Agent Mobility with Weak Local Inheritance and Transactional Remote Logic Invocation

Paul Tarau[1] Arun Majumdar[2]

[1]University of North Texas

[2]Vivomind Research LLC

LPMAS 2011

Motivation

- agent programming constructs have influenced a significant number of mainstream software components
- our new Java-based LeanProlog implementation is centered around “agent-like” *first class logic engine* constructs
- agent-oriented language design ideas can be used for a refactoring of Prolog’s interoperation with the external world, including interaction with other instances of the Prolog processor
- to ensure scalability and readiness for integration in cloud-computing services we need resource-aware distributed agent programming constructs that are as oblivious as possible to the actual location of an agent

Outline

- local inheritance mechanisms and interactions between agents
- “remote logic invocation” (RLI) mechanism
- agent spaces: a protocol for multi-agent coordination
- distributed programming constructs ensuring agent mobility
- typical use cases of the framework
- conclusion

A Bird's view of the Lightweight Prolog Agent Layer

- *agents* are implemented as *named* Prolog dynamic databases
- each agent has a process where its *home* is located - called an *agent space*
- they share code using a simple "Twitter-style" mechanism that allows their *followers* to access their predicates
- agents can *visit* other spaces located on local or remote machines - where other agents might decide to follow their replicated "avatars"
- the state of its avatar is dynamically updated when a state change occurs in the agent's code space
- communication between agents, including avatar updates, is supported by a remote predicate call mechanism between agent spaces, designed in a way that each call is atomic and guaranteed to terminate

First Class Logic Engines

- a *logic engine* is a Prolog language processor reflected through an API that allows its computations to be controlled interactively from another *engine*
- very much the same thing as a programmer controlling Prolog's interactive toplevel loop:
  - launch a new goal
  - ask for a new answer
  - interpret it
  - react to it
- logic engines can create other logic engines as well as external objects
- logic engines can be controlled cooperatively or preemptively

Agents as Dynamic code in Multiple Named Databases

- Lean Prolog provides dynamic database operations acting on multiple named databases
- the API is a set of predicates like
  - `db_assert(Database,Clause)`,
  - `db_retract(Database,ClauseHead)` etc.
- default database (named `'$'`) is used for operations without an explicit database argument
- the default database is shared i.e. when no definition exists in a named database, calls are redirected to predicate definitions in the default database
- the state of an agent (seen as a set of dynamic Prolog clauses) is contained entirely in a database with a name derived from the name of the agent

A “Twitter-style” agent inheritance mechanism

- agents share compiled code
- agents inherit code automatically from the default dynamic database
- an agent can decide to “follow” a set of other agents
- the set of agents followed by a given agent can be updated dynamically at any time

Example

Running the goal

```
?-cindy@[alice, bob].
```

ensures that agent `cindy` follows agents `alice` and `bob` but later `cindy` can change her mind and issue

```
?-cindy@[alice, dylan].
```

from which point in time `cindy` follows `alice` and `dylan`

- we have a fully dynamic “one-level” inheritance mechanism
- inheritance is “weak” i.e. non-transitive
- “all-or-nothing”: an agent inherits a *complete predicate definition* from the first of the followed agents that provides it

Example of Local Agent Interactions

```
local_agent_test:-
  assert(friends(cool_people)),
  alice@[bob,cindy], % alice follows bob and cindy
  alice@assert(like(macs)),
  alice@assert(like(popcorn)),
  alice@assert(hate(candy)),
  alice@((hate(pcs):-true)), % shorthand for assert
  cindy@[alice,bob], % cindy starts following alice
  bob@((like(X):-alice@hate(X))),  % bob likes what alice hates
  foreach(cindy@friends(X),println(friends:X)),
  foreach(bob@like(X),println(bob:likes(X))),
  foreach(alice@like(X),println(alice:likes(X))),
  foreach(cindy@hate(X),println(cindy:hates(X))).
```

Results of the Interaction

When running the predicate one can observe the fairly natural semantics of “following” another agent, e.g. that `bob` likes candy because `cindy` hates it.

```
?- local_agent_test.
friends : cool_people
bob : likes(candy)
bob : likes(pcs)
alice : likes(macs)
alice : likes(popcorn)
cindy : hates(candy)
cindy : hates(pcs).
```

Calling Remote Predicates

- as high level as possible remote execution mechanism - we call it Remote Logic Invocation (RLI)
- $\Rightarrow$ based on Java's *Remote Method Invocation* (RMI) communication layer
- communication is a good thing, if used with moderation :-)
- $\Rightarrow$ usually only one RMI port for each Prolog process is enough
- the port is shared among the agents

A simple Java API, reflected in Prolog

```
package rli;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerStub extends Remote {
  public Object rli_in() throws RemoteException;

  public void rli_out(Object T) throws RemoteException;

  public int rli_ping() throws RemoteException;

  public int rli_stop_server() throws RemoteException;

  public Object rli_call(Object T) throws RemoteException;
}
```

Agent Spaces

- an *agent space* is seen as a container for a group of agents usually associated with a Prolog process and an RLI server
- we assume that the name of the space is nothing but the name of the RLI port
- we make sure that on each host, a “broker”, keeping track of various agents and their homes, is started, when needed
- `start_space(BrokerHost,ThisHost,Port)` starts, if needed, the unique RLI service associated to a space and registers it with the broker (that it starts as well, if needed!)
- communication with agents inhabiting an agent space happens through this unique port - typically one per process
- $\Rightarrow$ all RLI calls to a given port are atomic and terminating

