Intelligent Agents for the Internet: a programming language approach

ULISSES FERREIRA

ulisses@ufba.br

Universidade Federal da Bahia
Departamento de Ciência da Computação
Av. Adhemar de Barros s/n, Ondina,
Salvador, BA, 40.170-110, Brasil

The University of Edinburgh
Division of Informatics
James Clerk Maxwell Building
King's Buildings, Mayfield Road
Edinburgh, EH9 3JZ, Scotland

Abstract. This article outlines some particular programming language features for development of Intelligent Agents on the Internet, stressing an approach that may support this kind of application and others in a straightforward way, as well as the ability to deal with partiality. At the end of this article it is briefly described an implemented programming language that adopts such approach.

1 Introduction

It is amazing how the Internet has grown during the last few years, and this growth rate will probably continue to be high along the next few years. Nowadays, almost everybody who has access to any university in Brazil has a personal e-mail account and uses a web browser, and many more people use the Internet. Some mobile-code languages[6], such as Java, have been recently designed for this new platform. Yet, the increasing complexity of software Intelligence naturally demands their behaviour to become closer to humans', which in turn demands higher-level languages and tools.

Apart from the Internet, since Prolog was designed, several programming techniques and paradigms have been proposed and developed. Not only this, but hardware has speeded up dramatically, memory has grown considerably, prices have fallen and the memory abstraction models for programming have changed. Nowadays, any object-oriented language may be used efficiently in any personal computer.

However, up to now, almost all languages designed for the Internet are not declarative and, because of this, are not very suitable for some sorts of complex systems that represent some knowledge, reason about it and are able to learn. Flexibility and simplicity are important advantages of declarative languages. This is one of the reasons that support the idea that, new programming languages and tools for global systems, including Agents[18], are natural solutions for some kinds of problems. Another reason is the idea of moving computation: not only of moving program code but also the computational state. Thus, a program that starts running on some site may move on to another site, perform some tasks, move on to a third site, perform some other tasks and so forth.

In terms of programming languages and tools for Intelligent Systems, up to now much implementation work has been done using traditional languages, such as Lisp, Prolog and their derivatives. At a secondary level, special tools for expert systems and some imperative languages, such as POP-11[3], have been used. Some languages and tools make use of the object-oriented approach, some make use of production systems, some with certainty factors attached to rules, and some others were designed for other tasks, such as STRIPS[2] for planning. In almost all cases, it can be observed that list is one of the basic types supported by these languages and tools. Besides lists, most of them provide some sort of pattern matching and many of them does not provide types, which suggest flexibility of their programs. On the other hand, some A.I. programs require quick behaviour, which suggest efficient languages, such as Fortran and C, which may be a surprise.

The Closed-World Assumption (CWA) of Logic Programming may be generalised to programming. For instance, since Pascal, many languages have claimed to be strongly typed, which means that their compilers guarantee that the programs are correct with respect to the language type system. However, in the context of global computation, where pieces of code may be linked on the fly[11], although compilers may catch almost all type errors, type safety at compiling time is no longer possible. The programmer no longer sees the whole system as a closed world because his or her program will probably make use of someone else's code and data. Therefore, in the context of global computation, new metaphors and paradigms should emerge for programming.

Agent is a term that has been used in several senses in the literature. However, almost all of them agree that Agents have a high degree of autonomy and hence should contain a helpful amount of knowledge. In this article, although not an exhaustive list, a programming language for Agents should support systems that may have the following abilities: Agents have their own identity, carry out plans and act physically on the environment; Agents are aware of their own resources and may carry them; Agents migrate, interact locally and learn; Agents recognise other agents and generate new ones; Agents deduce; Agents search the environment, and are aware of other agent's resources; Agents induce and make synthesis; Agents make plans, may be organised in hierarchies and certainly have a contribution to the whole environment; Agents cooperate in groups; Agents reason over beliefs, incomplete information and uncertainty. Resources[4] are pieces of passive data that are spread out on the Internet. With the above abilities and others, a question arises: what kind of programming language would be best for development of Intelligent Agent Systems on a new platform such as the Internet?

2 Related Work

Pontelli and Gupta have proposed W-ACE[27], a logic language for Intelligent Internet programming, although limited to the context of Logic Programming. The project is being carried out by the Laboratory for Logic and Databases at New Mexico State University.

As the Internet is a new platform, there is not much implementation work already done in programming languages and tools for Intelligent Agents on it. Only a few languages, such as Agent Tcl[17], Telescript[32] and Tycoon[21][20] implement *strong mobility*[10], which is the ability of running programs to move on to another site by performing the *migrate* instruction. Java and Obliq provide *weak mobility*, which is the ability of a running program to link to a coming code previously requested by the program.

