Programming with Answer Stream Generators

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Answer Stream Generators encapsulate the work of coroutining logic engines as an algebra of stream operations that extends Prolog with lazy evaluation and support for infinite answer streams. We implement this algebra on top of SWI-Prolog's Engine API and design an embedded language interpreter providing a compact notation for its composition mechanisms. Answer Stream Generators can be used to unify encapsulation of state originating from declarative constructs, procedural side effects and input-output interactions. They enhance the expressiveness of Prolog with language constructs comparable to generators in languages like Python, go, C#, Ruby or Lua, and language constructs implicitly available in non-strict functional programming languages like Haskell.

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 surviver in the complex ecosystem of programming languages. The goal of this paper is dual-purpose in this sense: we encapsulate an aspect of a pragmatic language extension, first-class logic engines (Tarau 1999; Tarau 2000; Tarau 2008a; Tarau 2008b; Tarau 2012), into a set of operations organized compositionally in the form of Answer Stream Generators, language constructs similar to Python's generators and sharing features with coroutining constructs now present in a several other widely used programming languages.

TODO: expand

Motivation built around some of Prolog's expressiveness limitations:

- procedural state-management constructs
- fixed depth-first and strict evaluation semantics

Reasons for some of the new things we propose:

- what generators can bring in languages like Python
- what lazy evaluation brings to Haskell
- we want to add streams to handle potentially very large amounts of data flowing over the internet
- the need to update Prolog's language constructs to closely match new developments adopted by some of today's most successful programming languages (e.g. mention Python's yield C++ fibers, go-language's goroutines)

2 SWI Prolog's engine implementation

An engine can be seen as a Prolog virtual machine that has its own stacks and machine state. Unlike normal SWI-Prolog threads (Wielemaker et al. 2012; Wielemaker 2003) though, they are

not associated with an operating system thread. Instead, you ask 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 cause 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 though the terminal.

Implementing the engine API dies not assume that a Prolog system supports multi-threading, It only assumes that the virtual machine is fully reentrant, it can be queried and it can stop, yield data and resumes execution as a coroutine.

3 The Answer Stream Generator API

We design Answer Stream Generators as a wrapper around the SWI-Prolog engine implementation. We also keep the goals and answer templates that started the engine to make them reusable. At the same time this external wrapper ensures that engines stay garbage collectable on termination.

no need to actually place all the upcoming code in the paper - but it is there for now to quickly explain what is going on

We start by describing the Answer Stream Generators API. The predicate `new_generator` holds when a new generator has been created consisting of a new engine, a goal and its answer template, the latest two needed for restarting the engine.

```
new_generator(X,G,engine(E,X,G)):-engine_create(X,G,E).
```

The predicate `clone_generator` creates a new generator from a generator's goal and answer template. If the goal's execution is side-effect free, the clone will behave the same way as the cloned generator.

```
clone_generator(engine(_,X,G),engine(E,X,G)):-engine_create(X,G,E).
```

The predicate `stop_generator` stops a generator and allows reclamation of its resources but it will signal that it is tt done on any future calls to it. It offers a few variants, including a "self-destruct" predicate of arity 0 .

```
stop_generator(E,R):-is_done_generator(E),!,R=E.
stop_generator(engine(E,X,G),engine(done,X,G)):-engine_destroy(E).

stop_generator(E):-stop_generator(E,_).

stop_generator:-engine_self(E),engine_destroy(E).
```

The predicate `ask_generator` queries a generator unless its engine is `done`. It marks the generator as `done` after its first failure. *This ensures an engine can be garbage collected by making its handle unreachable.*

```
ask_generator(engine(done,_,_),_):-!,fail.
ask_generator(engine(E,_,_),X):-engine_next(E,A),!,X=A.
ask_generator(Estate,_):-nb_setarg(1,Estate,done),fail.
```

The predicate `generate_answer` allows a generator to produce answers from inside a forward moving loop. As such it can be used to expose an **AND-stream** seen as progressing in a linear

recursive loop, as an **OR-stream**, undistinguishable from answers generated by an engine on backtracking.

```
generate_answer(X):-engine_yield(X).
```

The predicate `is_done_generator` succeeds if a generator is marked with `done` on its first failure.

```
is_done_generator(engine(done,_,_)).
```

4 The AND-stream / OR-stream Duality

We call *AND-stream / OR-stream duality* the ability to generate answer streams via backtracking (OR-streams) or as part of a forward moving recursive loop (AND-streams). As the examples that follow will, show, being oblivious to the choice of generation method they encapsulate is a key contributor to the "expressiveness lift" answer generators bring to Prolog.

