Rational Programming

Yoav Shoham
http://cs.stanford.edu/∼shoham

Original version: January, 1999

June 5, 2003

1 Introduction

Since its inception, AI has played an important role in the development of novel programming paradigms. For example, and without getting too deeply into questions of credit assignment, there is no doubt that AI has played a critical role in the development of functional programming (e.g., McCarthy's LISP [McC60]), object oriented programming (e.g., Hewitt's Actors [Hew77]), and logic programming (e.g., Green [Gre69], Colmerauer [Col82], and Kowalski [Kow79]).

In recent years AI hasn't had impact of this magnitude on programming languages. At the same time, within AI there have been several creative marriages between PL notions and AI notions (the latter primarily from the knowledge representation and reasoning area). There are many more similar marriages that can be forged. What they all have in common (at least the ones I have in mind) is the incorporation of some elements of *rationality* in the programming language. (As we'll see, in some instances the situation is reversed; programming language constructs are used to represent elements of rationality.)

The fairly recent work we are alluding to includes (in rough historical order) McCarthy's Elephant2000 [McC90], Shoham's Agent Oriented Programming [Sho93], Wellman's Market Oriented Programming [Wel93], Koller and Pfeffer's GALA [KP95], and Pfeffer's IBAL [Pfe01]. These are discussed further below.

To the extent that one wishes to base rationality at least in part on logical thinking, logic programming (LP) can be thought of as embodying an element of rationality. Notice though that LP embodies logic in a rather particular way. First, it is based on first-order logic; one can imagine other logics being used, and in fact several have been proposed (for example, modal logic [ABndC$^+$86]). Second, logic is embraced in an extreme fashion. For one thing, *all* elements of the programming language (program, data, variables, constants) are constructs of first-order logic; one could imagine incorporating logic into the language without having it take over the language completely (indeed, early incarnation of LP ideas tried various mixtures of LP and functional programming). And most importantly, not only is the syntax of the language logical, but its semantics are based directly on an inference procedure for first-order logic (namely, first-order resolution). This is perhaps the most radical element of LP as we know it, its purported declarative nature (I say 'purported'

since, as I've written elsewhere, the relative success of Prolog was derived in part precisely because it departed from the declarative ideal). These two elements, logical constructs and implicit process specification, together embody a particular version of constraint satisfaction (namely, unification).

Thus LP embodies several important (though not universal) tenets of AI: logical representation, theorem proving, and constraint satisfaction. It was already mentioned that LP embodies them in a particular way, and one question this raises is whether they can be incorporated differently. But perhaps more interestingly, one can ask whether other elements of rational thinking can be codified and pressed into programming service. For example, one can ask whether several modern techniques from the areas of knowledge representation and reasoning can be used. Is there an interesting notion of *probabilistic programming*? Of *default programming?* I think the answer to these questions is yes, and will return to them later.

Notice that all elements of rationality mentioned so far – logic, probability, defaults – are taken from the realm of reasoning in the sense of thinking about what is or is not the case. Rationality however is usually taken to refer to not only to the quality of such thinking, but also (and perhaps primarily) the quality of choice making. Can we take notions of motivation and self interest – utilities, goals, preferences, and so on – and meaningfully incorporate them in a programming language? Again, I believe the answer is yes. In fact, here is where I think some of the most interesting opportunities lie, especially as we enter the age of aggressive use of the internet, and envision needing to write programs that embody our wishes and execute over long periods of time, in the process interacting with other programs that embody the wishes of their respective programmers. It would be convenient if one were able to meaningfully impart to such programs certain motivations, preferences, and goals, and be confident that the programs will execute in accordance with these motivations etc. as prescribed by some clear theory of rationality.

2 A look at existing examples

Elephant2000 is a language that was described by McCarthy in informal notes [McC90]. While it has several strands and was never fully fleshed out, two ideas stand out. The first one is the temporal interpretation of program variables. In Elephant2000 one can refer to the time at which variables were assigned values, so that in particular one can say "if the variable was assigned the value 5 and since then the value has not changed, then ..." The other strand is the incorporation of *speech acts* into the language. Although full discussion of speech-act theory is not possible here, for the reader not familiar with it we mention that it is an influential linguistic theory, which holds that communication acts are just like any other acts in that they have intended and actual effects, and can be reasoned about accordingly.