Apart from mobility, some research work has been carried out in the combination of logic with object-oriented programming[22], and some languages which combine both paradigms have appeared, such as ObjLog[12], Coral++[28] and Prolog++[25], while others, such as λ-Prolog[26] and Mercury[31], combine logic with functional programming. Parlog[8] is a Prolog extension which includes parallel computation. However, in spite of the great importance of these languages, none of them provide representation of incomplete information. LIFE[1] is another important language, which combines Logic, Inheritance, Functions and Equations. Despite its power, it also fails to represent lack of information.

Still concerning integration between paradigms, there are some object-based languages which support concurrent programming, such as Java, Modula-3 and others, while other logic programming languages contain λ-terms, such as λ-Prolog and its extension Forum[23], which is based on linear logic.

3 Language Support for Intelligent Agents

In the presence of global computation[5], some of the characteristics of current languages and tools should be revisited. As an example, the CWA[29], supported by Prolog, sometimes seems to be inadequate as sometimes connections fail and users want programs to carry on running without information, and to give answers or to act accordingly. In Prolog, what is not present in the sequence of clauses is regarded as false, which contrasts with the idea of global computation, where the Internet may be seen as a huge and unreliable computer instead of a network. On the other hand, CWA is still very useful, e.g. to write a relation that informs whether an element is member of a given list. As another example, object-oriented languages as they have been traditionally conceived, are not very suitable for implementing globally mobile Agents[7] because the Internet is a public network where security[13] is a critical issue and hence Agents need to negotiate and agree before performing tasks. In particular, while objects perform tasks for free, Agents are able to provide services, perhaps charging for them.

In order to design a programming language suitable for Agents, some language support should be considered. A hybrid paradigm may be a politically correct attitude towards programming. Besides mobility, this hybrid paradigm may include logic programming, frames[24] or objects, production systems, imperative features and pure functions, combined in the same language, for the following reasons: the application suggests the natural paradigm to use; a hybrid paradigm provides a very flexible support to develop Intelligent Systems. The common features, such as I/O statements, tend to simplify the language in comparison to three or four simpler languages. Programming is also a matter of taste and culture. For example, while Lisp was prefered in the USA, Prolog was prefered in Europe and Japan. Finally, different parts of a system may be programmed by using different paradigms while they communicate to each other. For example, a programmer may wish to make a logic-programming query from some imperative code fragment, or alternatively to perform some action supported by some fuzzy inference.

In the presence of many languages and applications, a question arises: what kind of support a language should offer to be suitable for development of Intelligent Agent Systems on a new platform such as the Internet?

Because the Internet will never be an efficient nor a reliable environment, computation under lack of information, default values, plausible-reasoning techniques (such as probabilistic and abductive reasonings[19]), mobility and concurrency should be increasingly more important in programming. For instance, after the time-out of some remote value request, some default value may be assumed. The concept of "measure of data quality" is emphasised here. Thus, an expression may be computed with default values for some *unknown* subexpressions and results in *unknown*. In some cases, it may even result in a known value.

In the following subsections, some of the characteristics of the present hybrid-paradigm language will be described briefly.

3.1 Computation under Lack of Information

Lack of information is a topic which fits well in the context of global computation since sometimes connections fail and the programming language should provide mechanisms to allow the program to carry on running without some specific piece of information.

The notion of Computation under lack of information is combined with the concepts of *unknown value*, frames and inheritance of values of *slots* or *field inheritance*.

In frames, a slot may contain either a value or a procedure. I will use the term *field* for the former kind of slot and will use the term *method* for the latter. With the same analogy with class-based languages, I will use the terms *subframe* and *superframe* as relative hierarchical relationships between two frames. Thus a superframe A represents a broader concept from which properties are inherited by another frame B. In this case, B is a subframe of A.

3.1.1 The Unknown Value

Each data type in the language has its range of values and a special value, called *unknown*, that represents lack of information. All expressions in the language consider this special value, even statements such as the *if-then-else* and *while* were adapted to deal with this value. If everything is known in the program, it behaves exactly like a program in any language without this special value.

Every *unknown* value may have an implicit list of reasons for the value being unavailable. In the case of a question or form, the application user may refuse to give information. In this case, the variable receives the *unknown* value plus the information that its value has been asked to the user but, for some reason, he or she refused to give the information. This additional information is important because the system should recognise that repeating that question is annoying. A different situation is when a value is unsuccessfully requested from the network. In this case, after the proper time-out, the variable receives the *unknown* value plus the information that its value is due to a temporary machine failure. This additional information is relevant because the system should recognise that it is worth trying the same request later.