The simplest example of *AND-stream* is implemented by the generator `nat`. It defines the infinite stream of natural numbers by yielding an answer at each step of a recursive loop.

```
nat(E):-new_generator(_,nat_goal(0),E).

nat_goal(N):-
  SN is N+1,
  generate_answer(N),
  nat_goal(SN).
```

Alternatively, one could define and equivalent generator `nat_` as an OR-stream, with answers produced via backtracking.

```
nat_(E):-new_generator(N, between(0,inf,N), E).
```

Note also that both AND-streams and OR-streams can be infinite, as in the case of the generators `nat` and `nat_`. While one can see backtracking over an infinite set of answers as a "naturally born" OR-stream, the ability of the generators to yield answers from inside an infinite recursive loop is critical for generating infinite AND-streams.

Note also that generating an answer stream by either of the above methods is immaterial to the user of the generator which can be seen as a "black box".

The predicate `in/2`, defined as an `xfx` operator, "morphs" iteration over a stream of answers into backtracking.

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

X in E:-ask_generator(E,A),select_from(E,A,X).

select_from(_,A,A).
select_from(E,_,X):-X in E.
```

Example 1
The combined effect of the two engines turns the linear-recursive loop of `nat`, generating an AND-stream of answers into an OR-stream, thus, from the outside, one can explore them via backtracking. Note that `nat` is an infinite stream and so is its view from outside.

```
?- nat(E),X in E.
E = engine(<engine>(3,0x7f825943fbe0), _2248, nat_goal(0)),
X = 0 ;
E = engine(<engine>(3,0x7f825943fbe0), _2248, nat_goal(0)),
X = 1 ;
E = engine(<engine>(3,0x7f825943fbe0), _2248, nat_goal(0)),
X = 2 ;
...
```

Sequences enumerated on backtracking can be encapsulated as generators to provide a library of "built-in generators". The predicate `range/2` (with thanks to Python for the name!) creates a generator over a finite interval, closed on the left and open on the right:

```
range(From,To,E):-Max is To-1,new_generator(I,between(From,Max,I),E).
```

Assuming a default range from 0 we add:

```
range(To,E):-range(0,To,E).
```

5 The Operational Equivalence between Answer Stream Generators and Lazy Lists

One can turn lists into generators and (finite) generators into lists quite easily.

```
list2generator(Xs,engine(E,X,G)):-G=member(X,Xs),engine_create(X,G,E).

finite_generator2list(E,Xs):-findall(X,X in E,Xs).
```

Example 2

```
?- list2generator([a,b,c],E),finite_generator2list(E,Xs).
E = engine(done, _4962, member(_4962, [a, b, c])),
Xs = [a, b, c].
```

Moreover, if one uses SWI-Prolog's `lazy_lists` library, generators can be turned into finite or infinite lists and vice-versa. In fact, `list2generator` works as is on infinite lazy lists. In the case of `generator2list` we just replace `findall` with `lazy_findall`.

```
generator2list(E,Xs):-lazy_findall(X,X in E,Xs).
```

One can observe the two sides of the transformation after defining `lazy_nats` as follows.

```
lazy_nats(Xs):-lazy_findall(X,between(0,infinite,X),Xs).
```

Example 3
Equivalence between infinite generators and infinite lazy lists.

