Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects

Paul Tarau

Department of Computer Science
University of North Texas
P.O. Box 311366
Denton, Texas 76203
E-mail: tarau@cs.unt.edu

Abstract. On top of a simple kernel (Horn Clause Interpreters with LD-resolution) we introduce **Fluents**, high level stateful objects which empower and simplify the architecture of logic programming languages through reflection of the underlying interpreter, while providing uniform interoperation patterns with object oriented and procedural languages. We design a Fluent class hierarchy which includes first-class stateful objects representing the meta-level Horn Clause Interpreters, file, URL, socket Readers and Writers, as well as data structures like terms and lists, with high-level operations directly mapped to iterative constructs in the underlying implementation language. Fluents melt naturally in the fabric of Logic Programming languages and provide elegant composition operations, reusability, resource recovery on backtracking and persistence. The Web site of our Kernel Prolog prototype, `http://www.binnetcorp.com/kprolog/Main.html` allows the reader to try out online the examples discussed in this paper.

Keywords: *Logic Programming Language Design and Implementation, Interoperation of Declarative and Stateful Languages, Meta-Programming and Reflection*

1 Introduction

Despite significant syntactic, semantic and implementational variations, Logic Programming languages share a common kernel: *Horn Clause Resolution*[1], a semantically and operationally well understood calculus. As it is the case with pure functional programming languages, this calculus allows reasoning with referentially transparent, stateless entities.

However, the resolution process as such, is obviously not stateless, as it proceeds in time, step by step. If we want to preserve the ability to *reflect* in the object language the resolution process provided by the underlying interpreter,

[1] The most commonly used variation is Prolog's LD-resolution which combines a depth-first search rule with a left-to-right selection rule.

even simple abstractions like the sequence of alternative answers computed by the interpreter, will require non-trivial additional programming language constructs. While implementing reflection mechanisms, a careful language designer will be quickly faced with the need to pass Occam's razor to keep in check the explosion of redundant ontology.

Evolving algebras [6] have shown that programming languages can be seen as a combination of a basic, *terminating step* and some form of *iterative closure* operation. Linear logic [5, 1] has provided a more accurate description of the state of the proof process, with emphasis on seeing formulas as *resources*, with special notation indicating if they are unique or reusable.

Independently, the same need for *state representation with minimal new ontology* arises from the need for simplified *interoperation* of declarative languages with conventional software and operating system services which often relay on stateful entities.

Through constructs ranging from plain file or socket streams in C, to lazy list streams in languages like Scheme, iterators in Java or C++, monadic constructs [18, 2] in Haskell or in λ-Prolog, declarative I/O in Mercury [12], share the need for *abstracting away the nature of the stepping process* in a (finite or infinite, actual or generated as needed) sequence. Moreover, in the case of a declarative language implemented in a procedural or object oriented language, a uniform reflection mechanism is needed, for consistent modeling of stateful external objects providing native services.

This paper will introduce a concept of first class *fluents* on top of Horn Clauses with LD-Resolution to provide reflection of the underlying interpreter and interoperation with external stateful components, in a uniform way.

When seen from inside an Interpreter, other Interpreters will appear as instances of Fluents (Sources) producing a stream of answers. Through a set of suitable abstractions, they will be put to work as reusable components cooperating through independent resolution processes.

We will also describe a set of Fluent constructors which create Fluents from conventional data structures like lists, strings, files, terms and clauses and then provide Fluent Composers - allowing to elegantly combine them as building blocks for software components.

We will provide two compact meta-interpreters showing how backtracking and forward derivation can both be reflected (and controlled) at source level.

As a practical outcome, we provide a redesign of some key Prolog built-ins, of possible use in the next iteration of the ISO Prolog standardization process.

2 First Class Horn Clause Interpreters

2.1 Fluents: from Reflection to Interoperation with External Objects

We will build *Kernel Prolog* as a collection of Horn Clause Interpreters running LD-resolution on a default clause database and calling built-in operations. Each

of them has a constructor which initializes them with a *goal* and an *answer pattern*. In fact, they will be seen as possibly infinite *sources of answers* which can be explored one by one. The object encapsulating the state of the interpreter is very similar to a file descriptor encapsulating the advancement of a file reader. We will call such stateful entities evolving in time *Fluents*.

