I.C. Prolog II : a Language for Implementing Multi-Agent Systems*

Damian Chu
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate
London SW7 2BZ
email: dac@doc.ic.ac.uk

25 March, 1993

Abstract

This paper examines how we may prototype Multi-Agent Systems. We informally enumerate the low-level technical support needed for such systems and show how IC-Prolog II is a good candidate language. IC-Prolog II is a new implementation of Prolog that is particularly suited to distributed applications. It features multiple threads, high-level communication primitives and an object-oriented extension. A fully worked example of specifying an agent is given to illustrate use of the language. This shows that it is possible to give a high-level description of an agent, and that this description can be executed directly, making fast prototyping of agents a reality. With this new tool, researchers in Multi-Agent Systems may gain practical experience in exploring ideas on a real implementation.

Familiarity with the Prolog programming language is assumed.

1 Introduction

In recent years, interest in distributed computing has grown very rapidly since centralised systems have proved unable to cope with the information technology explosion. For reasons of performance, flexibility, practicality and cost, centralised mainframe computers have been replaced by networks of workstations. Since the data on knowledge bases are now distributed among many computers, there is a need for systems that enable these autonomous knowledge bases to cooperate. We can view these cooperative knowledge-based systems as examples of Multi-Agent Systems, whereby each knowledge base is an independent agent. To specify Multi-Agent Systems, we need a notation for describing an agent.

The use of logic to represent knowledge has a long history leading back to the Ancient Greeks. More recently, the discovery of the resolution principle [Rob65] and its application to predicate logic [Kow74] has led to Prolog, a programming language based on Horn Clauses, a subset of First Order Predicate Logic. Prolog enables the programmer to represent knowledge in a high-level symbolic manner [Kow74], and uses SLDNF resolution as the inference mechanism. Prolog has been widely used for implementing knowledge based systems, deductive databases and expert systems [SB89]. In the rest of this paper, we assume that the reader is familiar with Prolog. For excellent introductions to the language, see [CM84], [SS86] and [O'K90]. Though there are many implementations of Prolog, few address the needs of distributed computing.

IC-Prolog II (`ICP` for short) [CC92] is a new implementation of Prolog which addresses this need. It contains many new features such as multiple threads, high-level communication primitives and

*Published in the Proceedings of the Special Interest Group on Cooperating Knowledge Based Systems, Keele September 1992

an object-oriented extension which makes ICP especially suitable for implementing Multi-Agent Systems.

The structure of the paper is as follows : in section 2, we will examine the technical requirements for implementing Multi-Agent Systems. We then present in section 3 a fully worked (albeit very simple) example of a Multi-Agent System using ICP. We will explain features of the language as the need arises. We present our conclusions in section 4.

2 Technical Requirements for Multi-Agent Systems

Multi-Agent Systems covers many research topics such as agent architecture, problem decomposition, task allocation, cooperation, communication, knowledge representation, conflict resolution etc. Researchers have tended to either build systems tailor-made for their particular research area/application domain, and thus are not generally applicable or they have been forced to explain their ideas in theoretical discussions only, since there was no easy way for them to test their ideas on a real implementation. This is an unfortunate situation, as valuable insights can often be gained from carrying out experiments on prototype systems.

If we abstract away from the issues and look at the underlying support needed, we can draw up a list of requirements for building Multi-Agent Systems. Note that this is not a list of requirements for a Multi-Agent System since such a list would be highly subjective and would involve deciding on issues such as whether a Truth Maintenance System is required. We concentrate instead on the technical requirements that a language should satisfy in order to program Multi-Agent Systems.

In Multi-Agent Systems, an agent needs knowledge to operate effectively. This includes knowledge about its own capabilities, knowledge about other agents with which it can interact, knowledge about how it can communicate with other agents and knowledge about particular application domains. We use the term *capability* to mean a task which an agent can perform. An agent needs to be able to manipulate this knowledge and draw conclusions from it. The knowledge needed is not restricted to these, but it seems obvious that knowledge representation and inference are essential components of any language for Multi-Agent Systems.

The agent is likely to be interacting with more than one agent at any given moment so an ability to handle multiple interactions concurrently is needed. Since the agents in Multi-Agent Systems are distributed, an ability to communicate over the network is essential. Furthermore, this communication should be asynchronous in the sense that sending a message should not have to wait for it to be received before proceeding. Synchronous communication forces a tight coupling between agents that goes against the philosophy of Multi-Agent Systems.

