Lazy Stream Programming in Prolog

Paul Tarau
paul.tarau@unt.edu

Jan Wielemaker
J.Wielemaker@vu.nl

Tom Schrijvers
tom.schrijvers@cs.kuleuven.be

In recent years, stream processing has become a prominent approach for incrementally handling large amounts of data, with special support and libraries in many programming languages. Unfortunately, support in Prolog has so far been lacking and most existing approaches are ad-hoc. To remedy this situation, we present *lazy stream generators* as a unified Prolog interface for stateful computations on both finite and infinite sequences of data that are produced incrementally through I/O and/or algorithmically.

We expose stream generators to the application programmer in two ways: 1) through an abstract sequence manipulation API, convenient for defining custom generators, and 2) as idiomatic lazy lists, compatible with many existing list predicates. We define an algebra of stream generator operations that extends Prolog via an embedded language interpreter, provides a compact notation for composing generators and supports moving between the two isomorphic representations.

As a special instance, we introduce answer stream generators that encapsulate the work of coroutining first-class logic engines and support interoperation between forward recursive *AND-streams* and backtracking-generated *OR-streams*.

Keywords: lazy stream generators, lazy lists, first-class logic engines, stream combinators, AND-stream / OR-stream interoperation, Prolog extensions

1 Introduction

Initial design as well as evolution of successful programming languages often walks a fine line between semantic purity and pragmatic expressiveness. With its declarative roots and creative pragmatic additions Prolog is a long-time survivor in the complex ecosystem of programming languages. We believe that its longevity is due not only to its elegant semantics but also to its creative adaptations to emerging programming language features that respond to evolving software development requirements.

Stream processing—now prevalent in widely used programming languages languages like Java, Python, C#, go or JavaScript—offers a uniform and (mostly) declarative view on processing finite and infinite[1] sequences. Besides the expressiveness boost it provides, its advent has been driven by the need for processing big data. This big data problem manifests itself in static incarnations like very large training sets for machine learning, or as dynamic event streams coming from Web search queries and clicks, or from sensor networks supporting today's fast spreading IoT infrastructure.

The main goal of this paper is to extend Prolog with state-of-the-art lazy stream processing capabilities like those available in other languages. While some languages facilitate such an extension with features like generalized iterators (Python) or a lazy evaluation semantics (Haskell), Prolog presents two major obstacles that make this task particularly challenging.

The first obstacle is presented by Prolog's fixed depth-first search resolution and strict evaluation semantics. While Prolog's depth-first search mechanism can be complemented with alternative search strategies, as shown in [13] by overriding its disjunction operator, the evaluation mechanism remains

[1] We use "infinite" here as a short hand for data or computation streams of unpredictable, large or very large size.

Submitted to:
ICLP 2019

ultimately eager. When programming with lists or DCGs, one chains recursive steps in the body of clauses connected by conjunctions.

The second obstacle, a consequence of Prolog's incremental evolution as a programming language, is the presence of procedural state-management and I/O constructs that are interleaved with its native declarative programming constructs. These range from random generator streams to file and socket I/O and dynamic database operations. While monadic constructs in functional languages [10, 18] offer a unified view of declarative and procedural state-management operations, most logic programming languages still lack a unified approach providing a uniform interface to this mix of declarative and procedural language constructs.

We manage to overcome these obstacles and provide lazy stream processing for Prolog in a way that uniformly encapsulates different streaming mechanisms—state transformers, lazy lists and first-class logic engines [14, 17, 15], recently added to SWI-Prolog[2], into a set of operations organized compositionally in the form of *stream generators*. Our generators work in a way similar to Python's *yield* mechanism [11, 3] and they share features with coroutining constructs now present in a several programming languages including C#, go, Javascript and Lua. At the same time, they lift Prolog's expressiveness with lazy evaluation mechanisms similar to non-strict functional programming languages like Haskell [8] or functional-logic languages like Curry [1].

We organize our generators as an algebra, wrapped as a library module with a declarative interface, to avoid exposing operations requiring an implementation with a clear procedural flavor to the Prolog application programmer.

By defining a functor that transports operations between isomorphic generators and lazy lists, we offer a choice between abstract sequence operations and the concrete list view familiar to Prolog users.

