Agent Programming in Ciao Prolog

Francisco Bueno and the CLIP Group*
Facultad de Informática – UPM
bueno@fi.upm.es

The agent programming landscape has been revealed as a natural framework for developing "intelligence" in AI. This can be seen from the extensive use of the agent concept in presenting (and developing) AI systems, the proliferation of agent theories, and the evolution of concepts such as agent societies (social intelligence) and coordination.

Although a definition of what is an agent might still be controversial, agents have particular characteristics that define them, and are commonly accepted. An agent has autonomy, reactivity (to the environment and to other agents), intelligence (i.e., reasoning abilities). It behaves as an individual, capable of communicating, and capable of modifying its knowledge and its reasoning.

For programming purposes, and in particular for AI programming, one would need a programming language/system that allows to reflect the nature of agents in the code: to map code to some abstract entities (the "agents"), to declare well-defined interfaces between such entities, their individual execution, possibly concurrent, possibly distributed, and their synchronization, and, last but not least, to program intelligence.

It is our thesis that for the last purpose above the best suited languages are logic programming languages. It is arguably more difficult (and unnatural) to incorporate reasoning capabilities into, for example, an object oriented language than to incorporate the other capabilities mentioned above into a logic language. Our aim is, thus, to do the latter: to offer a logic language that provides the features required to program (intelligent) agents comfortably.

The purpose of this talk, then, is not to introduce sophisticated reasoning theories or coordination languages, but to go through the (low-level, if you want) features which, in our view, provide for agent programming into a (high-level) language, based on logic, which naturally offers the capability of programming reasoning.

The language we present is Ciao, and its relevant features are outlined below. Most of them have been included as language-level extensions, thanks to the extensibility of Ciao. Hopefully, the Ciao approach will demonstrate how the required features can be embedded in a logic programming language in a natural way, both for the implementor and for the programmer.

---

*Daniel Cabeza, Manuel Carro, Jesús Correas, José M. Gómez, Manuel Hermenegildo, Pedro López, Germán Puebla, and Claudio Vaucheret

State and its Encapsulation: from Modules to Objects The state that is most relevant for programming intelligence is the state of knowledge. Classically, a logic program models a knowledge state, and, also, logic languages provide the means to manipulate and change this state: the well-known assert/retract operations. These can be used for modeling knowledge evolution. On the other hand, state of data and its evolution are arguably best modeled with objects. Also, objects are a basic construct for capturing "individuality" in the sense of agents.

What is needed is a neat differentiation between the code that represents the state of knowledge and the code that represents the evolution of knowledge. A first step towards this is providing state encapsulation. This can be achieved with a well-defined, well-behaved module system. This is one of the main principles that has informed the design of Ciao. Having a language with modules and encapsulated state, the step towards objects is an easy one: the only extra thing needed is instantiation. Once we add the ability to create instances of modules, we have classes. This has been the approach in O'Ciao, the Ciao sublanguage for object orientation.

Concurrency and Distribution These two features are provided in Ciao at two levels. At the language level, there are constructs for concurrent execution and for distributed execution. At the level of "individual entities", concurrency and distribution comes through via the concept of active modules/objects.

Reactivity and Autonomy: Active Modules/Objects A module/object is active when it can run as a separate process. This concept provides for "autonomy" at the execution level. The active module service of Ciao allows starting and/or connecting (remote) processes which "serve" a module/object. The code served can be conceptually part of the application program, or it can be viewed alternatively as a program *component*: a completely autonomous, independent functionality, which is given by its interface.

Active modules/objects can then be used as "watchers" and "actors" of and on the outside world. The natural way to do this is from a well-defined, easy-to-use foreign language interface that allows to write drivers that interact with the environment, but which can be viewed as logic facts (or rules) from the programmer's point of view. An example of this is the SQL interface of Ciao to sql-based external databases.

And More Adding more capabilities to the language, in particular, adding more sophisticated reasoning schemes, requires that the language be easily extensible. Extensibility has been another of the guidelines informing the design of Ciao: the concept of a package is a good example of this. Packages are libraries that allow syntactic, and also semantic, extensions of the language. They have been already used in Ciao, among other things (including most of the abovementioned features), to provide higher order constructions like predicate abstractions, and also fuzzy reasoning abilities.