Logic Engines as Interactors

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. We introduce a new programming language construct, *Interactors*, supporting the agent-oriented view that programming is a dialog between simple, self-contained, autonomous building blocks.
We define *Interactors* as an abstraction of answer generation and refinement in *Logic Engines* resulting in expressive language extension and metaprogramming patterns, including emulation of Prolog's dynamic database.
A mapping between backtracking based answer generation in the callee and "forward" recursion in the caller enables interaction between different branches of the callee's search process and provides simplified design patterns for algorithms involving combinatorial generation and infinite answer streams.
Interactors extend language constructs like Ruby, Python and `C#`'s multiple coroutining block returns through *yield* statements and they can emulate the action of monadic constructs and catamorphisms in functional languages.
Keywords: *generalized iterators, logic engines, agent oriented programming language constructs, interoperation with stateful objects, metaprogramming*

1   Introduction

Interruptible Iterators are a new Java extension described in [11, 12]. The underlying construct is the `yield` statement providing multiple returns and resumption of iterative blocks. It has been integrated in newer Object Oriented languages like `Ruby` [14, 19] `C#` [16] and `Python` [28] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [10] and ICON [6].

Our next stepping stone is the more radical idea of allowing clients to communicate to/from inside blocks of arbitrary recursive computations. The challenge is to achieve this without the fairly complex interrupt based communication protocol between the iterator and its client described in [11]. As a natural generalization, the need arises for a structured two-way communication between a client and the usually autonomous service the client requires from a given language construct, often encapsulating an independent component.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [15, 5] have

made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At a more theoretical level, it has been argued that *interactivity*, seen as fundamental computational paradigm, can actually expand computational expressiveness and provide new models of computation [32].

In a logic programming context, the Jinni agent programming language [23, 25] and the BinProlog system [24] have been centered around logic engine constructs providing an API that supported reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [22], that have allowed the separation of the first-order language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins.

Building upon the *Fluents* API described in [22], this paper will focus on bringing interaction-centered, agent oriented constructs from software design frameworks and design patterns to programming language level.

The resulting language constructs, that we shall call *Interactors*, will express control, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog builtins, including dynamic database operations.

As paradigm independent language constructs, *Interactors* are a generalization of *Coroutine Iterators* [10] and *Interruptible Iterators* [11].

Their extra expressiveness comes from the fact that they embed arbitrarily nested computations and provide linearized data exchange mechanisms with their internal components. In particular, a mapping between backtracking based answer generation in the callee, and forward recursion in the caller, facilitating interaction between different branches of the callee's search process, becomes possible.

As the operation of interrupting iterators already has a fairly complex semantics, interrupting general computational process is even trickier and more likely to be unsafe, as their state is unknown to the clients. Therefore, Interactors are designed to support cooperative data exchanges rather then arbitrary interrupts.

Independently, the need for *state representation with minimal new ontology* in declarative languages arises from seeking simplified *interoperation* with I/O and conventional software and operating system services that often relay on stateful entities. In this sense, Interactors solve for Logic Programming languages problems similar to the problems that constructs like Monads and Catamorphisms solve for Functional Languages [2, 8].

2   First Class Logic Engines

To make the paper self-contained, we will start with an overview of the *Logic Engine API* introduced in [22] to which our *Interactor API* is a natural extension.

2.1 Engines as a Reflection Layer

Speaking generically, an *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution [26, 21] on a given clause database, together with a set of built-in operations.

Each *Logic Engine* has a constructor which initializes it with a *goal* and an *answer pattern*. In fact, an engine can be seen as a generator of a (possibly infinite) *stream of answers* which can be explored one by one, i.e. an *Iterator* over a stream of answers. To use a simple analogy, the object encapsulating the state of the runtime interpreter is very similar to a file descriptor encapsulating the advancement of a file reader.

Logic Engines will have the ability to create and query other Logic Engines, as part of a general mechanism to manipulate `Interactors`.

`Interactors` encapsulating logic engines, like any other stateful objects, will have their independent life-cycles. This general mechanism will allow Logic Engines to interoperate with the underlying imperative implementation language, which provides them and requests from them various services through a hierarchy of `Interactors`.

