1 Inference and Computation Mobility with Jinni

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 introduce Jinni (**J**ava **IN**ference engine and **N**etworked **I**nteractor), a lightweight, multi-threaded, logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in networked client/server applications, as well as through applets over the Web.

Mobile threads, implemented by capturing first order continuations in a compact data structure sent over the network, allow Jinni to interoperate with remote high performance BinProlog servers for CPU-intensive knowledge processing and with other Jinni components over the Internet.

These features make Jinni a perfect development platform for intelligent mobile agent systems.

Jinni is fully implemented and is being used in a growing number of industrial and academic projects. The latest version is available from

`http://www.binnetcorp.com/Jinni`

Keywords: Java based Logic Programming languages, remote execution, Linda coordination, blackboard-based distributed logic programming, mobile code through first order continuations, intelligent mobile agents, distributed AI

1.1 Introduction

The paradigm shift towards networked, mobile, ubiquitous computing has brought a number of challenges which require new ways to deal with increasingly complex patterns of interaction: autonomous, reactive and mobile computational entities are needed to take care of unforeseen problems, to optimize the flow of communication, to offer a simplified, and personalized view to end users. These requirements naturally lead towards the emergence of *agent programs* with increasingly sophisticated inference capabilities, as well as autonomy and self-reliance.

Jinni is a new, lightweight, logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in networked client/server applications and thin client environments. By supporting multiple threads, control mobility and inference processing, Jinni is well suited for the development of intelligent mobile agent programs.

Jinni supports multi-user synchronized transactions and interoperates with the latest version of BinProlog [16] , a high performance, robust, multi-threaded Prolog system with ability to generate C/C++ code and standalone executables.

For acronym lovers JINNI can be read as: **J**ava **IN**ference engine and **N**etworked **I**nteractor, although its wishmaker status (high level, dense, network ubiquitous, mobile, etc. agent programming language) is an equally good reason for its name.

1.2 The world of Jinni

Jinni is based on a simple **Things**, **Places**, **Agents** ontology, borrowed from MUDs and MOOs [14,1,3,9,19,15].

Things are represented as Prolog terms, basically trees of embedded records containing constants and variables to be further instantiated to other trees.

Places are processes running on various computers with at least one *server component* listening on a port and a blackboard component allowing synchronized multi-user Linda [6,10] transactions, remote predicate calls and mobile code operations.

Agents are collections of threads executing a set of goals, possibly spread over a set of different **Places** and usually executing remote and local transactions in coordination with other **Agents**. Their state is distributed over the network of **Places** in the form of dynamic Prolog clauses and produced/consumed Linda facts[1]. Jinni does not provide a single abstract data type for agents or places because it is intended to be an infrastructure on top of which they are built in an application specific way. In a typical Jinni applications, as crisp abstractions emerge through development process, a hierarchy of Places and Agents is built. **Place** and **Agent** prototypes are clonable, support inheritance/sharing of **Things** and are easily editable/configurable using the visual tools of the underlying Java environments. Agent threads moving between places and agents moving as units can be supported. Places are used to abstract away language differences between processors, like for instance Jinni and BinProlog. They can contain the same or different code bases (contexts), depending on the applications requirements. As an extra feature, mobile code allows fast processing in Jinni by delegating heavy inference processing to high-performance BinProlog components.

[1] Jinni implements assert and retract in terms of non-blocking local Linda operations.

1.3 Jinni as a Logic Programming Java component

Jinni is implemented as a lightweight, *thin client* logic programming component, based as much as possible on fully portable, vendor and version independent Java code. Its main features come from this architectural choice:

- a trimmed down, simple, operatorless syntactic subset of Prolog,
- pure Prolog (Horn Clause logic) with leftmost goal unfolding as inference rule,
- multiple asynchronous inference engines running on separate threads,
- a shared blackboard to communicate between engines using a simple Linda-style subscribe/publish (in/out in Linda jargon) coordination protocol, based on associative search,
- high level networking operations allowing code mobility [2,12,11,5,20,13] and remote execution,
- a straightforward Jinni-to-Java translator allowing packaging of Jinni programs as Java classes
- ability to load code directly from the Web and to show third party Web documents (text, graphics, multi-media) by controlling applet contexts in browsers
- backtrackable assumptions [17,8] implemented through trailed, overridable undo actions, also supporting Assumption Grammars, a variant of extended DCGs

Jinni's spartan return to (almost) pure Horn Clause logic does not mean it is necessarily a weaker language. Expressiveness of full Prolog is easily attained in Jinni by combining multiple engines. The magic is similar to just adding another stack to a Push Down Automaton: this morphs it into a Turing machine[2]! As we will show in section 1.5.1 control constructs like `if-then-else`, negation as failure, once/1, and operations like (eager or lazy) findall are easily expressed in terms of cooperating engines without using CUT or side effects. Engines give transparent access to the underlying Java threads and are used to implement local or remote, lazy or eager findall operations, negation as failure, if-then-else, etc. at source level. Inference engines running on separate threads can cooperate through either predicate calls or through an easy to use flavor of the Linda coordination protocol.