3.1.2 Slots and Scripts

The *unknown* value also allows better control over the two basic operations on memory: reading and writing a value, which corresponds to using a variable and assigning a value to a variable, respectively. A variable in this language may be seen a *slot*, using Minsky's terminology. The programmer may define scripts for these operations. One of them is called *provision script*, which runs when the value of a variable is being requested during the evaluation of some expression and this variable has the *unknown* value. The corresponding script is triggered to provide a value that represents that variable, not necessarily assigning a value to that variable.

In the context of global computation, whenever an *unknown*-valued variable is being used in some expression, the corresponding provision script may request a value from the Internet. Once the script assigned a value to the variable, the expression evaluation continues and this script is no longer triggered when the same variable is used (unless the *unknown* value has been explicitly assigned to it), which makes the program cleaner and more efficient. If this variable is defined as a field of a frame, by default, all instances of this frame or of its subframes will have this implicit property unless it is overridden.

On the other hand, the programmer might want to write a script to be triggered whenever a value is being stored in a variable or a field. This improves safety and security of systems because the programmer is able to protect the variables in a dynamic and clean way. There are other applications for this *assignment script*, for instance, whenever a value is assigned to a variable, the corresponding assignment script may execute a statement to write the value in a file, or perhaps some constraints are checked. If this variable is defined as a field of a frame, by default, all instances of this frame or of its subframes will have this implicit property unless it is overridden.

3.1.3 Inheritance

Apart from providing method inheritance, as any OO languages do, in the absence of a script to provide a value for a field in a frame, the virtual machine tries to obtain the value from its superframe whenever it is applicable. The superframe may have a script to provide such value or already have the value itself. If not, the virtual machine continues to search in the frame hierarchy. In other words, in the absence of information, scripts should have higher precedence over field inheritance, because they are local to the concept being represented.

3.1.4 Abstract Negation, Logic Programming

The universe is full of laws of cause and consequence, both natural and artificial laws. This makes declarative programs easier to write and understand then imperative programs. In the case of Intelligent Agents, facts and rules are dynamically inserted in their knowledge bases as part of the learning process in an appropriate way.

A Prolog-like language, Kleene, has been designed to be part of the larger language. Kleene provides *abstract negation*[14]. The contribution of the Kleene language is to offer a model of logic programming with the ability to distinguish what is false from what is absent from the search space of the program (by using the *unknown* value), as well as allowing the programmer to write predicates in both open and closed-world assumptions. This proposal contrasts with Prolog and almost all logic programming languages defined so far, as they regard what is absent as *false*. As the negation as failure[9] is a concept which depends on the closed-world assumption, a new kind of negation has been needed to allow the ability to distinguish between *false* and *unknown*.

In comparison to the state of the art in terms of theoretical work, Extended Logic Programming[16][19], the present approach has the advantage of offering only one kind of negation, the abstract negation, while Extended Logic Programming offers two kinds of negation, namely, negation as failure and explicit negation, which makes the programming task harder.

The term *abstract* here suggests that in the body of a rule the program does not need to know whether the predicate has been defined with the open or closed-world assumption. This is particularly useful for Intelligent Agents roaming over the Internet because the predicate used in the body of some rule might be defined elsewhere in the globe in such a way that the programmer does not have access to its definition. It follows that, in global computation, the negation must be abstract with respect to the assumption about the world.

The proposed approach is particularly useful for global computation, because Agents may contain their own declarative knowledge bases and may also communicate, learn and move over the Internet, where sometimes pieces of knowledge are not available. Rules have the form below:

$$[not]\, p(t_1, \ldots, t_n) \Leftarrow [not]\, p_1(t_{1,1}, \ldots, t_{1,r}), \ldots, [not]\, p_m(t_{m,1}, \ldots, t_{m,s}).$$

where $\Leftarrow$ stands for an inference operator, either $\leftarrow$ or $:-$. The square brackets are at the meta level and indicate that the not operator is optional in those positions. The predicate symbols are denoted by p_i, for $1 \leq i \leq m$. It was shown that a slight change would be sufficient in the Prolog unification algorithm. Let us now consider the classical non-flying bird example, with both open- and closed-world assumptions, in a very informal way:

```
fly(X) <- bird(X), not penguin(X).
not fly(Y) <- penguin(Y).
bird(tweety).
penguin(Z) :- bird(Z), polar(Z).
```

The program states that birds fly except penguins, which are 'polar birds'. From the above sequence of rules, penguin is the only polar bird. Moreover, if it is not known that an object is a bird, it is also unknown whether this object flies or not. The forth clause is based on the CWA and behaves as in any General Logic Programs if one interprets *unknown* as *false*. By default, a predicate with only clauses without an explicit body is under an open world assumption, which means that failure in the search for a unifying clause results in *unknown*.