Each Logic Engine based Interactor works as a separate Horn Clause LD-resolution interpreter. The engine constructor, when called, initializes a new, lightweight interpreter, having its own stacks and heap.

The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand.

```
get(Interactor,AnswerInstance)
```

It tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. Note that once the atom `no` has been returned, all subsequent `get/2` operations on the same `Interactor` will return `no`. As in the case of `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then

backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the `stop/1` operation (that is also called automatically when no more answers can be produced):

```
stop(Interactor)
```

So far, these operations provide a minimal *Coroutine Iterator API*, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [22], as well as other higher order operations similar to Haskell's *fold* (subsection 3.3).

These interactor operations correspond to the *Answer Source* fluents described in [22], where a complete specification of their operational semantics is given.

3   From Fluents to Interactors

We will now describe the extension of the *Fluents* API of [22] that provides a minimal bidirectional communication API between interactors and their clients.

3.1   The Interaction Mechanism

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

A yield/return operation First, like the `yield return` construct of C# and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

will save the state of the engine and transfer control and a result `Term` to its client. The client will receive a copy of `Term` simply by using its `get/1` operation. Similarly to Ruby's `yield`, our `return` operation suspends and returns data from arbitrary computations (possibly involving recursion) rather than from specific language constructs like a `while` or `for` loop.

Note that an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/get` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism (see [22]) simply with

```
throw(E):-return(exception(E)).
```

When combined with a `catch(Goal,Exception,OnException)`, on the client side, the client can decide, upon reading the exception with `get/1`, if it wants to handle it or to throw it to the next level.

The mechanisms discussed so far are expressive enough, as described in [22], to implement at source level key built-in predicates of Prolog like `if-then-else`, `findall` and `copy_term`.

Interactors and Coroutining The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable its client to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine,Data)
```

used to send a client's data to an Engine, and

```
from_engine(Data)
```

used by the engine to receive a client's Data.

Using a metacall mechanism like `call/1` (which can also be emulated in terms of engine operations [22]), one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine,Goal, Answer):-
  to_engine(Engine,Goal),
  get(Engine,Answer).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

where **ask_engine** sends a goal (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an **engine_yield** operation.

As the following example shows, this allows the client to use from outside the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1=>S2),sum_loop(S2).

inc_test(R1,R2):-
   new_engine(_,sum_loop(0),E),
   ask_engine(E,(S1=>S2:-S2 is S1+2),R1),
   ask_engine(E,(S1=>S2:-S2 is S1+5),R2).

?- inc_test(R1,R2).
R1=the(0 => 2),
R2=the(2 => 7)
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed, with access to the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

We will call `Interactors API` the Horn Clause subset of Prolog with LD resolution together with the Logic Engine operations described so far. As we shown in [22], `call/1` itself can be emulated at source level with the `Logic Engine API`. As shown in subsection 4.1, the API will also allow emulating Prolog's dynamic database operations, providing runtime code creation and execution.

3.2 Using Interactors

To summarize, a typical use case for the *Interactor API* looks as follows:

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation
5. while cooperation between the *engine* and its client is assumed, the "*client* drives" ; failure of the goal injected in the *engine*'s computation space fails the *engine* ("fast fail" semantics)

A number of alternate semantics are possible, implementable at source level, on top of `to_engine/2` and `from_engine/1`. An alternate scenario, emphasizing more on error recovery than "fast fail" could be devised: when failure and/or exceptions are caught by the engine, the client is simply notified about them, while the engine's ability to handle future requests is preserved.

3.3 Interactors and Higher Order Constructs

As a first glimpse at the expressiveness of this API, we will implement, in the tradition of higher order functional programming, a *fold* operation [2] connecting results produced by independent branches of a backtracking Prolog engine:

```
efoldl(Engine,F,R1,R2):-
  get(Engine,X),
  efoldl_cont(X,Engine,F,R1,R2).

efoldl_cont(no,_Engine,_F,R,R).
efoldl_cont(the(X),Engine,F,R1,R2):-
  call(F,R1,X,R),
  efoldl(Engine,F,R,R2).
```

Classic functional programming idioms like *reverse as fold* are then implemented simply as:

```
reverse(Xs,Ys):-
  new_engine(X,member(X,Xs),E),
  efoldl(E,reverse_cons,[],Ys).

reverse_cons(Y,X,[X|Y]).
```

Note also the automatic *deforestation* effect [29, 13] of this programming style - no intermediate list structures need to be built, if one wants to aggregate the values retrieved from an arbitrary generator engine with an operation like sum or product.

4   Interactors and Interoperation with Stateful Objects

The gain in expressiveness coming directly from the view of logic engines as answer generators is significant. We refer to [22] for source level implementations of virtually all essential Prolog built-ins (exceptions included). The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

4.1   Dynamic Databases with Interactors

The key idea for implementing dynamic database operations with Interactors is to use a logic engine's state in an infinite recursive loop, similar to the coinductive programming style advocated in [20, 7], to emulate state changes in its client engine.

First, a simple difference-list based infinite server loop is built:

```
queue_server:-queue_server(Xs,Xs).

queue_server(Hs1,Ts1):-
  from_engine(Q),
    server_task(Q,Hs1,Ts1,Hs2,Ts2,A),
  return(A),
  queue_server(Hs2,Ts2).
```

Next we provide the queue operations, needed to maintain the state of the database.

```
server_task(add_element(X),Xs,[X|Ys],Xs,Ys,yes).
server_task(push_element(X),Xs,Ys,[X|Xs],Ys,yes).
server_task(queue,Xs,Ys,Xs,Ys,Xs-Ys).
server_task(delete_element(X),Xs,Ys,NewXs,Ys,YesNo):-
  server_task_delete(X,Xs,NewXs,YesNo).
```

Then we implement the auxiliary predicates supporting various queue operations:

```
server_task_remove(Xs,NewXs,YesNo):-nonvar(Xs),Xs=[X|NewXs],!,
  YesNo=yes(X).
server_task_remove(Xs,Xs,no).

server_task_delete(X,Xs,NewXs,YesNo):-select_nonvar(X,Xs,NewXs),!,
  YesNo=yes(X).
server_task_delete(_,Xs,Xs,no).

server_task_stop(E):-stop(E).

select_nonvar(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
select_nonvar(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],
  select_nonvar(X,Xs,Ys).
```

Finally, we put it all together, as a dynamic database API:

```
% creates a new engine server
% providing Prolog database operations
new_edb(Engine):-new_engine(done,queue_server,Engine).

% adds an element to the end of the database
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).

% adds an element to the front
edb_asserta(Engine,Clause):-
  ask_engine(Engine,push_element(Clause),the(yes)).

% returns a instances of asserted clauses
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).

% delete an element of the database
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
```

```
      ask_engine(Engine,delete_element(Clause),the(yes(Clause))).

% removes a database
edb_delete(Engine):-stop(Engine).
```

The database will now generate the equivalent of `clause/2`, ready to be passed to a Prolog metainterpreter.

```
test_clause(Head,Body):-
  new_edb(Db),
    edb_assertz(Db,(a(2):-true)),
    edb_asserta(Db,(a(1):-true)),
    edb_assertz(Db,(b(X):-a(X))),
  edb_clause(Db,Head,Body).
```

Externally implemented dynamic databases are also made visible as `Interactors` and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases are provided. This simplifies adding module, object or agent layers at source level. By combining database and communication (socket or RMI) Interactors, software abstractions like mobile code and autonomous agents are built as shown in [27]. Interoperation with External Stateful Objects like file systems or Prolog language extensions as dynamic databases is also simpler as implementation language operations can be applied to Interactors directly. Moreover, Prolog operations traditionally captive to predefined list based implementations (like DCGs) can be made generic and mapped to work directly on Interactors encapsulating file, URL and socket Readers.

5 Refining Control and Simplifying Algorithms with Interactors

5.1 Refining control: a backtracking if-then-else

Modern Prolog implementations (SWI, SICStus, BinProlog) also provide a variant of `if-then-else` that either backtracks over multiple answers of its `then` branch or switches to the `else` branch if no answers in the `then` branch are found. With the same API, we can implement it at source level as follows [1]:

```
if_any(Cond,Then,Else):-
  new_engine(Cond,Cond,Engine),
  get(Engine,Answer),
  select_then_or_else(Answer,Engine,Cond,Then,Else).
```

---

[1] We have included this example because it expresses a form of control that cannot be implemented at source level. Although discussed in a posting of the author in comp.lang.prolog, this example has never been part of a reviewed publication.