Kernel Prolog Interpreters will possess, through built-in calls, the ability to create and query other Interpreters, as part of a general mechanism to a manipulate **Fluents**. *Fluents* encapsulating interpreters, like any other stateful objects, will have their independent life-cycles.

This general mechanism will allow Kernel Prolog interpreters to interoperate with the underlying object oriented implementation language, which will provide to and request from the interpreters, various services through a hierarchy of **Fluents**.

2.2 Interpreters as Answer Sources

Answer Sources can be seen as generalized iterators, allowing a given program to control answer production in another. Each Answer Source works as a separate Horn Clause LD-resolution interpreter (a very compact Java implementation of such an interpreter is given in the APPENDIX).

The **Answer Source** constructor initializes a new interpreter.

```
answer_source(AnswerPattern,Goal,AnswerSource)
```

creates a new Horn Clause solver, uniquely identified by **AnswerSource**, which shares code with the currently running program and is initialized with resolvent **Goal**. **AnswerPattern** is a term, usually a list of variables occurring in **Goal**.

The **get/2** operation (to be provided by all **Sources**, see section 3) is used to retrieve successive answers generated by an Answer Source, on demand.

```
get(AnswerSource,AnswerInstance)
```

tries to harvest the answer computed starting from **Goal**, as a instance of **AnswerPattern**. If an answer is found, it is returned as **the(AnswerInstance)**, otherwise **no** is returned. Note that once **no** has been returned, all subsequent **get/2** on the same **AnswerSource** will return **no**. 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 operation. 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 Answer Source's iteration over answers. Note however that backtracking over the Answer Source's creation point as such, makes it unreachable and therefore subject to garbage collection.

Finally, an Answer Source is stopped with the **stop** operation (implemented by all **Sources**, see section 3).

```
stop(AnswerSource)
```

The **stop/1** operation is called automatically when no more answers can be produced as well as through the Fluent's **undo** operation on backtracking.

3 Fluent Classes and their Operations

After seeing how AnswerSources encapsulate interaction with an interpreter, we will proceed with building a class hierarchy which generalizes this interaction pattern to external objects. The crux of this design is to make stateful external objects and interpreters communicate with a given interpreter in a uniform way. This turns out to be a very natural process, as modern "pattern aware" [4] object oriented design usually results in "interpreter-like" classes providing their services through high level abstractions. For instance, the Java classes in the Collections framework (JDK 1.2 and later), closely model set and finite function mathematics and are usable without any reference to "data-structure level" implementation detail.

We will first describe the root of our hierarchy, the **Fluent** class, then give some examples of simple Fluents and operations on Fluents.

Fluents are created with specific *constructors*, usually by converting from other Fluents or conventional Prolog data structures like Terms, Lists or Databases. All **Fluents** are enabled with a **stop/1** operation which releases their resources (most **Fluents** also call **stop** on backtracking, through their internal **undo** operation).

In our Java based reference implementation, the Fluent class looks as follows:

```
// Constructor, which adds this Fluent to the parent's trail.
class Fluent extends SystemObject {
  Fluent(Prog p) {trailMe(p);}

  // add the fluent to the parent Interpreter's Trail
  protected void trailMe(Prog p) {
    if(null!=p) p.getTrail().push(this);
  }

  // usable (through overriding) to release resources
  // and/or stop ongoing computations
  public void stop() {}

  // release resources on backtracking, if needed
  protected void undo() {stop();}
}
```

Sources are **Fluents** enabled with an extra **get/2** operation. Typical **Sources** are Horn Clause Interpreters, File, URL or String Readers, Fluents built from Prolog lists, Fluents iterating over data structures like Vectors or Hashtables or Queues in the underlying implementation language. Note that the constructor

Fluent(Prog p) is trailed on the caller program **p**'s trail, and provides an **undo** operation to be called by **p** on backtracking, to release resources through the Fluent's **stop** method.

The Source abstract class looks as follows:

```
abstract class Source extends Fluent {

  Source(Prog p) {super(p);}

  abstract public Term get();
}
```

Sinks are fluents enabled with an extra **put/2** and **collect/2** operation. Typical Sinks are **ClauseWriters** or **CharWriters** targeted to TermCollectors (implemented as a Java **Vectors** collecting Prolog terms), **StringSinks** (implemented as a Java **StringBuffers** collecting String representations of Prolog terms).