These requirements can be summarised by saying that the implementation language should :

- be able to represent knowledge
- have some inference mechanism
- be multi-tasking
- have asynchronous communication
- be able to communicate over a network

Note that this specification makes no assumptions on how agents cooperate nor what are the contents/protocol of the messages passed between them. The specification provides only the infrastructure for Multi-Agent Systems. This infrastructure enables experiments on these higher level issues to be carried out.

3 Specifying an Agent in ICP

ICP satisfies the requirements for programming Multi-Agent Systems as described in the previous section. To illustrate this, we will give an example of how to specify an agent in a Multi-Agent System using ICP. The example is in the domain of arithmetic and shows how a group of agents can cooperate to solve arithmetic problems that none could solve on its own.

In our scenario, there are five basic capabilities that are distributed among three agents `a1`, `a2` and `a3`. A capability may be a task which an agent can perform or some knowledge that it is willing to share with other agents. In this example, agent `a1` can perform addition and subtraction, agent `a2` can multiply and divide, and lastly agent `a3` is the expert on calculating square roots.

Although the agents have different capabilities, the basic architecture can be the same. For this example, we will assume a simple agent architecture. Each agent consists of four components :

1. a *knowledge base* which includes knowledge about the agent's own capabilities, knowledge about other agents' capabilities and rules for problem decomposition,
2. a *planner* which decides how to solve each task,
3. a *monitor* which accepts new tasks and reports results,
4. a *communicator* which handles incoming and outgoing messages.

These components are shown in the agent architecture diagram of figure 1.

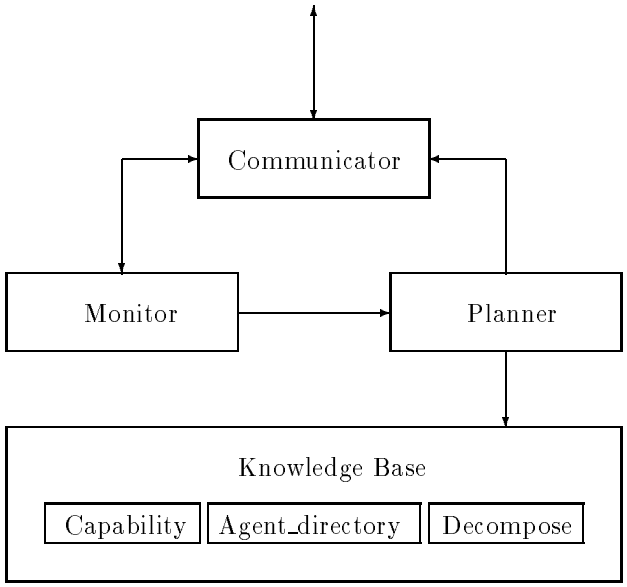

Figure 1: Architecture of the Simple Agent used in example

The basic mechanism in this agent is a follows: external requests to perform tasks arrive through the *communicator*; these are passed to the *monitor* which oversees the execution. Local requests go directly to the *monitor*. The task is given to the *planner*, which has a variety of methods of solving the problem including requesting help from other agents to solve smaller sub-problems. These sub-requests (if any) are transmitted through the *communicator*. The execution of the tasks are domain-specific. When the complete task is finished, the *monitor* sends back a report (again through the *communicator*) to the requesting agent.

Note that this architecture is not definitive. A fully-fledged agent may require other components though this simple design is a good first approximation. The agent designer is free to experiment with different architectures. The important point is how easily a given specification can be formalised as executable code, thus enabling fast prototyping of ideas.

3.1 Knowledge Base

In this section we will specify the *knowledge base* component as shown in the previous diagram. Programs written in Prolog have a flat structure. There is only one name space for all the predicate names. This is a problem if the programs are large, or if we would like to group together related

predicates for ease of maintenance. In effect, we would like a structuring mechanism for programs, a feature which is available in `ICP` through Logic & Objects.

Logic & Objects (*L&O* for short) is a object-oriented notation designed by McCabe [McC92]. It allows Prolog programs to be written in an object-oriented style. The *L&O* system is a preprocessor written in Prolog which converts these object-oriented programs into normal Prolog code, which is then executed on the host system. Thus, *L&O* can be thought of as a shorthand for the lengthy and laborious Prolog versions of the same program.