```
?- lazy_nats(Xs),list2generator(Xs,E),generator2list(E,Ys),member(Y,Ys).
E = engine(<engine>(20,0x7ff7a7502140), _14308, member(_14308, Xs)),
Ys = [0|_14334],
Y = 0,
lazy_list(lazy_lists:lazy_engine_next(<engine>(19,0x7ff7a7501b60), 1), Xs),
lazy_list(lazy_lists:lazy_engine_next(<engine>(21,0x7ff7a75021e0), 1), _14334) ;
E = engine(<engine>(20,0x7ff7a7502140), _14412, member(_14412, Xs)),
Ys = [0, 1|_14444],
Y = 1,
lazy_list(lazy_lists:lazy_engine_next(<engine>(19,0x7ff7a7501b60), 1), Xs),
```

```
lazy_list(lazy_lists:lazy_engine_next(<engine>(21,0x7ff7a75021e0), 1), _14444) ;
E = engine(<engine>(20,0x7ff7a7502140), _14516, member(_14516, Xs)),
Ys = [0, 1, 2|_14554],
Y = 2,
...
```

In fact, one can also use two predicates exported by the library `lazy_lists` to bypass the extra engine created by `lazy_findall` as follows.

```
generator2list_(engine(E,_,_),Xs):-lazy_list(lazy_engine_next(E, 1), Xs).
```

One can lift this equivalence between data objects to one between predicates using them via higher-order constructs like the following.

```
lend_operation_to_lazy_lists(Op,Xs,Ys,Zs):-
  list2generator(Xs,E1),
  list2generator(Ys,E2),
  call(Op,E1,E2,E3),
  generator2list(E3,Zs).

lazy_list_sum(Xs,Ys,Zs):-lend_operation_to_lazy_lists(dir_sum,Xs,Ys,Zs).

lazy_list_prod(Xs,Ys,Zs):-lend_operation_to_lazy_lists(cart_prod,Xs,Ys,Zs).
```

6 The Generator Algebra

We start with the simpler case of finite generators.

6.1 Operations on Finite Generators

Combining finite generators is easy, along the correspondence between direct sum and disjunction.

```
fin_dir_sum(E1,E2,E):-new_generator(R, (R in E1 ; R in E2), E).
```

The same applies to the cartesian product of finite generators, except that we need to clone the second generator to produce answers for each answer of the first. The reader familiar with linear logic might observe here that answer streams are by default usable only once and that an explicit copy operator is needed otherwise.

```
fin_cart_prod(E1,E2,E):-new_generator(R, fin_cart_prod_goal(E1,E2,R), E).

fin_cart_prod_goal(E1,E2,X-Y):-
  X in E1,
  clone_generator(E2,Clone),
  Y in Clone.
```

Example 4
Finite sum and product generators act as follows:

```
?- list2generator([a,b,c],E),range(0,3,F),fin_dir_sum(E,F,G),generator2list(G,Xs).
E = engine(<engine>(4,0x7f893803abf0), _5378, member(_5378, [a, b, c])),
F = engine(<engine>(6,0x7f893803af70), _5406, between(0, 2, _5406)),
G = engine(done, _5434, fin_dir_sum_goal(engine(<engine>(4,0x7f893803abf0), _5378,
```

```
member(_5378, [a, b, c])),engine(<engine>(6,0x7f893803af70), _5406,
between(0, 2, _5406)), _5434)),
Xs = [a, b, c, 0, 1, 2].
```

6.2 Extension to Infinite Answer Generators

In the case of possibly infinite generators, we ask each generator for one answer inside a linear recursive loop. We ensure that termination only happens if both generators terminate.

```
dir_sum(E1,E2,engine(E,X,G)):-
  G=dir_sum_goal(E1,E2,X),
  engine_create(X,G,E).

dir_sum_goal(E1,E2,X):-
  ( ask_generator(E1,X)
  ; ask_generator(E2,X)
  ; \+ (is_done_generator(E1),is_done_generator(E2)),
    dir_sum_goal(E1,E2,X)
  ).
```

Designing the recursive loop for possibly infinite products proceeds with a loop that will need to store finite initial segments of the generators as they grow into two lists, initially empty.

```
cart_prod(E1,E2,engine(E,X,G)):-
  G=cart_prod_goal(E1,E2),
  engine_create(X,G,E).

cart_prod_goal(E1,E2):-
  ask_generator(E1,A),
  cart_prod_loop(1,A,E1-[],E2-[]).
```

The algorithm, expressed by the predicate `cart_prod_loop` switches between generators while none of them is done. After that, it keeps progressing the active generator for new pairs, including those selecting an element from the stored lists of the terminated generator.

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

The predicate `flip` ensures correct order in a given pair as generators take turn being the active one in the recursive loop.

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

Example 5
Working with infinite sums and products. The slice/3 predicate (to be defined later) allows limiting output to a finite initial segment.