Remote or local dynamic database updates (with deterministic, synchronized transactions with immediate update semantics) make Jinni an extremely flexible Agent programming language. Jinni is designed on top of dynamic, fully garbage collectible data structures, to take advantage of Java's automatic memory management.

[2] Of course, Horn Clause logic is already Turing complete. What we mean here by expressiveness is just the informal concept of being able to express new constructs at source level in a natural way.

1.4 Basic agent programming with Jinni

Agents behaviors are implemented easily in terms of synchronized in/out Linda operations and mobile code. As an example of such functionality, we will describe the use of two simple chat agents, which are part of Jinni's standard library:

Window 1 : a reactive channel listener

```
?-listen(fun(_)).
```

Window 2 : a selective channel publisher

```
?-talk(fun(jokes)).
```

They implement a front end to Jinni's associative publish/subscribe abilities. The more general pattern `fun(_)` will reach all the users interested in instances of `fun/1`, in particular `fun(jokes)`. However, someone publishing on an unrelated channel e.g. with `?-talk(stocks(nasdaq)).` will not reach fun/1 listeners because stocks(nasdaq) and fun(jokes) channel patterns are not unifiable.
A more realistic stock market agent's buy/sell components look as follows:

```
sell(Who,Stock,AskPrice):-
  % triggers a matching buy transaction
  notify_about(offer(Who,Stock,AskPrice)).

buy(Who,Stock,SellingPrice):-
  % runs as a background thread
  % in parallel with other buy operations
  bg(try_to_buy(Who,Stock,SellingPrice)).

try_to_buy(Me,Stock,LimitPrice):-
  % this thread connects to a server side constraint and waits
  % until the constraint is solved to true on the server
  % by a corresponding sell transaction
  wait_for(offer(You,Stock,YourPrice),[ % server side mobile code
    lesseq(YourPrice,LimitPrice),
    local_in(has(You,Stock)),
    local_in(capital(You,YourCapital)), % server side 'local' in/1
    local_in(capital(Me,MyCapital)),    % operations
    compute('-',MyCapital,YourPrice,MyNewCapital),
    compute('+',YourCapital,YourPrice,YourNewCapital),
    local_out(capital(You,YourNewCapital)),
    local_out(capital(Me,MyNewCapital)),
    local_out(has(Me,Stock))
  ]).
```

Note that this example also gives a glimpse on Jinni's multithreaded client/server design (background thread launching with `bg`), as well as its *server side constraint solving ability* (`wait_for, notify_about`). We will now describe these features and Jinni's architecture in more detail.

1.5 What's new in Jinni

1.5.1 Programming with Engines

Jinni inherits from BinProlog's design the ability to launch multiple Prolog engines having their own state. An engine can be seen as an abstract datatype which produces a (possibly infinite) stream of solutions as needed. To create a new engine, we use:

```
  new_engine(Goal,AnswerTemplate,Handle)
```

Computation starts by calling `Goal` and producing, on demand, *instances* of `AnswerTemplate` The `Handle` is a unique Java Object denoting the engine, assigned to its own thread. It will be used, for instance, to ask answers, one at a time, or to kill the engine.

To get an answer from the engine we use:

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

Note that Answer is an instance of the AnswerTemplate pattern passed at engine creation time, by `new_engine`. Each engine can be seen as having its own virtual garbage collection process. Engines backtrack independently using their (implicit) choice-point stack and trail during the computation of an answer. Once computed, an answer is copied from an engine to the master engine which initiated it. Extraction of answers from an engine is based on a *monitor object* which synchronizes the producer and the consumer of the answer.

When the stream of answers reaches its end, `ask_engine/2` will simply fail. The resolution process in an engine can be discarded at any time with `stop_engine/1`. This allows avoiding the cost of backtracking in the case when a single answer is needed. The following example in the Jinni distribution) shows how to extract one solution from an engine:

```
one_solution(X,G,R):-
  new_engine(G,X,E),
  ask_engine(E,Answer),
  stop_engine(E),eq(Answer,R).
```

Note that `new_engine/3` speculatively starts execution of Goal on a new thread and that either a term of the form **the(X)** or **no** is returned by `ask_answer`. Synchronization with this thread is performed when asking an answer, using a special monitor object. By extending the monitor `Answer` class, one can easily implement speculative execution allowing a bounded number of answers to be computed in advance (a form of OR-parallel execution). Note that answers are produced in standard Prolog (LD-resolution) order. This design choice is needed for improved predictability, keeping in mind that Jinni, as a multi-threaded environment, is already subject to more complex operational semantics.