The main contributions of this paper are:

- We present a simple and clean approach for setting up lazy streams, which uniformly encapsulates algorithms, lists, first-class logic engines and other data sources.
- We show how to expose lazy streams in the form of lazy Prolog lists that, just like conventional lists, can be inspected and decomposed with unification. Under the hood, lazy lists use attributed variables and destructive updates to extend the list when needed.
- We have implemented our approach in several libraries:
  1. Our `lazy_streams` library features a dozen generator predicates (stream sources), an API to query them, a set of generator operations, a generator expression interpreter offering a declarative view of these operations and an interface to the next library.
  2. Our `lazy_lists` library provides a dozen generator predicates for directly setting up lazy lists.
  3. Our `pure_input` library provides a range of predicates for reading files and sockets backed by lazy lists.

  The first is available as an SWI-Prolog library package,[3] while the other two are bundled as a SWI-Prolog standard libraries.[4]

The rest of the paper is organized as follows. Section 2 demonstrates our approach with some examples. Section 3 describes implementation of lazy stream generator constructors and their interface.

[2] `http://www.swi-prolog.org/pldoc/man?section=engines`

[3] `https://github.com/ptarau/AnswerStreamGenerators/raw/master/lazy_streams-0.5.0.zip`

[4] `http://www.swi-prolog.org/pldoc/doc/_SWI_/library/lazy_lists.pl`, `http://www.swi-prolog.org/pldoc/doc/_SWI_/library/pure_input.pl`

Section 4 introduces several operations on generators and overviews the embedded language interpreter organizing them as an algebra of generator combinators. Section 5 describes lazy functional language style generator operations and an example of I/O stream generator. Section 6 overviews implementation of lazy lists using attributed variables and introduces the iso-functor connecting them to lazy stream generators. Section 7 compares and discusses alternative implementation options of lazy list and stream generators. Section 8 overviews related work and section 9 concludes the paper. The Appendix contains benchmarks comparing implementation alternatives for finite and infinite lazy streams.

2 Overview

This section briefly introduces our lazy streams with a few examples.

The generators `pos/1` and `neg/1` produce the infinite streams of positive and negative integers. With `map/4` we combine these two streams element-wise; here they annihilate each other with `plus/3`. With `show/2` we display the first 10 elements of the resulting constant stream of zeroes.

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

With `list/2` we turn the regular list `[a,b,c]` into a stream. Then `convolution/3` computes its Cartesian product with the positive integers, following a convolution approach, and `show/2` displays the first 16 elements.

```
?- 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].
```

We also provide an embedded language interpreter to concisely express algebraic operations on streams. Here we use it to create the Cartesian product of the stream `[a,b]` with the stream `[1,2,3]`, the latter abbreviated by the shorthand `(1:4)`. The `in_/2` predicate enumerates the elements of the resulting stream through backtracking, like `member/2` does for regular lists.

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

3 Implementing Lazy Stream Generators

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.

3.1 The Stream Generator Interface

A generator is represented by a *closure* (assumed deterministic), which is a term that can be called with an additional argument to yield the next element in the stream. For instance, the generator for the constant stream of `1`s is `=(1)`, where `call(=(1),X)` instantiates `X` to the next element, which is always `1`. Typically the other arguments of the term represent the state and parameters of the generator, like `1` in `=(1)`. We require that the closure has at least one argument, which never takes the reserved value `done` in the course of its operation. When the closure has no more elements to yield, it fails.

Our `ask/2` predicate provides a basic interface to interact with generators:

```
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).
```

This code `calls` the generator to produce an element. The first time the generator fails, `stop/1` writes `done` into its first argument in a non-backtrackable fashion (and then propagates the failure). Subsequent `asks` simply read the argument with `is_done/1` and never invoke the generator again. This means that its resources can be garbage collected.

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

```
:-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.
```

Basic Stream Generators We package basic stream generators into a predicate that sets them up from given parameters. For instance, the constant stream is created by the `const/2` predicate which takes the constant value `C` as an input.

```
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.

```
rand(random).
```

We can also generate a stream by by incrementally evolving a state:

```
gen_next(F,State,X):-arg(1,State,X),call(F,X,Y),nb_setarg(1,State,Y).
```

Here `State` acts as a container for destructively updated values using the `nb_setarg/3` built-in.[5]

For instance, we can define the stream of natural numbers as an evolving state:

```
nat(gen_next(succ,state(0))).
```

The more general `gen_nextval/3` predicate supports generators for which the evolving state does not coincide with the elements of the stream.