```
?- nat(N),nat(M),dir_sum(N,M,E),slice(E,0,6,S),X in S,writeln(X),fail.
0
0
1
1
2
2
false.

?- nat(N),nat(M),cart_prod(N,M,E),slice(E,0,6,S),X in S,writeln(X),fail.
0-0
1-0
1-1
0-1
2-1
2-0
false.
```

6.3 An embedded language interpreter for the algebra

With our sum and product operations ready, we can proceed designing the embedded language facilitation more complex forms of generator compositions.

6.3.1 Deep Cloning

As generators are by default single-use, given that we store a goal and its answer template together with the engine activating it (assuming that the code is free of side-effects), it makes sense to define a deep cloning operation that creates a fresh equivalent engine from possible nested engines that may have "spent their fuel". Thus, besides sum and product the cloning algorithm will also need to support linear-logic inspired "`!`" operator. Note that sequences represented as lists that being immutable do not require copying.

```
:-op(100,fx,(!)).

deep_clone(engine(_,X,G),engine(CE,X,G)):-engine_create(X,G,CE).
deep_clone(E+F,CE+CF):-deep_clone(E,CE),deep_clone(F,CF).
deep_clone(E*F,CE*CF):-deep_clone(E,CE),deep_clone(F,CF).
deep_clone(!E,CE):-deep_clone(E,CE).
deep_clone([X|Xs],[X|Xs]).
```

6.3.2 Working with Sets

If generators work over sets rather than multisets or arbitrary sequences, duplicates need to be removed. This can be done either by sorting (which has the advantage of providing a canonical representation, but assumes finite streams) or by using a built-in like `distinct/2` which will also work with infinite generators (within the limits of actual memory available).

The predicate `setify` wraps a generator to ensure it produces a set of answers, with duplicates removed.

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

In fact, one could just apply the same modification to the goal of a generator and its answer template, but our assumption here is that the generator's engine might be a composition of several engine operations, possibly already in progress.

We can implement our generator algebra as an embedded language via a simple interpreter (although partial evaluation can make this more efficient).

```
eeval(engine(E,X,G),R):-!,R=engine(E,X,G).
eeval(E+F,S):-eeval(E,EE),eeval(F,EF),dir_sum(EE,EF,S).
eeval(E*F,P):-eeval(E,EE),eeval(F,EF),cart_prod(EE,EF,P).
eeval(!E,Bang):-deep_clone(E,DeepE),eeval(DeepE,Bang).
eeval({E},SetGen):-eeval(E,F),setify(F,SetGen).
eeval([X|Xs],E):-list2generator([X|Xs],E).
```

Similarly to the `in/2` predicate, we can make the action of the interpreter transparent, via the `in_/2` predicate, also defined as an operator.

```
:-op(800,xfx,(in_)).
X in_ E:-eeval(E,EE),X in EE.
```

Example 6
Applying the embedded language interpreter.

```
?- list2generator([a,b],E),forall(X in_ E +(!E * !E), writeln(X)).
a
a-a
b
b-a
b-b
a-b
E = engine(<engine>(7,0x7fe9b80517a0), _2298, member(_2298, [a, b])).

?- forall(X in_ {[a,b] + [b,a]}, writeln(X)).
a
b
true.
```

6.4 Some algebraic properties of sums and products

- monoid structure for + and *
- distributivity
- commutativity and associativity if the generated sequences represent sets

6.5 Lazy Functional Programming Constructs

6.6 Map-reduce mechanisms

The predicate `map_generator/3` creates a new generator that applies a predicate with 2 arguments to the answer stream of a generator.

```
map_generator(F,E,NewE):-new_generator(Y,map_goal(F,E,Y),NewE).

map_goal(F,E,Y):-X in E,call(F,X,Y).
```

The predicate `map_generator/4` creates a new generator that applies a predicate with 3 arguments to the answer stream of a generator. Note also that in case the answer-counts of the two streams are not the same, the one still active is stopped to make it subject to garbage collection.