An example statement in Elephant2000, taken from a hypothetical travel agency program, is

```
if ¬ full flt
then accept.request(
       make commitment (admit (psgr, flt)))
```

and its intuitive reading is "if a passanger has requested to reserve a seat on a given flight, and that flight is not full, then make the reservation."

Agent Oriented Programming (AOP) [Sho93] is similar in several respects. It too includes temporal operators in the language, though differently from Elephant2000. Interestingly, AOP too embraces speech acts as the form and meaning of communication among agents. The most significant difference from Elephant2000 is that AOP embraces in addition the notion of *mental state*. Agents in AOP each have a mental state, consisting of components such as beliefs and commitments, whose form and meaning stand in rough correspondence with their commonsense counterparts. In fact, the semantics of speech acts are defined in terms of this mental state; for example, the result of an INFORM is a new belief.

AOP is actually not a single language, but a general design that allows multiple languages and associated interpreters. These languages can differ on several dimensions, and in particular on the choice of mental state. As a proof of concept, a particular simple language was defined and implemented, called Agent0. An example statement in Agent0 (with some extra sugar added for readability) is

```
IF
 MSG COND: (?msgId ?someone REQUEST
             ((FREE-PRINTER 5min) ?time))
 MENTAL COND:
   ((NOT (CMT ?other (PRINT ?doc (?time+10min))))
    (B (FRIENDLY ?someone))
THEN COMMIT
 ((?someone (FREE-PRINTER 5min) ?time))
  (myself (INFORM ?someone (ACCEPT ?msgId)) now)
```

and its approximate meaning is "If you get a request to free the printer for five minutes at a certain future time, if you're not committed to finishing a print job within ten minutes of that time, and if you believe the requester to be friendly, then accept the request and tell them that you did."

Market Oriented Programming (MOP) [Wel93] embraces economic mechanisms, in particular price and auction mechanisms. In one of the first MOP applications Wellman implemented a distribution transportation problem by coding up a Walrasian auction. This worked roughly as follows. For each commodity (i.e., task) there was a clearing house. Each agent notified each clearing house of its corresponding buying curve (how much of the particular commodity – or transporation task – it would be willing to take on for any given "price"); based on these buying curves each clearing house determined the current "price"; and the process repeated until reaching an equilibrium.

GALA is a language in which to specify games in the sense of game theory [KP95]. Implemented in Prolog and inheriting a Prolog-like notation, in GALA one specifies a game by writing programming-language-style the sequence of actions of the game. Once the game is specified, the system first constructs an internal representation of the game, and then goes ahead and computes solutions for the game (i.e., the Nash equilibria).

Consider for example the "inspection game." In this game there are two agents: a country that would like to experiment with a weapon counter to an international treaty, and an inspector whose job it is to detect treaty violations. If the country pulls off the experiment without being caught it wins, if it gets caught conducting one it loses, and otherwise nothing special happens. Specifically, the game consists of N stages, in each of which the country

decides whether to violate and the inspector decides whether to inspect. The inspector can only inspect up to M times, for some M<N. Here's the core of the GALA program that implements this game:

```
flow :
 (repeat(stage, unless(no_more_inspections),
                unless(no_more_stages),
  determine_outcome)
stage:
 (choose(violator, X, (X = yes ; X = no)),
  choose(inspector, Y, (Y = yes ; Y = no)),
  reveal(violator, Y),
  violate gets X,
  inspect gets Y,
  if($inspect = yes,
     decrement $inspections_remaining 1),
  decrement $stages_remaining 1)
```

It's not possible here to give a complete description of GALA, but note some key features: using programming constructs such as `repeat` and `if`, denoting choice points of agents by `choose`, and capturing the information states of agents by `reveal`.