```
gen_nextval(Advancer,State,Yield):-
  arg(1,State,X1),
  call(Advancer,X1,X2,Yield),
  nb_setarg(1,State,X2).
```

For instance, this approach is useful to turn a list into a stream.

```
list(Xs, gen_nextval(list_step,state(Xs))).

list_step([X|Xs],Xs,X).
```

We have built similar stream generators in the library package `lazy_streams`, for a *range* of numbers, turning a finite list into a an infinite cycle of its elements, as well as stream transformers excising a finite slice of a larger, possibly infinite stream, with taking or dropping an initial segment of a stream as special cases.

[5] `http://www.swi-prolog.org/pldoc/doc_for?object=nb_setarg/3`

3.2 Answer Stream Generators

We can encapsulate first class logic engines as generators when more complex computations are needed for generating the streams, that cannot be expressed as simple step-by-step state transformations.

3.2.1 SWI Prolog's First-Class Logic Engine Implementation

A first-class logic engine [14, 15] can be seen as a Prolog virtual machine that has its own stacks and machine state. In their SWI-Prolog implementation, unlike normal Prolog threads [20, 19], they are not associated with an operating system thread. Instead, one asks an engine for a next answer with the predicate `engine_next/2`. Asking an engine for the next answer attaches the engine to the calling operating system thread and causes it to run until the engine calls `engine_yield/1` or its associated goal completes with an answer, failure or an exception. After the engine yields or completes, it is detached from the operating system thread and the answer term is made available to the calling thread. Communicating with an engine is similar to communicating with a Prolog system through the terminal: the client decides how many answers it wants returned and what to do with them.

Engines are created with the built-in `engine_create/3`, that uses a goal and answer template as input and returns an engine handle as output. SWI-Prolog's engines are created with minimal dynamic stack space and are garbage collected when unreachable.

Note that implementing the engine API does not need a Prolog system that supports multi-threading. It 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*.

3.2.2 Answer Stream Generators

The predicate `eng/3` creates a generator as a wrapper for the `engine_next(Engine,Answer)` built-in, encapsulating the answers of that engine as a stream.

```
eng(X,Goal,engine_next(Engine)):-engine_create(X,Goal,Engine).
```

An alternative constructor, `ceng/3`, is also available if one wants to preserve the goal and answer template, usable, for instance, to clone the engine's answer stream, an operation that makes sense only when the Prolog code it is based on, is free of side effects.

3.2.3 The AND-stream / OR-stream Duality

A key feature of first-class engines is that they support two ways of producing a stream of answers: 1) via backtracking (OR-streams), and 2) as part of a forward moving recursive loop (AND-streams).

The stream generator abstraction makes the user oblivious to this choice of generation method, and allows us to seamlessly replace one implementation for another. Consider for instance the two implementations of the stream of natural numbers below. The first implementation generates an AND-stream yielding the elements in a recursive loop.

```
and_nat_stream(Gen):-eng(_,nat_goal(0),Gen).

nat_goal(N):-succ(N,SN),engine_yield(N),nat_goal(SN).
```

The OR-stream implementation generates the successive elements via backtracking.

```
or_nat_stream(Gen):-eng(N, nat_from(0,N), Gen).

nat_from(From,To):- From=To ; succ(From,Next),nat_from(Next,To).
```

When using engines, both AND-streams and OR-streams can be infinite, as in the case of the generators `or_nat_stream` and `and_nat_stream`. While one can see backtracking over an infinite set of answers as a "naturally born" OR-stream, the ability of the engine-based generators to yield answers from inside an infinite recursive loop is critical for generating infinite AND-streams. Because the choice of generation method is immaterial to the user of the generator, the implementor can choose the most convenient or efficient approach.

4 The Generator Algebra

This section describes a set of stream combinators exposed as an algebra via an embedded language interpreter.

4.1 Operations on Finite or Infinite Stream Generators

Sums of Streams We define the interleaving of two streams to be their sum. This operation is captured in the predicate `sum(+Gen1,+Gen2, -NewGen)`, which advances by asking each generator, in turn, for an answer. When one generator terminates, it keeps progressing in the other.

