# Lazy Stream Manipulation in Prolog via Backtracking: The Case of 2P-Kt

Giovanni Ciatto[1][0000−0002−1841−8996], Roberta Calegari[2][0000−0003−3794−2942], and Andrea Omicini[1][0000−0002−6655−3869]

[1] Department of Computer Science and Engineering (DISI)
[2] Alma Mater Research Institute for Human-Centered Artificial Intelligence
Alma Mater Studiorum—Univerisità di Bologna, Italy
{giovanni.ciatto, roberta.calegari, andrea.omicini}@unibo.it

**Abstract.** The ability of *lazily* manipulating long or infinite *streams* of data is an essential feature in the era of data-driven artificial intelligence. Yet, logic programming technologies currently fall short when it comes to handling long or infinite streams of data. In this paper, we discuss *how* Prolog can be reinterpreted as a stream processing tool, and re-designed around an abstract state-machine capable of lazily manipulating streams of data via backtracking.

**Keywords:** Prolog · Stream Processing · 2P-Kt · State Machine

## 1 Introduction

Streams are a powerful abstraction in computer science as they enable the processing of huge amounts of data, especially when keeping all data in memory would be impractical or infeasible. In the era of the Internet of Things (IoT) and data-driven artificial intelligence (AI), the ability of manipulating possibly unlimited streams of data is a must-have for all programming paradigms and languages. Indeed, a growing amount of application scenarios are characterised by the pervasive exploitation of smart devices generating/capturing huge amounts of data, as well as of the software infrastructures aimed at processing them.

A stream is an *ordered sequence* of data that may or may not be limited in length. Depending on how the are *generated*, streams are either *cold* (a.k.a. *pull*) or *hot* (a.k.a. *push*). Each item of a cold stream is generated on the fly, as soon as a consumer *pulls* it from the stream. In the case of hot streams, instead, an external entity is supposed to be in charge of generating items and *pushing* them to the stream, so that consumers can retrieve them in a FIFO way.

Cold streams are the simplest ones. A cold stream can be naturally attained via functional programming and higher-order functions (e.g. `map`, `filter`, `reduce`): this is why mainstream programming languages such as Java, C#, Python, JavaScript, Scala, Kotlin, etc., are being extended to blend functional features and constructs for dealing with streams. Conversely, hot streams are more complex, as they require data to be buffered while waiting for consumption—making them ideal for *temporally* decoupling data consumers and producers. In

particular, hot streams are key enablers of advanced stream processing techniques, such as sliding windows, or complex event processing (CEP)—which are deeply entangled with the *time*-related aspects of data production.

In this scenario, logic programming (LP), as well, has its role to play, both in data-driven AI – in particular in relation to explainable systems [6] – or in the IoT [5]. For instance, LP and rule-based frameworks are generally recognised as well-suited to support CEP [1,2], as they are expressive enough to capture complex events from hot streams. Similarly, answer-set programming (ASP) has been extensively exploited as a means for reasoning over hot streams of data [11,4,3].

In this paper, we focus on the Prolog [10] programming language—arguably, the most popular LP language. Currently, Prolog can hardly be considered as a suitable stream-processing technology [15], as it provides minimal support for consuming both cold and hot streams. However, we believe that this should be reconsidered because Prolog already supports the lazy exploration of possibly infinite search spaces via *backtracking*. Thus, the problem with Prolog is not to discuss *whether* it supports stream processing or not, but rather *how*.

Existing solutions extend Prolog with syntactical, semantical, or library enhancements aimed at supporting cold streams explicitly. Conversely, in this paper, we discuss how Prolog can be reinterpreted as a *stream processing tool*, capable of manipulating both cold and hot streams of data. In particular, our solution does not affect the syntax (nor the operation) of the Prolog language. More precisely, we show how Prolog predicates may be interpreted as *generators* of streams to be lazily consumed via backtracking. Along this line, we present an abstract design for Prolog solvers based on finite-state machines, aimed at supporting our notion of generators. Finally, a practical demonstration based on the 2P-Kt technology [7] is discussed showing how generators may let a Prolog solver consume events from the external world in a transparent way.

## 2 Logic Solvers as Streams Prosumers

### 2.1 Logic solvers as stream producers

Logic solvers *à la Prolog* are typically queried *interactively* by LP users in different *modes*, which are naturally captured by the message passing perspective adopted in fig. 1. The most common mode of interaction among users and logic solvers is summarised in fig. 1a: users submit *queries* (a.k.a. *goals*) to a logic solver – e.g. a Prolog interpreter – via some *ad-hoc* operation—e.g., `solve`. Assuming that one or more *solutions* exist, the solver computes and returns one of them—typically in terms of a *unifying substitution*, assigning values to the query variables of interest for the user. However, the user may be interested in solutions other than the first one: so, the solver should expose one further operation – e.g., `next` – letting users asking for further solutions to some previously-submitted query. Finally, when no (more) solutions are available for a query, the solver can return one (last) answer carrying the *failed* substitution (represented by $\perp$ in fig. 1) instead of a unifier.