The Sink abstract class looks as follows:

```
abstract class Sink extends Fluent {

  Sink(Prog p) {super(p);}

  // sends T to the Sink for tasks as accumulation or printing
  abstract public int put(Term T);

  // returns data previously sent to the Sink
  // (if collection ability is present)
  public Term collect() {return null;}
}
```

Not surprisingly, *even Prolog databases* are first class citizens implemented as extensions of **Sources** which provide **add/2, remove/2, collect/2** operations.

Fluents can be seen as *resources* which go through state transitions as a result of **put/2**, **get/2** and **stop/1** operations. They end their life cycle in a stopped state when all the data structures and/or threads they hold are freed.

3.1 Fluent Composers

Fluent composers provide abstract operations on **Fluents**. They are usually implemented with lazy semantics.

For instance, **append_sources/3** creates a new **Source** with a **get/2** operation such that when the first **Source** is stopped, iteration continues over the elements of the second **Source**.

Compose_sources/3 provides a cartesian product style composition, the new **get/2** operation returning pairs of elements of the first and second **Source**.

Reverse_source/2 builds a new Source **R** from a (finite) Source **F**, such that **R**'s **get/2** method returns elements of **F** in reverse order.

Split_source/3 is a cloning operation creating two **Source** objects identical to the **Source** given as first argument. It allows writing programs which iterate over a given **Source** multiple times.

Sources and **Sinks** are related through a **discharge(Source,Sink)** operation which sends all the elements of the **Source** to the given **Sink**. This allows for instance copying in a generic way a stream of answers of an Interpreter as well as data coming from a URL, through a socket, to a file, without having to iterate explicitly or know details on how data is actually produced and what its concrete representation is.

3.2 Fluent Modifiers

Fluent modifiers allow dynamically changing some attributes of a give Fluent. For instance **set_persistent(Fluent,YesNo)** is used to make a Fluent survive failure, by disabling its **undo** method, which, by default, applies the Fluent's **stop** method on backtracking.

4 Source level extensions through new definitions

To give a glimpse to the expressiveness of the resulting language, we will now introduce, through definitions in Kernel Prolog, a number of built-in predicates known as "impossible to emulate" in Horn Clause Prolog (except by significantly lowering the level of abstraction and implementing something close to a Turing machine).

4.1 Negation and `once/1`

These constructs are implemented simply by discarding all but the first solution produced by a Solver.

```
% returns the(X) or no as first solution of G
first_solution(X,G,Answer):-
  answer_source(X,G,Solver),
  get(Solver,Answer),
  stop(Solver).

% succeeds by binding G to its first solution or fails
once(G):-first_solution(G,G,the(G)).

% succeeds without binding G, if G fails
not(G):-first_solution(_,G,no).
```

4.2 Reflective Meta-Interpreters

The simplest meta-interpreter **metacall/1** just reflects backtracking through **element_of/2** over deterministic Answer Source operations.

```
metacall(Goal):-
  answer_source(Goal,Goal,E),
  element_of(E,Goal).

element_of(I,X):-get(I,the(A)),select_from(I,A,X).

select_from(_,A,A).
select_from(I,_,X):-element_of(I,X).
```

We can see **metacall/1** as an operation which fuses two orthogonal language features provided by Answer Sources: *computing an answer of a Goal*, and *advancing to the next answer*, through the source level operations **element_of/2** and **select_from/3** which 'borrow' the ability to backtrack from the underlying interpreter. The existence of this simple meta-interpreter indicates that **answer_sources** lift expressiveness of first-order Horn Clause logic significantly.

Note that **element_of/2** *works generically on* **Sources** *and is therefore reusable, for instance, to backtrack over the character codes of a file or a URL.*

After showing that we can emulate metacalls, we will use, for convenience, variables directly in predicate call position.

Note also that an Answer Source enumerates elements of the transitive closure of the *clause unfolding* relation [15, 16].