```
sum(E1,E2,sum_next(state(E1,E2))).

sum_next(State,X):-State=state(E1,E2),ask(E1,X),!,
  nb_setarg(1,State,E2),
  nb_setarg(2,State,E1).
sum_next(state(_,E2),X):-ask(E2,X).
```

For instance,

```
?- pos(N),neg(M),sum(N,M,E),show(10,E).
[1,-1,2,-2,3,-3,4,-4,5,-5]
```

We name this operation the "sum" because it is clearly associative and, if the order of the elements is unimportant (with inputs seen as *multisets*), it is also commutative. Also, it has the empty stream, defined such that `ask/2` always fails on it, as its neutral element.

Products of Streams The Cartesian product is the product operation on two streams. We can easily implement it by means of convolution in a first-class logic engine. Our implementation uses a recursive loop that supports possibly infinite stream generators by storing their finite initial segments into two lists that start out empty.

```
prod(E1,E2,E):-eng(_,prod_goal(E1,E2),E).

prod_goal(E1,E2):-ask(E1,A),prod_loop(1,A,E1-[],E2-[]).
```

The algorithm, expressed by the predicate `prod_loop`, alternates between both generators while neither is done. After that, it keeps progressing in the remaining generator until that too is exhausted. Each time a generator produces a new element, it is paired with the previously produced elements of the other generator, which are stored in a list.

```
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.
```

The predicate `flip/5` builds a pair in the appropriate order when the generators take turns being active in the recursive loop.

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

Here is the product of the natural numbers with themselves as an 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]
```

The singleton stream with a known constant (e.g., `o`) is the neutral element for the product, if we consider any element `X` to be isomorphic to `o-X` and `X-o`. Moreover, the product is associative if we ignore the association in the pair representation (e.g, `1-(2-3)` seen as equivalent to `(1-2)-3`) and commutative if the order of the elements does not matter, when interpreting our streams as multisets. Finally, under the same assumptions, the product distributes over the sum.

Note: Our `lazy_streams` package also provides the `prod_/3` stream product operation that avoids the engine creation and collection overhead, coming from the use of the constructor `eng/3` by using a *Cantor unpairing function* to split natural numbers generated by `nat/1` that are used to index dynamic arrays growing with each new element consumed from the two input streams. By using the $N \to N^k$ generalized Cantor untupling function, implemented for instance in [16], one can obtain efficient generator product operations for k generators.

4.2 An Embedded Language Interpreter

With our sum and product operations ready, we can proceed with the design of the embedded language, facilitating more complex forms of generator compositions. The grammar of this embedded language of generator expressions F is:

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

The language 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 `eval_stream(+GeneratorExpression, -Generator)`, which turns a generator expression into a ready-to-use generator that combines the effects of its components.

```
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).
```

We already explained the auxiliary predicates used in the evaluator, except for `setify/2`. That predicate wraps a generator to ensure it produces a set of answers, with duplicates removed, using the built-in `distinct/2`[6].

```
setify(E,SE):-eng(X,distinct(X,X in E),SE).
```

This works for infinite generators (within the limits of available memory), in contrast with sorting, which also removes duplicates, but assumes the list is finite.

Lastly, we define `in_/2` as a variant of `in/2` that takes a generator expression rather than a generator.

```
:-op(800,xfx,(in_)).

X in_ GenExpr:-eval_stream(GenExpr,NewGen),X in NewGen.
```

Here is a small example that combines many of the features described above:

```
?- 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 ....
```

5 Other Stream Generator Operations

This section describes the additional stream operations in our `lazy_streams` package.

5.1 Lazy Functional Programming Constructs

Map The `map/3` predicate creates a generator that applies a binary predicate to the subsequent elements in a given stream to produce a new stream.

```
map(F,Gen,map_next(F,Gen)).

map_next(F,Gen,Y):-ask(Gen,X),call(F,X,Y).
```

Our package contains similarly defined `map/N+1` generators that apply a predicate with `N arguments` to `N-1` stream generators.