This mode of interaction is very effective since it enables the *lazy* enumeration of a possibly infinite amount of solutions. However, it comes with a few drawbacks. First, despite logic solvers are actually capable of generating streams of solutions, the notion of stream is somewhat *implicit* in the solver machinery—therefore, not explicitly exploitable. Second, solvers are *stateful*, in that they are responsible to keep track of the status of the interaction with each querying user.

To overcome these issues we suggest a shift of perspective, as depicted in fig. 1b. There, users and solvers interact in a *stream-oriented* mode, where the stream of solutions is *explicit* and the interaction between solvers and users is *stateless*. Thus, solvers expose just one operation – i.e., `solve` – accepting a user's query and returning a reference to the related *cold* stream of solutions. Users just need solvers to create solution streams that users can then lazily consume on demand. Of course, solutions can still be produced lazily behind the scenes: whenever a user tries to consume a new solution, it can be computed on the fly.

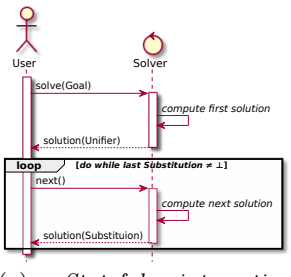
Thus, even though interaction does not change from the operational viewpoint, our approach overcomes the limits of traditional logic solvers: solution streams here are *explicitly* represented, and can therefore be *manipulated* as such.

### 2.2 Logic solvers as stream consumers

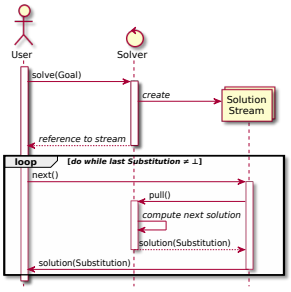
By adopting a message passing perspective, logic solvers do not interact with users only. Indeed, logic solvers typically act on a *knowledge base* (KB). In the general case, KBs are containers of the specific knowledge required by solvers to compute solutions to users' queries. For instance, KB for Prolog solvers contain both rules and facts as Horn clauses, and are either static or dynamic.

From an interaction perspective, however, a KB is just a component exploited by solvers as part of their resolution process. More precisely, solvers may need KB to retrieve some clauses, selected via unification, or, to retract or store some knowledge possibly learned/acquired during the resolution.

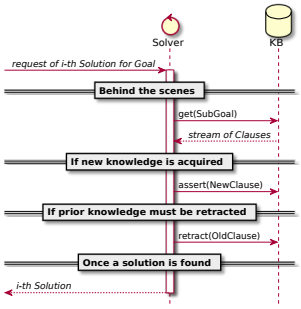
Fig. 1: Interaction modes between logic solvers and users or KB.




(a) *Stateful interaction* mode




(b) *Stream-oriented interaction* mode




(c) Interaction among a logic solver and its KB

In particular, clause retrieval highlights how the interaction between solver and KB can be described in terms of streams as well. As depicted in fig. 1c, clause retrieval from KB can be modelled as a operation – e.g., `get` – accepting a clause template $C$ and returning the *stream* of clauses unifying with $C$ currently stored into the KB. The solver can then consume the stream as needed, e.g. either lazily or not, depending on the search strategy adopted.

Finally, storing a clause in the KB can be modelled as an `assert` accepting a clause $C$ and adding it to the KB, whereas clause retraction can be modelled as a `retract` accepting a clause template $C$ and removing a clause $C'$ unifying with $C$. Both operations could be exploited either by the solver or by some *external* entity willing to affect the solver's knowledge.

### 2.3 Solvers vs. the World

Yet, how can logic solvers deal with event streams coming from the external world? Once KBs are recognised as individual entities, a trivial answer could be: *via KB*. External events may indeed be *reified* into actual knowledge to be stored into some solver's KB. In this scenario, external event streams should be translated into a sequence of `assert`ions aimed at injecting events into the KB, as facts. The solver could then lazily consume the events by `get`ting or `retract`ing the corresponding facts from the KB.

There are, however, two major drawbacks in this approach. First, the reification of events into KB requires space. Second, solvers do not necessarily have to process or *consume* reified events—thus a lot of space is wasted. Accordingly, a different approach is required to let solvers consume event streams from the external world without reifying them unnecessarily.