```
select_then_or_else(no,_,_,_,Else):-Else.
select_then_or_else(the(BoundCond),Engine,Cond,Then,_):-
  backtrack_over_then(BoundCond,Engine,Cond,Then).

backtrack_over_then(Cond,_,Cond,Then):-Then.
backtrack_over_then(_,Engine,Cond,Then):-
  get(Engine,the(NewBoundCond)),
  backtrack_over_then(NewBoundCond,Engine,Cond,Then).
```

5.2 Simplifying Algorithms: Interactors and Combinatorial Generation

Various combinatorial generation algorithms have elegant backtracking imple-
mentations. However, it is notoriously difficult (or inelegant, through the use of
impure side effects) to compare answers generated by different OR-branches of
Prolog's search tree.

Comparing Alternative Answers Such optimization problems can easily be
expressed as follows:

  – running the generator in a separate logic engine
  – collecting and comparing the answers in a client controlling the engine

The second step can actually be automated, provided that the comparison cri-
terion is given as a predicate

```
compare_answers(First,Second,Best)
```

to be applied to the engine with an `efold` operation

```
best_of(Answer,Comparator,Generator):-
  new_engine(Answer,Generator,E),
  efoldl(E,
    compare_answers(Comparator),no,
  Best),
  Answer=Best.

compare_answers(Comparator,A1,A2,Best):-
  if((A1\==no,call(Comparator,A1,A2)),
    Best=A1,
    Best=A2
  ).

?-best_of(X,>,member(X,[2,1,4,3])).
X=4
```

Counting Answers without Accumulating Problems as simple as counting the number of solutions of a combinatorial generation problem can become tricky in Prolog (unless one uses impure side effects) as one might run out of space by having to generate all solutions as a list, just to be able to count them. The following example shows how this can be achieved using an `efold` operation on an integer partition generator:

```
integer_partition_of(N,Ps):-
  positive_ints(N,Is),
  split_to_sum(N,Is,Ps).

split_to_sum(0,_,[]).
split_to_sum(N,[K|Ks],R):-N>0,sum_choice(N,K,Ks,R).

sum_choice(N,K,Ks,[K|R]):-NK is N-K,split_to_sum(NK,[K|Ks],R).
sum_choice(N,_,Ks,R):-split_to_sum(N,Ks,R).

positive_ints(1,[1]).
positive_ints(N,[N|Ns]):-N>1,N1 is N-1,positive_ints(N1,Ns).

% counts partitions by running
% the generator on an engine that returns
% 1 for each answer that is found
count_partitions(N,R):-
  new_engine(1,
    integer_partition_of(N,_),Engine),
  efoldl(Engine,+,0,R).
```

5.3 Encapsulating Infinite Computations Streams

An infinite stream of natural numbers is implemented simply as:

```
loop(N):-return(N),N1 is N+1,loop(N1).
```

The following example shows a simple space efficient generator for the infinite stream of prime numbers:

```
prime(P):-prime_engine(E),element_of(E,P).

prime_engine(E):-new_engine(_,new_prime(1),E).

new_prime(N):-N1 is N+1,
  if(test_prime(N1),true,return(N1)),
  new_prime(N1).

test_prime(N):-M is integer(sqrt(N)),between(2,M,D),N mod D =:=0
```

Note that the program has been wrapped, using the `element_of` predicate defined in [22], to provide one answer at a time through backtracking. Alternatively, a forward recursing client can use the `get(Engine)` operation to extract primes one at a time from the stream.

6 Interactors and Multi-Threading

While one can build a self-contained lightweight multi-threading API solely by switching control among a number of cooperating engines, with the advent of multi-core CPUs as the norm rather than the exception, the need for *native* multi-threading constructs is justified on both performance and expressiveness grounds. Assuming a dynamic implementation of a logic engine's stacks, Interactors provide lightweight independent computation states that can be easily mapped to the underlying native threading API.

A minimal native Interactor based multi-threading API, has been implemented in the Jinni Prolog system [25] on top of a new thread launching built-in