Reduce The predicate `reduce(+Closure,+Generator,+InitialVal, -ResultGenerator)` creates a generator that reduces a finite generator's elements with the given closure, starting with an initial value. Its only element is the resulting single final value. Similarly to Haskell's `foldl` and `foldr`, it can be used to generically define arithmetic sums and products over a stream.

```
reduce(F,InitVal,Gen, reduce_next(state(InitVal),F,Gen)).
```

[6] `https://www.swi-prolog.org/pldoc/doc_for?object=distinct/2`

It uses the predicate `reduce_next/4` that applies closure `F` to the state `S`, while generator `E` provides "next" elements.

```
reduce_next(S,F,E,R):- \+ is_done(E),
  do((
    Y in E, arg(1,S,X),
    call(F,X,Y,Z),
    nb_setarg(1,S,Z)
  )),
  arg(1,S,R).

do(G):-call(G),fail;true.
```

Note that by working in $O(1)$ space, with destructive updates, we can handle finite streams with an arbitrary number of elements.

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

```
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).
```

For example, we can compute the stream of cumulative sums of the natural numbers.

```
?- nat(E), scan(plus,0,E,F), show(11,F).
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55].
```

5.2 Stream Wrappers for I/O and Stateful Prolog Features

We can easily wrap file or socket readers as generators. This has the advantage that details like opening a file, reading and closing a stream stay hidden, as shown by the `term_reader/2` generator below.

```
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
  ).
```

6 Lazy Streams as Lazy Lists

In addition to the abstract datatype representation for lazy streams where the user interacts with them through a dedicated API, we also provide a more concrete representation for lazy streams in the form of *lazy lists*. Lazy lists look much like regular Prolog lists and, just like regular lists, they can be inspected and deconstructed with unification. The difference with regular lists is that lazy lists are not fully materialized from the start, but that, like lazy streams, their elements are computed on demand and thus can conceptually hold infinitely many elements.

A key advantage over the abstract stream representation is that much of the existing functionality for regular lists can be reused for lazy lists; a good example are DCGs, which can also be used to parse lazy lists.

6.1 Lazy List Representation

Our lazy list representation is based on the lazy function technique of Casas et al. [5, 6]. The main idea is to delay the evaluation of a computation until its result is actually needed. Initially, the result is represented by a logic variable. When this variable is inspected through unification (the need), a coroutine mechanism (e.g., `freeze/2` or attributed variables [7]) is triggered to perform the computation and deliver the actual value just in time for the inspection.

We apply this technique to compute a list incrementally as more and more of it is needed. Thus a lazy *list* is represented as a normal Prolog list where the *tail* is formed by an attributed variable. The following code illustrates this approach on the lazy list of natural numbers.

```
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].
```

A call to `simple:lazy_nats/1` creates an attributed variable for the lazy list that can be passed to a typical list traversal predicate. Such a predicate unifies this variable with either the empty list (`[]`) or a list cell (`[Head|Tail]`), which triggers the attributed variable hook and instantiates the variable to `[N|Tail]` where `Tail` is again an attributed variable. When the list traversal predicate recurses on `Tail`, this process is repeated.

Improvement There is a significant disadvantage to the above basic approach. As already observed by Casas et al., on backtracking, the lazily computed extension of the list is lost and possibly recomputed again on the next forward computation. A pathological case is that where the list traversal first tries to unify with the empty list and then with the non-empty list. For such a predicate, every element is computed twice, a first time when the unification with the empty list fails, and a second time when the unification with the non-empty list succeeds.

In addition to the recomputation overhead, this makes the implementation unsuitable for fetching data from an external source—like a network socket—that cannot backtrack. It is possible to keep a buffer to support re-fetching content from the socket but the amount of data we need to buffer depends on the unknown non-determinism in the Prolog code that processes the list and we cannot recover if the selected buffer size proves to be too short.

We provide a solution for this problem by using *non-backtrackable assignment* in the form of SWI-Prolog' `nb_setarg/3`, which assigns an argument in a compound term and is not undone on backtracking. We illustrate this idea on the lazy list of natural numbers:

```
lazy_nats_from(N,L) :- put_attr(L,lazy_streams,state(N,_)).

attr_unify_hook(State,Value) :-State=state(N,Read),
  ( var(Read) -> succ(N,M),nats(M,Tail),
     nb_setarg(2,State,[N|Tail]),arg(2,State,Value)
  ;  Value = Read
  ).
```

Compared to the basic version, we use a compound state here, where a the `Read` field is initially free and instantiated with the resulting list structure once the value has been computed. Because of the use of `nb_setarg/3`, the information recorded in `Read` survives backtracking.