```
map_generator(F,E1,E2,NewE):-new_generator(_,map_goal2(F,E1,E2),NewE).

map_goal2(F,E1,E2):-
  ( ask_generator(E1,X1)->Ok1=true;Ok1=fail),
  ( ask_generator(E2,X2)->Ok2=true;Ok2=fail),
  ( Ok1,Ok2->call(F,X1,X2,R),
    generate_answer(R),
    map_goal2(F,E1,E2)
  ; \+Ok1,Ok2->stop_generator(E2),fail
  ; Ok1,\+Ok2->stop_generator(E1),fail
  ).
```

The predicate `zipper_of` specializes `map_generator/4` to create pairs of answers produced by the 2 generators.

```
zipper_of(E1,E2,E):-map_generator(zip2,E1,E2,E).

zip2(X,Y,X-Y).
```

The predicate `reduce_with`, similar to `fold` in functional languages applies repeatedly an operation on the stream of answers of a generator.

It can be made to work in constant space, by backtracking internally over a given generator.

```
reduce_with(Op,E,R):-
  ask_generator(E,First),
  Res=result(First),
  ( Y in E,
    arg(1,Res,X),
    call(Op,X,Y,R),
    nb_setarg(1,Res,R),
    fail
  ; arg(1,Res,R)
  ).
```

It can be wrapped into a generator, to support our algebra of generators in a uniform way.

```
reducer(Op,E,NewE):-new_generator(R,reduce_with(Op,E,R),NewE).
```

Example 7

```
?- range(5,E),reducer(plus,E,NewE),X in NewE.
E = engine(<engine>(4,0x7ff879d64130), _3120, between(0, 4, _3120)),
NewE = engine(<engine>(3,0x7ff879d643f0), _3156, reduce_with(plus,
engine(<engine>(4,0x7ff879d64130), _3120, between(0, 4, _3120)), _3156)),
X = 10 ;
false.
```

- TODO mention here reducing a direct sum of generators with a min operation - e.g. for finding the best price among a stream of answers representing several vendors for the same product

6.7 Slicing Operations on Answer Generators

```
slice(E,From,To,NewE):-new_generator(X,in_slice_of(E,From,To,X),NewE).

in_slice_of(E,From,To, X):-
  From>=0,From<To,
  Ctr=c(0),
  X in E,
  arg(1,Ctr,K),K1 is K+1,nb_setarg(1,Ctr,K1),
  (
    K<From->fail
  ; K>=To->stop_generator(E),!,fail
  ; true
  ).

take(K,E,NewE):-slice(E,0,K,NewE).

drop(K,E,NewE):-slice(E,K,inf,NewE).
```

As an example, putting it all together, let's generate Pythagorean natural numbers X,Y,Z such that $X^2+Y^2=Z^2$.

```
pythagoras(Triplets):-
  nat(M),
  nat(N),
  cart_prod(M,N,Pairs),
  map_generator(mn2xyz,Pairs,Triplets).
```

For efficiency, we will use here the parametric solutions $X=M^2-N^2, Y=2MN, Z=M^2+N^2$.

```
mn2xyz(M-N,X^2+Y^2=:=Z^2):-N>0,M>N,
  X is M^2-N^2,
  Y is 2*M*N,
  Z is M^2+N^2.
```

We can extract a sample of the stream by creating a slice generator, together with an assertion testing its correctness as follows.

```
pythagorean_slice:-
  pythagoras(P),
  slice(P,100001,100005,S),
  forall(R in S,(assertion(R),writeln(R))).
```

Example 8
The result of extracting a slice from an infinite stream of Pythagorean triples.

