Lazy Stream Programming in Prolog

Paul Tarau    Jan Wielemaker    Tom Schrijvers

paul.tarau@unt.edu
J.Wielemaker@vu.nl
tom.schrijvers@cs.kuleuven.be

ICLP’2019, Las Cruces, NM
Motivation

- **stream processing** is now prevalent in: Java, Python, C#, go, Lua, JavaScript etc.
- processing *infinite sequences* is also available in non-strict functional languages like Haskell
- streams offers a uniform and (mostly) declarative view on processing finite and infinite sequences
  - local or network/cloud-based I/O operations
  - large training data sets for machine learning
  - dynamic event streams coming from Web search queries and clicks
  - sensor networks supporting today’s IoT infrastructure
- ⇒ extend Prolog with state-of-the-art lazy stream processing!
Overview

- answer stream generators that encapsulate the work of coroutining first-class logic engines
- support interoperation between forward recursive \textit{AND-streams} and backtracking-generated \textit{OR-streams}
- expose them to the application programmer either through an abstract sequence manipulation API or as lazy lists
- an isomorphism that transports data and operations between these two representations
- an algebra of stream generator operations that extends Prolog via an embedded language interpreter providing a compact notation for composition mechanisms
Contributions

- simple and clean approach for setting up lazy streams, which uniformly encapsulates algorithms, lists, first-class logic engines and other data sources
- by using attributed variables, we expose lazy streams in the form of lazy Prolog lists that, just like conventional lists, can be inspected and decomposed with unification
- an embedded language interpreter supporting an algebra of stream operations
- we have implemented our approach in several libraries:
  1. `lazy_streams`: generator builders, generator operations, an embedded generator algebra interpreter: a standalone package at https://github.com/ptarau/AnswerStreamGenerators/raw/master/lazy_streams-0.5.0.zip
  2. `lazy_lists`: generator predicates for directly setting up lazy lists (part of SWI-Prolog)
  3. `pure_input` library provides a range of predicates for reading files and sockets backed by lazy lists (part of SWI-Prolog)
A few use cases

- **map operation on infinite streams**

  ```prolog
  ?- pos(P), neg(N), map(plus, P, N, Zero), show(10, Zero).
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0].
  ```

- **convolution between an infinite and a finite stream**

  ```prolog
  ?- pos(P), list([a, b, c], L), convolution(P, L, C), show(16, C).
  [1-a, 1-b, 2-a, 1-c, 2-b, 3-a, 2-c, 3-b, 4-a, 3-c,
   4-b, 5-a, 4-c, 5-b, 6-a, 5-c].
  ```

- **evaluating stream expressions**

  ```prolog
  ?- X in_ [a, b]*(1:4).
  X = a-1 ; X = b-1 ; X = b-2 ; X = a-2 ; X = b-3 ; X = a-3.
  ```
The stream constructors

- *stream generators* are created by a family of constructors, encapsulating sequences produced algorithmically or as a result of state transformers interfacing Prolog with the “outside world.”
- A design philosophy similar to that of monads in functional languages.
- A generator is represented by a *closure* i.e., a term that can be called with an additional argument to yield the next element in the stream.
- When the closure has no more elements to yield, it fails.

```prolog
ask(E,_) :- is_done(E), !, fail.
ask(E,R) :- call(E,X), !, R = X.
ask(E,_) :- stop(E), fail.
```

where
```
is_done(E) :- arg(1,E,done).
stop(E) :- nb_setarg(1,E,done).
```
Some basic generators

- an infinite constant stream:
  \[ \text{const}(C, = (C)) . \]

- the `rand/1` predicate produces the `random/1` stream generator, which relies on externally maintained state to yield floating point numbers between 0 and 1
  \[ \text{rand}(\text{random}) . \]

- a stream created by incrementally evolving a state:
  \[ \text{gen_next}(F, \text{State}, X) :- \]
  \[ \text{arg}(1, \text{State}, X), \]
  \[ \text{call}(F, X, Y), \]
  \[ \text{nb_setarg}(1, \text{State}, Y) . \]

- here \text{State} acts as a container for destructively updated values using the \text{nb_setarg}/3 built-in
Iterating over the elements of a stream via backtracking

- the `in/2` predicate uses `ask/2` to produce the elements on backtracking

  ```prolog
  :-op(800, xfx, (in)).
  X in Gen:-ask(Gen, A), select_from(Gen, A, X).
  select_from(_, X, X).
  select_from(Gen, _, X):-X in Gen.
  ```

