Multi-Engine Horn Clause Prolog

Paul Tarau[1]

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

Abstract. We extend Horn Clause Prolog with two new primitives, `new_engine (+Goal, +Answer, -Engine)` and `new_answer(+Engine, -Answer)` for creating and exploring answer spaces of multiple interpreters (engines). We show that despite its ontological parsimony, the resulting language is comparable in practical expressivness with conventional full Prolog, by allowing compact definitions for negation, if-then-else, all solution predicates, I/O and reflective meta-interpreters. While not really needed anymore as a workaround for the lack of expressivenes of pure Prolog, a form of dynamic database operations can be emulated as well, using the state of multiple engines. With multiple engines laying the foundation for multi-threaded execution models of logic programming languages, the surprising result that a minimal extension of Horn Clauses in fact bridges the gap to full Prolog is significant as a basis for lightweight implementations of embedabble logic programming components. Our constructs have been used in the design of small footprint mobile code interpreters for a commercial multi-threaded agent programming language - Jinni - available from `http://www.binnetcorp.com/Jinni`. *An online version of the code described in the paper can be run over the net and is available for download from:* `http://pc87043.csci.unt.edu/bp_cgi/715/kernel_prolog/query.html`

Keywords: Prolog software design, expressiveness of logic programing languages, software patterns for Prolog programming, lightweight Prolog systems

1 Introduction

1.1 The problem

Although Prolog programmers usually agree on how to express various algorithms in side-effect free Prolog, a wide diversity of opinions exist on how to bridge the semantic gap between pure Prolog and full Prolog and how software patterns for pure Prolog can be reused in dealing with real-life programming ('full Prolog'), while remaining as close as possible to the well understood semantic base of the language.

A quick look at how beginners evolve into Prolog programmers mimics quite well the dynamics of the issues involved. The following pattern, we have witnessed a few times during the last two decades on comp.lang.prolog is tipical.

Once programming without destructive assignment is understood, the Prolog beginner manages to build *Horn Clause programs* relatively well.

Then he or she hits suddenly the next obstacle: the problem of *information exchange between different derivations (AND-branches).*

As Prolog explores different AND-branches through backtracking, their state is not simultaneously present. The (unrecommended) approach is saving state through dynamic database operations.

This is plagued not only by the loss of declarative semantics but also with quick deterioration of operational semantics: non-reentrancy of code, sensitivity to CUT, sensitivity to semantics of the underlying dynamic database implementation.

The comp.lang.prolog insider usually preaches abstinence: such information exchange is better to be avoided. Suggestions to handle the problem with negation, setof/bagof constructs, start to come out. The sudden conspiracy:-) among insiders is somewhat to forget mentioning that the solutions involve reexecution of goals, changes in complexity, needlessly performing all solution computations etc.

At this point, there are clear signs that a serious expressivness limit of the language has been reached.

The welcome-to-the-club initiation then takes the beginner to the semantics of database updates and views, the fineprint specification of setof and bagof, with delicatessen like the morphing of stack variables into heap variables in the middle of a the sorting process. Of course, not being a beginner anymore, the newly born Prolog insider is ready for the next iteration - explaining to other novices where the fun comes from.

It is a fact, at this point, that all the semantic simplicity of Horn Clause Logic Programming is gone.

As it is the case with the expressiveness of Finite State Automata vs Deterministic Finite State Automata, it is no secret that non-deterministic programs give a much simpler expression to common algorithms. This extends naturally to Horn Clause programs. In fact, as shown in [TD95], appropriate use of non-determinism also leads to better performance models and a more predictable real-time behavior by possibly eliminating garbage collection.

As programming gains in genericity and component reusability in the procedural, object oriented and functional paradigms, the need to combine answers obtained through different derivations (different AND branches) the very existence of which makes logic programming different, is, arguably, instrumental for the survival of logic programming languages, as it improves their ability to build generic, reusable components.