It is quite surprising how simply all essential control constructs of Prolog can be built on top of this `one_solution/3` primitive[3].

```
if(Cond,Then,Else):-
  one_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.

once(G):-one_solution(G,G,the(G)).

not(G):-one_solution(G,G,no).

copy_term(X,CX):-one_solution(X,true,the(CX)).

bg(Goal):-new_engine(Goal,_,_). % spawns a new background thread
```

Similarly, findall/3 is emulated easily by iterating over ask_engine/2 operations.

```
find_all(X,G,Xs):-
  new_engine(G,X,E),
  once(extract_answers(E,Xs)).

extract_answers(E,[X|Xs]):-
  ask_engine(E,the(X)),
  extract_answers(E,Xs).
extract_answers(_,[]).
```

Note that lazy variants of `findall` can be designed by introducing a stream-inspired concept of *lazy list*. This can be implemented by using a special JavaObject tail containing an Engine handle, which overrides default unification into a call to `ask_engine` to instantiate the tail to a new answer, if available, and to the empty list otherwise. To implement this functionality we would only have to extend Jinni's attributed variable class (AVar) with minimal new unification code.

[3] In fact, for efficiency reasons, now Jinni has a `first_solution/3` builtin, which does not require a separate engine. It is implemented simply by throwing a special purpose *exception* when the first solution is found.

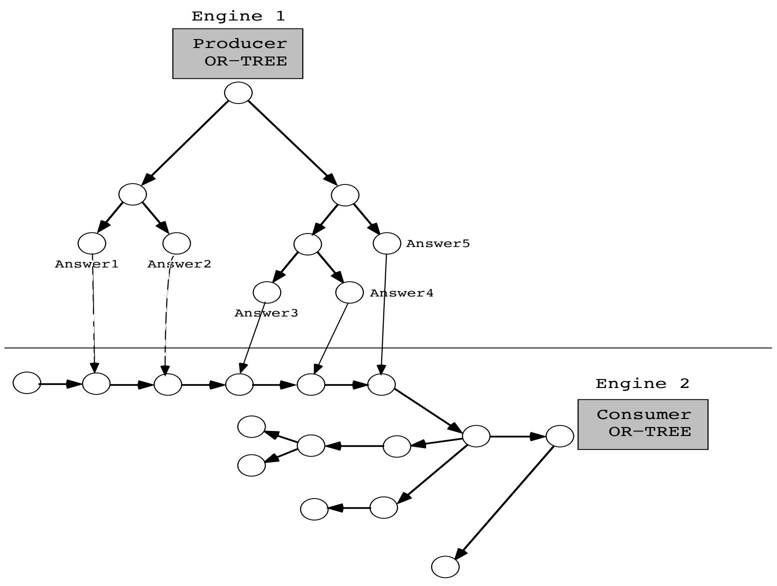

Fig. 1.1. Orthogonal Engines

Currently Jinni supports a programmer managed, *on demand* answer production through multiple *orthogonal*[4] *engines* (Fig. 1.1). We can see this programming pattern as the ability of an AND-branch of an engine to collect answers from multiple OR-branches of another engine. They give to the programmer the means to see the answers produced by an engine as a (lazy) data stream, and to control their production, in a way similar to Java's `Enumeration` interface.

In fact, by using orthogonal engines, a programmer does not really need to use findall and other similar predicates anymore - why accumulate answers eagerly on a list which will get scanned and decomposed again, when answers can be produced on demand? Still, encapsulating answers in lazy lists, can make their use even more transparent to Prolog programmers, as only one new construct

```
lazy_findall(AnswerTemplate, Goal, LazyAnswerList)
```

reusing a programmer's knowledge of `findall/3` would be needed.

1.5.2 Coordination and remote execution mechanisms

Our networking constructs are built on top of the popular Linda [6,10,7] coordination framework, enhanced with unification based pattern matching,

[4] We call them orthogonal (a geometric metaphor) as their execution proceeds independently.

remote execution and a set of simple client-server components melted together into a scalable peer-to-peer layer, forming a 'web of interconnected worlds' (Fig 1.2):

```
out(X): puts X on the server
in(X):  waits until it can take an object matching X from the server
all(X,Xs): reads the list Xs matching X currently on the server
the(Pattern,Goal,Answer): starts a thread executing Goal on server
```