In this work, we propose *generators* as the basic means to let solvers interact with the external world. A generator is a special Prolog primitive capable of affecting and inspecting the external world via some I/O facility (fig. 2). It is invoked by a solver and produces a *stream of facts* to be consumed by the same solver. However, from the solvers perspective, generators are ordinary built-in predicates denoted by *signatures*—i.e., name/arity couples of the form $p/n$.

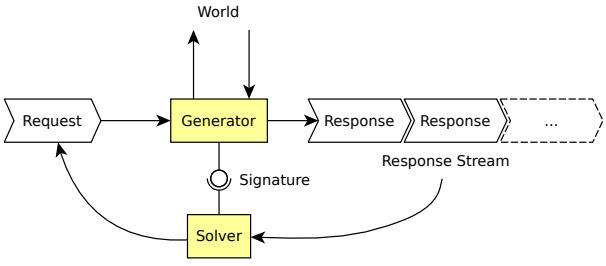



Fig. 2: Dataflow and component view of *generators*, i.e. solvers' gates towards the external world

More precisely, whenever the solver needs to compute the assignment of variables $T_i$ satisfying relation $p(T_1, \ldots, T_n)$, it can trigger the generator denoted by $p/n$ (if it exists), by sending the $p/n$ generator a *request* providing a snapshot of the current resolution context and possibly an initial assignment of some $T_i$. The generator answers by providing a stream of *responses* – each one with some possible complete assignment of $T_i$ – that the solver can consume accordingly to its resolution strategy—i.e., possibly later. To produce responses, generators may take into account several information sources – e.g., the resolution context, the external world – as a part of the request. They may also attempt to *affect* the external world via some I/O *action*—e.g., triggering a sensor.

Depending on the numbers of responses a generator provides, it can either be classified as either *functional* or *relational*. Functional generators produce just one response and their execution is therefore analogous to the execution of a function, as they consume an input and return a single result. Conversely, relational generators produce two or more responses.

### 2.4 Example: TSP in Prolog

Let us consider for instance the case of a user exploiting a standard Prolog system to solve arbitrary instances of the Traveling Salesman Problem (TSP).

Let us assume the system requires maps to be represented as facts in the form `path(+Src, +Dst, +Cst)` – each one representing an undirected path between two locations, and the estimated cost –, like e.g.:

```
path(bucarest, giorgiu, 90).
path(bucarest, pitesti, 101).
path(pitesti, 'rimnicu vilcea', 97).
path(pitesti, craiova, 138).
path('rimnicu vilcea', craiova, 146).
...
```

Under this assumption, Prolog exposes a predicate `tsp(?Cities, ?Circuit, ?Cost)` aimed at computing the best `Circuit` for some set of `Cities`, and the corresponding `Cost`—where, `Cities` is a set of cities, `Circuit` is a list of cities to be visited in a row, and `Cost` is an integer. Following a purely-logical interpretation, the predicate represents a ternary relation $\texttt{tsp} \subseteq 2^{\mathcal{C}} \times \mathcal{C}^* \times \mathbb{N}$ grouping subsets of cities, lists of cities, and non-negative integers, where $\mathcal{C}$ is the set of all cities mentioned in the KB as either the first or second argument of a `path/3` fact, and $\mathcal{C}^*$ is the Kleene-closure of $\mathcal{C}$. Thus, an assignment of the `Cities`, `Circuit`, and `Cost` variables satisfies the predicate if

- `Circuit` $\equiv [c_0, \ldots, c_{n-1}, c_0]$, and
- `Cities` $\equiv \bigcup_{i=0}^{n-1} \{c_i\}$, and
- $\forall i \in \{1, \ldots, n\}$ `path`$(c_{i-1}, c_{i \bmod n}, x_i) \in$ KB, and
- `Cost` $\equiv \sum_{i=1}^{n} x_i$, and
- `Cost` is *minimal*.

Accordingly, because of Prolog backtracking, a query of the form:

```
?- tsp(Cities, Circuit, Cost).
```

would enumerate all minimally-costly circuits of all possible subsets of cities in $\mathcal{C}$, and their costs—one for each solution. Users may partially instantiate some variable in order to contextualise their queries: for instance, a query of the form:

```
?- tsp({pitesti, craiova, 'rimnicu vilcea'}, [pitesti | Others], Cost).
```

would enumerate all minimally-costly circuits starting in Pitesti, and involving the cities Craiova, and Rimnicu Vilcea.

The predicate `tsp/3` could be implemented declaratively in Prolog. In its simplest formulation, the predicate may leverage Prolog's depth-first strategy, and its backtracking mechanism to lazily generate all the possible circuits and select the less costly one: not likely the best possible strategy, yet a working one. However, better strategies have been proposed in the literature for solving the TSP, with efficient implementations built upon them—rarely based on pure Prolog. Here, instead, generators make it possible to exploit external libraries for solving the TSP in Prolog as if they were implemented via LP.