In our example agent `a1` can perform addition and subtraction as well as knowing its own name. This may be specified using *L&O* as:

```
capability : {
   my_name(a1).
   add(X, Y, Sum) :- Sum is X + Y.
   subtract(X, Y, Difference) :- Difference is X - Y.
}.
```

Here `capability` is an object representing the capabilities of an agent. In *L&O* syntax the object name is followed by a colon and a list of Prolog clauses between braces. The clauses are the methods for that object. A condition may be prefixed by an object label e.g. `foo:bar` means send the message `bar` to the object `foo`. Note that constants begin with a lower-case letter following the Prolog convention that variables begin with upper-case letters.

The capabilities of addition and subtraction are (in this case) trivially implemented by calling upon the arithmetic primitives of Prolog. In other examples, the execution of a task may involve queries to large databases, substantial computation or even interfacing to hardware devices.

This agent knows about two other agents `a2` and `a3`, which have other capabilities. This can be coded as :

```
agent_directory : {
   capability(a2, [‘multiply/3, ‘divide/3]).
   capability(a3, [‘square/2, ‘squareroot/2, ‘cube/2]).
}.
```

This specifies that `a2` can do multiplication and division (both have 3 arguments - 2 input arguments for the operands and 1 output argument for the result) and that `a3` has three other skills. The backquote is *L&O* syntax to prevent evaluation.

The third part of the *knowledge base* is the rules for decomposing a problem into smaller sub-problems. We show here two rules — one for doubling a number and another for calculating the length of the hypotenuse given the other two sides of a right-angle triangle.

```
decompose : {
   rule(double(X,D), [ add(X,X,D) ]).
   rule(hypotenuse(X,Y,H), [ square(X,X2), square(Y,Y2),
                             add(X2,Y2,H2), squareroot(H2,H) ]).
}.
```

We can now describe the agent's *knowledge base* using the object `knowledge` as follows:

```
knowledge : {
   find_agent(Task, Agent) :-
      functor(Task, Cap, Arity),   /* get name of task     */
      capability(Agent, CapList),  /* get capability list  */
      on(‘Cap/Arity, CapList).     /* check if in the list */
}.
knowledge << capability.
knowledge << agent_directory.
knowledge << decompose.
```

Here we have made use of another facility of *L&O* – inheritance. This is denoted by the inheritance operator ‘`<<`’. The rule

```
knowledge << capability.
```

means the object `knowledge` inherits all methods from object `capability`. This allows queries such as

```
knowledge:my_name(X)
```

to be answered correctly even though there is no explicit method for it in the object `knowledge`. The query is answered through inheriting from the `capability` object. Multiple inheritance is represented by inheriting from more than one object. In our example, `knowledge` also inherits methods from the `agent_directory` and `decompose` objects.

Finally, the *find_agent* clause of the *knowledge base* is used to match an agent with a given task by searching for the name of the task (ignoring the parameters) in the inherited `agent_directory` object. `on/2` is the built-in primitive for testing list membership. Note that Prolog's backtracking enables multiple solutions to be found. This corresponds to the case when more than one agent can perform the given task.

The other agents `a2` and `a3` have different local knowledge. They have different capabilities, their own agent directories and rules for problem decomposition. The structure of the *knowledge base* however, is the same, as are the other components (*planner*, *monitor* and *communicator*) of the agent architecture. Using *L&O*, we are able to decouple the agent-specific and agent-independent code.

The agent-specific knowledge of agent `a2` is as follows:

```
/* agent a2 */
capability : {
   my_name(a2).
   multiply(X, Y, Product) :- Product is X * Y.
   divide(X, Y, Quotient) :- Quotient is X / Y.
}.

agent_directory : {
   capability(a1, [‘add/3, ‘subtract/3, ‘hypotenuse/3]).
   capability(a3, [‘square/2, ‘squareroot/2, ‘cube/2]).
}.

decompose : {
   rule(double(X,D), [ multiply(X,2,D) ]).
}.
```

Lastly, agent `a3` has the following agent-specific knowledge.

```
/* agent a3 */
capability : {
   my_name(a3).
   squareroot(X, Root) :- Root is sqrt(X).
}.

agent_directory : {
   capability(a2, [‘multiply/3, ‘divide/3, ‘double/2]).
   capability(a1, [‘add/3, ‘subtract/3, ‘double/2, ‘hypotenuse/3]).
}.

decompose : {
   rule(square(X, Sq), [ multiply(X,X,Sq) ]).
   rule(cube(X,C), [ square(X,S), multiply(S,X,C) ]).
}.
```

An interesting point is that `a3` can call upon either `a1` or `a2` to perform doubling since both agents have offered this capability.

