Sharing of side-effects under nondeterminism in Prolog

Mario Wenzel[1], Jonas Höfer[2]

Abstract: In this paper we explore a novel semantics of logic programming with side-effects under nondeterminism.

While Prolog executes every side-effect whenever a sub-goal entailing side-effects is encountered, regardless of nondeterminism, Curry disallows side-effects under nondeterminism completely.

We propose a semantics of sharing of effects between separate branches of nondeterministic computations in a way that both allows side-effects under nondeterminism but also isolates alternate computations from side-effects of other branches.

Keywords: Prolog; Side-Effects; continuations

1 Introduction

Nondeterminism is a useful abstraction in programming. It usually allows us to traverse alternate code paths without explicitly implementing a traversal strategy. These abstractions allow us to state problems in a declarative manner and the implementation, maybe with some hints on the search strategy, explores the search space and gives us one or many solutions that fit our initial parameters, if any are found.

Prolog, for example, is part of almost every computer science curriculum and has been associated with symbolic AI and planning for decades. Many other programming languages, like Curry [Ha97], and logic programming models, like Answer Set Programming [Li19], also employ a model of nondeterminism to answer queries or implement decision procedures.

Often times in Prolog programming the order in which the clauses defining a predicate occurs in the program and the order of the goals in the body of a rule is of vital importance when evaluating a user's query. It is said that an important part of the philosophy of logic programming is, that programs should be written to minimize the effect of these two factors as far as possible [Br05]. We call programs that do so fully or partly declarative.

[1] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, 06099 Halle (Saale), Germany
mario.wenzel@informatik.uni-halle.de
[2] University of Gothenburg , Department of Computer Science and Engineering,
Rännvägen 6B, 41258 Göteborg, Sweden
jonas.hofer@gu.se

cc by sa

Fully declarative programs are easier to reason about, as the order of the internal operations and implementation details have no bearing on the results the system gives in response to a query.

It is considered good Prolog programming style to make programs as declarative as possible. This can greatly reduce the likelihood of making errors that are hard to detect, particularly when backtracking is used [Br05].

Besides commutativity of the clauses and subgoals, another desirable property of a Prolog program is "separability", meaning that clauses (and predicates) can be read in isolation. Wrong usage of the cut, for example, may lead to predicates not being as general as possible or "missing" solutions. Additionally, one must know which alternatives may have been "cut" and are not relevant for this specific clause.

There are methods and best practices to write Prolog programs that are as declarative as possible. For example, using the Constraint Logic Programming modules it is often possible to avoid cuts and negations by using constraints, and grounding or instantiating values as late as possible.

Another important advice is to avoid predicates that entail side-effects, like `asserta/1` or `retract/1`. While it might be possible to avoid dynamic changes to the *internal* database of our program, or even track and backtrack those changes, this is not possible when we write programs explicitly for their side-effects. We want to read and write data, interactively react to external requests, or run sub-programs and start other processes.

Rather than just return a model of a logic program, logic programming languages like Prolog and Curry allow to encode side-effects, like reading files, prompt the user, or write to sockets. This allows us to write complex and interactive programs in those programming languages, employing nondeterminism for our search strategies and decision procedures. Issues start to arise when we have input and output under nondeterminism. The semantics heavily depends on the specifics of the employed search strategy, or even implementation details of the interpreter or compiler. More problematic, we often suspend our mental nondeterministic model of the program and produce rather imperative code.

In Prolog side-effects happen whenever a sub-goal corresponding to an IO-action is encountered as the next proof goal. The side-effect is executed and we go on to prove the next sub-goal. If we now need to backtrack over this IO-action, the side-effect is not taken back, as this is generally impossible due to the flow of time.

In Curry, on the other hand, if a sub-goal with side-effects is encountered under nondeterministic execution, we get an error. Depending on the interpreter used, the side-effects still happen (PAKCS) or the program terminates with the stated error (KiCS).

Both are valid strategies of handling side-effects. Prolog's strategy is simple but expressive though the programmer has to be aware of the evaluation semantics. The Curry approach,

while consistent, may be seen as too restrictive. At least one branch containing IO operations may be explored and the resulting side-effects of this branch would be consistent with a deterministic execution of this path. The Prolog approach may be seen as not restrictive enough, as every side effect in every branch happens, irrespective of computations and side-effects in other branches, that may already have irrecoverably destroyed the environment our branch expected. Therefore, alternate choices cannot be viewed independently of each other.