- with `in/2`, the generator emulates its declarative Prolog equivalent

- a declarative view of a procedural iteration mechanism: it works as if `member/2` were used on a list!
Backtracking with \texttt{in/2}

a stream backed by a list

?- \texttt{list([1,2,3],G),X in G.}\n\texttt{G = gen\_nextval(list\_step, state([2, 3])),}\n\texttt{X = 1 ;}\n\texttt{G = gen\_nextval(list\_step, state([3])),}\n\texttt{X = 2 ;}\n\texttt{G = gen\_nextval(list\_step, state([])),}\n\texttt{X = 3 ;}\n\texttt{false.}

elements of a random streams

?- \texttt{rand(G),X in G.}\n\texttt{G = random(),}\n\texttt{X = 0.7711731055905214 ;}\n\texttt{G = random(),}\n\texttt{X = 0.32254300867150587 .}
A steam *engine* pulling a stream of cars (some first class)
First class logic engines

- a first-class logic engine can be seen as a Prolog virtual machine that has its own stacks and machine state
- already present in BinProlog from around 1995
- in their SWI-Prolog implementation, unlike normal Prolog threads they are not associated with an operating system thread
- instead, one asks an engine for a next answer with the predicate `engine_next/2`
- implementing the engine API only assumes that the virtual machine is fully re-entrant, that it can be queried and that it can stop, yield data and resume execution as a coroutine
Generators built with first class logic engines

- the predicate \texttt{eng/3} creates a generator as a wrapper for the
texttt{engine\_next(Engine,Answer)} built-in, encapsulating the
answers of that engine as a stream

\begin{verbatim}
eng(X,Goal,engine\_next(Engine)) :-
    engine\_create(X,Goal,Engine).
\end{verbatim}
The AND-stream / OR-stream Duality

- first-class engines support two ways of producing a stream of answers
  - via backtracking (OR-streams)
  - as part of a forward moving recursive loop (AND-streams)
- the stream generator abstraction makes the user oblivious to this choice of generation method

\[\text{and\_nat\_stream}(\text{Gen}) :\neg\text{eng}(\_ , \text{nat\_goal}(0), \text{Gen}) .\]

\[\text{nat\_goal}(N) :\neg\text{succ}(N, SN), \text{engine\_yield}(N), \text{nat\_goal}(SN) .\]

operationally equivalent to:

\[\text{or\_nat\_stream}(\text{Gen}) :\neg\text{eng}(N, \text{nat\_from}(0, N), \text{Gen}) .\]

\[\text{nat\_from}(\text{From}, \text{To}) :\text{From} = \text{To} ; \text{succ}(\text{From}, \text{Next}), \text{nat\_from}(\text{Next}, \text{To}) .\]
Stream operations: direct sum

- **the predicate** `sum(+Gen1,+Gen2, -NewGen)` **advances by asking each generator, in turn, for an answer**
  
  \[
  \text{sum}(E1,E2,\text{sum\_next}(\text{state}(E1,E2))).
  \]

  \[
  \text{sum\_next}(\text{State},X) :- \text{State}=\text{state}(E1,E2),\text{ask}(E1,X),!,
  \text{nb\_setarg}(1,\text{State},E2),
  \text{nb\_setarg}(2,\text{State},E1).
  \text{sum\_next}(\text{state}(\_\_,E2),X) :- \text{ask}(E2,X).
  \]

- **when one generator terminates, we keep progressing in the other**

- **works on finite or infinite streams:**
  
  ?- pos(N),neg(M),sum(N,M,E),show(10,E).
  \[1,-1,2,-2,3,-3,4,-4,5,-5]\]
Stream operations: direct product, with engines

```prolog
prod(E1,E2,E) :- eng(_,prod_goal(E1,E2),E).
prod_goal(E1,E2) :- ask(E1,A), prod_loop(1,A,E1-[],E2-[]).
```

- **prod_loop** runs inside an engine, and returns pairs

```prolog
prod_loop(Ord1,A,E1-Xs,E2-Ys) :-
    flip(Ord1,Ord2,A,Y,Pair),
    forall(member(Y,Ys), engine_yield(Pair)),
    ask(E2,B), !,
    prod_loop(Ord2,B,E2-Ys,E1-[A|Xs]).
prod_loop(Ord1,_A,E1-_Xs,_E2-Ys) :-
    flip(Ord1,_Ord2,X,Y,Pair),
    X in E1, member(Y,Ys), engine_yield(Pair), fail.
```