Agent Mobility and State Synchronization

- our concept of agent mobility is derived directly from the unique nature of a Prolog program - that can be seen as a set of predicate definitions built each from an ordered set of clauses
- $\Rightarrow$ changes of an agent's state can be safely propagated from an agent space to another without the need to spawn a thread for each otherwise possibly non-terminating remote procedure call
- we will now show that it is possible to keep such calls always local using code replication

Visiting an Agent Space

- an agent can *visit* one or more agent spaces at a given time
- when calling the predicate `visit(Agent,Host,Port)` an agent broadcasts its database and promises to broadcast its future updates
- “avatar”: an agent is represented at a remote space by a replica of its set of clauses
- the predicate `take_my_clauses(Agent,Host,Port)` remotely asserts the agent’s clauses to the database of the agent’s “avatar”
- only the agent’s *own code* goes and not the code that the agent inherits locally

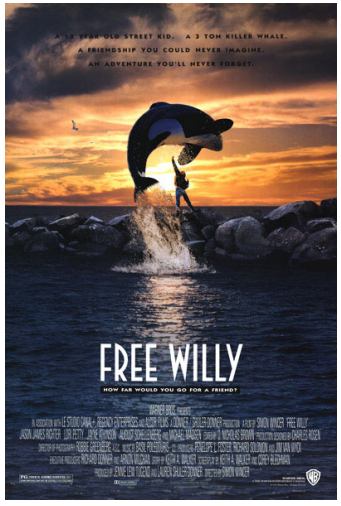
Propagation of Updates

- as the agent keeps track of all the locations where it has dispatched avatars, it will be able to propagate updates to its database using atomic, guaranteed to terminate RLI calls
- an agent is also able to `unvisit` a given space - in which case the code of the avatar is completely removed and broadcasts of updates to the unvisited space are disabled

Remote Followers

- an agent can have followers in various spaces that it visits
- followers inherit the code of the avatar - and therefore all their calls stay local
- why this makes sense:
  - for instance, an agent asked to find neighboring gas stations should do it based on the GPS location of the agent space it is visiting
  - execution is local - possible non-termination or lengthy execution does no block communication ports

“Free Will” $\Rightarrow$ Flexibility



- it is an agent’s autonomous decision to visit a given agent space
- it is an agent’s autonomous decision to become a follower locally or mediated through an avatar
- “free will” on both sides provides flexibility and enables implementation of “anthropomorphic" mechanisms for negotiation, reputation building and trust between agents in a given application

An Example of Distributed Agent Interaction: Space 1

`french_space`: initializes an agent space where salutations occur in French, inhabited by agent `alice`

```
french_space:-
  start_space(french_space),
  assert(when_arriving_say(bonjour)),
  assert(when_leaving_say(aurevoir)),
  rli_wait(english_space), % synchronizing spaces
  sleep(3), % assuming bob has arrived by now
  alice@[bob], % alice adopts good manners from bob
  alice@salutations, % and applies them
  alice@visit(english_space). % then she visits bob's home
```

An Example of Distributed Agent Interaction: Space 2

`english_space`: salutations occur in English, inhabited by `bob`

```
english_space:-
  start_space(english_space),
  assert(when_arriving_say(hello)),
  assert(when_leaving_say(goodbye)),
  bob@[], bob@visit(french_space),
  bob@((salutations:- % bob brings some politeness
        when_arriving_say(A),
        println(A),
        sleep(5), % we can do something more interesting here
        when_leaving_say(B),
        println(B))),
  sleep(3), bob@unvisit(french_space), % bob leaves
  % assuming that alice is visiting later
  alice@[bob], % good manners are borrowed from bob
  alice@salutations. % and exercised right away
```

An Example of Distributed Agent Interaction: the Result

`alice` greeting `bob`, who visits `french_space`

```
?- french_space.
bonjour
aurevoir
```

`alice` greeting `bob` when she visits `english_space`

```
?- english_space.
hello
goodbye
```

- when `bob` visits, he promises that future code updates will follow him - resulting in the predicate `salutations` being brought to `french_space`
- since `alice` follows him this predicate becomes available to her
- the salutation messages executed by `alice` depend on local facts and come out in the local language

Limitations

- the predicates described so far are just basic building blocks
- initial synchronization is needed to ensure all spaces are up and running
- complex multi-agent interactions require “discovery of services”
- other interaction patterns are needed for more complex agent coordination, e.g. publish/subscribe or Linda blackboards (see COORD’2011 paper)
- parallelism for performance is available independently using higher-order predicates like multi-fold, multi-all (see DAMP’2011 paper)

Conclusion

- we have described a distributed multi-agent architecture that, despite its simplicity exhibits some novelty in terms of the way agents inherit dynamic code and the way they engage in communication with other agents
- our concept of agent mobility is based on a simple remote logic invocation mechanism
- while quite straightforward – by limiting calls to database updates – it provides remote code sharing without requiring potentially non-terminating remote predicate calls
- our framework supports a dynamic a “free will" mechanism that agents can exercise when using other agents’ knowledge bases
- not covered: various reasoning mechanisms that agents can implement - these are seen as independent of the infrastructure itself
- We thank for support from NSF (research grant 1018172) and Viviomind Research LLC

Questions?

A few related papers are at:

- http://logic.cse.unt.edu/tarau/research/LeanProlog