The `all/2` operation, fetching the list of all matching terms is used instead of (cumbersome) backtracking for alternative solutions over the network. Note that the only blocking operation is `in/1`. Blocking `rd/1` is easily emulated in terms of `in/1` and `out/1`, while non-blocking `rd/1` is emulated with `all/2`. For expressiveness, the following derived operations are provided:

- `cout/1`, which puts a term on the blackboard only if none of is instances are present,
- `cin/1` which works like `in/1` but returns immediate failure if a matching term is absent
- `when/1` (a more efficient a non-blocking rd/1)

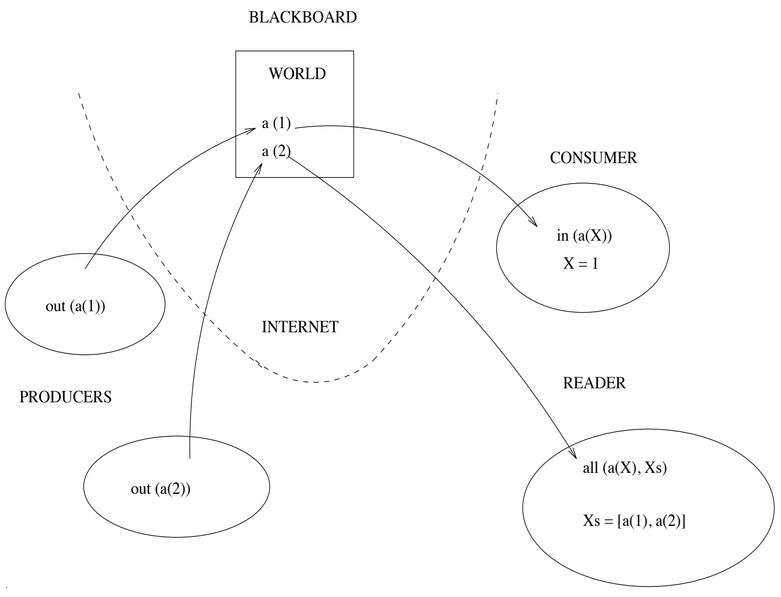

Fig. 1.2. Basic Linda operations

1.5.3 Server-side constraint solving

A natural extension to Linda is to use constraint solving for the selection of matching terms, instead of plain unification. This is implemented in Jinni through the use of 2 builtins:

Wait_for(Term,Constraint): waits for a Term such that Constraint is true on the server, and when this happens, it removes the result of the match from the server with an in/1 operation. Constraint is either a single goal or a list of goals [G1,G2,..,Gn] to be executed on the server.

Notify_about(Term): notifies the server to give this term to any blocked client which waits for it with a matching constraint i.e.

```
notify_about(stock_offer(nscp,29))
```

would trigger execution of a client having issued

```
wait_for(stock_offer(nscp,Price),less(Price,30)).
```

The use of server side constraint execution was in fact suggested by a real-life stock market application. In a client/server Linda interaction, triggering an atomic transaction when data verifying a simple arithmetic inequality becomes available, would be expensive. It would require repeatedly taking terms out of the blackboard, through expensive network transfers, and put them back unless the client can verify that a constraint holds. Our server side implementation checks a constraint only after a match occurs between new incoming data and the head of a suspended thread's constraint checking clause, i.e. a basic indexing mechanism is used to avoid useless computations. On the other hand, a mobile client thread can perform all the operations atomically on the server side, using local operations on the server, and come back with the results. The (simplified) server side fragment showing the implementation of `wait_for and notify_about` is as follows:

```
wait_for(Pattern,Constraint):-
  if(take_pattern(available_for(Pattern),Constraint),
     true,
     and(
      local_out(waiting_for(Pattern,Constraint)),
      local_in(holds_for(Pattern,Constraint))
     )
  ).

notify_about(Pattern):-
  if(take_pattern(waiting_for(Pattern,Constraint),Constraint),
     local_out(holds_for(Pattern,Constraint)),
     local_out(available_for(Pattern))
  ).

% takes the first matching Pattern for which Constraint holds
take_pattern(Pattern,Constraint):-
```

```
  local_all(Pattern,Ps),
  member(Pattern,Ps),
  Constraint,
  local_cin(Pattern,_).
```

Note that each time the head of the waiting clause matches incoming data, its body is (re)-executed. It would be interesting to explore use of *memoing* to reduce re-execution overhead. Although termination of constraint checking is left in the programmer's hand, only one thread is affected by a loop in the code, the server's integrity as such not being compromised. We think that improvement of implementation technology for server side constraint solving in a blackboard based framework rises some challenging open problems. Moreover, incorporating server-side symbolic constraint reducers (CLP, FD or interval based) can dramatically improve performance for large scale problems.