However, to be able to re-use such nondeterministic generators as components, we need to control the production of their answers and we need to be able to consider alternative solutions together.

1.2 The contributions

We will describe in the paper a shortcut for the problem of information exchange between AND-derivations, which encourages using simple non-deterministic generators, encapsulated as separate engines (Horn Clause interpreters).

Through a set of suitable abstractions, they will be put to work as reusable components cooperating through independent resolution processes.

While the semantics of a multi-engine program cannot be expressed in a trivial way in terms of one-engine Horn Clause resolution[1], their operational semantics is fairly simple. Moreover, the usual declarative semantics, covers pure components running on each engine, while the more powerful glue language used to coordinate engines can be kept as a separate, meta-level component.

The first contribution of this paper is a new, 'axiomatic' redesign of a language having the expressivenes of full Prolog on top of Horn Clause Prolog, using only two predefined operations. Our design covers negation, limitted prunning through once/1 and if-then-else/3, all-solution predicates, I/O and goes beyond standard Prolog in providing forms of lazy, on-demand generation of sets of solutions. With these primitives available, there's no real need for Prolog's cut operation anymore and the resulting language has the benefit of allowing program transformations like unfolding to be performed safely.

After filling the semantic gap between Horn Clause programs and full Prolog, we further extend the expresiveness of the resulting language by emulating some

[1] This is the very reason we need to introduce them as an extra construct!

basic linear and intitionistic logic operators and on top of them a higher level *answer producer/ answer consumer construct* construct.

The software engineering virtues of intuitionistic implications for expressing modules and local information have been advocated by Dale Miller back in the 80's, during the original design of Lambda-Prolog [Mil89,Mil91] but, unfortunately, they have not been used yet in major commercial Prolog implementations or real life software projects. In this context, the second contribution of the paper is exploring the synergy of our 'first-order' engines, with linear and intuitionistic assumptions[2] by designing a high level `Answer<=Producer=>Consumer` primitive allowing alternative OR-answers from the Producer engine to be seen as linear assumptions by the `Consumer` engine. The construct works as if goal `Answer1 -o Answer2 -o ... -o Consumer` were executed on `Consumer`'s engine, with `-o` denoting linear implication and Answer1,Answer2,... being the instances of `Answer`, produced by executing `Producer`. These operations allow for a (possible infinite) stream of answers of an engine, computed through alternative derivations, to be used as needed in one and the same derivation of another engine.

We have tested our design through two different implementations, in the latest versions of BinProlog and Jinni, available from `http://www.binnetcorp.com`. The *engine operations* described in this paper have been used extensively in implementing some advanced networking constructs like mobile code and client/server applications in [Tar99a,TD98,Tar98c].

2 Kernel Prolog = Horn Clauses + Engines

2.1 Primitive Engine Operations

At an abstract level only two new primitive operations are needed to extend Horn Clause programs with *engines*. Each engine works as a separate Horn Clause interpereter. *Engine constructors initialize engines with a goal and an answer pattern.*

```
new_engine(Goal,AnswerPattern,Handle)
```

creates a new Horn Clause solver, uniquely identified by Handle, which shares code with the currently running program and is initialized with resolvent Goal. AnswerPattern is a term, usually a list of variables occuring in Goal. *Answer constructors build successive answers generated by an engine, on demand.*

```
new_answer(Handle,AnswerInstance)
```

harvests the result of a new AND-branch of the search tree generated by Goal, as a instance of AnswerPattern. If AnswerPattern is **no** that the engine is stopped, otherwise it returns a new answer, of the form **the(AnswerInstance)** or **no** if no more answers are available. Note that once **no** has been returned, all subsequent uses of this primitive will return **no**. Note also that no bindings are propagated to the original Goal or AnswerPattern when **new_answer** retrieves an answer, i.e. AnswerInstance is obtained by first standardazing apart (renaming) the variables in Goal and AnswerPattern, and then backtracking over its alternative answers in a separate Prolog interpreter.