For instance, we assume that a "ACME TSP" C library exists that solves TSP efficiently, which can be wrapped within a relational generator `tsp/3` to be exploited by a Prolog solver. Generator `tsp/3` should work as follows:

1. whenever the Prolog solver encounters a `tsp(Cities, Circuit, Cost)` sub-goal, it triggers the generator via a *request* containing a snapshot of the current KB and the *actual* values of `Cities`, `Circuit`, and `Cost`;
2. the generator reads *(i)* the map graph from the KB snapshot, and *(ii)* the cities from the actual value of `Cities`;
3. the generator generates the stream of all the possible subsets of $\mathcal{C}$ and selects the ones unifying with the actual value of `Cities`, thus: if `Cities` is bound to a particular sub-set of cities, then the stream has just one element, otherwise it may have several ones;
4. for each sub-set of cities in the stream, the generator triggers ACME TSP and computes the corresponding TSP solution, if any;
5. every time it is triggered, ACME TSP computes zero or more solutions for the TSP and returns them to the generator;
6. for each TSP solution of each selected instantiation of `Cities`, the generator yields a response to the solver;
7. each response may either contain a unifier – assigning `Cities` to the selected list of cities, `Circuit` to the minimally-costly circuit for those cities, and `Cost` to the cost of that circuit – or a failed substitution—informing the solver the `tsp/3` predicate should fail;
8. the solver can consume the response stream lazily via backtracking.

In other words, generators can be exploited as a means to wrap external data producers and let the solver consume the data they produce via streams. In Prolog, streams of this sort are lazily consumed via ordinary backtracking: the solver lazily generates a new choice point for each element in the stream and handles

them as usual. Solvers of different sorts may consume the stream differently—e.g. buffering (some slice of) it, or, handling each datum concurrently.

## 3 Solvers as Streams Prosumers via State Machine

In order to design a Prolog solver supporting our notion of generator, we enhance the Prolog state machine proposed in [13] with the capability of lazily consuming streams of data coming from either a generator or the KB (fig. 3). In particular, we change how the state machine manages the resolution of (sub-)goals, by supporting the selection of a generator as a means to provide one or more solutions for (sub-)goals, other than the ordinary selection of rules from the KB.

The state machine in fig. 3 stems from the acknowledgement that a Prolog solver may solve a (sub-)goal by either selecting a generator or a number of logic rules from the KB. In both cases, a stream of data must be lazily consumed by the solver—either carrying generator responses or clauses from the KB.

Whenever a stream of data needs to be processed, there are essentially two major phases: the *opening* of the stream – where a channel between the stream producer and its consumer is created –, and the *consumption* of the stream—where items from the stream are sequentially processed. To support both phases, two more locations are included – namely Generator Selection and Generator Execution – respectively aimed at triggering a generator and consuming the response stream it provides. Furthermore, to support a stream-oriented interaction among the solver and its KB, we model rule management as well through two locations, namely Generator Selection and Generator Execution, respectively aimed at querying the KB, and consuming the rule stream it provides.

All the other aspects are handled in the same way as in [13]. Thus, state machine execution is triggered whenever a user submits a query to the solver: when this is the case, execution starts from the Goal Selection location. Then, it may go through any location until it eventually reaches some *final* one (End or

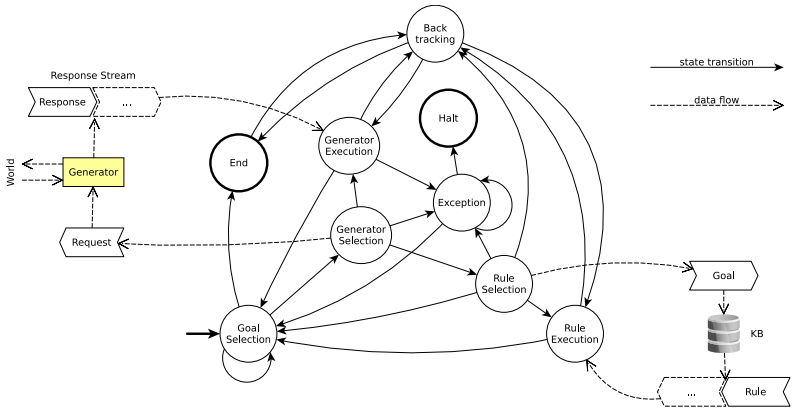



Fig. 3: Handling generators with enhanced Prolog state machine

Halt), where a new solution is yielded—which the user can eventually consume. Once a solution is consumed, the user can either submit a new query or ask for the next solution. In the former case, the automaton is reset to the Goal Selection location. Conversely, the latter case is only possible if the last solution was provided by the End location. In that case, the automaton backtracks and looks for the next solution. This may involve stepping through Backtracking, then moving back into the Generator (resp. Rule) Execution, in order to consume one more element from some previously-opened response (resp. caluse) stream.