With the above technique we have implemented the `pure_input` library, which supports a lazy list view of files and sockets, as well as the generic `lazy_lists` library. The general goal to create a lazy list is `lazy_list(:Next, +State0, -List)`. This executes `call(Next, State0, State1, Head)` to produce the next element.

Lazy lists allow Prolog to handle infinite data streams in limited memory, provided that garbage collection can reclaim the already processed part of the list. This is possible if the user code does not keep a reference to the head of the list. A particular pitfall here is nondeterminism: even when the current branch no longer needs the head of the list, the runtime environment may have to hold onto it for the sake of unexplored alternative branches. Hence, non-determinism can only be mixed with lazy lists if every choicepoint is resolved (i.e., no unexplored alternatives remain) after examining only a bounded number of additional elements. If this condition is met, the attributed variable trigger, which advances the stream, and garbage collection, which reclaims the unused prefix of the list, together ensure that the in-memory window of the stream is finite.

6.2 From Lazy Streams to Lazy Lists

Now we show how to convert from lazy streams to lazy lists and back by providing an isomorphism between both representations. The former conversion is interesting because lazy lists may present a convenient, tangible representation. In contrast, the latter may be more convenient for defining new generators, and avoids confusion with regular lists. Indeed, although infinite lazy lists look like regular lists, they don't work well with all regular list predicates. For instance,

```
?- lazy_nats(Ns),maplist(succ,Ns,Ps).
... loops forever ...
```

The problem is that while the list is infinite and lazy, `maplist/3` is eager and only works on finite lists. By exploiting the isomorphism between the two representations we can easily import the lazy `map/3` from streams to get a `lazy_maplist/3`.

Isomorphism Two predicates witness the isomorphism between the representations. The predicate `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, and using `ask/2` to advance that state (which is in fact already handled by the generator), and produce a new element.

```
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.

```
lazy2gen(Xs, Gen):-list(Xs, Gen).
```

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.

```
iso_fun(F,From,To,A,B):-call(From,A,X),call(F,X,Y),call(To,Y,B).
```

We have also defined similar code for predicates with other arities and modes.

This allows us to define lazy version of `maplist`:

```
lazy_maplist(F,LazyXs,LazyYs):-iso_fun(map(F),lazy2gen,gen2lazy,LazyXs,LazyYs).
```

where `map/3` is the stream generator from Section 5.1. Here is an example of the result.

```
?- 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,...
```

Inversely, an alternative `sum_/3` operation can be implemented quite easily with lazy lists. Our `lazy_streams` package uses this technique to borrow it with help from the `iso_fun/6` predicate.