Another difficulty is that side-effects in Prolog are extra-logical in the sense that they are not modelled. They "just happen", when entering a subgoal that entails the side-effect. And while on a logical level the subgoals of a query such as `"write(a), write(b)."` are commutative, it is quite clear, that we can not reasonably expect commutativity from the side-effects. The effects happened and are ordered in time. Switching the side-effect-entailing predicates in the query changes the order of side-effects happening. Though for a single side-effect it is not clear why anything should change when commuting an effectful predicate with a side-effect free one. But consider the query `"(X=1;X=2), write(a)."` that prints either one or two *a*s, depending on whether `write` appears before or after the alternative unifications.

In this paper we explore a different approach to handling side-effects under nondeterminism. We mark IO-operations and suspend computation on all alternate branches when a goal with side-effects is encountered. Once all branches have either returned a valid solution (e. g., a variable binding) or reached a suspended goal, we use a selection predicate and execute some or all of the suspended side-effects.

This allows us to share side-effect between branches, if they evaluate the same IO-operation. Furthermore, it allows us to fail branches with different IO actions than the one we chose. This ensures a consistent sequence of IO actions for the branches. This approach effectively isolates the branches against effects of other branches of the computation. We may use different procedures to select the IO action we allow to happen. We implemented a leftmost strategy similar to SLD resolution, where the leftmost IO operation is always chosen and shared with all branches with compatible IO. But we also implemented a consensus-based selection method, where the IO operation, that most branches expected to happen, is executed. Finally, we also recover the original Prolog semantics within our generic framework.

In Section 2 we will recapitulate SLD-resolution with a special focus on side-effects and their visibilty throughout the resolution process. In Section 3 we lay out our core idea of sharing compatible side-effects between alternate branches of our resolution procedure. In Section 4 we show the broadness of our approach through several examples, and conclude with some open questions in Section 5.

2 Resolution-Trees and Side-Effects

The derivations of a Prolog program may be represented by a possibly infinite tree called SLD-Tree. We recapitulate an example from the literature [NM90] for the SLD-Tree of $\leftarrow$ grandfather(a,X) (Figure 1) of the program

(i) grandfather(X,Z) $\leftarrow$ father(X,Y) $\wedge$ parent(Y,Z).
(ii) parent(X,Y) $\leftarrow$ father(X,Y).
(iii) parent(X,Y) $\leftarrow$ mother(X,Y).
(iv) father(a,b).
(v) mother(b,c).

Nodes represent remaining proof goals. Alternate child branches for the same proof goal are alternate rule applications (i. e., choice points). A □ node means, that there exists an answer substitution for a successful SLD-resolution along this path of rule applications. Prolog systems use the ordering of the clauses to impose an ordering on the edges descending from a node of the SLD-Tree, i. e., leftmost alternatives in the SLD-Tree correspond to topmost clauses in the program. In Figure 2 we take a more abstract view of an SLD-Tree, by deleting all □ nodes (ignoring whether this particular subtree yields an answer or not) and replacing all other nodes with ○, ignoring the particulars of each proof goal.

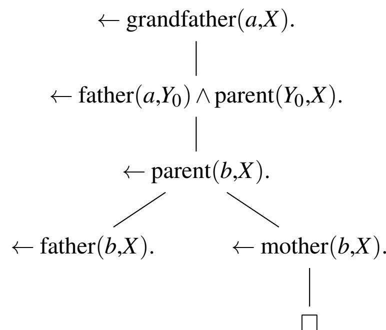

Fig. 1: SLD-Tree of $\leftarrow$ grandfather(a,X)

When all nodes correspond to predicates that are free of side-effects, the order of resolution steps is generally unobservable. The only information we receive as a user is the order of answer substitutions (or solutions). When we consider nodes that have side-effects attached, the side effect happens when the node is entered during tree traversal, the side effect may be observable from that point on forward.

Let us consider the predicate `read/1` that reads a term from the standard input of the program, with the side-effect that this term is now gone from the input stream. We mark the node ● with the first subgoal read(X) in the SLD-Tree in Figure 3. The nodes where

X is now bound and the return value of the operation with side-effects are visible are the descendents marked ●. From this point on, the side-effect is observable in a non-pure manner for all derivations that come after, as marked by the solid line denoting time. All nodes along the solid line will see the input stream altered. Any changes are visible in neighbouring branches, which are effectively "the other clauses" for some predicates. We clearly violate our separability principle.