Overall, our state machine affects the operation of a Prolog solver as follows:

1. [Generator Selection] whenever a new sub-goal is selected, the solver looks for a generator whose signature matches the sub-goal one;
2. [Generator Execution] if some are found, the solver considers the first response in the stream as a solution to the goal, and generates choice points for subsequent responses;
3. [Rule Selection] otherwise, if no generator is selected for the current sub-goal, some rule is looked for instead, whose head unifies with the sub-goal;
4. [Rule Execution] if any such rule is found, resolution can proceed by addressing the rule's body as the next goal to be proved;
5. [Backtracking] otherwise, if no rule is found, the sub-goal is considered failed and resolution must backtrack.

Location Exception completes the picture by intercepting exceptions – possibly thrown by generators as part of some response of theirs –, via the standard `catch/3` predicate.

Ordinary Prolog built-in primitives naturally fit the picture as they are reinterpreted as generators by solvers. For instance, the `is/2` predicate can be considered a functional generator accepting a variable and an expression and returning a single response assigning the variable to the value attained by reducing the expression – if possible –, or an exception—in case the expression cannot be reduced. Conversely, the `member/2` predicate can be considered as a relational generator, enumerating all the possible items in a list. Accordingly, the aforementioned Generator Selection location is where built-in primitives are selected for execution in place of rules from the KB.

## 4 Backatrackable Predicates as Streams in 2P-Kt

In order to demonstrate the feasibility of our approach, we propose a case study based on 2P-Kt. 2P-Kt [7] is a Kotlin-based ecosystem for LP, including general API for stream-oriented logic solvers of any sort. Regardless of the particular logic, inference rule, or search strategy of choice, a logic solver is modelled in 2P-Kt as a prosumer of streams: it produces output streams of solutions and consumes input streams generated by generators. A Prolog solver implementation is available as well, leveraging the state-machine-based design presented in section 3. Furthermore, 2P-Kt involves an API for writing generators in Kotlin, by blending an imperative, object-oriented, and functional programming style.

In this section, we first illustrate briefly the portion of the 2P-Kt API involving solvers and generators, then we discuss an example generator implementing the TSP example from 2.4.

### 4.1 2P-Kt Solvers and Generators API

Figure 4 provides an overview of the 2P-Kt API. Here we focus on the resolution-related portion of this API (cf. [9] for further details). There, logic solvers are modelled as instances of the `Solver` type defined as follows:

```
interface Solver {
    val staticKb: Theory
    val dynamicKb: Theory
    val libraries: Libraries
    fun solve(goal: Struct): Sequence<Solution>
}
```

Essentially, a logic solver is any entity exposing a method `solve` which accepts a logic `Structure` – i.e., a particular case of logic `Term` in the 2P-Kt type system – as the input goal, and produces a `Sequence` – i.e., a *lazy* stream in the Kotlin type system – of logic `Solution`s as output. Furthermore, 2P-Kt requires each logic solver to be composed by at least three more entities, namely: *(i)* a `staticKb` and *(ii)* a `dynamicKb`, both of type `Theory` – that is, an ordered and indexed container of logic clauses, retrievable via unification –, and *(iii)* a `libraries` container of type `Libraries`—which, within the scope of this section, is essentially an implementation of the structure indexing generators.

Each `Solution` in 2P-Kt may be of any of three sorts, namely `Yes`, `No`, and `Halt`, representing the positive, negative, and exceptional case, respectively. All solutions carry the original query they are answering to, other than the `Substitution` they are answering through. So for instance, objects of type `Solution.Yes` always contain an object of type `Substitution.Unifier`, whereas other sorts of solutions always contain an object of type `Substitution.Fail`. Similarly, objects of type `Solution.Halt` carry the uncaught exception which interrupted the resolution process.

