Computational Logic as a Dual Process Model of Thought

Bob Kowalski
Imperial College London

10 February 2006

Abstract In the *dual process model* of thinking, developed in Cognitive Psychology, two kinds of thinking operate in tandem. *Intuitive* thinking is automatic, effortless and largely subconscious, while d*eliberative* thinking is controlled, effortful and mostly conscious. Logic is normally associated only with deliberative thinking. However, in this paper I argue that Computational Logic can be used to model both *intuitive* and *deliberative* thinking, as well as some of the relationships between them.

In Computing, Computational Logic is a wide-spectrum language, which can be used to model both high-level specifications as well as low-level condition-action rules and neural network, stimulus-response associations. Moreover, the same kinds of forward and backward reasoning that are used at runtime to execute logic programs can also be used to transform/compile high-level specifications into lower-level form, and sometimes to decompile lower-level programs into higher-level form. I argue that this use of logic in Computing can be extended to provide a logical and computational model of the dual process theory of human thinking.

Introduction

Computational logic [1] is the use of logic as a computational formalism. It extends normal logic programming (Horn clauses plus negation as failure) by means of such features as explicit negation, integrity constraints and the amalgamation of object language and meta-language. It has been used as a high-level representation language in software engineering and artificial intelligence, and also to formalise argumentation in legal reasoning [2].

Many of the applications of computational logic, at this high level, focus on its use for declarative knowledge representation. In these applications, the procedural interpretation of logic programs, in which implications are used backwards to reduce goals to sub-goals, is merely an implementation technique. At this level, computational logic can be regarded as a candidate formalisation of system 2 processes in a dual process model of thought [3].

However, for many other applications, the procedural interpretation is used explicitly to represent goal-reduction procedures. Sometimes, this is because high-level knowledge is itself procedural in nature. At other times, it is because, for the sake of efficiency, high-

level declarative representations need to be transformed into more efficient, lower-level, procedural form.

Abductive logic programming (ALP) [4] extends normal logic programming by the inclusion of integrity constraints and forward reasoning. In ALP agents [5], ALP is used to model the thinking component of an agent's interaction with the world. In this model, the agent's beliefs are modelled by normal logic programs and its goals are modelled by integrity constraints.

Integrity constraints in ALP agents serve as maintenance goals, which include the condition-action rules of production systems as a special case. Reasoning forwards[1] from observations, using maintenance goals and beliefs, generalises the forward execution of condition-action rules in production systems. Abductive logic programs, therefore, inherit some of the psychological plausibility of cognitive models based on production systems, such as SOAR [6] and ACT-R [7].

The relationship between logic programming and ACT-R has been noted by Stenning and van Lambalgen [8, 9]. They also argue that logic programs with integrity constraints can model human performance on such problems as the Byrne suppression task [10], which is often interpreted as showing that people are not logical.

Computational logic is a very high-level representation language. However, already in the early days of computing, propositional logic was used to model low-level electronic circuits and computer hardware. Moreover, in recent years, a number of researchers have shown both how to represent neural networks directly as low-level logic programs [11] and, conversely, how to compile arbitrary logic programs into lower-level neural networks [12, 13]. As Stenning and vanLambalgen [8, 9] argue, logic programs corresponding directly to neural networks can be regarded as a candidate formalisation of system 1 processes in a dual process model of thinking.

Thus computational logic can be used as both a high-level and a low-level computer language, and therefore as a model of both system 1 and system 2 kinds of thinking. It can also model some of the relationships and interactions between system 1 and system 2.

In computing, programs are often transformed or compiled into lower level form and sometimes decompiled into higher-level form. In logic programming, in particular, transformation [14, 15] of high-level representations into more efficient, lower-level form can be performed by applying at compile time reasoning steps that would otherwise need to be applied at run time. The use of program transformation and decompilation in computational logic is similar to some of the relationships between system 1 and system 2 processes in human thinking.

---

[1] Forward and backward reasoning are informal terms for special cases of the resolution rule of inference. Forward reasoning, in particular, is a form of *modus ponens* together with a form of universal instantiation that is restricted to matching the premise of the inference with a condition of an implication.

Claiming that computational logic might be able to model system 1 and system 2 levels of thinking should not be confused with claiming that people reason logically with abstract, meaningless hypotheses. On the contrary, the proposal made here is only that computational logic might be able to model the way that people represent and process their interactions with the concrete, real world.