Let us call Kernel Prolog the language obtained by adding to Horn Clauses (pure Prolog) the two previously introduced built-in operations **new_engine/3** and **new_answer/2** and an infinite supply of new constant symbols denoting engines[3].

[2] λProlog and Liniear Logic based constructs present in BinProlog

[3] The embeded lightweight interpreters described in [Tar98c] used as a scripting layer in our networked agent programming language Jinni, are essentially implementations of Kernel Prolog. BinProlog's engines also support equivalent functionality, except that the original builtins signal the end of a solution stream with failure, instead of using the no/0, the/1 functors for returning results.

2.2 Source level extensions through new definitions

We will know introduce through definitions in Kernel Prolog, a number of 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). We have added a **K** to the usual names of the primitives, to make the code presented in the paper runnable directly in BinProlog [Tar98a], without name conflicts.

Negation and `once/1` These constructs are implmented simply by discarding all but the first solution produced by an engine.

```
% returns the(X) or no as first solution of G
first_solutionK(X,G,Answer):-
  new_engineK(G,X,E),
  new_answerK(E,R),
  Answer=R.

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

% succeds withount binding G, if G fails
notK(G):-first_solutionK(_,G,no).
```

Metacalls Metacalls can be seen as instances of engine operations. We add explicit backtracking to explore the answers one by one.

```
callK(Goal):-
  new_engineK(Goal,Goal,E),
  collect_callK(E,Goal).

% backtracks over the answer sequence
collect_callK(E,Goal):-
  new_answerK(E,the(Answer)),
  collect_moreK(E,Answer,Goal).

collect_moreK(_,Answer,Answer).
collect_moreK(E,_,Answer):-collect_callK(E,Answer).
```

If-then-else Once we have first solution and metacall, emulating if-then-else is easy (probably Prolog folklore).

```
% executes Then if Cond succeeds otherwise executes Else
ifK(Cond,Then,Else):-
  first_solutionK(successful(Cond,Then),Cond,R),
  select_then_elseK(R,Cond,Then,Else).

% selects and calls the Then or Else part of a conditional
select_then_elseK(the(successful(Cond,Then)),Cond,Then,_):-callK(Then).
select_then_elseK(no,_,_,Else):-callK(Else).
```

All-solution predicates We only describe findall here, as other all solution predicates can be built on top of it.

```
% if G has a finite number of solutions
% returns a list Xs of copies of X each
% instantiated correspondingly
findallK(X,G,Xs):-
```

```
   new_engineK(G,X,E),
   new_answerK(E,Answer),
   collect_all_answersK(Answer,E,Xs).

% collects all answers of an Engine
collect_all_answersK(no,_,[]).
collect_all_answersK(the(X),E,[X|Xs]):-
  new_answerK(E,Answer),
  collect_all_answersK(Answer,E,Xs).
```

Term copying and instantiation state detection As standardazing variables apart upon return of answers is part of the semantics of `new_answerK/2`, term copying is just computing a first solution to `true/0`. Implementing `varK/1` is then simply possibility of having copies unifiable with two distinct constants.

```
% creates a copy of G with variables uniformly
% substituted with new variables not occuring
% in the current resolvent)
copy_termK(X,CX):-first_solutionK(X,true,the(CX)).

% true if X is currently a free varaible
varK(X):-copy_termK(X,a),copy_termK(X,b).
```

How tor run all this? To make the code actually runnable as is[4], the following BinProlog implementation of the two primitives is provided:

```
new_engineK(Goal,AnswerPattern,Handle):-
  open_engine(Goal,AnswerPattern,Handle).

new_answerK(Engine,Answer):-
  ask_engine(Engine,X)->Answer=the(X)
; Answer=no.
```

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** - this justifies the name Kernel Prolog for the resulting language. As Kernel Prolog contains negation as failure, following [DEDC96] we can, in principle use it for an executable specification of full Prolog.