Generators are modelled in 2P-Kt as functions of the type:

```
typealias Generator = (Request) -> Sequence<Response>
```

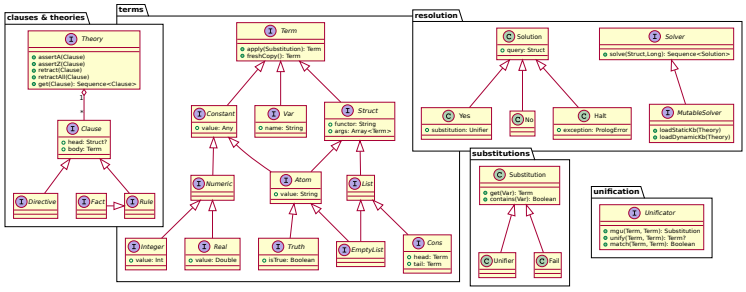



Fig. 4: Overview on the public API of 2P-Kt

i.e., functions accepting a `Request` as input and returning a `Sequence` of `Responses` as output. There, `Request` is a container of all the information needed at runtime to produce a sequence of `Responses`:

```
class Request(
    val context: ExecutionContext,
    val signature: Signature,
    val arguments: List<Term>
) {
    fun solve(subQuery: Struct): Sequence<Solution>
    fun replySuccess(): Response
    fun replyFail(): Response
    fun replyWith(substition: Substitution): Response
    fun replyException(exception: TuPrologRuntimeException): Response
}
```

These include: *(i)* a snapshot of `ExecutionContext` at invocation time – in turn including a snapshot of the solver's `staticKb` and `dynamicKb` –, *(ii)* the `Signature` of the invoked generator, and *(iii)* the `List` of `Terms` storing actual `arguments` provided to the generator upon invocation. Furthermore, each instance of `Request` exposes a bunch of methods – namely, the many `reply*()` ones –, aimed at generating a new `Response` for that particular `Request`. As `Responses` are mere containers of `Solutions`, there are many variants of the `reply*()` methods, each one aimed at generating a given sort of responses – e.g. responses carrying positive/negative/exceptional solutions – for the subgoal that triggered the generator. Finally, each request supports the spawning of an inner resolution process via its `solve(...)` method. This method creates a novel sub-solver through which generator implementors can resolve sub-queries as part of some generator execution.

Thanks to this design, any Kotlin method of the form:

```
fun method(request: Request): Sequence<Response> = sequence {
  request.arguments[i] // read the i-th actual argument
  request.context.staticKb[h] // read clauses in KB whose head matches h
  solve(goal) // perform sub-queries

  val substitution = (arg0 mguWith value0) + (arg1 mguWith value1) + ...

  yield(request.replyWith(substitution))
  // or
  yield(request.replyFail())
  // or
  yield(request.reply*(...))
}
```

can be considered a generator in the eyes of a logic solver. This leverages a particular feature of Kotlin, namely the `sequence { ... }` blocks, which let developers write stream *generators* by blending the imperative and functional programming styles. This is possible because of the `yield(value)` method which users may call inside `sequence { ... }` blocks in place of `return value` to provide values to the stream.

So, for instance, to implement the predicate `natural/1` – which holds true for all natural numbers –, one may write the following generator:

```
fun natural(request: Request): Sequence<Response> = sequence {
    var n = 1
    while (true) {
        yield(Integer.of(n))
        n++
    }
  }.map {
    request.replyWith(request.arguments[0] mguWith it)
  }
```

A Prolog solver would then treat such a generator as a backtrackable predicate. Thus, in Prolog, one may use the goal `natural(X)` to enumerate all the natural numbers.

Summarising, 2P-Kt generators API supports the creation of backtrackable Prolog predicates out of lazy data streams.

### 4.2 Travelling Salesman Problem in 2P-Kt

The real potential of generators is revealed when they are exploited by solvers to manage input data streams from the external world. There, the external world may be any source of data, there including other solvers, possibly of different nature. For example, generators may be exploited to let a Prolog solver call a TSP solver to efficiently compute solutions for TSP instances, as discussed in section 2.4. Accordingly, here we demonstrate how a generator of such a sort may be realised through 2P-Kt.

In [8] we provide a GitHub repository hosting the source code of a 2P-Kt generator leveraging Google OR-Tools [12] to efficiently solve TSP instances. Google OR-Tools is a C++ library proving many constraint programming and operative research tools – there including routing-related facilities –, and some JVM bindings which let us exploit such tools in Kotlin.

Accordingly, our repository includes some scripts aimed at automating the compilation and execution of a simple demo involving a command-line TSP-enabled Prolog interpreter. Following the discussion from section 2.4, such a Prolog interpreter exposes a `tsp/3` predicate aimed at enumerating the minimally-costly circuits for any given set of cities, provided that the interpreter's KB contains several `path/3` facts describing the connections among those cities. As an ordinary Prolog interpreter, such facts may be either consulted from a `.pl` file or dynamically asserted via `assert/1`.