Used for system 1 thinking, for example, ALP reasons forward to derive concrete actions from concrete experiences, in the manner of condition-action rules. Used for system 2 thinking, it reasons forwards to derive relevant abstractions from concrete experiences and to derive high-level achievement goals from high-level maintenance goals. It reasons backwards from achievement goals, reducing goals to sub-goals and eventually to candidate actions. Before executing actions in the environment, it reasons forwards hypothetically from the candidate actions, to simulate their likely consequences, to help in deciding what to do. The use of ALP for system 2 thinking to simulate and monitor candidate actions by forward reasoning can also be combined with its use for system 1 thinking to generate candidate actions using lower-level condition-action rules.

An Example

Suppose you are walking along the street minding your own business, when someone suddenly punches you in the face. It would be natural to react, without thinking, by punching the other person straight back. However, a more philosophical agent might be inclined to think logically at a higher level.

The first thing for the philosophical agent to do is to take stock of the experience. This can be done by reasoning forward from a record of the experience, to assess its significance, using such beliefs as:

A person has just punched me.
If a person punches me, then the person attacks me.

Therefore:

A person has just attacked me.

This conclusion directly matches the condition of a "maintenance goal":

If a person attacks me, then I defend myself against the attack.

One more step of forward reasoning derives the "achievement goal":

I defend myself against this attack.

At this point, the philosophical agent would need to consider alternative means of defence, of which one of the simplest is simply to respond in kind:

To defend myself against an attack,
I attack the person with a similar attack.

This goal-reduction procedure combines backward reasoning with a logical implication:

I defend myself against an attack
if I attack the person with a similar attack.

The sub-goal, reached by one step of backward reasoning, is the candidate action:

I attack the person by punching the person back.

Assuming there are no other alternative candidate actions, it takes only a simple act of will to turn the hypothetical, candidate action into a real act of punching the other person back in the face. If the philosophical agent is lucky, then there will still be enough time left to successfully maintain the top-level goal of self-defence. However, if the agent is not, it will be too late for the agent to perform any kind of action at all.

It's easy to regard this scenario as a caricature of logical reasoning. However, the end result of all this reasoning is equivalent to just one step of forward reasoning with the report of the experience and a lower-level maintenance goal (which behaves like a condition-action rule):

A person has just punched me.
If a person punches me, then I attack the person by punching the person back.

With enough foresight, the philosophical agent could have transformed the high-level representation of its goals and beliefs into this lower-level form, by performing the necessary reasoning steps in advance, before the need arises. A less philosophical agent, on the other hand, might be content to use the lower-level representation directly, without being aware of any higher-level goals and beliefs.

So far in this example, from the perspective of dual process theory, we have seen two ways of processing the experience. The philosophical agent uses system 2 thinking, either directly or in an optimised form as a system 1 response. The less philosophical agent is capable of only a system 1 response.

However, there is more to the story. Whether the candidate action of punching back is generated by system 1 or system 2, there is still the possibility of using system 2 to simulate the candidate action, by reasoning forward to derive its possible consequences, using such implications as:

If I attack a person and the person is stronger than me,

then the person will probably attack me back and hurt me.

If, indeed, the attacking person is stronger than me, then the foreseeable, likely consequences are not very desirable. It might be better to consider alternative means of defence, such as those associated with such other beliefs as:

I defend myself against an attack
if I run away from the attack.

I defend myself against an attack
if the person probably attacked me to achieve a higher-level goal (such as robbery)
and I convince the person to achieve the higher-level goal
in an alternative way that is less harmful to me.

I could use forward reasoning to simulate each of the alternative options and then choose the alternative that has highest expected utility[2], or I could use heuristic rules to choose between the alternatives instead. This kind of hypothetical reasoning might be viewed as a system 2 way of processing the candidate actions generated by system 1 or system 2.

Computational Logic as a Wide-Spectrum Computer Language

The example highlights the logical side of computational logic. However, much of its merit as a model of dual process thinking comes from its computational character.

Viewed as a computational formalism, computational logic is much higher-level than the vast majority of programming languages, and is used instead as a program specification language in software engineering, query language in database systems or knowledge representation language in artificial intelligence. However, partly because it can be used to represent procedural knowledge, including both goal-reduction procedures and condition-action rules, it can also be used as a lower-level programming language. This ability to represent both high level and low level computation contributes to its ability to serve as a dual process model of thought.