2.3 A reflective meta-interpreter

Finally, let us show that Kernel Prolog has a simple reflective interpreter (actually much simpler than full Prolog - which has to deal with CUT).

```
% basic reflective meta-interpreter

solveK(G):-
  onceK(reduceK(G,NewG)),
  callK(NewG).

% reflective reducer
% the simplest such beast is: reduceK(X,X).

reduceK(G,G):-is_builtin(G).
reduceK(',',(A,B),',',(solveK(A),solveK(B))).
reduceK(G,',',(clause(G,Gs),solveK(Gs))).
```

[4] The reader is invited to experiment online with the code in this paper following the link provide at the end of the abstract

The presence of such a simple reflective meta-interpreter makes Kernel Prolog a very interesting target for program transformations like unfolding, and in particular, for partial deduction based source-level optimizations.

2.4 Emulating dynamic database state with engines

While not really needed at this point as a workaround for the lack of expressivenes of pure Prolog:-), let us show how a form of dynamic database operations can be emulated using the state of multiple engines.

Apparently, this looks a difficult problem because `new_engine` and `new_answer` operations provide a fixed communication flow between the answer producer and the answer consumer engine. Note however, that we can use the new engine creation operation to emulate both linear (usable once) and conventional assert-style operations, by keeping the set of engines representing references to 'dynamic clauses' on a list, as in:

```
assertK(Clause,Engines,[E|Engines]):-
  new_engineK(repeat,Clause,E).

linear_assertK(Clause,Engines,[E|Engines]):-
  new_engineK(true,Clause,E).

clauseK(Engines,Head,Body):-
  member(E,Engines),
  new_answerK(E,the((Head:-Body))).
```

The trick in the case of conventional `assertK` is to force indefinite backtracking inside the producer engine, with `repeat/0`, a pure predicate defined as `repeat. repeat:-repeat.`

On the other hand, the usable once, `linear_assertK` operation only executes the goal `true/0` - and therefore it will only succeed once, before returning the answer `no`.

We have not shown how a `retract`-like operation is implemented - but this can easily achieved by removing the engine reference from the list of engines, so that `clauseK/3` cannot see it anymore.

Note that these dynamic database operations are in fact backtrackable, as they depend on the list of engines maintained by the consumer program. As such, they are closer to BinProlog's linear and intuitionistic assumptions [TDF96]. Scoped implications (as in Lambda Prolog, Lygon [WH95], Lolli [HM94,Hod94]) can be implemented using logical varables to close the scope of an implication[5].

However, assuming that engine numbers are allocated in increasing order or that the list of engines is available as a global state variable, failure persistent dynamic database operations can be emulated as well.

Note that our reflective meta-interpreter can be easily extended to deal with dynamic clauses, as follows.

```
% extended meta-circular interpreter with
% emulated dynamic clauses

% given a set of engines Es representing
% dynamic clauses, solve the goal G
% based on reflection and use of the
% new clauses encapsulated in the engines

solveK(Es,G):-
  onceK(reduceK(Es,G,NewG)),
```

[5] This is how they are actually implemented in BinProlog [Tar98a]

```
  callK(Es,NewG).

reduceK(_,G,G):-is_builtin(G).
reduceK(Es,','(A,B),','(solveK(Es,A),solveK(Es,B))).
reduceK(Es,G,','(clauseK(Es,G,Gs),solveK(Es,Gs))).

callK(_,G):-callK(G).
callK(Es,G):-clauseK(Es,G,B),callK(Es,B).
```

2.5 Basic file and socket I/O in Kernel Prolog

Let us now incorporate basic I/O in our design. Let us supose that 2 reserved engines (**stdin, stdout**) are built-in in the interpreter.

Then basic I/O would look as follows:

```
get_code(X):-new_answer(stdin,Answer),convert_code(Answer,X).

convert_code(the(X),X).
convert_code(no,-1).

put_code(X):-new_answer(stdout,X).
```

Streams used for file and socket I/O can be seen as special purpose engines with operations like:

```
open_stream(File,Stream):-
  new_engine(file(File),_,Stream).

get_code(Stream,Code):-
  new_answer(Stream,Answer),
  convert_code(Answer,Code).
```

Note that one can assume that arithmetics is part of Kernel Prolog through the usual extension of Horn Clauses with successor arithmetics.