The actual operational behaviour of predicate `tsp/3` is governed by the `Tsp` generator whose source code (stub) is shown in fig. 5 (cf. [8] for full source code). The `Tsp` generator is a singleton object of type `TernaryRelation` – i.e., a particular sort of `Generator`, tailored on ternary predicates –, whose main behaviour is encapsulated within the `computeAll` method.

The `Tsp` object is also endowed with a method – namely, `tsp` – which returns a sequence of circuits and costs for any given list of cities provided as input. Such method assumes each input city to be represented by a logic term – in particular, a constant –, and outputs circuits represented as logic lists of cities represented in the same way. Behind the scenes, the `tsp` interacts both the Prolog interpreter's KB to read distances among cities, and a Google OR-Tool solver for computing all possible solution to a particular TSP instance.

The `computeAll` handles the situation where the Prolog interpreter meets a (sub-)goal of the form `tsp(Cities, Circuit, Cost)`—where all variables may be partially or totally uninstantiated. The method operation can then be described as a pipeline of *lazy* operations applied to the actual arguments of `tsp/3`, which we refer as `fst`, `snd`, and `trd` within the method. Accordingly, the method firstly performs a sub-query aimed at computing the set of all cities currently

contained into the KB (cf. variable `allCities` in fig. 5). The sub-query is a Prolog goal of the form `path(_, _, _)`, whose solutions are all eagerly consumed and their first and second arguments – which are assumed to be city names – are merged into a set, to remove duplicates. Then, all possible permutations of all possible subsets of `allCities` are lazily generated. However, only the subsets of cities that unify with `fst` are actually selected (this may be just one set of cities if `fst` refers to a fully instantiated set of cities) for the next steps of the computation. Then, for all selected sets of cities, all possible solutions to the corresponding TSP instance are computed. Finally, each possible circuit (resp. cost) computed for each TSP instance is unified with `snd` (resp. `trd`). Failed unifications are of course dropped, while the successful ones are converted into responses of the `tsp/3` generator.

It is worth to highlight that the whole pipeline is *lazy*. This implies that even once the first TSP solution has been presented to the user, the other ones are still to be computed.

## 5 Conclusion and Future Work

In this paper we address the issue of stream processing in logic programming.

In particular, we discuss how logic solvers can be naturally conceived as lazy prosumers of data streams as they *(i)* lazily *produce* data streams thanks to their interactive nature, *(ii)* lazily *consume* data streams as part of their resolution process—e.g. when they access knowledge bases. Furthermore, we show how logic solvers can support the processing of input data stream via the notion of predicates as *generators*, which we introduce in this paper. Summarising, generators are reactive computational units which logic solvers may trigger so

```
import it.unibo.tuprolog.core.List as LogicList

object Tsp : TernaryRelation<ExecutionContext>("tsp") {
  init { com.google.ortools.Loader.loadNativeLibraries() }

  private fun Request<ExecutionContext>.tsp(cities: List<Term>): Sequence<Pair<LogicList, Integer>> { ... }

  // other utility methods

  override fun Request<ExecutionContext>.computeAll(fst: Term, snd: Term, trd: Term): Sequence<Response> {
    val allCities = solve(Struct.template("path", 3))
      .filterIsInstance<Solution.Yes>()
      .map { it.solvedQuery }
      .flatMap { sequenceOf(it[0], it[1]) }
      .toSet()

    return allCities
      .subsets()
      .flatMap { it.permutations() }
      .map { it to (Set.of(it) mguWith fst) }
      .filter { (cities, substitution) -> cities.isNotEmpty() && substitution is Unifier }
      .flatMap { (cities, substitution) -> tsp(cities).map { it.addLeft(substitution) } }
      .map { (substitution, circuit, cost) -> substitution + (snd mguWith circuit) + (trd mguWith cost) }
      .filterIsInstance<Unifier>()
      .map { replySuccess(it) }
  }
}
```

Fig. 5: 2P-Kt generator implementing the `tsp/3` predicate

as to receive data streams from the external world. This may be useful, for instance, to let a solver delegate some part of its resolution process to some external entity—assuming that it is optimised to the purpose.

To demonstrate the feasibility of our approach in the specific (and technically most relevant) case of Prolog, we propose a generator-enabled modelling of Prolog solvers as state machines, formalising the lazy consumption of streams via backtracking. The proposed formalisation preserves the standard operation of Prolog and requires no modification to the language, while enabling Prolog solvers to process data streams.

Finally, we discuss the use case of 2P-Kt [7], a Kotlin-based technology for LP including an implementation of Prolog solvers relying on our state-machine-based formalisation. We then exploit 2P-Kt to show how generators can be used to bridge different sorts of solvers together via a few lines of Kotlin code.