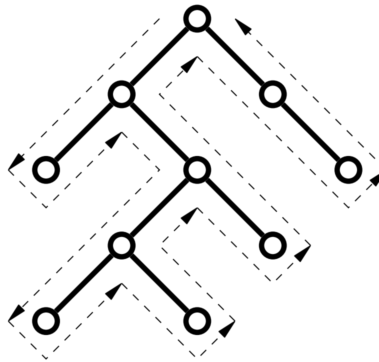

Fig. 2: Depth-first search with backtracking through SLD-Tree [NM90]

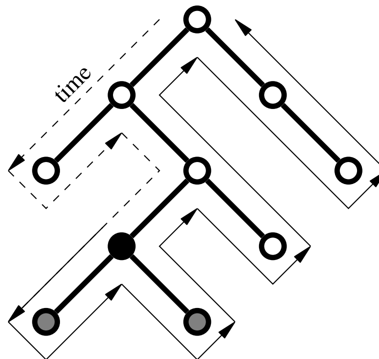

Fig. 3: Depth-first search with backtracking through SLD-Tree with side-effects

If the two nodes marked earlier have no solution, the side-effect is visible even though no solution "under it" is. This means a caller may observe side-effects, even though their call "failed" and did not yield solutions.

Let us again consider the query `(X=1;X=2), write(a).` from the introduction. By having the side-effect after the alternatives, instead of before, we basically push down a side-effect node in the tree and duplicate it, replicating the "problem" for every alternative.

3 Shared World State for Alternate Derivations

During SLD-resolution on all paths from the root of the resolution tree to either TRUE or FAIL-nodes, a sequence of IO actions happen. An IO action happening changes the state of the system and can not be taken back.

Definition 1 (Input Argument)**.** An input argument is an argument denoting the parameters of an IO action. At call time it must be instantiated to a term. ■

Definition 2 (Output Arguments)**.** An output argument is an argument denoting the result of an IO action. At call time it may or may not be bound. If the argument is bound at call time, the goal behaves as if the argument were unbound, and then unified with that term after the goal succeeds. ■

Definition 3 (IO predicate)**.** A predicate that entails IO. ■

Definition 4 (IO action)**.** An occurrence of an IO predicate within the goal list of an SLD-resolution step. ■

Definition 5 (Compatibile IO actions)**.** A pair of IO actions is compatible iff they have the same predicate and their input arguments can be unified. ■

- Let some resolution step A_0 that entails IO leads from a system state (or environment) ε_0 to ε_1. During the backtracking operation, the side effects that lead from ε_0 to ε_1 can not be taken back.
- Let the resolution step A_1 be the alternative to A_0, as it is another clause for the first subgoal, i. e., A_0 may be a second rule that has a head that unifies with the selected resolution literal.
- Then A_1 (given it also entails IO) originally was to lead from system state ε_0 to ε_1', but now, as ε_0 is irrevocably gone, has to work with the environment ε_1.
- A_1, through the global state, is now “aware” that it is not the “first” resolution step, even though the only available information to that resolution step or path should be the already bound variables. The branches are not isolated.

Let a be a Prolog predicate that can be proven in two ways by two different write operations:

```
1 a :- writeln(1).
2 a :- writeln(2).
```

Let ε be the environment before we try to prove a (line 1). Both rules expect the environment before their write-operation to be ε, but after we have proven a using the first rule, the environment is now ε'. This is an "unexpected" situation for the second rule (line 2).

Of course, in general in a nondeterministic reading of the program it is not clear that 1 should even be written before 2. For a parallel evaluation of the rules, the order of side-effects has to be painstakingly recovered [KPS88]. On the other hand, if both rules share the same effect, we get the output twice, even though the second write may very well just evaluate to `true`, as its effect has already happened.

- In our model we evaluate all alternate branches and suspend the computation of a branch either if it is finished with a pure result, i. e., a variable binding, or its next goal is an IO action.
- We employ a selection predicate that nondeterministically returns lists of suspended computations that are either pure or have mutually compatible IO actions. The selection predicate has the form `select(+ListOfComputations, -ListOfComputations)` and may be nondeterministic.
- The lists are handled sequentially
  - If the computations are pure, the variable binding is returned.
  - If the computations are not pure, the IO action is executed and its result is fed back into each branch in the list, i. e., it is shared between the branches.
- All branches and IO actions that are not selected into a list fail.

This allows us to effectively isolate rules and branches against each other. For example, for a selection predicate that returns a list of identical operations, we can give the guarantee that if a branch succeeded with a variable binding, its IO actions are the only IO actions that happened up to this point.

Note that in order to "collect" all continuations, our program is not allowed admit non-termination in any branch due to infinite derivation depth (`nt :- nt.`), or a predicate with an infinite number of choice points (`repeat`, but also `length`, if the first argument is not bound).