1.5.4 Mobile Code: for expressiveness and for acceleration

An obvious way to accelerate slow Prolog processing for a Java based system is through use of native (C/C++) methods. The simplest way to accelerate Jinni's Prolog processing is by including BinProlog through Java's JNI (as implemented in the latest version of our BinProlog/C/Java interface).

However, a more general scenario, also usable for applets not allowing native method invocations is the use of a *remote accelerator.* This is achieved transparently through the use of *mobile code.*

Code, state and computation mobility The Oz 2.0 distributed programming proposal of [20] makes *object mobility* more transparent, although the mobile entity is still the *state* of the objects, not *live code.*

Mobility of *live code* is called *computation mobility* [4]. It requires interrupting execution, moving the state of a runtime system (stacks, for instance) from one site to another and then resuming execution. Clearly, for some languages, this can be hard or completely impossible to achieve.

General Magic's Telescript and Odissey [11] agent programming framework, IBM's Java based *aglets* [12] as well as Luca Cardelli's Oblique [2] have pioneered implementation technologies achieving *computation mobility.*

Jinni's live code mobility In the case of Jinni, computation mobility is used both as an *accelerator* and an *expressiveness lifting* device. A live thread will migrate from Jinni to a faster remote BinProlog engine, do some CPU intensive work and then come back with the results (or just sent back results, using Linda coordination). A very simple way to ensure atomicity and security of complex networked transactions is to have the agent code move to the site of the computation, follow existing security rules, access possibly large databases and come back with the results.

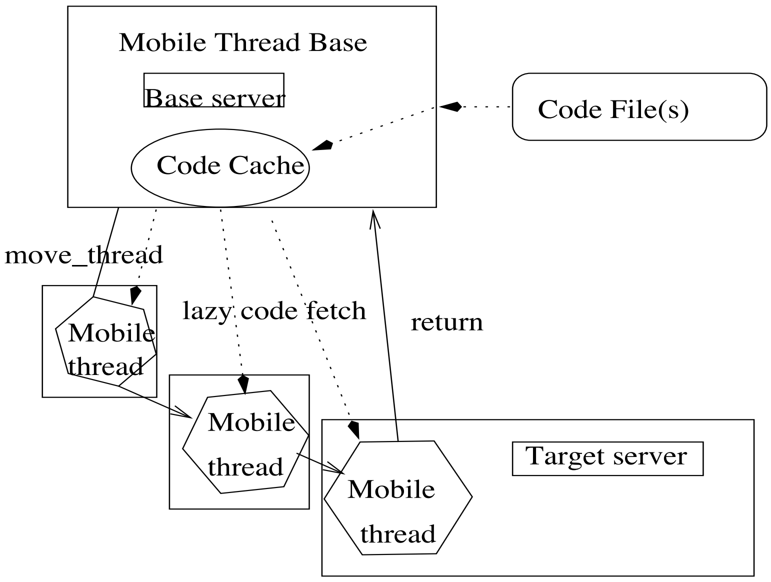

Fig. 1.3. Launching a mobile thread from its base

Jinni's mobile computation is a scaled down, simplified subset of BinProlog's mobile computation facilities. They are both based on the use of *first order continuations* i.e. encapsulated future computations, which can be easily suspended, moved over the network, and resumed at a different site. As continuations are first-order objects both in Jinni and BinProlog, the implementation is straightforward [?] and the two engines can interoperate transparently by simply moving computations from one to the other.

The target side waits in server mode. Once the continuation is received on the target side, the source might spawn a server thread as well, ready to execute code fetching and persistent database update requests from its mobile counterpart on the target side.

Fig. 1.3 shows the connections between a mobile thread and its base.

In the case of Jinni two simple **move/0** and **return/0** operations are used to transport computation to the server and back. The client simply waits until computation completes, when bindings for the first solution are propagated back:

```
Window 1: a mobile thread

?-there,move,println(on_server),member(X,[1,2,3]),
        return,println(back).
back
X=1;
no.
```

```
Window 2: a server

?-run_server.
on_server
```

In case return is absent, computation proceeds to the end of the transported continuation. Note that mobile computation is more expressive and more efficient than remote predicate calls as such. Basically, it *moves once*, and executes on the server *all future computations* of the current AND branch until a return instruction is hit, when it takes the remaining continuation and comes back. This can be seen by comparing real time execution speed for:

```
?-there,for(I,1,1000),run(println(I)),fail.

?-there,move,for(I,1,1000),println(I),fail.
```

While the first query uses `run/1` each time to send a remote task to the server, the second moves once the full computation to the server where it executes without further requiring network communications. Note that the `move/0, return/0` pair cut nondeterminism for the transported segment of the current continuation. This avoids having to transport state of the choice-point stack as well as implementation complexity of multiple answer returns and tedious distributed backtracking synchronization. Surprisingly, this is not a strong limitation, as the programmer can simply use something like:

```
?-there,move,findall(X,for(I,1,1000),Xs),return,member(X,Xs).
```

to emulate (finite!) nondeterministic remote execution, by collecting all solutions at the remote side and exploring them through (much more efficient) local backtracking after returning.