In our perspective, this work represents one further step towards the *practical* exploitation of LP – and, in particular, Prolog – as a general means for stream processing. Notably, our contribution presents some similarities with other works [15,14]. In particular, similarly to [15], we focus on letting Prolog manipulate streams of data; while, similarly to [14], we provide a mechanism to let logic solvers delegate computations to extenal entities. However, differently from [15], we require no variation to the syntax, functioning, or libraries of Prolog; while, unlike [14], we focus on Prolog rather than ASP.

A number of issues remain uncovered in this work, and will be the subject of our future research. Among the many, the most relevant issues concern *time* and *side effects*. In particular we plan to explore the temporal dimension in LP-based stream processing, by providing for instance some means to support time-dependent or time-limited data streams. Similarly, we would like to explore the intricacies related to the processing of data streams which may affect the internal state of a logic solver – e.g. by affecting the KB – in a predictable way.

## Acknowledgments

Andrea Omicini has been supported by the H2020 Project "AI4EU" (G.A. 825619). R. Calegari has been supported by the H2020 ERC Project "CompuLaw" (G.A. 833647).

## References

1. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A rule-based language for complex event processing and reasoning. In: Hitzler, P., Lukasiewicz, T. (eds.) Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6333, pp. 42–57. Springer (2010). https://doi.org/10.1007/978-3-642-15918-3_5
2. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Real-time complex event recognition and reasoning–a logic programming approach. Applied Artifical Intelligence **26**(1-2), 6–57 (2012). https://doi.org/10.1080/08839514.2012.636616

3. Beck, H., Dao-Tran, M., Eiter, T.: Lars: A logic-based framework for analytic reasoning over streams. Artificial Intelligence **261**, 16–70 (2018). https://doi.org/10.1016/j.artint.2018.04.003
4. Beck, H., Eiter, T., Folie, C.: Ticker: A system for incremental ASP-based stream reasoning. Theory and Practice of Logic Programming **17**(5-6), 744–763 (2017). https://doi.org/10.1017/S1471068417000370
5. Calegari, R., Ciatto, G., Mariani, S., Denti, E., Omicini, A.: LPaaS as micro-intelligence: Enhancing IoT with symbolic reasoning. Big Data and Cognitive Computing **2**(3) (2018). https://doi.org/10.3390/bdcc2030023
6. Calegari, R., Ciatto, G., Omicini, A.: On the integration of symbolic and sub-symbolic techniques for XAI: A survey. Intelligenza Artificiale **14**(1), 7–32 (2020). https://doi.org/10.3233/IA-190036
7. Ciatto, G.: 2P-Kt, https://github.com/tuProlog/2p-kt
8. Ciatto, G.: Travelling salesman problem (TSP) in 2p-kt, https://github.com/tuProlog/ortools-tsp-example
9. Ciatto, G., Calegari, R., Siboni, E., Denti, E., Omicini, A.: 2P-Kt: logic programming with objects & functions in Kotlin. In: Calegari, R., Ciatto, G., Denti, E., Omicini, A., Sartor, G. (eds.) WOA 2020 – 21th Workshop "From Objects to Agents". CEUR Workshop Proceedings, vol. 2706, pp. 219–236. Sun SITE Central Europe, RWTH Aachen University, Aachen, Germany (Oct 2020), http://ceur-ws.org/Vol-2706/paper14.pdf
10. Colmerauer, A., Roussel, P.: The birth of prolog. In: Lee, J.A.N., Sammet, J.E. (eds.) History of Programming Languages Conference (HOPL-II). pp. 37–52. ACM (Apr 1993). https://doi.org/10.1145/154766.155362
11. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005. pp. 90–96. Professional Book Center (2005), http://ijcai.org/Proceedings/05/Papers/1353.pdf
12. Perron, L., Furnon, V.: OR-tools, https://developers.google.com/optimization/
13. Piancastelli, G., Benini, A., Omicini, A., Ricci, A.: The architecture and design of a malleable object-oriented Prolog engine. In: Wainwright, R.L., Haddad, H.M., Menezes, R., Viroli, M. (eds.) 23rd ACM Symposium on Applied Computing (SAC 2008). vol. 1, pp. 191–197. ACM, Fortaleza, Ceará, Brazil (16–20 Mar 2008). https://doi.org/10.1145/1363686.1363739
14. Redl, C.: The DLVHEX system for knowledge representation: recent advances (system description). Theory and Practice of Logic Programming **16**(5-6), 866–883 (2016). https://doi.org/10.1017/S1471068416000211
15. Tarau, P., Wielemaker, J., Schrijvers, T.: Lazy stream programming in Prolog. Electronic Proceedings in Theoretical Computer Science **306**, 224–237 (Sep 2019). https://doi.org/10.4204/eptcs.306.26