3.2 Planner

The next agent component is the *planner*. The *planner* receives tasks either from the *monitor* or those generated internally as a result of problem decomposition. The *planner* first tries to complete tasks using the agent's own capabilities by querying its own *knowledge base*. If the task is not one that it can do itself, the *planner* tries to find another agent who can perform it, sends a request to that agent and waits for a reply through the *communicator*. A unique communication ID is generated for each such request. If none of the agents have that capability, the *planner* will try to decompose the problem and recursively plan the smaller problems. Finally, if decomposition fails, then the whole planning process has failed. We encode this behaviour concisely as follows:

```
planner : {
   plan(Task) :-
      knowledge:Task.              /* try to solve it myself */
   plan(Task) :-
      knowledge:find_agent(Task, Agent),
      communicator:send_request(Agent, Task, Id),
      communicator:wait_reply(Id, Msg),
      Task = Msg.                  /* this fails if Msg = 'failure' */
   plan(Task) :-
      knowledge:rule(Task, Subtasks),  /* decompose problem */
      execute(Subtasks).

   execute([]).                    /* empty set succeeds trivially  */
   execute([First|Rest]) :-        /* solve set of tasks by ...     */
      plan(First),                 /* planning one of the tasks ... */
      execute(Rest).               /* and recursively do the rest   */
}.
```

3.3 Monitor

We turn our attention now to the *monitor*. This component is responsible for supervising the execution of a task and to report the results back to the originating agent who sent the request. If the task cannot be completed, the 'failure' message is reported back instead. Requests initially come from external agents through the *communicator*, or are generated locally. In both cases, they are passed on to the *planner* component. This behaviour is repesented as follows:

```
monitor : {
   request(Sender, Task, Id) :-
      planner:plan(Task), !,    /* task succeeded */
      communicator:send_reply(Sender, Task, Id).

   request(Sender, Task, Id) :-
      /* failed to complete task, report failure */
      communicator:send_reply(Sender, failure, Id).

   solve(Task) :-               /* local tasks */
      planner:plan(Task), !.
   solve(Task) :-               /* problem unsolvable */
      writeseqnl(user, ['cannot solve', Task]).

}.
```

3.4 Communicator

Finally, we come to the *communicator* component. This is responsible for sending messages to other agents using *mailboxes* to handle communication, and for redirecting new requests to the *monitor*. Mailboxes are high-level abstractions for inter-agent communication designed and implemented by V. Benjumea. A mailbox is simply a repository for messages. Messages can be sent to and removed

from a mailbox. For two threads to communicate, the sender places a message in the mailbox, and the receiver removes it from the same mailbox. In `ICP`, a mailbox is created using

```
mbx_create(-Id)
```

'`-`' is used to denote an output argument. Input arguments are represented by '`+`'. In this case, `mbx_create/1` returns an identifier naming the newly created mailbox. Mailbox identifiers are globally unique in the network, so the exact same identifier may be used by any agent on the network. To send and receive messages to/from mailboxes, we use

```
mbx_send(+Id, +Message)
mbx_recv(+Id, -Message)
```

A name may be associated with a mailbox identifier using

```
mbx_bind(+Id, +Name)
```

This *registers* the name so that other agents may find out the identifier by querying the name using

```
mbx_getid(+Name, -Id)
```

Finally, to destroy a mailbox, use the primitive

```
mbx_close(+Id)
```

Maiboxes are implemented very efficiently. More specifically, the creation of a new mailbox does not require a new TCP socket to be opened.

Using mailboxes, we are able to describe the behaviour of the *communicator*. The *communicator* handles the forwarding of requests to external agents. It does this by creating a new mailbox for the reply, looking up the mailbox identifier for the specified agent, and sending a message to it. The message consists of the sender agent's name, the task and the reply mailbox ID.

