Interoperating Logic Engines

Paul Tarau[1] and Arun Majumdar[2]

[1] Department of Computer Science and Engineering
University of North Texas
Denton, Texas, USA
`tarau@cs.unt.edu`

[2] Vivomind Intelligence, Inc.
Rockville, Maryland, USA
`arun@vivomind.com`

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.

As a first step toward a declarative semantics, we sketch a pure Prolog specification showing that Interactors can be expressed at source level, in a relatively simple and natural way.

Interactors extend language constructs like Ruby, Python and `C#`'s multiple coroutining block returns through *yield* statements and they can emulate the action of fold operations and monadic constructs in functional languages.

Using the Interactor API, we describe at source level, language extensions like dynamic databases and algorithms involving generation of infinite answer streams.

Keywords: Prolog language extensions, logic engines, semantics of metaprogramming constructs, generalized iterators, agent oriented programming language constructs.

1 Introduction

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 [1] 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 [2].

In a logic programming context, the Jinni agent programming language [3] and the BinProlog system [4] 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* [5], 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 [5], 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.

Interruptible Iterators are a new Java extension described in [6]. The underlying construct is the `yield` statement providing multiple returns and resumption of iterative blocks, i.e. for instance, a `yield` statement in the body of a `for` loop will return a result for each value of the loop's index.

The `yield` statement has been integrated in newer Object Oriented languages like `Ruby` [7,8] `C#` [9] and `Python` [10] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [11] and ICON [12].

Interactors can be seen as a natural generalization of Interruptible Iterators and Coroutine Iterators. They implement the 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 [6]. Towards this end, Interactors provide 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.

2 First Class Logic Engines

Our *Interactor API* is a natural extension of the *Logic Engine API* introduced in [5]. 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 [13] on a given clause database, together with a set of built-in operations. 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. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas. In our actual implementation, with all data areas dynamic, engines are lightweight and engine creation is extremely fast.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine. The query

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

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. As in the case of the `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 might or might not reclaim resources held by the engine. In our actual implementation we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected.

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` [5], as well as higher order operations like *fold* and *best_of*.

3 From Fluents to Interactors

We will now describe the extension of the *Fluents* API of [5] that provides a minimal bidirectional communication API between interactors and their clients.

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

3.1 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/2` 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 [5]) simply by defining

```
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/2`, if it wants to handle it or to throw it to the next level.

3.2 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 an engine's 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.

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

Using a metacall mechanism like `call/1` (which can also be emulated in terms of engine operations [5]) or directly through a source level transformation [14], one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine,Query, Result):-
  to_engine(Engine,Query),
  get(Engine,Result).

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

The predicate `ask_engine/3` sends a query (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an `engine_yield` operation. The query is typically a goal or a pattern of the form `AnswerPattern:-Goal` in which case the engine interprets it as a request to instantiate `AnswerPattern` by executing `Goal` before returning the answer instance.

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. The goal can then access 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.

4 A (Mostly) Pure Prolog Specification

At a first look, Interactors deviate from the usual Horn Clause semantics of pure Prolog programs. A legitimate question arises: are they not just another procedural extension, say, like assert/retract, setarg, global variables etc.?

We will show here that the semantic gap between pure Prolog and its extension with Interactors is much narrower than one would expect. The techniques that we will describe can be seen as an executable specification of Interactors within the well understood semantics of logic programs (SLDNF resolution).

Toward this end, we will sketch an emulation, in pure Prolog, of the key constructs involved in defining Interactors.

There are four distinct concepts to be emulated:

1. we need to eliminate backtracking to be able to access multiple answers at a time
2. we need to emulate copy_term as different search branches and multiple uses of a given clause require fresh instances of terms, with variables standardized apart
3. we need to emulate suspending and resuming an engine
4. engines should be able to receive and return Prolog terms

We will focus here on the first two, that are arguably less obvious, by providing actual implementations. After that, we will briefly discuss the feasibility of the last two.

4.1 Metainterpreting Backtracking

First, let's define a clause representation, that can be obtained easily with a source-to-source translator. Clauses in the database are represented with difference-list terms, structurally isomorphic to the binarization transformation described in [14]. The code of a classic Prolog naive reverse + permutation generator program becomes:

```
:-op(1150,xfx,<=).