If our interpreter can access a single unfolding step through a similar Fluent, a *finer grained meta-interpreter* can be built as follows. Let's introduce a new Fluent,

```
unfolder_source(Clause,Source)
```

which, given a Clause produces a stream of clauses obtained by unfolding the first atom on the right side against a matching clause in the database. Each step is described through an (associative) clause composition operation $\oplus$ as follows:

Let $\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n$ and $\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m$ be two clauses (suppose $n > 0, m \geq 0$). We define

$$(\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n) \oplus (\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m) = (\texttt{A}_0\texttt{:-B}_1\texttt{,...,B}_m\texttt{,A}_2\texttt{,...,A}_n)\theta$$

with $\theta = \text{mgu}(\texttt{A}_1,\texttt{B}_0)$. If the atoms $\texttt{A}_1$ and $\texttt{B}_0$ do not unify, the result of the composition is denoted as $\perp$ (failure). Furthermore, we consider $\texttt{A}_0\texttt{:-true,A}_2\texttt{,...,A}_n$ to be equivalent to $\texttt{A}_0\texttt{:-A}_2\texttt{,...,A}_n$, and for any clause C, $\perp \oplus \texttt{C} = \texttt{C} \oplus \perp = \perp$. As usual, we assume that at least one operand has been renamed to a variant with variables standardized apart.

We can now build a meta-interpreter which implements the transitive closure of the unfolding operation $\oplus$ (provided as the get/2 operation of an Unfolder Source in the underlying implementation language), combined with backtracking trough **element_of/2**.

```
unfold_solve(Goal):-unfold(':-'(Goal,Goal),':-'(Goal,true)).

unfold(Clause,Clause).
```

```
unfold(Clause,Answer):-
  unfolder_source(Clause,Unfolder),
  element_of(Unfolder,NewClause),
  unfold(NewClause,Answer).
```

Note that this meta-interpreter will provide both backtracking and recursion for implementing Prolog's LD-resolution search. Clearly, alternative search mechanisms can be programmed quite easily.

4.3 If-then-else

Once we have *first_solution* and *metacall* operations, emulating if-then-else is easy.

```
% if Cond succeeds executes Then otherwise Else
if(Cond,Then,Else):-
  first_solution(successful(Cond,Then),Cond,R),
  select_then_else(R,Cond,Then,Else).

select_then_else(the(successful(Cond,Then)),Cond,Then,_):-Then.
select_then_else(no,_,_,Else):-Else.
```

4.4 All-solution predicates

All-solution predicates like findall/3 can be obtained by collecting answers through recursion.

```
% if G has a finite number of solutions
% returns a list Xs of copies of X each
% instantiated correspondingly
findall(X,G,Xs):-
   answer_source(X,G,E),
   get(E,Answer),
   collect_all_answers(Answer,E,Xs).

% collects all answers of a Solver
collect_all_answers(no,_,[]).
collect_all_answers(the(X),E,[X|Xs]):-
   get(E,Answer),
   collect_all_answers(Answer,E,Xs).
```

Note that, again, the **collect_all_answers** operation is generic, and works on any **Source**. This suggest providing a built-in Source-to-List converter **source_list/2** which can be made more efficient in the underlying implementation language where iteration replaces **collect_all_answers/3**'s recursion while also the eliminating interpretation overhead.

The alternative definition of findall/3 becomes simply:

```
findall(X,G,Xs):-
   answer_source(X,G,Solver),
   source_list(Solver,Xs).
```

4.5 Term copying and instantiation state detection

As standardizing variables apart upon return of answers is part of the semantics of `get/2`, term copying is just computing a first solution to `true/0`. Implementing `var/1` uses the fact that only free variables can have copies unifiable with two distinct constants.

```
copy_term(X,CX):-first_solution(X,true,the(CX)).
var(X):-copy_term(X,a),copy_term(X,b).
```

The previous definitions have shown that the resulting language subsumes (through user provided definitions) constructs like negation as failure, if-then-else, once, **copy_term**, **findall** - this justifies its name *Kernel Prolog*. As Kernel Prolog contains negation as failure, following [3] we can, in principle, use it for an executable specification of full Prolog.