In computing, the highest level languages lie on the border between artificial and natural languages. Here, controlled natural languages [16, 17], in particular, can be executed by translating them into symbolic logic and using an automated theorem-prover. Many of these languages are translated into the logic programming sublanguage of logic and are executed either directly in Prolog or indirectly by a meta-interpreter [18] written in Prolog.

In computing, high-level representations are generally easier to develop and maintain, but low-level programs are normally more efficient. In high-level representations, goals,

---

[2] Such a combination of ALP and Decision Theory has been developed by David Poole in his Independent Choice Logic [20].

beliefs and procedures are represented explicitly. In low-level representations, goals, if they exist at all, are generally only implicit and emergent.

As a result of these differences, high-level representations are normally developed first. If direct execution at the high level is too inefficient, they are compiled into a lower-level language or transformed into a more efficient representation in the same high-level language. The process of compilation/transformation can be iterated any number of levels, down to the level of hardware.

In the case of logic programming, high-level programs can be transformed [15] by applying backward and forward reasoning in advance. The resulting transformed programs are more efficient than the original programs, because much of the reasoning that would need to be performed over and over when needed has been performed once and for all beforehand. In ALP, this technique can be used to transform high-level maintenance goals and beliefs into lower-level condition-action rules.

In computing, high-level and low-level representations are used in tandem. High-level representations are developed and then compiled/transformed to a more efficient, lower level program, for routine problem-solving tasks. If the lower-level program fails, because it is incorrect or incomplete, or if it needs to be changed to provide enhanced functionality, then the higher-level representation is changed instead. After the change, the new high-level representation is recompiled/transformed into the lower level.

Alternatively, programs are sometimes implemented directly at a low-level, without the intervention of any higher-level representation. These low-level programs might be legacy systems developed in the early days of computing, before the advent of higher-level languages. Or they might be adaptive, self-programming systems in domains where high-level abstractions do not exist, as is the case with some systems developed using genetic algorithms or artificial neural networks.

Nonetheless, even in this alternative case, it is often possible to decompile the lower-level programs and reformulate them in higher-level representations, for the sake of greater intelligibility and so that they can more easily be changed. In many cases, however, the lower-level program can only be approximated by a higher-level representation. This can happen when the higher-level representation is written in a highly structured language and the lower-level program is written in an unstructured manner.

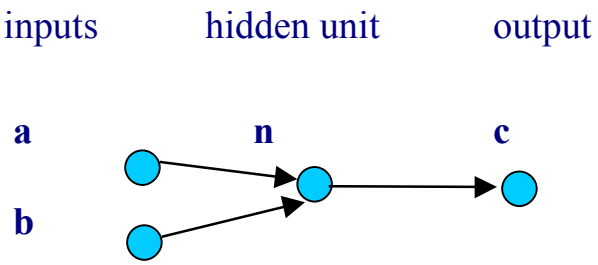
In computing, the lowest-level languages lie on the border between software and hardware. Indeed, the very borderline itself is blurred by the use of micro-programmable hardware, which can be programmed to emulate higher-level machines, and reconfigurable hardware, which can be tailored to implement a specific task, such as image processing or pattern matching, as quickly as a dedicated machine.

In the analogy between software and the mind, computer hardware is analogous to the neural networks of the brain.

Logic programs and artificial neural networks

A number of authors have investigated the relationship between logic programs and artificial neural networks. Starting with [12], Holldobler, Kalinke and their colleagues [13] showed how to compile any logic program with negation as failure into a recurrent neural network. Conversely, Poole et al [11] and D'Avila Garcez et al [19] show how to decompile neural networks into logic programs.

D'Avila Garcez et al, for example, consider the network:



where the units, **a**, **b**, **n** have weights +5, – 5, 1 respectively; the hidden node **n** and output node **c** both have threshold .5; and the activation function **f** is the standard sigmoid function that coerces the real numbers into the range [0,1].

The representation of Poole et al [11] simulates such a network by a Horn clause program:

> ***n*** *holds with strength* S_n
> *if* ***a*** *holds with strength* S_a
> *and* ***b*** *holds with strength* S_b
> *and* $S_n = \boldsymbol{f}(5S_a - 5S_b - .5)$
>
> ***c*** *holds with strength* S_c
> *if* ***n*** *holds with strength* S_n
> *and* $S_c = \boldsymbol{f}(S_n - .5)$

Here, the variables S_a, S_b, S_n, S_c represent the activation values of the units, **a**, **b**, **n**, **c**, respectively and all lie in the range [0,1], where 0 is associated with the Boolean value false and 1 with true.