clauses([
    [app([],A,A)|B]<=B,
    [app([C|D],E,[C|F])|G]<=[app(D,E,F)|G],

    [nrev([],[])|H]<=H,
    [nrev([I|J],K)|L]<=[nrev(J,M),app(M,[I],K)|L],

    [perm([],[])|N]<=N,
    [perm([O|P],Q)|R]<=[perm(P,S),ins(O,S,Q)|R],

    [ins(T,U,[T|U])|V]<=V,
    [ins(W,[X|Y],[X|Z])|X0]<=[ins(W,Y,Z)|X0]
]).
```

Note that we can assume that variables are local to each clause and therefore they have been standardized apart accordingly[1].

First, let's define the basic inference step (equivalent to an LD-resolution step, [13]) as a simple "arrow composition" operation:

```
compose(F1,F2,A<=C):-copy_term(F1,A<=B),copy_term(F2,B<=C).
```

We can now add a new "arrow" to a list of existing arrows, provided that the composition succeeds:

[1] Allowing shared variables would bring a different, but nevertheless interesting semantics, with "inter-clausal variables" seen as write-once global variables.

```
match_one(F1,F2,Fs,[NewF|Fs]):-compose(F1,F2,F3),!,NewF=F3.
match_one(_,_,Fs,Fs).
```

We can see an arrow as representing the current goal. The next step is to let an arrow select from a *list* of clauses the ones that match:

```
match_all([],_,Fs,Fs).
match_all([Clause|Cs],Arrow,Fs1,Fs3):-
  match_one(Arrow,Clause,Fs1,Fs2),
  match_all(Cs,Arrow,Fs2,Fs3).
```

We can add a stopping condition to mark the success of an LD-derivation as matching an arrow of the form `Answer<=[]`

```
derive_one(Answer<=[],_,Fs,Fs,As,[Answer|As]).
derive_one(Answer<=[G|Gs],Cs,Fs,NewFs,As,As):-
  match_all(Cs,Answer<=[G|Gs],Fs,NewFs).
```

With these building blocks in place, the result of the LD-derivations of *all answer instances* of a query can be defined as:

```
all_instances(AnswerPattern,Goal,Clauses,Answers):-
  Gs=[AnswerPattern<=[Goal]],
  derive_all(Gs,Clauses,[],Answers).
```

where `derive_all` lifts the derivation process to progressively solve all existing and newly generated goals:

```
derive_all([],_,As,As).
derive_all([Arrow|Fs],Cs,OldAs,NewAs):-
  derive_one(Arrow,Cs,Fs,NewFs,OldAs,As),
  derive_all(NewFs,Cs,As,NewAs).
```

Finally, we can integrate the clause database:

```
all_answers(X,G,R):-clauses(Cs),all_instances(X,G,Cs,R).
```

and try out a few goals:

```
?- all_answers(Xs+Ys,app(Xs,Ys,[1,2,3]),Rs).
Rs = [[]+[1, 2, 3], [1]+[2, 3], [1, 2]+[3], [1, 2, 3]+[]]

?- all_answers(P,perm([1,2,3],P),Ps).
Ps = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

Note, that for non-ground queries, answers computed this way keep variable equalities as expected:

```
?- List=[A,B,B,A],all_answers(R,nrev(List,R),Rs).
List = [A, B, B, A],
Rs = [[_A, _B, _B, _A]]
```

Note that, except for relying on `copy_term` and a cut that can be replaced with a negation as failure, the metainterpreter is entirely written in pure Prolog.

4.2 Emulating `copy_term`

We can emulate the effect of `copy_term` in the previously described metainterpreter by observing that a logical variable can be "split" into two new ones and consequently a Prolog term can be recursively deconstructed and rebuilt as two fresh terms, identical to it up to uniform variable renamings.

```
fork_term('$v'(T1,T2), R1,R2):-R1=T1,R2=T2.
fork_term(T, T1,T2):-
  nonvar(T),functor(T,F,N),(F/N) \== ('$v'/2),
  functor(T1,F,N),functor(T2,F,N),
  fork_args(N,T,T1,T2).

fork_args(0,_,_,_).
fork_args(I,T,T1,T2):-I>0,
  I1 is I-1,arg(I,T,X),
  fork_term(X,A,B),
  arg(I,T1,A),arg(I,T2,B),
  fork_args(I1,T,T1,T2).
```

One can see that this produces indeed two fresh copies of the original term:

```
?- fork_term(f(A,B,g(B,A)),T1,T2).
A = '$v'(_A1, _A2),
B = '$v'(_B1, _B2),
T1 = f(_A1, _B1, g(_B1, _A1)),
T2 = f(_A2, _B2, g(_B2, _A2)).
```

Note that `functor` and `arg` can be seen as generic abbreviations for predicates describing the building/decomposition operations for each function symbol occurring in the program and `$v/2` can be assumed to be any function symbol not occurring in the program. Along the lines of [15] one can see that this functionality can be also expressed through a simple program transformation provided that `nonvar/1` can be expressed using negation as failure as

```
  nonvar(X):- not(X=0),not(X=1).
```

We will obtain a slightly different definition of composition, that would require replacing both the clause and the resolvent with one of the copies while using the other pair of copies for the arrow compositions.

```
compose(F1,F2, A<=C, NewF1,NewF2):-
   fork_term(F1,A<=B,NewF1),
   fork_term(F2,B<=C,NewF2).
```

One can now see that after propagating the extra arguments through the clauses of the metainterpreter described in subsection 4.1, together with the source level transformations we just mentioned, a metainterpreter that does not require `copy_term` can be derived.