- **flip**

```
flip(1,2,X,Y,X-Y). flip(2,1,X,Y,Y-X).
```

- **Example:**

```
?- nat(N),nat(M),prod(N,M,R),show(12,R).
[0-0, 1-0, 1-1, 0-1, 2-1, 2-0, 2-2, 1-2, 0-2, 3-2, 3-1, 3-0]
```
An algebraic view: the embedded Language interpreter

\[
F \ ::= \ F_1 + F_2 \quad \text{(sum)} \quad \mid \quad \{F\} \quad \text{(setification)} \\
\mid \quad F_1 \times F_2 \quad \text{(product)} \\
\mid \quad N : M \quad \text{(range)} \\
\mid \quad [X|Xs] | [] \quad \text{(list)} \\
\mid \quad X^G \quad \text{(engine)} \\
\mid \quad A \quad \text{(constant)} \\
\mid \quad E \quad \text{(stream)}
\]

- the language of stream expressions comprises lists, engines, ranges, constants and existing streams, as well as their sums, products and setification (i.e., removing duplicates)
- we have implemented it with a simple interpreter, the predicate \text{eval\_stream}(+\text{GeneratorExpression}, -\text{Generator}), which turns a generator expression into a ready-to-use generator that combines the effects of its components
The stream expression evaluator

- recursing over stream expressions

  eval_stream(E+F,S):- !,
  eval_stream(E,EE),eval_stream(F,EF),sum(EE,EF,S).

  eval_stream(E*F,P):- !,
  eval_stream(E,EE),eval_stream(F,EF),prod(EE,EF,P).

  eval_stream(E:F,R):- !,range(E,F,R).

  eval_stream([],L):-!,list([],L).

  eval_stream([X|Xs],L):-!,list([X|Xs],L).

  eval_stream({E},SetGen):-!,eval_stream(E,F),setify(F,SetGen).

  eval_stream(X^G,E):-!,eng(X,G,E).

  eval_stream(A,C):-atomic(A),!,const(A,C).

  eval_stream(E,E).

- the one-liner that trims duplicates, with engines

  setify(E,SE):-eng(X,distinct(X,X in E),SE).
we define \texttt{in\_}/2 as a variant of \texttt{in}/2 that takes a generator expression rather than a generator

\begin{verbatim}
:-op(800, xfx, (in\_)).
X in\_ GenExpr:-eval_stream(GenExpr, NewGen), X in NewGen.
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
?- X in\_ ([a, b, a] + (1:3)*c).
X = a ; X = 1-c ; X = b ; X = 1-c ;
X = 2-c; X = 1-c ; X = 2-c ...
\end{verbatim}
Lazy Functional Programming constructs: map and reduce

- map/3 creates a generator that applies a binary predicate to the subsequent elements in a given stream to produce a new stream:

  \[ \text{map}(F, \text{Gen}, \text{map\_next}(F, \text{Gen})) \]

  \[ \text{map\_next}(F, \text{Gen}, Y) : \text{ask}(\text{Gen}, X), \text{call}(F, X, Y). \]

- note that we do not need to explicitly iterate – like with most other stream operations, we only need to specify a single computation step!

- reduce(+Closure,+Generator,+InitialVal,-ResultGenerator) creates a generator that reduces a finite generator’s elements with a closure, starting with an initial value:

  - its only element is the resulting single final value (i.e., like Haskell’s foldl and foldr)
  - used to generically define arithmetic sums and products over a stream
Scan: collecting prefix computation results

- `scan(+Closure, +Generator, +InitialVal, -ResultGenerator)`
- similar to `reduce/4` but also yields all intermediate results
- this is also meaningful for infinite streams

```prolog
scan(F,InitVal,Gen,scan_next(state(InitVal),F,Gen)).

scan_next(S,F,Gen,R) :- arg(1,S,X),
    ask(Gen,Y),
    call(F,X,Y,R),
    nb_setarg(1,S,R).
```

- the stream of cumulative sums of the natural numbers:

```prolog
?- nat(E), scan(plus,0,E,F), show(11,F).
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55].
```
Stream Wrappers for I/O and Stateful Prolog Features

- we wrap file or socket readers as generators
- this has the advantage that details like opening a file, reading and closing a stream stay hidden, e.g., as in `term_reader`:
  
  ```prolog
  term_reader(File, next_term(Stream)) :- open(File, read, Stream).

  next_term(Stream, Term) :-
      read(Stream, X),
      ( X\==end_of_file->Term=X
       ; close(Stream), fail
      ).
  ```

- again, note that we do not need to explicitly iterate
- we only specify a single step and the end condition
Lazy Streams as Lazy Lists

- a representation for lazy streams that’s familiar to the Prolog user
- lazy lists look much like regular Prolog lists
- they can be inspected and deconstructed with unification
- for instance, DCGs can be used to parse lazy lists
Implementing streams as lazy lists