1.6 Jinni's Logical Engine

The inference rule of Jinni is called LD-resolution, consisting of repeatedly unfolding the leftmost goal in the body of the resolvent, seen as a clause with its head containing the answer pattern and its body the current state of the goal stack. It has the advantage of giving a more accurate description of Prolog's operational semantics than SLD-resolution.

With the notations of [18] each inference step is described as an algebraic clause composition operation $\oplus$ which unfolds the leftmost body-goal of the first argument.

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

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

with θ = mgu(A_1,B_0). If the atoms A_1 and B_0 do not unify, the result of the composition is denoted as $\perp$. Furthermore, we consider A_0:-true,A_2,...,A_n to be equivalent to A_0:-A_2,...,A_n, and for any clause C, $\perp \oplus$ C = C $\oplus \perp$ = $\perp$. As usual, we assume that at least one operand has been renamed to a variant with variables standardized apart.

Jinni's main interpreter implements iteration of the $\oplus$ composition operation and backtracking over a set of clauses seen as a Java Enumeration type.

The implementation of Jinni's interpreter is minimalistic. A simple goal stack mechanism equivalent to BinProlog's *binarization* is used, giving full access to continuations as first order objects. The OR-stack is represented implicitly through the interpreter, with a Java Enumerations keeping the state of the current clause cursor and the top of trail cursor maintained directly as part of the state of a Java Stack object.

```
/**
  Main Jinni interpreter. Uses goal, which is part
  of the state of each program as a kind of goal-stack
  + current state of the computed answer substitution.
*/
private final void solve() throws OnceException {
  begin:
  for(;;) {
    goal=reduceBuiltins(goal); // loop over inline builtins
    if(goal.getBody() instanceof True) {
      if(first_only) throw new OnceException(goal);
      else { // send answer to controlling thread
        sendAnswer(goal);
        break begin;
      }
    }
    int oldTrailTop=trail.size();
    Clause reduced_goal=goal;

    // get set of clauses of current predicate
    Enumeration e=Init.bboard.toEnumerationFor(
      goal.getFirst().getKey());

    if(tracing>0) jtrace(reduced_goal,e);

    // backtrack over matching clauses
    while(e.hasMoreElements()) {
      Term T=(Term)e.nextElement();
      if(!(T instanceof Clause)) continue;

      // undo old bindings
      trail.unwind(oldTrailTop);
```

```
        // resolution step, over goal/resolvent of the form:
        // Answer:-G1,G2,...,Gn.

        goal=T.toClause().unfold_with_goal(reduced_goal,trail);
        if(null==goal) continue;

        if(!e.hasMoreElements()) {
           continue begin; // Last Call Optimization
        }
        solve(); // recursive call
      } // end of backtrack
      break begin;
    }
  }
```

We plan to accelerate Jinni by partially evaluating this interpreter to a Prolog-to-Java translator, along the lines of the jProlog[5] compiler co-developed with Bart Demoen.

1.7 A Meta-circular Interpreter for Jinni

The Prolog subset supported by Jinni has been designed with the idea of supporting program transformations and meta-interpretation.

In particular, in the absence of CUT, building a meta-interpreter which interprets itself according to the semantics of the object language is fairly easy. The following interpreter is *meta-circular* i.e. it has the property that `solve(solve(G))` computes the same answers as `solve(G)` or `G`, for any Jinni goal.

```
% meta-circular interpreter

solve(G):-
  once(reduce(G,NewG)),
  NewG.

% reflective reducer
% the simplest such beast is, of course: reduce(X,X).

reduce(G,G):-is_builtin(G).
reduce(',' (A,B),',' (solve(A),solve(B))).
reduce(G,',' (clause(G,Gs),solve(Gs))).
```

Note that reflection through a metacall to `NewG` is used in `solve/1`, after the `reduce/2` step is performed. Reflection is also apparent on builtins and, as it is usual with Prolog meta-interpreters, backtracking through clause/2

[5] Available at: http://www.cs.unt.edu/ tarau/

is reflected from the meta-language into the object language directly. The reader might ask why our meta-interpreter is different from the usual one. The answer is simple: if-then-else logic in the absence of CUT would make the naively adapted Prolog meta-interpreter unpleasantly complex.

1.8 Mutual Agent/Host Security: the *Bring Your Own Wine* Principle

Jinni has currently a `login + password` mechanism for all remote operations, including mobile code. However, the combination of meta-interpretation and computation mobility opens the door for experimenting with novel security mechanisms.