4.3 Implementing Suspend/Resume and Term/Exchanges

The metainterpreter described in subsection 4.1 can be easily modified to return the current goal list when observing a `return(X)` instruction and then be resumed at will, by adding a clause similar to the one handling the case `Answer<=[]`. At this point, data exchange operations and `to_engine` and `from_engine` can be implemented through an extra argument added to the metainterpreter.

5 Interactors and Higher Order Constructs

As a first glimpse at the expressiveness of the Interactor API, we will implement, in the tradition of higher order functional programming, a *fold* operation [16] 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 [17] 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.

6 Emulating Dynamic Databases with Interactors

The gain in expressiveness coming directly from the view of logic engines as answer generators is significant. We refer to [5] for source level implementations of virtually all essential Prolog built-ins. The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

The key idea for implementing dynamic database operations with Interactors is to use a logic engine's state in an infinite recursive loop.

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).

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).
```

Next, we put it all together, as a dynamic database API.

We can create a new engine server providing Prolog database operations:

```
new_edb(Engine):-new_engine(done,queue_server,Engine).
```

We can add new clauses to the database

```
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).

edb_asserta(Engine,Clause):-
  ask_engine(Engine,push_element(Clause),the(yes)).
```

and we can return fresh instances of asserted clauses

```
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).
```

or remove them from the the database

```
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
  ask_engine(Engine,
    delete_element(Clause),the(yes(Clause))).
```

Finally, the database can be discarded by stopping the engine that hosts it:

```
edb_delete(Engine):-stop(Engine).
```

The following example shows how the database generates 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).
```

As a side note, combining this emulation with the metainterpreter described in section 4, provides an executable specification of Prolog's dynamic database operations in pure Prolog, worth investigating in depth, as future work.

Externally implemented dynamic databases can also be made visible as Interactors and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases can be provided. This simplifies adding module, object or agent layers at source level. By combining database and communication Interactors, software abstractions like mobile code and autonomous agents can be built as shown in [18]. Encapsulating external stateful objects like file systems or external database or Web service interfaces as Interactors can provide a uniform interfacing mechanism and reduce programmer learning curves in practical applications of Prolog.

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.

7 Simplifying Algorithms: Interactors and Combinatorial Generation

Various combinatorial generation algorithms have elegant backtracking implementations. 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.