Finally, IBAL [Pfe01] allows for specification of probabilistic and decision-theoretic agents.[1] The semantics of various expressions are extended to include probability distributions over values, as well as utilities of various outcomes. For example, an assignment operator $x = e$ now defines a stochastic experiment that generates the value of x. This in itself allows IBAL to express many *generative* models, such as Bayesian networks and hidden Markov models. The language then goes further to allow for *rejective* models, such as *product of experts*. There data are generated uniformly and then are passed to a series of experts, who may accept/reject it, thus inducing a probability distribution. This is accomplished by introducing *observations*. An observation is a statement in IBAL that asserts that a certain boolean predicate is true. Given probabilistic nature of all values, this allows IBAL to *condition* variables given observations, which is accomplished by an underlying Bayesian inference engine. Observations also provide the basis for integrating learning, in the form of Bayesian parameter estimation, into the IBAL framework.

IBAL goes further and incorporates decision theoretic concepts by allowing *decision* and *reward* declarations in program blocks. These declarations specify which actions/information are available to the decision maker, as well as utilities associated with those actions. For example, the following is a core IBAL representation of a typical Markov Decision Problem:

```
MDP(s) = {
(* takes current state as argument *)

(*define a transition function*)
transition(s,a)= {
```

[1] The following description of IBAL was supplied by Eugene Nudelman.

```
   if a |= 'a1 and s |= 's1
   then dist[0.3:'s1,0.7:'s2]
   else ...
  }

(*define an immediate reward function for each state-action pair*)
reward(s,a) = { reward case (s,a) of [('s1,'a1):3, ('s1, 'a2):5, ...] }

(*make a decision given current state*)
choose a from a1, a2 given s

(*declare system dynamics*)
next_state = transition(s,a)
current_reward = reward(s,a)
future_reward = MDP(next_state)

(* discount rewards *)
reward 1 current_reward
reward 0.9 future_reward
}
```

This defines an MDP with discounted rewards and a stochastic transition function. The *choose* statement states that the program must choose one of the actions (a1, a2) that maximizes the expected utility of the block. Utility is specified via the *reward* statements. Note the declarative nature of IBAL: the way to choose an optimal strategy is left unspecified, that is done by an underlying inference engine.

2.1 Some observations

Clearly, while they all play on connections between notions of rationality and computation, the five pieces of work just described are different in many ways. Here are some observations, which will help put this work into perspective, and place some structure on the space of rational programming.

To start with the obvious, the five appeal to different elements of, and versions of, rationality, from temporal reasoning to speech acts to mental state to auctions to games. Despite this, note that at least four of the systems are "multi-agent" in one sense or another. Both Elephant2000 and AOP are multi-agent languages in the sense that they provide primitives for structured interaction among different programs (or "agents"); both appeal to speech acts for structured communication, and in addition in AOP one explicitly models other agents using the notion of mental state. GALA and MOP are multi-agent in a stronger sense; they embody specific interaction mechanisms, and not merely tools with which to program such interactions. MOP uses auction mechanisms, and GALA uses an algorithm that computes Nash equilibria. It is not clear to what extent IBAL is multi-agent, however. Certainly it allows one to easily define a time line of a game. However, currently it doesn't have any notion of game-theoretic solution concepts, or any other built-in notion related to the interaction among multiple agents.

I believe that programming in a multi-agent environment lies at the heart of the new opportunity to innovate in the area of programming languages and systems. Inter-application communication has been around for many years, and has been pushed aggressively in industry in recent years (at some point the battle lines were drawn by Microsoft pushing OLE/COM and OMG pushing CORBA, which later morphed into a DCOM/Java matchup; today the action is around defining XML-based standards). However, all the work has been put into enabling the mechanics of interaction (in which we include also the interesting work on mobile code; see below). The field is wide open as far as effecting meaningful interactions among disparate programs, each of which answers to different masters, which is in large measure what the five systems discussed are about.

Next, notice that neither MOP nor GALA are programming languages or systems *per se.* MOP is 'programming' in the sense that linear programming and integer programming are. That is, MOP is simply an optimization method that happens to be based on an auction-like procedure. It is 'multi agent' in that the underlying procedure is inherently distributed, consisting of the individual agents bidding independently. However all agents are programmed by the same entity, and all in the service of the same optimization problem.