```
?- pythagorean_slice.
184575^2+113792^2=:=216833^2
184828^2+112896^2=:=216580^2
185079^2+112000^2=:=216329^2
185328^2+111104^2=:=216080^2
true.
```

7 Wrappers on I/O and stateful procedural Prolog constructs

- e.g. a file or socket reader can be wrapped as a generator

- event streams can be seen as generators
- as an example, clause/2, wrapped as an engine allows exploring a predicate in a froward loop rather then on backtracking

TODO: wrap up a file reader as an engine - maybe from the pio package?

8 Engines and Multithreading? - if space permits

TODO: discuss similarities and differences, possibly show an example where engines implement cooperative multitasking or coordination

9 Engine Implementation Support

TODO: discuss SWI-Prolog's lightweight engine implementation

- mention also that pseudo-engines, implemented more efficiently with state preserved with things like nb_setarg (e.g. counters or aggregates) can be encapsulated as special instances of the engine or generator API
- such pseudo-engines can mimic generators the same way as a C-based built-in like `memberchk` emulates an equivalent Prolog definition
- mention also expressing a similar set of operations with OR-continuations?

10 Discussion

- generators vs. lazy list
  - — one can access the N-th element of a generator in $P(1)$ space
  - — lazy lists need $O(N)$ for the same
  - — but, lazy lists are reusable while generators need explicit cloning
  - — lazy lists operate via a convenient but *concrete* list syntax
  - — generators represent *abstract* lazy sequences on which one operates directly (via a somewhat procedural API) or declaratively via an algebra encapsulated as an embedded language
- discuss similarities of purpose with SWI-Prolog's solution sequences API
- can generators be combined with the **tor** library (Schrijvers et al. 2014) to make more efficient algorithms when working on infinite streams or implementing pseudo-engines?
- is the linear logic view of generators needed or one should make the cloning operation automatic
- should deep cloning be computed symbolically by applying program transformation directly to the goals encapsulated by the engines

11 Related work

Maybe?

- some history - see (Tarau 1999; Tarau 2000; Tarau 2008a; Tarau 2008b) (Tarau 2011) (Tarau 2012)
- work on delimited continuations (SCHRIJVERS et al. 2013), hookable disjunction (Schrijvers et al. 2014)
- work on pipelines (Pieters and Schrijvers 2019)

TODO: discuss alternative ways to achieve the same in Prolog e.g. tor (Schrijvers et al. 2014)

TODO: briefly discuss and compare with Python and mention other generator and coroutine implementations e.g. goroutines, fibers, generator libraries for other languages etc.

OPEN QUESTION: can a similar answer generator API be implemented it terms of attributed-variables, TOR, delimited continuations? That would avoid creating new virtual machines. A different semantics would need to be faced: would such streams be subject to backtracking. Is that good or bad? a way to do that would be to attach a goal+template as attributes of a variable and then have the unify hook make the goal call itself with a new argument, while extracting its "answer" in a lazy_findall-like mechanism. Thus the simpler question becomes: can something like lazy_findall be implemented without engines?

12 Conclusion

Generators make Prolog more expressive, in line with developments in today's popular functional and procedural languages, by supporting infinite answer streams and lazy functional programming operations.

They give the programmer a sequence or set abstraction that allows organizing sequence processing as AND-streams or OR-streams of answers.

Generator cloning, along the lines of similar linear logic operations, offers the programmer a choice between flexibility and optimal resource utilization.

The generator algebra together with its the embedded language interpreter supports the writing compact and elegant code. TODO: expand

References

PIETERS, R. P. AND SCHRIJVERS, T. 2019. Faster coroutine pipelines: A reconstruction. In *21st International Symposium on Practical Aspects of Declarative Languages (PADL 2019)*.

SCHRIJVERS, T., DEMOEN, B., DESOUTER, B., AND WIELEMAKER, J. 2013. Delimited continuations for prolog. *Theory and Practice of Logic Programming 13,* 4-5, 533546.

SCHRIJVERS, T., DEMOEN, B., TRISKA, M., AND DESOUTER, B. 2014. Tor: Modular search with hookable disjunction. *Science of Computer Programming 84*, 101 – 120. Principles and Practice of Declarative Programming (PPDP 2012).

TARAU, P. 1999. Multi-Engine Horn Clause Prolog. In *Proceedings of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, G. Gupta and E. Pontelli, Eds. Las Cruces, NM.

TARAU, P. 2000. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In *Computational Logic–CL 2000: First International Conference*, J. Lloyd, Ed. London, UK. LNCS 1861, Springer-Verlag.

TARAU, P. 2008a. Logic Engines as Interactors. In *Logic Programming, 24-th International Conference, ICLP*, M. Garcia de la Banda and E. Pontelli, Eds. Springer, LNCS, Udine, Italy, 703–707.

TARAU, P. 2008b. Interactors: Logic Engine Interoperation with Pure Prolog Semantics. In *Proceedings of CICLOPS 2008, 8th International Colloquium on Implementation of Constraint and LOgic Programming Systems*, M. Carro and B. Demoen, Eds. 17–32.

TARAU, P. 2011. Coordination and Concurrency in Multi-engine Prolog. In *COORDINATION*, W. D. Meuter and G.-C. Roman, Eds. Lecture Notes in Computer Science, vol. 6721. Springer, Berlin Heidelberg, 157–171.

TARAU, P. 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, 97–126.

WIELEMAKER, J. 2003. Native preemptive threads in SWI-Prolog. In *Practical Aspects of Declarative Languages*, C. Palamidessi, Ed. Springer Verlag, Berlin, Germany, 331–345. LNCS 2916.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12*, 67–96.

Appendix

not to be included in the paper, added here just to make a few tests get into the Prolog file extracted from the LaTeX document