With these additions, it turns out that Kernel Prolog is a fairly expressive languge. Unlike pure Horn Clause Prolog and unlike weaker lanaguges based on Prolog + negation, which have been extensively studied in the past, Kernel Prolog has enough power to be used as a basis for building compact Prolog implementations as well as for modular library components.

2.6 Semantic Issues in Kernel Prolog

While providing a formal semantics of Kernel Prolog needs more research and it might need methods beyond the techniques used for less expressive logic languages, let us give a sketch of the issues involved.

Note that kernel Prolog is a superset of Horn Clauses with a form of metacall facility. Related semantic issues have been studied [War81,HL89] and are now fairly well understood.

Like in the case of `findall`, solutions returned from an engine contain new variables. As such, their formal description cannot be covered through a straightforward adaptation of work like [Llo87,Apt90]. The issues are in fact similar with those involved in describing formally AND-parallel execution of Prolog programs [SH91,Pon97], with resolution theory needing to be adapted to deal with separate computations of answers within the same AND-branch.

Clearly, the two new builtins added to Horn Clause Prolog lack an obvious declarative semantics. On the other hand, the *dialog* between a *master* Horn Clause engine creating a new engine for execution of one of its subgoals and the *slave* engine which

returns new answers on demand is well known to any Prolog user as it is essentially the same as querying a Prolog interpreter through a toplevel command interpreter[6].

However, in a more declarative framework, it can be the case that a suitable semantics for our engine operations can be given in terms similar to that of streams in functional languages.

Multi-threaded execution of Kernel Prolog Although engines were running on separate threads in our first Jinni implementation of Kernel Prolog, we have decided later to keep engine operations separate from the underlaying multi-threading model, as they are basically orthogonal concepts.

Still, flexible combination of engines and multi-threading is possible, with minimal new ontology. Let us suppose that two multi-threading control operations are available:

```
% Launches Goal on separate (background) Thread
bg(+Goal,-ThreadHandle)

% waits for Thread to terminate and propagates bindings for JoinPattern
join(+Thread,JoinPattern)
```

Then, something like

```
?-new_engine(Goal,Answer,Engine),
  bg(new_answer(Engine),Thread),
  ...work on something else...,
  join(Thread,Answer),
  ...use Answer...
```

would allow speculative computation of answers as well as synchronization and data exchange on completed tasks. A more flexible communication pattern (which includes coordination through *blackboard constraints* [Tar99b]) has been implemented in both Jinni and BinProlog. Expressing such functionality in terms of 'pure' Kernel Prolog is subject of ongoing work.

3 Memory managment and performance issues for multiple engine systems

As this paper focuses on a logical reconstruction of Prolog on top of its logical Horn Clause kernel, and not on implementation of engines as such, we will only overview here a more detailed design (actually implemented in BinProlog [Tar98a]) which allows precise control on allocation and deallocation of memory resources.

To create a new engine having its own stack group (heap, local stack and trail) we use:

```
create_engine(HeapSize,StackSize,TrailSize,Handle)
```

The Handle is a unique integer denoting the engine for further processing. To 'fuel' the engine with a goal and an expected answer variable we use:

```
load_engine(Handle,Goal,AnswerVariable)
```

No processing, except the initialization of the engine, takes place and no answer is returned as result of this operation.

To get an answer from the engine we use:

[6] The original motivation for Kernel Prolog was encapsulating this interaction with a human oracle in a way that makes it programmable - i.e. transforms Horn Clause Prolog into a real life agent programming language. This objective has been achieved through the implementation of Jinni [Tar98c].

```
ask_engine(Handle,Answer)
```

When the stream of answers reaches its end, `ask_engine/2` will simply fail.