In a different setting, the same is essentially true of GALA. In fact, when one writes a GALA program one isn't specifying a process but a game; once the game is specified, the GALA engine generates an internal game representation and proceeds to solve it. Furthermore, unlike in the case of logic programming, there are no other data structures or variable values that are computed and returned to the user, nor any side effects (except those brought about by the running of arbitrary Prolog programs, which GALA allows). Thus, rather than apply an element of rationality to the specification of a computational process, GALA uses notions of computation (namely, programming constructs) to specify an element drawn from the realm of rational behavior (namely, a game). Thus perhaps it is more appropriate to describe this work as *programmatic rationality* rather than rational programming.

It's not clear to me at this point how best to view IBAL in this respect. My initial feeling is that the purely probabilistic component adopts a rational programming stance and the decision theoretic component a programmatic rationality one (which would make the otherwise very interesting IBAL problematic in this respect). But I believe that IBAL is too young to pronounce judgment here.

In general, in looking at these systems we note a tradeoff between power and generality. If one hard codes a particular interaction mechanism into the system one provides much power to the user, but the power comes at the price of flexibility; in the extreme case, one ends up with a library one can link into a given application, but not a general purpose programming tool. The alternative is to provide *some* interaction structure, on top of which one can implement many different interaction mechanisms; the downside of this approach is that there might be too little power in the primitives to justify giving up the convenience of familiar standard languages.

Finally, note that the five systems discussed differ on how many people are expected to do the programming in the context of a given application. Elephant2000 and AOP lie at one extreme – although one could write self-contained applications in e.g., AOP just as one does in an object oriented programming, the strength of the system is in allowing one's program to interact with any number of other programs about which there is very little a priori information. GALA lies at the other extreme – there is exactly one programmer, or rather, game specifier. MOP lies inbetween; there is one programmer who sets up the market

economy, but then multiple participants who might participate via programs, the writing of which (as discussed above) currently lies outside the purview of MOP. IBAL was really designed for single-agent applications, therefore it assumes a single user.

2.2 What next?

There are many new directions in which to push the paradigm of rational programming; here are a few.

- *Default* and *probabilistic programming.* If one can program with first-order logic, why shouldn't one be able to program with default logic or probabilities?

  IBAL already introduces variables with probabilistic interpretation. Essentially, an IBAL program specifies joint conditional distributions over values of its blocks/variables. For these variables, the notion of 'assignment' is generalized to that of 'observation'; observing (or setting) the value of one variable can change the values of others.

  In addition to probabilistic variables one could imagine default ones. That is, one could imagine that a variable might take on one of two kinds of value – definite and default. We would presumably have two sorts of assignment statement, perhaps `:=` (assign default value) and `:==` (assign definite value). The intention is of course that definite assignment can override any existing value, but default assignments can only override default values (they have no effect when applied to a variable with a definite value). However, it might also be useful to unassign a variable its definite value without assigning it a new one, so that (for example) it can be assigned a new default value. Tests too can come in default and definite flavors, such as `=` (equal by default) and == (definitely equal) (borrowing from Prolog notation).

  This mechanism is reminiscent of Prolog's negation-as-failure, except that here we are not restricted to a particular default mechanism (that derived from the procedural semantics of Prolog), but may implement any default theory we wish.

- *Where to GO?*

  An intriguing notion in programming languages, which was given particular relevance in the context of wide-area networks, is *mobile code.* Languages for mobile 'agents' contain primitives for migrating the program in the network; for simplicity we will refer to this command as GO, which is the command used in the Telescript language [Whi94]. When the program interpreter comes across this command, it freezes its state (including program stack and variable heap), ships itself across the network to the specified destination, and (all subject to permissions) continues to execute at that location.

  To date the question of where the object should ship itself has been resolved simply by specifying an exact address. But what if the object goes through a reasoning process before it decides where best to continue to execute? This reasoning could take into account known capabilities of different venues; it could learn over time where best to go; and it could enter into negotiations with various sites about the conditions for its executing there (for example, the amount of computing resources it is allocated, or the hosting fees.) This is an excellent domain in which to apply theories of uncertainty,

learning, knowledge representation, planning, and computational economics. In particular, there is an opportunity to apply ideas from mechanism design, to ensure efficient use of network resources by the self-interested mobile objects.