The core ideas of the IO-aware semantics is: **There is only one world.** This means:

- When a query is fully evaluated and all solutions have been returned, there is a sequence of IO actions $A_0 \ldots A_n$ that have happened. Every path in the resolution tree from the root node to a leaf node, both FAIL and TRUE-nodes, contains either the full sequence of IO actions or a prefix of it.

- When multiple alternative branches entail an IO action, only one side-effect actually happens.
- All branches with compatible IO actions to the action that has happened receive the result of the IO action, i. e. its output arguments and may continue with the computation.
- All branches with incompatible IO actions could never have a prefix of $A_0 \dots A_n$ as actions that happened and therefore fail before they execute the incompatible action.

If some branch succeeds, the programs observable behaviour is identical to the one without backtracking (i. e. we can program as if we guess right every time).

4 Selection Predicates and Further Examples

We have implemented three selection predicates or strategies. As an example we use the following predicate `e`. where we wrapped the usual Prolog writes into our own suspendable writes with the shareable semantics.

```
1 e(W) :- nio_write("B"), W = 0.
2 e(W) :- nio_write("A"), W = 1.
3 e(W) :- nio_write("A"), W = 2.
4 e(W) :- W = 3.
```

In this example, the computations are suspended at the impure writes (three writes, one with `B` and two with `A`) and a pure computation with a binding of `W` to `3`.

4.1 Selector for Prolog Semantics

This selector recovers the original Prolog semantics with the output being the same as the corresponding Prolog program:

```
IO write B
pure W = 0 ;
IO write A
pure W = 1 ;
IO write A
pure W = 2 ;
pure W = 3 .
```

4.2 Selector for Leftmost Side-Effect

This selector first selects all pure computations and returns the results and then returns a list of all suspended computations with IO actions compatible to the first (i. e., leftmost) IO action. All other suspended computations fail:

```
pure W = 3 ;
IO write B
pure W = 0 .
```

This selection function is (in the sense that SLD-resolution is) not complete, as we cut off other branches that do not fit our selected IO action. As described previously, there is an argument to evaluate IO actions with an environment that was changed in "parallel" branches as `false`, as we are unable to recover the original semantics of that IO action.

4.3 Selector for Side-Effect obtained through Consensus

The consensus selector finds all maximal lists of suspended impure computations where all computations are pairwise compatible. We then return the longest of these lists.

```
pure W = 3 ;
io write A
pure W = 1 ;
pure W = 2 .
```

4.4 Further Examples

Through the sharing of arguments from side-effects it is possible to call predicates with side-effects in a more general manner than usually allowed (i. e., with binding patterns that would lead to a runtime error), as long as the unification is specialized enough.

- This allows either rules 1 and 3 or 2 and 3 to succeed together:

```
1 shared_write1 :- nio_write(pair(7,_)).
2 shared_write1 :- nio_write(pair(9,8)).
3 shared_write1 :- nio_write(pair(_,8)).
```

- This also allows for **stealing** of arguments between branches:

```
1 shared_write2 :- nio_write("ABC").
2 shared_write2 :- nio_write(M), write("We stole: "), writeln(M).
```

Another application we propose are **"linear protocols"**, where some rather linear back-and-forth of resource allocations and communications between two computers is written down without a case distinction between protocol versions. This possibly allows us to "auto-select" the proper protocol without having to extract code for, i. e., establishing a connection (the starred predicates are wrapped IO):[3]

```
1 protocol(Server) :- openConnection*(Server,Connection),
  ↪ getCapabilites*(Connection, Capabilites), supportsA(Capabilites),
  ↪ doA*(Connection), close*(Connection).
2 protocol(Server) :- openConnection*(Server,Connection),
  ↪ getCapabilites*(Connection, Capabilites), supportsB(Capabilites),
  ↪ doB*(Connection), close*(Connection).
3 protocol(Server) :- openConnection*(Server,Connection),
  ↪ getCapabilites*(Connection, Capabilites), close*(Connection), fail.
```

In this case, the opening of the connection is shared between all branches, as is the retreival of the server's supported features. All network operations happen only once. We then check whether the server supports a specific feature or protocol version. Branches or protocol variants that are not supported by the server fail. `doA` and `doB` can not be shared. So once we get to a `do`-goal, all other branches with differing goals will fail. If none of our implementations is supported by the server, the last rule just closes the connection again and returns with failure.

4.5 Implementation

The implementation uses algebraic effects [PP02] and handlers [PP09]. Our implementation of effect handlers follows Schrijvers et al. [Sc13] and Saleh; Schrijvers [SS16].