To destroy an engine an free its memory areas we use:

```
destroy_engine(Handle)
```

If we want to submit another query before using the complet stream of answers, it is more efficient to reuse an existing engine with `load_engine/3`, instead of destroying it and creating a new one. It is safe to discard an engine on failure, before retuning the answer **no**. It is also safe to discard an engine on backtracking over **new_egine/3**.

As engines are assigned to real processors in multi-threaded in implementations like BinProlog's new native win32 threads package [Tar98b], this reusability of a given engine for execution of multiple goals allows tighter programmer control over the resurces of a system.

Automatic garbage collection of engines can be achieved with a simple reference counting mechanism or by extending the reachabilty logic of Prolog's garbage collector to engine handles.

In BinProlog, engine creation/destruction is a constant time operation. In our newer Jinni implementation [Tar99b], engine memory is also allocated on demand, through the underlying Java system, with engine creation requiring small constant initial space and engines being garbage collected automatically.

Note that it is always possible to move to the underlying implementation language frequently used engine operations if the (constant) overhead of having them defined at source level is judged too high.

4 Linear Assumptions and Orthogonal Engines

4.1 A high-level Producer/Consumer interaction pattern

We have sketched how linear and intuitionistic implications can be expressed in Kernel Prolog. To make the code in the paper fully executable we will use here their equivalent BinProlog implementation.

We will now show, how, with engines and assumptions, we can provide a high level, encapsulated Consumer/Producer interaction pattern.

Let's start by describing the new construct, through the following example:

```
?- answer(X)<=member(X,[1,2,3])=>consumer(Xs),write(Xs),nl.
```

where `consumer/1` is defined as:

```
consumer([X|Xs]):-answer(X),consumer(Xs).
consumer([]).
```

will execute exactly as the goal

```
?- answer(1) -o answer(2) -o answer(3) -o consumer(Xs),write(Xs),nl.
```

in a logic language providing linear implication.

4.2 Communication through Linear Assumptions

This construct, implemented as a unique primitive, is based on the synergy between multiple engines and intuitionistic and linear implications (as well as their continuation passing couterparts).

The concrete syntaxe is $Answer \Leftarrow Producer \Rightarrow Consumer$ or $Answer \Leftarrow Producer$ when the Consumer is the (implicit) current AND-continuation.

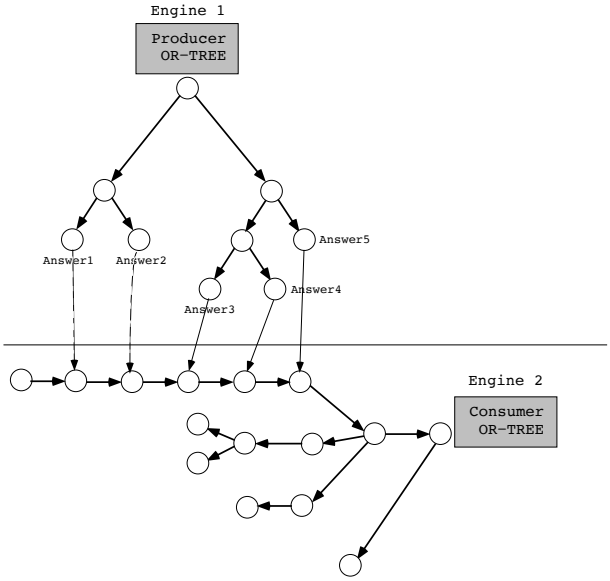

Fig. 1. Orthogonal engines

Producer is started on a separate engine. It generates linear facts of nased on the pattern `Answer`, from computed answers of Goal, originating from *distinct OR-branches*. They are usable by `Consumer` as if they were assumed through use of linear implication, i.e. they are consumed one-by-one. This hides the intricacies of starting a new 'orthogonal' logic engine into a simple, easy to use abstraction.

