proc. of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020)*. Springer LNCS 12560, pp. 57–73. DOI: 10.1007/978-3-030-75333-7_4.

Hanus (ed.), M. (2016). *Curry: an integrated functional logic language (vers. 0.9.0)*. Available at http://www.curry-lang.org.

Hoare, C.A.R. (1978). "Communicating sequential processes". In: *Communications of the ACM* 21.8, pp. 666–677. DOI: 10.1145/359576.359585.

Lloyd, J.W. (1987). *Foundations of logic programming*. Springer, second, extended edition.

López-Fraguas, F. and J. Sánchez-Hernández (1999). "TOY: a multiparadigm declarative system". In: *Proc. of RTA'99*. Springer LNCS 1631, pp. 244–247.

Partain, W. (1993). "The nofib benchmark suite of Haskell programs". In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer, pp. 195–202.

Robinson, J.A. (1965). "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM* 12.1, pp. 23–41.

Wadler, P. (1997). "How to declare an imperative". In: *ACM Computing Surveys* 29.3, pp. 240–263.

Wittorf, M.A. (2018). "Generic translation of Curry programs into imperative programs (in German)". MA thesis. Kiel University.