4.6 Implementing Exceptions

While it is possible to implement exceptions at source level as shown in [17], through a continuation passing program transformation (binarization), an efficient, constant time implementation can simply allow the interpreter to return a new answer pattern as indication of an exception. We have chosen this implementation scenario in our Kernel Prolog compiler which provides a **return/1** operation to exit an engine's emulator loop with an arbitrary answer pattern, possibly before the end of a successful derivation.

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

catch(Goal,Exception,OnException):-
  answer_source(answer(Goal),Goal,Source),
  element_of(Source,Answer),
  do_catch(Answer,Goal,Exception,OnException,Source).

do_catch(exception(E),_,Exception,OnException,Source):-
  if(eq(E,Exception),
     OnException % call action if matching
     throw(E)    % throw again otherwise
  ),
  stop(Source).
do_catch(the(Goal),Goal,_,_,_).
```

The **throw/1** operation returns a special exception pattern, while the **catch/3** operation stops the engine, calls a handler on matching exceptions or re-throws non-matching ones to the next layer.

5 Built-ins as a Library of Fluents

Modular extension of Kernel Prolog through new built-ins is based on an Object Oriented hierarchy of Fluents.

5.1 Lists and Terms as Source Fluents

Sequential Prolog data structures are mapped to Fluents naturally. For instance, **list_source/2** creates a new Fluent based on a List, such that its **get/2** operation will return one element of the list at a time. Similarly **term_source/2** creates a Fluent from an N-argument compound term, such that its **get/2** method will return first its function symbol then each argument. They are directly usable for composition/decomposition operations like **univ/2** (also known as `=../2`):

```
univ(T,FXs):-if(var(T),list_to_fun(FXs,T),fun_to_list(T,FXs)).

list_to_fun(FXs,T):-list_source(FXs,I),source_term(I,T).
fun_to_list(T,FXs):-term_source(T,I),source_list(I,FXs).
```

As they can be converted easily to/from Prolog data-structures, Fluents are usable as *canonical representation for data objects* as well as for *computational processes* (like in the case of **answer_sources**). *Fast iteration on Fluents, using loops over efficient native data structures in the implementation language, replace recursion in the object language.* This makes it possible to build high performance Fluent based logic programming implementations in relatively slow languages like Java (preliminary benchmarks indicate that our ongoing Jinni 2000 implementation is within an order of magnitude of the fastest C-based Prolog implementations, and it is likely to match quite closely slower ones like SWI Prolog). Interoperation with external objects is also simpler as implementation language operations can be applied to Fluents directly.

5.2 File, URL and Database I/O in Kernel Prolog

File and URL I/O operations are provided by encapsulating Java's Reader and Writer classes as Fluents. Clause and character Readers are seen as instances of Sources and therefore benefit from Source composition operations. Moreover, Prolog operations traditionally captive to predefined list based implementations (like DCGs) can be made generic and mapped to work directly on Sources like file, URL and socket Readers.

Dynamic clause databases are also made visible as **Fluents**, and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases are provided, to simplify adding module, object or agent layers at source level. By combining database and communication (socket or RMI) Fluents abstractions like mobile code are built easily and naturally.

5.3 Memoing Fluents

Most Fluents are designed, by default, to be usable only once, and to release all resources held (automatically on backtracking or under programmer's control when their **stop** operation is invoked). While Fluent operations like **split_fluent/3**

can be used[2] to duplicate most Source Fluents, the following alternative provides a more efficient alternative.

A **Memoing Fluent** is built on top of a Source Fluent by progressively *accumulating* computed values in a List or dynamic array. A Memoing Fluent can be shared between multiple consumers which want to avoid recomputation of a given value.

5.4 Fluent based Lazy Lists

Lazy Lists can be seen as an instance of Memoing Fluents: they accumulate successive values of a Source Fluent in a (reusable) list. The simple Lazy List abstraction in our reference implementation works as follows:

```
source_lazy_list(Source, LazyList)
```

creates a new LazyList object from a Source object:

```
lazy_head(LazyList, LazyHead)
```

extracts the current head element of the list. Iteration over the list is provided by

```
lazy_tail(LazyList, LazyTail)
```

which returns LazyTail, a new lazy list encapsulating the next stage of the Source fluent.

While complete automation of lazy lists through a form of attributed variable construct is possible, we have chosen a simpler implementation scenario based on the previously described operations, mainly because overriding unification with execution of an arbitrary procedure would introduce potential *non-termination* - something which would break the very idea of keeping the execution mechanism as close as possible to basic Horn Clause resolution, as available in classic Prolog.

Based on these operations, a lazy **findall/3** is simply:

```
% creates lazy list from an answer source
lazy_findall(X,G,LazyList):-
  answer_source(X,G,S),
  source_lazy_list(S,LazyList).
```

In fact, the behavior of the lazy list encapsulating **lazy_findall's** advancement on alternative solutions produced by an Answer Source, is indistinguishable from a lazy list constructed from an ordinary **list_source**:

[2] The astute reader might notice that Linear Logic provers provide similar operations. This is by no means accidental, a resource conscious proof procedure will usually provide explicit means to implement multiple use of a resource.

```
% creates a lazy list from a List
lazy_list(List,LazyList):-
  list_source(List,S),
  source_lazy_list(S,LazyList).
```

The following operations are centered around the **lazy_tail/3** advancement operation, which produces a lazily growing reusable list. This list is explored with **lazy_element_of/2** in a way similar to the way ordinary lists are explored with **member/2** and ordinary Sources are explored with **element_of/2**.

```
% explores a lazy list in a way compatible with backtracking
% allows multiple 'consumers' to access the list, end ensures that
% the lazy list advances progressively and consistently
lazy_element_of(XXs,X):-
  lazy_decons(XXs,A,Xs),
  lazy_select_from(Xs,A,X).

% backtracks over the lazy list
lazy_select_from(_,A,A).
lazy_select_from(XXs,_,X):-lazy_element_of(XXs,X).