Let us consider the (open) problem of mutually protecting a mobile agent from its (possibly malicious) host as well as the host from the (possibly malicious) agent. Protecting the host from the agent is basically simple and well known. It is achieved through building a *sandbox* around the code interpreter as in Java. The sandbox can filter (usually statically) the instruction set, ensuring, for instance, that local file operations are forbidden.

However, protecting the agent from injection of a malicious continuation from the host, to be executed after its return is basically an open problem.

We will sketch here a solution dealing with both problems.

It is known that (most) language interpreters are Turing-equivalent computational mechanisms, i.e. it is not statically decidable what they will do during their execution. For instance, we cannot statically predict if such an interpreter will halt or not on arbitrary code.

The main idea is very simple: *a mobile agent will bring its own (Turing equivalent) interpreter*[6], give it to the host for static checking of sandbox compliance. Note that a sufficient condition for an interpreter to be sandbox compliant is that it *does not use reflection* and *it only calls itself or builtins provided by the sandbox.* Clearly, this can be statically checked, and ensures protection of the host against a malicious agent[7]. Protecting the mobile agent who brought its own meta-interpreter is clearly simpler than running over an unknown/statically unpredictable interpreter provided by the host. Moreover, in the presence of first order continuations, the agent can check properties of future computations before actually executing potentially malicious code[8]. Note that by bringing its Turing-equivalent interpreter, the agent can make

[6] Inspired from the technique, some restaurants in Canada apply to wine, to avoid paying expensive licensing fees: they ask you to bring your own. Subtle side effects on the customer's mind are therefore also her own responsibility.

[7] In multi-threaded systems like Jinni, non-termination based resource attacks are not an issue, as the interpreter can be made to run on its own thread and therefore it cannot bloc the host's server mechanism.

[8] In fact, in the case of Jinni's mobile code, the returning continuation is unified with the one left home, as the natural way to propagate bindings computed

sure that its own security checking mechanisms cannot be statically detected by the host. Clearly, supposing the contrary would imply that a malicious host would also solve the halting problem.

1.9 Application domains

Jinni's client and server scripting abilities are intended to support platform and vendor independent Prolog-to-Java and Prolog-to-Prolog bidirectional connection over the net and to accelerate integration of the effective inference technologies developed the last 20 years in the field of Logic Programming in mainstream Internet products.

The next iteration is likely to bring a simple, plain English scripting language to be compiled to Jinni, along the lines of the LogiMOO prototype, with speech recognizer/synthesizer based I/O. A connection between Jinni and its Microsoft Agent counterpart *Genie* are among the high priority tasks likely to be left to the growing community of Jinni co-developers[9].

Among the potential targets for Jinni based products: lightweight rule based programs assisting customers of Java-enabled appliances, from Web based TVs to mobile cell phones and car computers, all requiring knowledge components to adjust to increasingly sophisticated user expectations.

A stock market simulator is currently on the way to be implemented based on Jinni, featuring user programmable intelligent agents. It is planned to be connected to real world Internet based stock trade services.

Jinni's key features are currently being ported to BinProlog, which will support a similar multi-threading and networking model and at considerably higher engine performance, while transparently interoperating with Jinni through mobile code, remote predicate calls and Linda transactions.

1.10 Conclusion

The Jinni project shows that Logic Programming languages are well suited as the basic glue so much needed for elegant and cost efficient Internet programming. The ability to compress so much functionality in such a tiny package shows that building logic programming components to be integrated in emerging tools like Java might be the most practical way towards mainstream recognition and widespread use of Logic Programming technology. Jinni's emphasis on functionality and expressiveness over performance, as well as its use of integrated multi-threading and networking, hint towards the priorities we consider important for future Logic Programming language design.

remotely. As far as the continuation contains no metacalls or clause database operations, no malicious actions as such can be attached by the visited host

[9] Jinni's sustained growth is insured through a relatively unconventional *bazaar* style development process, similar to Linux and more recently Netscape client products.

Acknowledgments

We thank for support from NSERC (grants OGP0107411), the Université de Moncton, Louisiana Tech University as well as from E-COM Inc. and the Radiance Group Inc. Special thanks go to Koen De Bosschere, Phil Cohen, Veronica Dahl, Bart Demoen, Ed Freeman, Don Garrett, Armin Minkler, Stephen Rochefort and Yu Zhang for fruitful interaction related to the design, implementation and testing of Jinni. We also thank an anonymous referee for his salient comments helping to significantly improve the presentation of this paper.