The Horn clause program, augmented with an appropriate definition of the activation function, is an exact representation of the neural network. Moreover, forward reasoning with the logic program simulates forward execution of the network. Both the logic program and the neural network are low-level representations, in which weights and hidden units do not have a high-level conceptual interpretation.

In contrast with Poole et al, D'Avila Garcez et al decompile the same network, under the assumption that the strengths of the inputs are either 0 or 1, into a logic program with negation as failure:

c *holds if* ***a*** *holds and* ***b*** *does not hold.*

More generally, they show how to decompile any regular neural network into a higher-level, purely qualitative representation as an extended logic program with both explicit negation and negation as failure.

Conclusion

Compared with other dual process theories, computational logic is more formal, more computational and more logical. It combines system 1 processing at the neural network level, with condition-action rules (at the interface between systems 1 and 2) and system 2 processing at the level of maintenance goals, goal-reduction, and simulation of candidate actions. It also provides systematic transformation, compilation, and decompilation techniques, which relate one level of thinking to another.

References

1. Kakas, A.C., Sadri, F. (Eds.): Computational Logic: Logic Programming and Beyond. Springer Verlag. LNAI 2407 and LNAI 2408 (2002)

2. Prakken, H., Sartor, G.: The Role of Logic in Computational Models of Legal Argument: A Critical Survey. In Kakas, A.C., Sadri, F. (Eds.): Computational Logic: Logic Programming and Beyond. Springer Verlag (2002)

3. Evans, J.: In two minds: dual process accounts of reasoning. Trends in Cognitive Science, 7, (2003) 454-459

4. Kakas, A.C.., Kowalski, R., Toni, F.: The Role of Logic Programming in Abduction. In: Gabbay, D., Hogger, C.J., Robinson, J.A. (eds.): Handbook of Logic in Artificial Intelligence and Programming 5. Oxford University Press (1998) 235-324

5. Kowalski, R., Sadri, F.: From Logic Programming towards Multi-agent Systems. Annals of Mathematics and Artificial Intelligence. Vol. 25 (1999) 391-419

6. Laird, J.E , Newell, A., Rosenbloom, P.S.: SOAR: An architecture for general intelligence. Artificial Intellilgence. 33:11, (1987) 1-64

7. Anderson, J., Lebiere, C.: The atomic components of thought. NJ: Erlbaum (1998)

8. Stenning, K., van Lambalgen, M.: Semantic Interpretation as Computation in Nonmonotonic Logic: The Real Meaning of the Suppression Task. Cognitive Science: A Multidisciplinary Journal, Vol. 29, No. 6: pages 919-960.

9. Stenning, K., van Lambalgen, M.: Human Reasoning and Cognitive Science MIT Press (To be published 2006)

10. Byrne R. M.: Suppressing valid inferences with conditionals. Cognition. (1989)

11. Poole, D.L., Mackworth, A.K., Goebel, R.: Computational intelligence: a logical approach. Oxford University Press (1998)

12. Holldobler, S. Kalinke, Y.: Toward a new massively parallel computational model for logic programming. In *Proceedings of the Workshop on Combining Symbolic and Connectionist Processing, ECAI 94*, (1994) 68-77

13. Hitzler, P., Holldobler, S., Seda, A.K.: Logic programs and connectionist networks. Journal of Applied Logic (2004)

14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. New York: Prentice Hall (1993)

15. Pettorossi, A., Proietti, M.: Program Derivation= Rules+ Strategies. In Kakas, A.C., Sadri, F. (Eds.): Computational Logic: Logic Programming and Beyond. Springer Verlag.; Vol. 2407 (2002)

16. Fuchs, N., Schwitter, R.: Attempto Controlled English (ACE) CLAW 96, Proceedings of The First International Workshop on Controlled Language Applications, Katholieke Universiteit (1996)

17. Schwitter, R.: http://www.ics.mq.edu.au/~rolfs/controlled-natural-languages/

18. Costantini, S.: Meta-reasoning: A Survey. In Kakas, A.C., Sadri, F. (Eds.): Computational Logic: Logic Programming and Beyond. Springer Verlag. (2002)

19. d'Avila Garcez, A.S., Broda, K., Gabbay, D.M.: Symbolic knowledge extraction from trained neural networks: A sound approach. Artificial Intelligence 125 (2001) 155–207

20. Poole, D.L.: The independent choice logic for modeling multiple agents under uncertainty. Artificial Intelligence. Vol. 94 (1997) 7-56