To answer the query $fly(tweety)$, the system unifies the goal with the head of the first rule, binding X to $tweety$. Then, the system finds the subgoal $bird(tweety)$ which unifies the third clause. *not penguin(tweety)* is the next subgoal to be explored in the first rule. The subgoal unifies the fourth rule, biding Z to $tweety$. As the subgoal $bird(tweety)$ had already been proven, the next subgoal is $polar(tweety)$. Finally, the system does not unify any clause and, because of this, this subquery results in $unknown$. The forth body results in *unknown* and hence the subquery *penguin(tweety)* results in *false*, as the fourth rule adopts the CWA. Then, *not penguin(tweety)* results in *true*, making the body of the first rule to become *true*. Finally, the query results in *true*.

While an Agent may be defined as a set of frames, frames may have their own sequence of rules written in Kleene. Due to the *unknown* value, when the partial result of a query is *unknown*, the search may continue in the superframe or, alternatively, it may be delegated to another Agent. The integration of frames with logical clauses seems to be an attractive form of knowledge representation for Agents.

3.2 Uncertainty

The present hybrid-paradigm language considers multi-valued reasoning. Instead of the *boolean* data type, the language provides the *logic* data type. Besides the values *true*, *false* and *unknown*, the *logic* type has a confidence factor between -1 and $+1$ along with two *truth thresholds* from which one of the three truth values is obtained.

Many systems that are able to make decisions require some sort of inference that makes use of uncertainty, instead of rules from the first-order predicate logic. The ability to represent and reason under uncertainty is a very flexible feature that a language may have, and is surely very helpful for computation on an unreliable media. The present language provides the MYCIN[30] model of uncertainty because of its simplicity, while other models may be programmed.

3.3 Frames

Frame is an old idea due to Marvin Minsky. One of the main differences between frames and objects is that frames inherit both methods and values of fields while in OOP, classes inherit methods but not values of fields. The *unknown value* allows a frame to search automatically a default value in its superframe. Therefore, changing a field value in a frame corresponds to changing the default value of its subframes. Each field in a frame may have a script attached to it, which indicates that when its value is needed in some expression and it is *unknown*, the script will be triggered instead of inheriting a default value from a superframe. Thus the purpose of the script is to provide a default value more specific than the one in the superframe and hence has higher priority during the evaluation than inheritance.

As an example of this approach, "Brazilian" is a prototype implemented as a frame with a field called *LifeExpectancy* with its corresponding general value. A subframe "Carioca" has a provision script for the corresponding field, which assigns *1.05*Super.LifeExpectancy* to it (this proportion is probably not real, just to exemplify). As Brazilian's lives will become better soon, the corresponding value is increased and the script for *Carioca. LifeExpectancy* should provide a value proportionally increased. This mechanism is not supported by class-based object-oriented languages, since the links between classes and subclasses cease as the objects are constructed.

4 Conclusion

In the presence of the *unknown* value in the language, some language features present in other languages disappeared. For example, in terms of OOP, the language provides single inheritance, but a class with multiple-inheritance may be easily implemented by programming scripts to solve conflicts. Also, nothing prevents an implementation which creates objects as they are being referred, instead of using an operator such as *new*. The *unknown* value also permits constraint programming, which is not in the current definition.

The logic programming sublanguage, Kleene, is as simple as Prolog but more expressive and allows to program without the Closed-World Assumption. This language is not totally implemented, only a compiler in C and an abstract interpreter in Prolog were written. The other paradigms are running however, and an Expert System in this language, SENERGIA, has been used in Mataripe Refinery, Petrobrás.

As a future work, the language may provide a model that allows dynamically resources hire, such as CPU hire, provided that Agents *identities* are guaranteed.

The language is implemented in two modules: a compiler and a virtual machine, both having a bit more than 20,000 lines of C source code altogether. Concerning the object code, the compiler has roughly 250 kbytes while the virtual machine has roughly 200 kbytes. The system is available for downloading from its web page[15], runs on Unix machines and may be useful for experiments. Some small examples are also running from this web page. Mobility is implemented on the e-mail service. Thus, mobile Agents currently migrate by e-mail, while virtual machines check regularly e-mail messages in the inbox. Once it recognises a coming computation, the virtual machine loads its code, data and state, and then continue the process.