```
run_bg(Engine,ThreadHandle)
```

This runs a new Thread starting from the engine's `run()` method and returns a handle to the Thread object. To ensure that access to the Engine's state is safe and synchronized, we hide the engine handle and provide a simple producer/consumer data exchanger object, called a `Hub`. The complete multi-threading API, partly designed to match Java's own threading API is:

- `bg(Goal)`: launches a new Prolog thread on its own engine starting with `Goal`.
- `hub_ms(Timeout,HubHandle)`: constructs a new `Hub` returned as a `HubHandle` - a synchronization device on which N consumer threads can wait with `collect(HubHandle,Data)` for data produced by M producers providing data with `put(HubHandle,Data)`. However, if a given consumer waits more than `Timeout` milliseconds it returns and signals failure. As usual in Java, `Timeout=0` means indefinite suspension. If the thread is meant to interact with the parent, a `HubHandle` can be given to it as an argument.
- `current_thread(ThreadHandle)`: returns a handle to the current thread - that might be passed to another thread wanting to join this one.
- `join_thread(ThreadHandle)`: waits until a given thread terminates.
- sleep_ms(Timeout): suspends current thread for `Timeout` milliseconds.

A number of advanced multi-threading libraries have been designed around this basic API. For instance, *AND-synchronization* ensures waiting until N-tasks are finished. *Barriers* ensure that a number of threads wait jointly and when they all finish, a Runnable action is executed.

Associative Interactors The message passing style interaction shown in the previous sections between engines and their clients, can be easily generalized to associative communication through a unification based blackboard interface [3,

4]. Exploring this concept in depth promises more flexible interaction patterns, as out of order `ask_engine` and `engine_yield` operations would become possible, matched by association patterns.

7 Interactors Beyond Logic Programming Languages

We will now compare Interactors with similar constructs in other programming paradigms.

7.1 Interactors in Object Oriented Languages

Extending Interactors to mainstream Object Oriented languages is definitely of practical importance, given the gain in expressiveness. For instance, in the implementation of lightweight Prolog engines, Interactor based interfaces can provide a uniform interoperation mechanism with most of the system built at source level. Such an interface has been used in the Java-based Jinni Prolog system [25] to provide a uniform view of various Java services to Prolog's logic engines. In simple cases, like file operations, that can be suspended and resumed at will, such an interface is usable directly. In more complex cases, coroutining behavior can be achieved by adding `switch/case` statements to keep track of advancement of the control flow in a method[2]. An elegant open source Prolog engine `Yield Prolog` has been recently implemented in terms of Python's *yield* and `C#`'s *yield return* primitives [9]. Extending Yield Prolog to support our Interactor API only requires adding the communication operations `from_engine` and `to_engine`. In older languages like `Java`, `C++` or the `Objective C` dialect used in Apple's iPhone SDK, one needs to implement a more complex API, including a `yield return` emulation.

7.2 Interactors and similar constructs in Functional Languages

Interactors based on logic engines encapsulate future computations that can be unrolled on demand. This is similar to lazy evaluation mechanisms in languages like Haskell [18]. Interactors share with Monads [17,30] the ability to sequentialize functional computations and encapsulate state information. As a minor detail, the returned values consisting of terms of the form `the(Answer)` and `no`, like the `Maybe Monad`'s `Just a` and `Nothing` types, are used to encode possible failure of a computation. With higher order functions, monadic computations can pass functions to inner blocks. On the other hand, our `ask_engine` / `engine_yield` mechanism, like Ruby's `yield`, is arguably more flexible, as it provides arbitrary switching of control (coroutining) between an Interactor and its client. Our ability to define Prolog's `findall` construct [22] as well as `fold` operations in terms of Interactors, is similar to definition of comprehensions [31, 1] in terms of Monads, and List comprehensions in particular.

---

[2] Or just wait until Java borrows from `Ruby` or `C-sharp` something similar to the `yield` or `yield return` statements.

8   Conclusion