References

1. The Avalon MUD. http://www.avalon-rpg.com/.
2. K. A. Bharat and L. Cardelli. Migratory Applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995. http://gatekeeper.dec.com/ pub/DEC/SRC/research-reports/ abstracts/src-rr-138.html.
3. BlackSun. CyberGate. http://www.blaxxun.com/.
4. L. Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, pages 3–6. Springer-Verlag, LNCS 1228, 1997.
5. L. Cardelli. Mobile ambients. Technical report, Microsoft, 1998. http://www.research.microsoft.com/users/adg/Research/Ambit/default.html.
6. N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.
7. S. Castellani and P. Ciancarini. Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 89–106, Cesena, Italy, April 1996. Springer.
8. V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.
9. K. De Bosschere, D. Perron, and P. Tarau. LogiMOO: Prolog Technology for Virtual Worlds. In *Proceedings of PAP'96*, pages 51–64, London, Apr. 1996.
10. K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.
11. GeneralMagicInc. Odissey. Technical report, 1997. available at http://www.genmagic.com/agents.
12. IBM. Aglets. Technical report, 1999. http://www.trl.ibm.co.jp/aglets.
13. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
14. T. Meyer, D. Blair, and S. Hader. WAXweb: a MOO-based collaborative hypermedia system for WWW. *Computer Networks and ISDN Systems*, 28(1/2):77–84, 1995.
15. P. Tarau. Logic Programming and Virtual Worlds. In *Proceedings of INAP96*, Tokyo, Nov. 1996. keynote address.

16. P. Tarau. BinProlog 7.0 Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.
17. P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.
18. P. Tarau and K. De Bosschere. Memoing with Abstract Answers and Delphi Lemmas. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Springer-Verlag, pages 196–209, Louvain-la-Neuve, July 1993.
19. P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, Sept. 1996. http://clement.info.umoncton.ca/ lpnet.
20. P. Van Roy, S. Haridi, and P. Brand. Using Mobility to Make Transparent Distribution Practical. Technical report, 1997. manuscript.

Appendix: A Quick Introduction to Jinni

Getting started

Using Jinni through an applet The latest version of Jinni is available as an applet at:

```
http://www.binnetcorp.com/Jinni
```

After enter a query like:

```
append(Xs,Ys,[1,2,3]).
```

the applet will display the results in its Prolog console style lower window.

Using Jinni in command line mode: Consulting a new program:

```
?-[<myprog>].
```

will read in memory the file `<myprog>.pro` program replacing similarly named predicates with new ones. It is actually a shorthand for reconsult/1. To accumulate clause for similarly named predicates, use consult/1. The shorthand:

```
?-co.
```

will reconsult again the last reconsulted file.

Client/server interaction To try out Jinni's client/server abilities, open 3 shell windows:

```
Window 1

java  Jinni
..............
?-run_server.


Window 2

?-there.
?-in(a(X)).


Window 3

?-there.
?-out(a(hello)).
```

When entering the out command in Window 3 you will see activity in Window 2. Through the server in Window 1, Window 3 has communicated the word "hello" returned as a result of the in query in Window 2!

Bi-directional Jinni / BinProlog talk

As client, Jinni talks to BinProlog servers with `out, cin` and `all` commands and with `the(Answer, Goal, Result)` or `all(Answer, Goal, Results)` remote execution queries. To try this out, start an unrestricted BinProlog server with:

```
  ?-trust.
```

BinProlog's `trust/0` starts a password protected server, willing to accept remote predicate calls and mobile code. BinProlog's default *run_server/0* only accepts a *limited set* (mostly Linda operations - a form of *sandbox* security). As a server, Jinni understands out, all, cin, rd, in commands coming from BinProlog clients and uses multiple threads to synchronize them as well as `the(Answer,Goal,Result)` or `all(Answer,Goal,ListOfResults)` remote execution queries. The most natural use is a Java server embedded into a larger application which communicates with Prolog clients. Jinni-aware BinProlog clients or servers are available from

```
  http://www.binnetcorp.com/BinProlog
```

Secure operations can be performed using Jinni's and BinProlog login and password facilities. Both Jinni and BinProlog support computation mobility.

The **move/0** command transport execution from **Jinni** client to a **BinProlog** server for accelerated execution. For instance the Jinni command:

```
?-there,move,for(I,1,1000),write(I),nl,fail.
```

would trigger execution in the much faster BinProlog server where the 1000 numbers will be printed out.

Remote exection is deterministic and restricted to a segment of the current AND-continuation. The command:

```
?-there,move,findall(I,for(I,1,10),Is),return,member(I,Is).
```

will return the values for **Is** computed on the **Jinni** server, which can be explored, after **return/0**, through local backtracking by **member/2**, on the client side. This combination of move-findall-return-member shows that implementing code mobility as deterministic remote execution of a segment of the current AND-branch does not limit its expressiveness.