References

[1] H. Ait-Kaci et al. *The Wild LIFE Handbook*. Digital, Paris Research Laborator, 1994. http://www.isg.sfu.ca/life/.

[2] J. Allen and J. Hendler, editors. *Readings in Planning*. Representation and Reasoning. Morgan Kaufmann, San Mateo, California, 1990.

[3] R. Barrett, A. Ramsay, and A. Sloman. *POP-11: A practical language for artificial intelligence*. Ellis Horwood series in computers and their applications. Halstead Press, Chichester New York, 1985.

[4] J. Bredin, D. Kotz, and D. Rus. Market-based resource control for mobile agents. In *Proceedings of Autonomous Agents '98*, pages 197–204, 1998. Earlier version published as Darmouth College, Department of Computer Science technical report TR97-326.

[5] L. Cardelli. Global computation. *ACM Computing Surveys*, 28A(4), 1996.

[6] L. Cardelli. *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, chapter Mobile Computation. Springer-Verlag, Linz, Austria, 1997.

[7] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? – update. In *Mobile Object Systems: Towards the Programmable Internet*, pages 46–48. Springer-Verlag, Apr. 1997. Lecture Notes in Computer Science No. 1222.

[8] K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[9] K. L. Clark. *Logic and Data Bases*, chapter Negation as Failure, pages 293–322. Plenum Press, New York, 1978.

[10] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing mobile code languages. In *Mobile Object Systems: Towards the Programmable Internet*, pages 93–110. Springer-Verlag, Apr. 1997. Lecture Notes in Computer Science No. 1222.

[11] D. Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[12] P. Dugerdil. *Contribuition à l'étude de la représentation des connaissances fondée sur les objects: le language ObjLog*. PhD thesis, GRTC, Université d'Aix-Marseille III, France, 1987.

[13] W. M. Farmer, J. D. Guttman, and V. Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 118–130, Rome, Italy, Sept. 1996. Springer-Verlag Lecture Notes in Computer Science No. 1146.

[14] U. Ferreira. Abstract negation in logic programming. *URLs http://www.cs.tcd.ie/˜ferreirj/plain.html and http://www.ufba.br/˜plain*, 1998.

[15] U. Ferreira. Plain: A hybrid-paradigm programming language. *URL http://www.ufba.br/˜plain*, 1998.

[16] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing. Ohmsha Ltd and Spring-Verlag*, pages 365–385, 1991.

[17] R. S. Gray. Agent tcl: A transportable agent system. In *Proceedings of the CIKM'95 Workshop on Intelligent Information Agent*, 1995.

[18] M. N. Huhns and M. P. Singh, editors. *Readings in Agents*. Morgan Kaufmann, San Francisco, California, 1997.

[19] A. C. Kakas, R. A. Kowalski, and F. Toni. *The Role of Abduction in Logic Programming, in Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.

[20] B. Mathiske, F. Matthes, and J. W. Schmidt. On migrating threads. Technical report, Fachbereich Informatik Universitat Hamburg, 1994.

[21] F. Matthes, S. Mussig, and J. W. Schmidt. Persistent polymorphic programming in tycoon: An introdution. Technical report, Fachbereich Informatik Universitat Hamburg, 1993.

[22] F. G. McCabe. *Logic and Objects*. Prentice Hall International Ltd, 1992.

[23] D. Miller. The forum specification language. *http://www.cis.upenn.edu/˜dale/forum*, 1998.

[24] M. Minsky. A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laborator, 1974.

[25] C. Moss. *Prolog++ The Power of Object-oriented and Logic Programming*. Addison-Wesley, 1994.

[26] G. Nadathur and D. Miller. An overview of λprolog. In *Proceedings of the 5th International Conference on Logic Programming*, Cambridge, MA, 1989. MIT Press.

[27] E. Pontelli and G. Gupta. Web-ace: A logic language for intelligent internet programming. *http://www.cs.nmsu.edu/lldap/prj_lp/web*, 1998.

[28] R. Ramakrishnan et al. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Databases*, 1993.

[29] R. Reiter. *Logic and Data Bases*, chapter On Closed World Data Bases, pages 55–76. Plenum Press, New York, 1978.

[30] E. H. Shortlife. *Computer-Based Medical Consultations: MYCIN*. New York, 1976. Elsevier.

[31] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, 1995. also in http://www.cs.mu.oz.au/research/mercury.

[32] J. White. *Telescript Technology: the Foundation for the Electronic Marketplace*. General Magic, Inc., 1994.