Logic Engines encapsulated as Interactors have been used to build on top of pure Prolog (together with the Fluent API described in [22]) a practical Prolog system, including dynamic database operations, entirely at source level. Interactors allowed to communicate between distinct OR-branches as a practical alternative to the use of side effects and have provided elegant implementations of control structures and higher order predicates.

In a broader sense, Interactors can be seen as a starting point for rethinking fundamental programming language constructs like Iterators and Coroutining in terms of language constructs inspired by *performatives* in agent oriented programming. If the concept catches on, we expect it to impact on programmer productivity and simplification of software development at large.

Beyond applications to logic-based language design, we hope that our language constructs will be reusable in the design and implementation of new functional and object oriented languages.

References

1. Yves Bekkers and Paul Tarau. Monadic Constructs for Logic Programming. In John Lloyd, editor, *Proceedings of ILPS'95*, pages 51–65, Portland, Oregon, December 1995. MIT Press.
2. Richard S. Bird and Oege de Moor. Solving optimisation problems with catamorphism. In Richard S. Bird, Carroll Morgan, and Jim Woodcock, editors, *MPC*, volume 669 of *Lecture Notes in Computer Science*, pages 45–66. Springer, 1992.
3. N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.
4. K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, January 1996.
5. FIPA. FIPA 97 specification part 2: Agent communication language, October 1997. Version 2.0.
6. Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in Icon. *ACM Trans. Program. Lang. Syst.*, 3(2):144–161, 1981.
7. Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In Verónica Dahl and Ilkka Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2007.
8. Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.*, 9(4):355–372, 1999.
9. Jeff Thompson. Yield Prolog. Project URL http://yieldprolog.sourceforge.net.
10. Barbara Liskov, Russell R. Atkinson, Toby Bloom, J. Eliot B. Moss, Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer, 1981.
11. Jed Liu, Aaron Kimball, and Andrew C. Myers. Interruptible iterators. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 283–294. ACM, 2006.
12. Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In Verónica Dahl and Philip Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2003.

13. Simon Marlow and Philip Wadler. Deforestation for higher-order functions. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 154–165. Springer, 1992.
14. Yukihiro Matsumoto. The Ruby Programming Language. June 2000.
15. James Mayfield, Yannis Labrou, and Timothy W. Finin. Evaluation of KQML as an Agent Communication Language. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *ATAL*, volume 1037 of *Lecture Notes in Computer Science*, pages 347–360. Springer, 1995.
16. Microsoft Corp. Visual `C#`. Project URL http://msdn.microsoft.com/vcsharp.
17. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
18. Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. September 2002. http://haskell.org/definition/haskell98-report.pdf.
19. Koichi Sasada. YARV: yet another RubyVM: innovating the ruby interpreter. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 158–159. ACM, 2005.
20. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483. Springer, 2007.
21. Paul Tarau. A Simplified Abstract Machine for the Execution of Binary Metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, 7 1991.
22. Paul Tarau. 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, July 2000. LNCS 1861, Springer-Verlag.
23. Paul Tarau. Orthogonal Language Constructs for Agent Oriented Logic Programming. In Manuel Caro and Jose F. Morales, editors, *Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems*, Saint-Malo, France, September 2004.
24. Paul Tarau. BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 2006.
25. Paul Tarau. The Jinni 2006 Prolog Compiler: a High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. Technical report, BinNet Corp., 2006.
26. Paul Tarau and M. Boyer. Nonstandard Answers of Elementary Logic Programs. In J.M. Jacquet, editor, *Constructing Logic Programs*, pages 279–300. J.Wiley, 1993.
27. Paul Tarau and Veronica Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(3):359–380, May 2001. Cambridge University Press.
28. Guido van Rossum. A Tour of the Python Language. In *TOOLS (23)*, page 370. IEEE Computer Society, 1997.
29. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.
30. Philip Wadler. The essence of functional programming. In *ACM Symposium POPL'92*, pages 1–15. ACM Press, 1992.
31. P.L. Wadler. Comprehending monads. In *ACM Conf. Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM Press.
32. Peter Wegner and Eugene Eberbach. New Models of Computation. *Comput. J.*, 47(1):4–9, 2004.