The converse of sending a request is to receive one. In this case the message is passed to the local *monitor* as a new thread of computation. `ICP` is a multi-threaded Prolog which means that multiple computations may be executed concurrently. This is not the same as co-routining which demands that control is passed explicitly between the threads. In `ICP`, the different threads execute independently and concurrently, unaware of each other's existence unless they wish to communicate. This is achieved by employing a time-slice mechanism. New threads are created using the `fork/1` primitive.

Two other tasks performed by the *communicator* are the sending and receiving of results. This is achieved by simple mailbox communication using the dedicated mailbox created specifically for each reply. The mechanism for receiving replies is that a blocking read is set up on a mailbox, which is unblocked when the reply is eventually received.

The implementation of the *communicator* is shown in figure 2. The calls to `write/2` and `writeseqnl/2` are not essential as they are for the displaying of trace messages only.

3.5 Agent Initialisation

To start an agent, we need to give it a unique mailbox ID. This mailbox ID must be accessible globally by other agents so that they can communicate with the new agent. We can register this mailbox ID globally by binding it with the agent name. Once the mailbox is created, we fork a process to read the incoming messages.

```
init_agent :-
   knowledge:my_name(MyName),
   mbx_create(Id),              /* new mailbox ID */
   mbx_bind(Id, MyName),        /* register ID globally */
   writeseqnl(user, ['initialised agent :', MyName]),
   fork(read_messages(Id)).
```

```
communicator : {
   /* sending a request to another agent */
   send_request(Agent, Task, Id) :-
      mbx_create(Id),    /* create unique Id for reply */
      agent_id(Agent, Mbx),
      knowledge:my_name(Name),
      writeseqnl(user, ['sending request to', Agent, ':', Task]),
      mbx_send(Mbx, request(Name, Task, Id)).

   /* receiving an incoming request */
   request(Sender, Task, Id) :-
      fork(`(monitor:request(Sender, Task, Id))).

   /* sending results back to the requestor */
  send_reply(Who, Msg, Id) :-
      writeseqnl(user, ['sending reply to', Who, ':', Msg]),
      mbx_send(Id, Msg).

   /* blocking read, waiting for reply */
   wait_reply(Id, Msg) :-
      write(user, 'waiting for reply ... '), flush,
      mbx_recv(Id, Msg),
      writeseqnl(user, ['received reply :', Msg]),
      mbx_close(Id).

   /* look up mailbox ID of agent */
   agent_id(Agent, Mbx) :-
      mbx_getid(Agent, Mbx), !.
   agent_id(Agent, Mbx) :-
      writeseqnl(user, ['unknown agent :', Agent]).
}.
```

Figure 2: Implementation of the Communicator

The `read_messages/1` program is a loop that reads messages and passes them to the communicator. It is thus the mechanism by which messages are transformed into *communicator* tasks.

```
read_messages(Id) :-
   mbx_recv(Id, Msg),
   writeseqnl(user, ['MESSAGE RECEIVED :', Msg]),
   communicator:Msg,    /* pass all incoming messages to ... */
   read_messages(Id).   /* the communicator and loop again   */
```

3.6 Sample Trace

To test our simple arithmetic agents, we started `ICP` on three separate machines running the agents `a1`, `a2` and `a3` respectively. The query

```
| ?- monitor:solve(hypotenuse(3,4,X)).
```

was posed to agent `a2` and resulted in the trace shown in figure 3. The three columns show the trace messages displayed by each of the agents `a1`, `a2` and `a3`. The ordering of the lines indicate the time sequence. (notes: Numbers preceded by an underscore denote variables whose values are not yet known. The large numbers are the mailbox identifiers.)