- we delay the evaluation of a computation until its result is actually needed
- the lazy list is represented as a Prolog list where the tail is formed by an attributed variable
- when inspected through unification, attributed variables trigger the computation and deliver the actual value
- building the list happens incrementally, when needed, but what has been computed can be reused

Example: the lazy list of natural numbers:

```prolog
simple.lazy_nats(List):- simple.lazy_nats_from(0,List).

simple.lazy_nats_from(N,List):- put_attr(List,simple,N).

simple:attr_unify_hook(N,Value):- succ(N,M),
  simple.lazy_nats_from(M,Tail), Value = [N|Tail].
```
The isomorphism: from/to lazy streams to/from lazy lists

- two predicates witness the isomorphism between the representations
- `gen2lazy(+Generator,-LazyList)` turns a possibly infinite stream generator into a lazy list by using the generator as the state on which the lazy list is based. It uses `ask/2` to advance that state and produce a new element.

  ```prolog
  gen2lazy(Gen,Ls):- lazy_list(gen2lazy_forward,Gen,Ls).
  gen2lazy_forward(E,E,X):-ask(E,X).
  ```

- The opposite direction is even easier, as the `list/2` generator also works on lazy lists!

  ```prolog
  lazy2gen(Xs,Gen):-list(Xs,Gen).
  ```
Transformers: defining the iso-functor

- we can easily transport not just the data representations but also the operations acting on them
- in category theory, this concept is formally known as an iso-functor, a mapping that transports morphisms between objects from one category to another and back
- the predicate `iso_fun(+Operation, +SourceType, +TargetType, +Arg1, -Result)` generically transports a predicate of the form `F(+Arg1, -Arg2)` to a domain where an operation can be performed and brings back the result
  
  ```prolog
  iso_fun(F,From,To,A,B):-call(From,A,X),call(F,X,Y),call(To,Y,B).
  ```

- we have implemented variants for several arities and modes
Borrowing \texttt{map} from lazy streams, for infinite lazy lists

- \texttt{maplist} will loop on lazy lists as its halting condition is unification against the empty list.
- When a list is infinite this never happens.
- We borrow \texttt{map} to define a lazy version of \texttt{maplist}:

\begin{verbatim}
lazy_maplist(F,LazyXs,LazyYs):-
    iso_fun(map(F),lazy2gen,gen2lazy,LazyXs,LazyYs).
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
?- lazy_nats(Ns),lazy_maplist(succ,Ns,Ps),prefix([A,B,C],Ps).
Ns = [0,1,2|__20314], Ps = [1,2,3|__20332], A=1,B=2,C=3, ...
% ^^^^^ ^^^^^
\end{verbatim}
Borrowing `lazy_sum` form lazy lists

- **direct sum proceeds by interleaving elements of the two streams**

```
sum_(E1,E2, E):- iso_fun(lazy_sum,gen2lazy,lazy2gen,E1,E2, E).

lazy_sum(Xs,Ys,Zs):- lazy_list(lazy_sum_next,Xs-Ys,Zs).

lazy_sum_next([X|Xs]-Ys,Ys-Xs,X).
lazy_sum_next(Xs-[Y|Ys],Ys-Xs,Y).
```

- **a single step of interleaving is expressed declaratively in**

`lazy_sum_next/3`
Discussion: pros and cons of lazy streams vs. lazy lists

- lazy streams: abstract sequence interface
- lazy lists: the concrete list-based view
- one can access the \( n \)-th element of a generator in \( O(1) \) space
- lazy lists might or might not need \( O(n) \) for that, depending on possible garbage collection of their unused prefix
- for stream generators no garbage collection is needed when working destructively in constant space
- their results can be exposed declaratively via the stream algebra
- lazy lists are reusable, while new generators must be created to revisit a sequence
- \( \Rightarrow \) they offer similar services, but as they interoperate with help of \texttt{iso_fun} predicates, one can choose the implementation most suitable for a given algorithm
Conclusions

- we have described a unified approach to program with finite and infinite stream generators that enhances Prolog with operations now prevalent in widely used programming languages like Python, C#, go, JavaScript, Ruby and Lua
- our streams support lazy evaluation mechanisms comparable to those in non-strict functional languages like Haskell
- as a special instance, we have defined generators based on first-class logic engines that can encapsulate both AND-streams and OR-streams of answers
- we have provided an embedded interpreter for our generator algebra to enable declarative expression of stream algorithms in the form of compact and elegant code
- we have provided a lazy list representation for our streams, which interacts nicely with unification and typical Prolog list code and an iso-functor that supports transport of operations between lazy lists and generators
Thank you!

QUESTIONS?