% returns a head/tail pair of a non-empty lazy list
lazy_decons(XXs,X,Xs):-
  lazy_head(XXs,X),
  lazy_tail(XXs,Xs).
```

A minor change in Prolog's chronological backtracking is needed however: only the creation point of the lazy list is subject to trailing, and the complete lazy list is discarded at once. This is achieved in our reference implementation by giving to each lazy list its own (dynamically growing) trail, and by providing an **undo** operation which rewinds the trail completely when backtracking passes the lazy list object's creation point.

6 Related work

Similar to the Answer Sources described in this paper, *engine* constructs have been part of systems like Oz [11, 10] BinProlog [13] and Jinni [14].

The main differences with Oz engines are:

- while Oz designers have chosen not to handle backtracking in exchange for the ability of sharing variables between different threads, Kernel Prolog provides *encapsulated backtracking*, local to a given **Answer Source**
- Oz engines are not separated from the underlying multi-threading model, they are not simple Horn Clause processors, they are part of Oz's computation spaces - which include threads and constraint stores
- in Oz, answers are returned only when a computation space is stable - the engine mechanism in Oz is overloaded as a synchronization device - which in our case is an orthogonal concept

– Oz engines have been designed for a different purpose, i.e. to program alternative search algorithms or for local constraint propagation, while our objective is a uniform reflection mechanism for multiple first order Horn Clause interpreters and interoperation with (other) external stateful objects

Fluents share some design objectives with Haskell's IO Monad approach [9] - which essentially encapsulates the state of the external world in a single stateful entity on which IO operates as a sequence of transitions. Our fluents can be seen as an abstraction for multiple stateful worlds organized as a typed inheritance hierarchy and specialized toward *source* and *sink* roles - corresponding to abstract *read* and *write* operations. Note however that some sink fluents provide a **collect** operation allowing to build new sources. Arguably, fluents offer a more flexible management of input and output flows than the monolithical IO Monad. In fact, John Hughes recent proposal to replace monads with the more powerful concept of arrow [7] with emphasis on directionality hints towards possible evolution towards a fluent-like concept.

Java's own design of Reader and Writer class trees and the ability to transform streams into new streams with stronger properties or elements of a different granularity (which in fact serves as the implementation bases for some of our fluents, behind the scenes) and its recently introduced Collection framework [8] also show convergence towards similar design patterns.

Our previous work on the Jinni agent programming language [14] and BinProlog [13] has described similar engine constructs. However, the key idea of seeing engines as instances of Fluents, the separation of engines from the multithreading mechanism, the reconstruction of Prolog's built-ins as a hierarchy of Fluent classes and the interoperation of external objects encapsulated as Fluent instances are definitely new.

New languages based on relatively pure subsets of Prolog like Mercury [12] have been designed as targets of more efficient implementation technologies and for their reliability in building large software systems. While Horn Clauses with negation have been extensively studied and some of the techniques described in this paper might be well known to experienced Prolog programmers, the very idea of systematically exploring the gains in expressive power as a result of having multiple pure Prolog interpreters as first order objects, has not been explored yet, to our best knowledge.

7 Future work

We have recently finished the first cut of a fast WAM based implementation of Kernel Prolog in Java and integrated it with the interpreter described in this paper. Preliminary benchmarks indicate being constantly within one order of magnitude from the fastest C-based Prolog implementations.

This opens the door for a number of real-life software applications.

The advent of component based software development and intelligent appliances requiring small, special purpose, self contained, still powerful processing

elements, makes Kernel Prolog an appealing implementation technique for building logic programming components. In particular, in the case of small, wireless interconnected devices, subject to severe memory and bandwidth limitations, compact and orthogonally designed small language processors are instrumental.

Our ongoing commercial Palm Prolog and Prolog-in-Java implementations use respectively C and Java variants of a fast Horn Clause LD-resolution WAM emulator based on the Kernel Prolog design described in this paper. This high-performance Kernel Prolog compiler (subject of an upcoming paper) will also provide support for Agent Classes - a new form of code structuring which promises to bring logic programming to functionality beyond the usual object oriented Prolog extensions, within a declarative framework.

Here are a few open issues and some other ongoing or projected Kernel Prolog related developments:

- executable specification of ISO Prolog in terms of Kernel Prolog
- a study of Kernel Prolog's invariance under program transformations (unfolding)
- type checking / type inference mechanisms for Kernel Prolog
- lightweight engine creation and engine reuse techniques for Kernel Prolog
- Kernel Prolog as a basis of embedded Prolog component technology and Prolog based Palm computing

8 Conclusion

We have provided a design for the uniform interoperation of Horn Clause Solvers with stateful entities (Fluents) ranging from external procedural and object oriented language services like I/O operations, to other, 'first class citizen' Horn Clause Solvers. As a result, a simplified Prolog built-in predicate system has emerged.

By collapsing the semantic gap between Horn Clause logic and (most of) the full Prolog language into three surprisingly simple, yet very powerful operations, we hope to open the doors not only for an implementation technology for a new generation of lightweight Prolog processors but also towards a better understanding of the intrinsic elegance hiding behind the core concepts of the logic programming paradigm.

Our Horn Clause Solvers encapsulated as Fluents provide the ability to communicate between distinct OR-branches as an practical alternative to the use assert/retract based side effects, in implementing all-solution predicates. Moreover, lazy variants of all solution predicates are provided as a natural extension to Fluent based lazy lists.

Finally, high level Fluent Composers allow combining component functionality in generic, data representation independent ways.

References

1. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, Jerusalem, Israel, 1990. MIT Press.
2. 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.
3. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, Berlin, 1996. ISBN: 3-540-59304-7.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995. ISBN: 0201633612.
5. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, (50):1–102, 1987.
6. Yuri Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the EATCS*, 43:264–284, 1991.
7. John Hughes. Generalizing Monads to Arrows. Technical report. available from: http://www.cs.chalmers.se/ rjmh/Arrows/.
8. Sun Microsystems. The Java Collections Framework. Technical report. available from: http://java.sun.com/products/jdk/1.2/docs/guide/collections/.
9. Simon Peyton Jones and John Hughes. Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999. available from: http://www.haskell.org/onlinereport/.
10. Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.
11. Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, NY, USA, November 1994. The MIT Press.
12. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The Mercury Language Web Site. 1998. http://www.cs.mu.oz.au/research/mercuryl.
13. Paul Tarau. BinProlog 7.0 Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.
14. Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.
15. 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.
16. Paul Tarau and Michel Boyer. Elementary Logic Programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, August 1990.
17. Paul Tarau and Veronica Dahl. Logic Programming and Logic Grammars with First-order Continuations. In *Proceedings of LOPSTR'94, LNCS, Springer*, Pisa, June 1994.
18. Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 1–17, 1993.