We wrap the suspendable IO predicates and use a "runner" predicate to run a supplied goal with a given selector predicate (`?- run_nio_generic(consensus_selector, e(W)).`)

As different IO predicates have different positions of input and output arguments, it is necessary to wrap all wanted IO predicates individually. Internally, we want to make sure that `read`, for example, is always called with a fresh variable (such that it always succeeds), and this binding is then fed back into all branches, regardless of their actual argument. We want the branches to fail at this point then. Similarly, while the IO actions `read(a)` and `read(b)` are shareable (the same `read(X)` that is then unified with `a` and `b`), `write(a)` and `write(b)` are not.

[3] Of course, closing an opened connection or other resource should be done using `setup_call_cleanup`, which cannot be implemented in Prolog.

5 Conclusion and Open Questions

We have shown that there is some design space between Prolog's decision to run IO under nondeterminism "whenever", and Curry's decision not to allow IO under nonderterminism at all.

Our solution is running just a single strand of side-effects and each alternative or branching computation either uses the side-effects along this singular strand, or fails. All computations that produce a solution have "seen" a consistent environment and have been effectively shielded from side-effects that are not compatible with this computation's "history" of side-effects.

There are quite a few open questions that both lend themselves to further research and a frank discussion:

- A `writeln(T)` action may either be wrapped with its own predicate or expanded to the compound `write(T), write('\n')`. If we want to maximize sharing between branches, which is the useful approach? Do we want to decompose writes of strings into multiple writes of single character? Is it useful to share having written half a string?
- Do larger interactive programs (with reading configuration files, listening to incoming network requests, performing logging and maintenance tasks) even have codepaths that lend themselves to sharing?
- To prevent the duplication of IO under nondeterminism, specifically for the example of linear protocols, is it always possible and somewhat straightforward to just rewrite the original Prolog program?
- Is it possible to recover the Curry semantics with this approach (i. e., selector predicate)?
- We exchange depth-first for (not quite) breadth-first search through the SLD-Tree (we advance each branch until we reach IO). That could use a lot of memory and lose a lot of performance for some applications. If we restrict ourselves to the selector for leftmost side-effects, we probably can go back to something similar to SLD-resolution where we just need to additionally track a list of IO actions and their results. With only a single selector, is this still useful and is it worth it, in terms of resource usage?
- Should there be some control commands for the branches to interact with the IO that was done? If so, which are they? At first glance, it seems reasonable that a branch could say "well, that read did not succeed for me. But no harm done. Other branches may ignore that IO."

References

[Br05] Bramer, M.: Logic Programming with Prolog. Springer, 2005, ISBN: 978-1-85233-938-8.

[Ha97] Hanus, M.: Curry: A Multi-Paradigm Declarative Language (system description). In (Bry, F.; Freitag, B.; Seipel, D., eds.): Twelfth Workshop Logic Programming, WLP 1997, 17-19 September 1997, München, Germany, Technical Report PMS-FB-1997-10. Ludwig Maximilians Universität München, 1997.

[KPS88] Kalé, L. V.; Padua, D. A.; Sehr, D. C.: OR parallel execution of Prolog programs with side effects. J. Supercomput. 2/2, pp. 209–223, 1988, URL: https://doi.org/10.1007/BF00128177.

[Li19] Lifschitz, V.: Answer Set Programming. Springer, 2019, ISBN: 978-3-030-24657-0.

[NM90] Nilsson, U.; Maluszynski, J.: Logic, programming and Prolog. Wiley, 1990, ISBN: 978-0-471-92625-2.

[PP02] Plotkin, G. D.; Power, J.: Notions of Computation Determine Monads. In (Nielsen, M.; Engberg, U., eds.): Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings. Vol. 2303. Lecture Notes in Computer Science, Springer, pp. 342–356, 2002, URL: https://doi.org/10.1007/3-540-45931-6%5C_24.

[PP09] Plotkin, G. D.; Pretnar, M.: Handlers of Algebraic Effects. In (Castagna, G., ed.): Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Vol. 5502. Lecture Notes in Computer Science, Springer, pp. 80–94, 2009, URL: https://doi.org/10.1007/978-3-642-00590-9%5C_7.

[Sc13] Schrijvers, T.; Demoen, B.; Desouter, B.; Wielemaker, J.: Delimited continuations for prolog. Theory Pract. Log. Program. 13/4-5, pp. 533–546, 2013, URL: https://doi.org/10.1017/S1471068413000331.

[SS16] Saleh, A. H.; Schrijvers, T.: Efficient algebraic effect handlers for Prolog. Theory Pract. Log. Program. 16/5-6, pp. 884–898, 2016, URL: https://doi.org/10.1017/S147106841600034X.