```
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).
```

7 Discussion

The abstract sequence interface of the `lazy_streams` package and the concrete list-based view provided by the `lazy_lists` library offer similar services, but as they interoperate with help of `iso_fun` predicates, one can choose the implementation most suitable for a given algorithm. For instance, one can access the nth 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. With most stream generators no garbage collection is needed when working destructively in constant space, while their results can be exposed declaratively via the stream algebra. On the other hand, lazy lists are reusable, while new generators must be created to revisit a sequence. The Appendix in the extended version of this paper[7] also shows with benchmarks that stream generators are faster than lazy lists.

Some algorithms can be most easily expressed using first-class logic engines, but avoiding engines when possible reduces memory footprint and can avoid term copying. Thus, one might ask if a predicate like `lazy_findall` could be implemented without using first class logic engines, e.g., it terms of attributed-variables, TOR [13], delimited AND-continuations [12]. This seems unlikely as these techniques are all subject to backtracking and cannot store state that survives it. First-class engines can be simulated [17], but that is impractically slow. A more promising alternative is reflecting Prolog's backtracking mechanism by implementing OR-continuations at abstract machine level.

8 Related work

A relational view of stream processing and querying has been present in the database community [9, 2], and within logic programming [4], the latter with focus on reasoning about streams. Our lazy stream generators share with Python [11] the encapsulation of a stream into a mechanism providing its elements on-demand. But, by contrast to Python's generator operations (see library *itertools*), we ensure that everything scales up to work on infinite streams. Adoption of a very similar mechanism by other widely

[7]

used languages validates the claims of enhanced expressiveness that generators can bring. The basic idea of using coroutining in the form of first-class logic engines has been present in the BinProlog system [15] as early as 1995 and the attributed variables mechanism [7] that we used to implement lazy lists, has been present even earlier, originally introduced to support constraint programming. However, putting it all together in the form of a mature API is a novel contribution of this paper, as well as the uniform view, encapsulating into an open-source library our mix of declarative and procedural implementation techniques.

9 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, while also supporting 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. Moreover, 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.

In addition, we have provided a lazy list representation for our streams, which interacts nicely with unification and typical Prolog list code. Our iso-functor supports transport of operations between lazy lists and generators, which allows us to choose the simplest or most efficient implementation of stream operations. In terms of impact, there are 83 github sites using the `lazy_lists` library, and Ogborne's analysis tools for the Enron e-mail corpus[8] already make good use of our `pure_input` library based on them. We plan to explore further applications and expose our libraries to the wider community.

Acknowledgments

This work has been partially supported by NSF grant 1423324 and by the Flemish Fund for Scientific Research (grant G0D1419N).

References

[1] Sergio Antoy (2005): *Evaluation Strategies for Functional Logic Programming*. *J. Symb. Comput.* 40(1), pp. 875–903, doi:10.1016/j.jsc.2004.12.007.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani & Jennifer Widom (2002): *Models and Issues in Data Stream Systems*. In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, ACM, New York, NY, USA, pp. 1–16, doi:10.1145/543613.543615.

[3] David M. Beazley (2009): *Python Essential Reference*, 4th edition. Addison-Wesley Professional.

[4] Harald Beck, Minh Dao-Tran, Thomas Eiter & Michael Fink (2015): *LARS: A Logic-based Framework for Analyzing Reasoning over Streams*. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, AAAI Press, pp. 1431–1438, doi:10.1016/j.artint.2018.04.003.

[5] Amadeo Casas, Daniel Cabeza & Manuel V. Hermenegildo (2006): *A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems*. In Masami Hagiya & Philip

[8] `https://github.com/Anniepoo/enron`

Wadler, editors: *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, *Lecture Notes in Computer Science* 3945, Springer, pp. 146–162, doi:10.1007/11737414_11.

[6] Amadeo Casas, Daniel Cabeza Gras & Manuel V Hermenegildo (2005): *Functional notation and lazy evaluation in Ciao*. *CICLOPS 2005*. https://cliplab.org/papers/lazy-functions-ciclops05.pdf.

[7] Christian Holzbaur (1992): *Metastructures vs. attributed variables in the context of extensible unification*. In Maurice Bruynooghe & Martin Wirsing, editors: *Programming Language Implementation and Logic Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 260–268, doi:10.1007/3-540-55844-6_141.

[8] Paul Hudak, John Hughes, Simon Peyton Jones & Philip Wadler (2007): *A History of Haskell: Being Lazy with Class*. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, ACM, New York, NY, USA, pp. 12–1–12–55, doi:10.1145/1238844.1238856.