The actual implementation is given in the Appendix (see also file `library/engines.pl`, available from the `http://www.binnetcorp.com` BinProlog download page).

The following example shows how an OR-stream of answers, seen as linear assumptions from an 'orthogonal' engine, can be used in an different AND-branch:

```
?- a(X)<=member(X,[1,2,3])=>(a(A),a(B),a(C),write([A,B,C]),nl).

[1,2,3]
```

5 Related work

Engine-like constructs [Sch97] have been part of systems like Oz [HVRS97,VRHB+97] and have been used in the past for encapsulated search - arguably more flexible than Prolog's fixed search mechanism. The main differences with our engines come from their different intended use:

- Oz engines are not separated from the underlaying multi-threading model
- Oz engines are not simple Horn Clause processors, they are part of Oz's more complex, multi-paradigm execution model
- Oz engines are used for a different purpose, i.e. to program alternative search algorithms or visual debuggers, while our objective in this paper is a reconstruction of Prolog based on a Horn Clause kernel

New languages based on relatively pure subsets of Prolog like Mercury [SHC98] have been designed as targets of more efficient implementation technologies and for their reliability in building large software systems. Flexible execution order is being used in systems like SICStus, GNU-Prolog and B-Prolog to support constraint solving and in systems like XSB for synchronizing answer production and answer consumption in

tabling. 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 prower as a result of having multiple pure Prolog interpreters as first order objects, has not been explored yet, to our best knowledge. While our techniques have been described in a BinProlog context, they can be ported to any Prolog or Prolog-like language implementation which is *reentrant* - i.e. able to run more than one instance of the interpreter or runtime system at the same time - a requierment also for supporting native multi-threading.

6 Future work

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 prgramming 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 implementation will use a fast Horn Clause only reentrant emulator, to be extended towards full Prolog based on the Kernel Prolog design described in this paper.

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

- a study of Kernel Prolog's invariance under program transformations (unfolding)
- expressivenes of multi-engine Datalog (conjectured as being Turing-equivalent)
- type checking / type inference mechanisms for Kernel Prolog
- lightweight engine creation and engine reuse techniques for Kernel Prolog
- adapting Kernel Prolog for spoken I/O, using a new prototype based Natural Language style predicate syntax
- Kernel Prolog as a basis of embedded Prolog component technology and Prolog based Palm computing
- implementation techniques for lighweight Web based Kernel Prolog interpreters
- executable specification of ISO Prolog in terms of Kernel Prolog

7 Conclusion

First order logic engines are introduced to overcome two major expressiveness problems of pure logic programming languages: the inability to abstractly specify communication between distinct OR-branches and the ability to describe at source level various non-sequential execution models. We have shown also that meta-programming and dynamic database operations are easily emulated in Kernel Prolog. By collapsing the semantic gap between Horn Clause logic and (most of) the full Prolog language into two 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.

References

[Apt90] K.R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, North-Holland, 1990.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, Berlin, 1996. ISBN: 3-540-59304-7.

[HL89] P.M. Hill and J.W. LLoyd. Analysis of metaprograms. In H. Abramson and M.H. Rogers, editors, *1st Workshop on Meta-Programming in Logic Programming*, Bristol, UK, 1989. MIT Press.

[HM94] Joshua S. Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[HVRS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An Overview of the Design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, 1997. ACM Press.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic computation — Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.

[Mil89] D. A. Miller. Lexical scoping as universal quantification. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Cambridge, Massachusetts London, England, 1989. MIT Press.

[Mil91] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[Pon97] Pontelli, E. and Gupta, G. W-ACE: A Logic Language for Intelligent Internet Programming. In *Proc. of IEEE 9th ICTAI'97*, pages 2–10, 1997.

[Sch97] 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.

[SH91] Kish Shen and Manuel V. Hermenegildo. A simulation study of Or- and independent And-parallelism. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming Proceedings of the 1991 International Symposium*, pages 135–151, Cambridge, Massachusetts London, England, 1991. MIT Press.