We see that agent `a2` was not able to solve the initial hypotenuse problem, so it sent a request to `a1` to solve it. `a1` was able to decompose the problem into four parts (2 squares, 1 addition and 1 square root) of which it could only perform one – the addition. The other three parts were sent

```
a1      a2      a3
==      ==      ==
        sending request to a1 : hypotenuse(3,4,_51)
        waiting for reply ...
MESSAGE RECEIVED : request(a2,hypotenuse(3,4,_205),131073)
sending request to a3 : square(3,_215)
waiting for reply ...
                MESSAGE RECEIVED : request(a1,square(3,_203),65537)
                sending request to a2 : multiply(3,3,_63)
        MESSAGE RECEIVED : request(a3,multiply(3,3,_205),196609)
        sending reply to a3 : multiply(3,3,9)
                received reply : multiply(3,3,9)
                sending reply to a1 : square(3,9)
received reply : square(3,9)
sending request to a3 : square(4,_221)
waiting for reply ...
                MESSAGE RECEIVED : request(a1,square(4,_440),65537)
                sending request to a2 : multiply(4,4,_63)
        MESSAGE RECEIVED : request(a3,multiply(4,4,_445),196609)
        sending reply to a3 : multiply(4,4,16)
                received reply : multiply(4,4,16)
                sending reply to a1 : square(4,16)
received reply : square(4,16)
sending request to a3 : squareroot(25,_65)
waiting for reply ...
                MESSAGE RECEIVED : request(a1,squareroot(25,_678),65537)
                sending reply to a1 : squareroot(25,5)
received reply : squareroot(25,5)
sending reply to a2 : hypotenuse(3,4,5)
        received reply : hypotenuse(3,4,5)
```

Figure 3: Trace of messages exchanged to calculate hypotenuse

to `a3`. Note that `a3` twice needed the help of `a2` to solve multiplications while it was calculating squares. `a2` was able to respond even though it was still waiting for the results of the original hypotenuse query. This was possible because of the multi-threading capability of `ICP`. The answer 5 was received by `a2` in the last step of the trace, indicating that the problem was successfully solved through close collaboration between the agents.

4 Conclusion

`ICP` is a new Prolog system developed at Imperial College. `ICP` has many features which are not available in standard Prolog systems. These include a multi-threading capability, high level communication primitives and an object-oriented extension. There are many other features in `ICP` which are useful for programming Multi-Agent Systems, but which we have not discussed because they were not required for the simple example. These features include many-to-many communication, secure communication, timeouts for receiving messages, interface to TCP/IP, foreign language interface and a complete Parlog sub-system [Gre87] for those applications where fine-grained parallelism is important. *L&O* is also a much richer notation than we have space to give justice to.

We have shown that `ICP` satisfies our informal list of technical requirements for a Multi-Agent Systems implementation language. Furthermore, the worked example illustrates that `ICP` is capable of describing agent behaviour in a high-level manner.

For the sake of clarity, brevity and completeness, I chose to model a simple agent architecture

and a simple application domain. The reader should realise that the example can be easily modified or expanded to model more complex agent architectures, alternative cooperation strategies, dynamic replanning and all the other issues of interest to researchers in Multi-Agent Systems. Even with this simple architecture, we can very easily build distributed applications in domains far more complex than arithmetic. Cooperating expert systems can be modelled in this way.

The implementation language does not impose a particular viewpoint on Multi-Agent Systems. Instead it provides an infrastructure so that prototypes can be easily constructed and experiments performed. We believe that as such, `ICP` will be a useful tool to researchers in the field.

IC-Prolog II is available by anonymous ftp from `src.doc.ic.ac.uk` (Internet: 146.169.2.1) in the directory

```
computing/programming/languages/prolog/icprolog
```

References

[CC92] Yannis Cosmadopoulos and Damian A. Chu. *IC Prolog II version 0.94 Reference Manual.* London, 1992.

[CM84] Wiliam F. Clocksin and Christopher S. Mellish. *Programming in Prolog, 2nd Edition.* Springer Verlag, New York, 1984.

[Gre87] Steve Gregory. *Parallel Logic Programming in PARLOG.* Addison-Wesley Publishers Ltd., 1987.

[Kow74] Robert A. Kowalski. Predicate logic as a progamming language. In *Proceedings of IFIP-74*, pages 569–574, Amsterdam, 1974. North Holland.

[McC92] Frank G. McCabe. *Logic and Objects.* Prentice-Hall International (UK), 1992.

[O'K90] Richard A. O'Keefe. *The Craft of Prolog.* MIT Press, Cambridge, Mass, 1990.

[Rob65] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[SB89] Leon Sterling and Randall D. Beer. Metainterpreters for expert system construction. *Journal of Logic Programming*, 6(1-2):163–178, 1989.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Mass, 1986.