[9] Yan-Nei Law, Haixun Wang & Carlo Zaniolo (2011): *Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams*. *ACM Trans. Database Syst.* 36(2), pp. 8:1–8:32, doi:10.1145/1966385.1966386.

[10] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Information and Computation* 93, pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[11] Guido van Rossum & Fred L. Drake (2011): *The Python Language Reference Manual*. Network Theory Ltd.

[12] Tom Schrijvers, Bart Demoen, Benoit Desouter & Jan Wielemaker (2013): *Delimited continuations for Prolog*. *Theory and Practice of Logic Programming* 13(4-5), pp. 533–546, doi:10.1017/S1471068413000331.

[13] Tom Schrijvers, Bart Demoen, Markus Triska & Benoit Desouter (2014): *Tor: Modular search with hookable disjunction*. *Science of Computer Programming* 84, pp. 101–120, doi:10.1016/j.scico.2013.05.008.

[14] Paul Tarau (2000): *Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects*. In John Lloyd, editor: *Computational Logic–CL 2000: First International Conference*, London, UK, doi:10.1007/3-540-44957-4_82. LNCS 1861, Springer-Verlag.

[15] Paul Tarau (2012): *The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines*. *Theory and Practice of Logic Programming* 12(1-2), pp. 97–126, doi:10.1017/S1471068411000433.

[16] Paul Tarau (2013): *Compact Serialization of Prolog Terms (with Catalan Skeletons, Cantor Tupling and Gödel Numberings)*. *Theory and Practice of Logic Programming* 13(4-5), pp. 847–861, doi:10.1017/S1471068413000537.

[17] Paul Tarau & Arun Majumdar (2009): *Interoperating Logic Engines*. In: *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*, Springer, LNCS 5418, Savannah, Georgia, pp. 137–151, doi:10.1007/978-3-540-92995-6_10.

[18] Philip Wadler (1993): *Monads and composable continuations*. *Lisp and Symbolic Computation*, pp. 1–17, doi:10.1007/BF01019944.

[19] Jan Wielemaker (2003): *Native Preemptive Threads in SWI-Prolog*. In Catuscia Palamidessi, editor: *Practical Aspects of Declarative Languages*, Springer Verlag, Berlin, Germany, pp. 331–345, doi:10.1007/978-3-540-24599-5_23. LNCS 2916.

[20] Jan Wielemaker, Tom Schrijvers, Markus Triska & Torbjorn Lager (2012): *SWI-Prolog*. *Theory and Practice of Logic Programming* 12, pp. 67–96, doi:10.1017/S1471068411000494.

A Benchmarks

To get a relative idea of the performance overhead of different approaches, we present here several benchmark results.

A.1 Infinite Streams

First we compare the different runtimes for accessing the Nth element in the infinite stream of natural numbers represented in different ways.

The different representations we use are the direct stream generator presented in Section 3, engines encapsulated as streams for both OR and AND answers, the direct lazy list representation of Section 6, and a lazy list version of `findall/3` in library `lazy_lists`.

Table 1 shows the benchmark results for $N = 2^{23}$, obtained on a 128GB 18-core iMacPro, with 8 GB given to SWI-Prolog to avoid stack overflows.

Clearly, the direct generator is the fastest, while the engine-based generators are about 5 times slower (with little difference between AND and OR). The lazy lists are more costly as they trigger the attributed variable hook at every element, involving a metacall and an `nb_setarg/1` operation. The lazy findall is the worst, combining the cost of attributed variables with that of engines.

A.2 Finite Streams

We also consider two benchmarks that process finite streams, in order to compare them to regular lists where the elements are not produced on demand but are already pre-computed and pre-loaded in memory. The two benchmarks both process the sequence of numbers 1 to N (with $N = 2^{23}$). The first sequentially steps to the last number, while the second backtracks over all elements.

We consider two generator approaches: the first fetches the elements from the list, while the second creates the range algorithmically. In addition we consider an engine-as-stream and a lazy findall that both backtrack through the list to produce their elements.

Table 2 shows the results. As to be expected, the list version is by far the fastest. There is a price to be paid for using generators, and as we saw above, lazy findall is even more expensive. Yet, the largest price of all is to compute the elements algorithmically on demand. When backtracking through all elements, the overhead is lower than when going straight for the last element.

But, more importantly, our benchmarks show that differences between implementations are all within a $O(1)$ time complexity margin from each other. We believe that the (fairly small) constant overhead is acceptable, in exchange for supporting lazy stream operations in a way that is as convenient as possible to the application programmer.

	Time (s)	Relative (×)
generator	3.0	1.00 ×
OR engine	16.5	5.50 ×
AND engine	15.6	5.20 ×
lazy list	18.8	6.27 ×
lazy findall	28.6	9.53 ×

Table 1: Benchmark results for accessing the 2^{23}th element in an infinite stream.

	Last Element		All Elements	
	Time (s)	Relative (×)	Time (s)	Relative (×)
list	0.1	1 ×	0.5	1.0 ×
list generator	3.8	38 ×	3.1	6.2 ×
range generator	26.7	277 ×	28.1	56,2 ×
OR engine	3.3	33 ×	3.7	7,4 ×
lazy findall	13.7	137 ×	15.9	31,8 ×

Table 2: Benchmark results for processing a stream of 2^{23} elements.