[SHC98] Zoltan Somogyi, Fergus Hederson, and Thomas Conway. The Mercury Language Web Site. 1998. http://www.cs.mu.oz.au/research/mercuryl.

[Tar98a] 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.

[Tar98b] Paul Tarau. BinProlog 7.0 Professional Edition: User Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

[Tar98c] Paul Tarau. Towards Inference and Computation Mobility: The Jinni Experiment. In J. Dix and U. Furbach, editors, *Proceedings of JELIA'98, LNAI 1489*, pages 385–390, Dagstuhl, Germany, October 1998. Springer. invited talk.

[Tar99a] Paul Tarau. A Logic Programming Infrastructure for Mobile Internet Agents. In *The Logic Programming Paradigm: Current Trends and Future Directions.* Springer, LNAI, 1999.

[Tar99b] Paul Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.

[TD95] Paul Tarau and Bart Demoen. Higher-Order Programming in an OR-intensive Style. In Manuel Hermenegildo and Pedro Lopez, editors, *Proceedings of the 1995 COMPULOG-NET Workshop and Area Meeting on Parallelism and Implementation Technology*, 1995.

[TD98] Paul Tarau and Veronica Dahl. Mobile Threads through First Order Continuations. In *Proceedings of APPAI-GULP-PRODE'98*, Coruna, Spain, July 1998.

[TDF96] Paul Tarau, Veronica Dahl, and Andrew Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In Joxan Jaffar and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, December 1996. "Springer".

[VRHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhouer. Mobile Objects in Distributed Oz. *ACM TOPLAS*, 1997.

[War81] D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.

[WH95] Michael Winikoff and James Harland. Implementing the Linear Logic Programming Language Lygon. In John Lloyd, editor, *International Logic Programming Symposium*, pages 66–80, Portland, Oregon, December 1995. MIT Press.

Appendix - Orthogonal Engines With Linear Assumptions

```
% executes Producer on an 'orthogonal' OR-engine
Answer<=Producer=>Consumer:-!, % the consumer is explicit
  cross_run(Producer,Answer,Consumer).
Answer<=Producer:-             % the consumer is in the 'future'
  cross_run_with_current_continuation(Producer,Answer).

% transfers Answer from Producer to Consumer
cross_run(Producer,Answer,Consumer):-
  % new engine with initial heap,stack,trail
  default_engine_params(H,S,T),
  cross_run(H,S,T,Producer,Answer,Consumer).

% creates Answer pulling mechanism in Consumer
% for 2 implication variant
cross_run(H,S,T,Producer,Answer,Consumer):-
   compound(Answer),copy_term(Answer,A),
   new_engine(H,S,T,Producer,Answer,Handle),
   (A:-ask_or_kill(Handle,A))=>Consumer,
   destroy_engine(Handle).

% transfers Answer from Producer to future Consumer
% to be found/created later as the current continuation is unfolded
cross_run_with_current_continuation(Producer,Answer):-
   default_engine_params(H,S,T),
   cross_run_with_current_continuation(H,S,T,Producer,Answer).

% creates Answer pulling mechanism in Consumer
% for 1 implication variant
cross_run_with_current_continuation(H,S,T,Producer,Answer):-
   compound(Answer),copy_term(Answer,A),
   new_engine(H,S,T,Producer,Answer,Handle),
   assumei((A:-ask_or_kill(Handle,A))).

% queries and discards engines
ask_or_kill(Handle,Answer):-ask_engine(Handle,Answer),!.
ask_or_kill(Handle,_):-destroy_engine(Handle),fail.

% creates a new engine with initial Heap, Stack, Trail sizes
new_engine(H,S,T,Goal,Answer,Handle):-
  create_engine(H,S,T,Handle),
  load_engine(Handle,Goal,Answer).

% overridable default engine sizes for heap,stack,trail
default_engine_params(H,S,T):-assumed(engine_params(H,S,T)),!.
default_engine_params(128,32,32).
```