```
t1:-
  range(0,3,E1),
  %range(10,14,E2),
  list2generator([a,b,c],E2),
  range(100,102,E3),
  eeval(E1+E3+E2+!E3, E),
  forall(X in E,writeln(X)).

t2:-
  range(0,2,E1),
  list2generator([a,b,c],E2),
  eeval(E1*E2, E),
  forall(X in E,writeln(X)).


t3:-
  range(0,3,E1),
  %range(10,14,E2),
  list2generator([a,b,c],E2),
  range(100,102,E3),
  eeval(E1*E3+E2*!E3, E),
  forall(X in E,writeln(X)).

t4:-
  range(3,E),
```

```
  eeval(!(E*E)*!(E+E)*E,NewE),
  forall(X in NewE,ppp(X)).

t5:-
 eeval([the]*[cat,dog,robot]*[walks,runs],E),eeval(!E,EE),
  stop_generator(E,_),
  forall(X in EE,ppp(X)).

t6:-nat(E),take(4,E,F),forall(X in F,ppp(X)).

t7:-range(20,E),drop(15,E,F),forall(X in F,writeln(X)).

t8:-range(5,E),map_generator(succ,E,NewE),forall(X in NewE,writeln(X)).


t9:-range(0,10,E1),range(100,110,E2),
    dir_sum(E1,E2,E),
    forall(R in E,writeln(R)).

t10:-range(0,10,E1),range(100,110,E2),
    map_generator(plus,E1,E2,E),
    forall(R in E,writeln(R)).

t11:-range(0,10,E1),
    list2generator([a,b,c,d],E2),
    zipper_of(E1,E2,E),
    forall(R in E,writeln(R)).


t12:-range(10,E),reduce_with(plus,E,Sum),writeln(Sum).

go:-
   member(G,[t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12]),
   ( nl,writeln(G),G,fail
   ; current_engine(E),ppp(E),fail
     % tests that all stays clean
   ).

% helpers
ppp(X):-portray_clause(X).
```

One can also transfer these operations to use a concrete lazy list syntax as follows.

```
% adaptor for lazy_list syntax

% finite printer for generators
ppg(K,E):-take(K,E,F),forall(X in F,ppp(X)).

% finite printer for lazy lists
ppl(K,Xs):-findnsols(K,X,member(X,Xs),Rs),!,ppp(Rs).

% positive integers
pos(Xs) :-lazy_findall(X, between(1, infinite, X), Xs).

% negative integers
```

```
neg(Xs) :-lazy_findall(X, (between(1, infinite, X0), X is -X0), Xs).

lazy_take(K,Xs,Ys):-
   list2generator(Xs,E1),
   take(K,E1,E2),
   generator2list(E2,Ys).

% etc.
% this can be generalized with higher order mapping predicates
% like one would do in  Haskell

l1:-pos(Ns),neg(Ms),lazy_list_sum(Ns,Ms,Xs),ppl(20,Xs).
l2:-pos(Ns),neg(Ms),lazy_list_prod(Ns,Ms,Xs),ppl(20,Xs).
```

One can test this port form engines to list as follows:

```
?- l1.
[1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 7, -7, 8, -8, 9, -9, 10, -10].
true.

?- l2.
[1- -1, 2- -1, 2- -2, 1- -2, 3- -2, 3- -1, 3- -3, 2- -3, 1- -3, 4- -3, 4- -2, 4- -1, 4- -4, 3- -4, 2- -4,
```

TODO This actually exhibits a nice isomorphism - thus the algebra of generators can be mechanically transferred to an algebra of lazy lists.