- *Trusting software.* The internet is an anonymous space. Whom do you trust on the internet? More importantly, whom should your agent trust? How does your agent come to *be* trusted? In real life we trust people either by reasoning about their motivations, or by learning their behavior over time. Can these lessons be transferred to the software arena?

  One direction this can be pursued is by generalizing existing work on belief revision. In a multi-agent setting a program will need to make decisions about whether to believe facts it is told by other programs. Current theories of belief change assume complete gullibility, and hence provide no guidance. It seems to me that one can develop a theory of *information pedigrees*, and incorporate it into a multi-agent programming system. One element of such a language might be a three-argument assignment operation, as in

  ```
  Variable := Value {Source}
  ```

  with the intended interpretation that a particular source is attempting to set the value of the variable; the semantics of this operation would appeal to the pedigree of the source. Here some ideas from qualitative belief fusion [IS01], which introduces the notion of information pedigree, may prove useful.

- *Utilitarian programming.*

  Could a language be developed that allows the user to specify his/her preference relation or utility function, so that the loader and/or interpreter generate warnings and errors when the program attempts to take action that is inconsistent with these? A prime application for such a language could be trading programs, for example ones written for the Trading Agent Competitions run over the past several years. One could imagine a programmer specifying a trading policy as well some of its preferences and beliefs, and the compiler noting dominated strategies, potentials for breaking a budget, and other potential sources of trouble. This is of course a very challenging direction, and the boundary between a programming tool and a decision-support system is not clear to me. Design of the component with which the user specifies preferences is in itself a good challenge for AI; some of the recent work in modular utility representation and utility elicitation might come in handy.

- *Human-oriented programming.* Imagine that a program interpreter automatically tracks the user's interaction with the implemented system, builds a model of the user's level of interest and proficiency, and customizes its behavior accordingly. Such activities are already being carried out (for example, and until it was phased out, in Microsoft's context-sensitive help that is based on a Bayesian-net model of the user [Cor96, Hor]), but not as part of the development effort, and certainly not aided automatically by the development system. Could we get to the point at which user-monitoring is achieved as an integral part of programming? If so, can it be facilitated by having the right programming constructs, as opposed to simple-minded instrumentation of the interpreter?

After all, the programmer is in the best position to understand what goal the program is trying to achieve, and, given the right tools, s/he could give valuable hints to the user-tracking component of the interpreter. Here, too, it seems that representing and eliciting the user's preferences and skills will be an interesting challenge in itself.

References

[ABndC+86] R. Artuaud, P. Bieber, L. Fari nas del Cerro, J. Henry, and A Herzig. Molog: Manuel d'utilisation. Internal report, I.R.I.T., Université P. Sabatier Toulouse (France), 1986.

[Col82] A. Colmerauer. Prolog-ii: Manuel de reference and modele theorique. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix, Marseille II, 1982.

[Cor96] "Microsoft Corporation". Intellisense in microsoft office 97, September 1996.

[Gre69] C. C. Green. Application of theorem proving to problem solving. In *Proceedings IJCAI 1*, pages 219–239. IJCAI, 1969.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hor] E. Horvitz. http://www.research.microsoft.com/ research/dtg/horvitz/lum.htm.

[IS01] P. Maynard-Reid II and Y. Shoham. Belief fusion: Aggregating pedigreed belief states. *Journal of Logic, Language, and Information*, 10(2):183–209, 2001.

[Kow79] R. Kowalski. *A Logic for Problem Solving.* North-Holland, New York, 1979.

[KP95] D. Koller and A. Pfeffer. Generating and solving imperfect information games. In *Proc. IJCAI-95.* Morgan Kaufmann Publishers, 1995.

[McC60] J.M. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(3):184–195, 1960.

[McC90] J. McCarthy. Elephant 2000: A programming language based on speech acts, 1990. unpublished manuscript.

[Pfe01] A. Pfeffer. Ibal: A probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 733–740, Seattle, WA, 2001.

[Sho93] Y. Shoham. Agent Oriented Programming. *Journal of Artificial Intelligence*, 60(1):51–92, 1993.

[Wel93] M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of AI Research*, 1:1–23, 1993.

[Whi94] J. White. Telescript technology (white paper). Technical report, General Magic, Inc., 1994.