7.1 Comparing Alternative Answers

Optimization problems, selecting the "best" among answers produced on alternative branches 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 criterion 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
```

Clearly, a similar mechanism can be used to count the number of solutions without having to accumulate them to a list.

7.2 Encapsulating Infinite Computations Streams

An infinite stream of natural numbers is implemented 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 [5], 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.

8 Applications of Interactors and Practical Language Extensions

Interactors and Multi-Threading. As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog and SWI-Prolog [19,20], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

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 [3] on top of a simple thread launching built-in:

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

This runs a new Thread starting from the engine's `run()` predicate 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`. Some key components of the multi-threading API, partly designed to match Java's own threading API are:

- `bg(Goal)`: launches a new Prolog thread on its own engine starting with `Goal`.
- `hub_ms(Timeout,Hub)`: constructs a new `Hub` - a synchronization device on which `N` consumer threads can wait with `collect(Hub,Data)` (similar to a synchronized `from_engine` operation) for data produced by `M` producers providing data with `put(Hub,Data)` (similar to a synchronized `from_engine` operation.

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 [21]. 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.

9 Interactors Beyond Logic Programming Languages

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

9.1 Interactors in Object Oriented Languages

Extending Interactors to mainstream Object Oriented languages is definitely of practical importance, given the gain in expressiveness. An elegant open source Prolog engine `Yield Prolog` has been recently implemented in terms of Python's *yield* and `C#`'s *yield return* primitives [22]. 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 `Objective C` one needs to implement a more complex API, including a `yield return` emulation.

9.2 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 [23]. Interactors share with Monads [24] the ability to sequentialize functional computations and encapsulate state information. 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. The ability to define Prolog's `findall` construct as well as `fold` operations in terms of Interactors, is similar to definition of comprehensions [24] in terms of Monads.

10 Conclusion

We have shown that Logic Engines encapsulated as Interactors can be used to build on top of pure Prolog a practical Prolog system, including dynamic database operations, entirely at source level. We have also provided a sketch of an executable semantics for Logic Engine operations in pure Prolog. This shows that, in principle, their exact specification can be expressed declaratively.

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.

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.

Among real world applications of these ideas, we have been pursuing a new model of natural language understanding [25] where multiple concurrently processing agents, using lightweight interpretation engines implemented as interactors transform text into semantic model structures for reasoning in the Oil and Gas exploration and production domain.

References

1. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In: Wooldridge, M., Müller, J.P., Tambe, M. (eds.) IJCAI-WS 1995 and ATAL 1995. LNCS, vol. 1037, pp. 347–360. Springer, Heidelberg (1996)
2. Wegner, P., Eberbach, E.: New Models of Computation. Comput. J. 47(1), 4–9 (2004)
3. Tarau, P.: Orthogonal Language Constructs for Agent Oriented Logic Programming. In: Carro, M., Morales, J.F. (eds.) Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems, Saint-Malo, France (September 2004)
4. Tarau, P.: BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp. (2006)
5. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861. Springer, Heidelberg (2000)
6. Liu, J., Kimball, A., Myers, A.C.: Interruptible iterators. In: Morrisett, J.G., Jones, S.L.P. (eds.) POPL, pp. 283–294. ACM, New York (2006)

7. Matsumoto, Y.: The Ruby Programming Language (June 2000)
8. Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter. In: Johnson, R., Gabriel, R.P. (eds.) OOPSLA Companion, pp. 158–159. ACM, New York (2005)
9. Microsoft Corp.: Visual C# . Project, `http://msdn.microsoft.com/vcsharp`
10. van Rossum, G.: A Tour of the Python Language. In: TOOLS (23), p. 370. IEEE Computer Society, Los Alamitos (1997)
11. Liskov, B., Bloom, T., Schaffert, J.C., Snyder, A., Atkinson, R., Moss, E., Scheifler, R.: CLU. LNCS, vol. 114. Springer, Heidelberg (1981)
12. Griswold, R.E., Hanson, D.R., Korb, J.T.: Generators in Icon. ACM Trans. Program. Lang. Syst. 3(2), 144–161 (1981)
13. Tarau, P., Boyer, M.: Nonstandard Answers of Elementary Logic Programs. In: Jacquet, J. (ed.) Constructing Logic Programs, pp. 279–300. J.Wiley, Chichester (1993)
14. Tarau, P., Boyer, M.: Elementary Logic Programs. In: Deransart, P., Małuszyński, J. (eds.) PLILP 1990. LNCS, vol. 456, pp. 159–173. Springer, Heidelberg (1990)
15. Warren, D.H.D.: Higher-order extensions to Prolog – are they needed? In: Michie, D., Hayes, J., Pao, Y.H. (eds.) Machine Intelligence 10. Ellis Horwood (1981)
16. Bird, R.S., de Moor, O.: Solving optimisation problems with catamorphism. In: Bird, R.S., Woodcock, J.C.P., Morgan, C.C. (eds.) MPC 1992. LNCS, vol. 669, pp. 45–66. Springer, Heidelberg (1993)
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
18. Tarau, P., Dahl, V.: High-Level Networking with Mobile Code and First Order AND-Continuations. Theory and Practice of Logic Programming 1(3), 359–380 (2001)
19. Carro, M., Hermenegildo, M.V.: Concurrency in prolog using threads and a shared database. In: ICLP, pp. 320–334 (1999)
20. Wielemaker, J.: Native preemptive threads in SWI-prolog. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 331–345. Springer, Heidelberg (2003)
21. De Bosschere, K., Tarau, P.: Blackboard-based Extensions in Prolog. Software — Practice and Experience 26(1), 49–69 (1996)
22. Jeff Thompson: Yield Prolog. Project, `http://yieldprolog.sourceforge.net`
23. Peyton Jones, S.L. (ed.): Haskell 98 Language and Libraries: The Revised Report (September 2002), `http://haskell.org/definition/haskell98-report.pdf`
24. Wadler, P.: Comprehending monads. In: ACM Conf. Lisp and Functional Programming, Nice, France, pp. 61–78. ACM Press, New York (1990)
25. Majumdar, A.K., Sowa, J.F., Stewart, J.: Pursuing the goal of language understanding. In: Eklund, P., Haemmerlé, O. (eds.) ICCS 2008. LNCS, vol. 5113, pp. 21–42. Springer, Heidelberg (2008)