Toward Aligning Computer Programming with Clear Thinking via the Reason Programming Language[*]

Selmer Bringsjord & Jinrong Li
Department of Cognitive Science
Department of Computer Science
Rensselaer AI & Reasoning Laboratory
Rensselaer Polytechnic Institute (RPI)
Troy NY 12180 USA
Contact Author Email: `selmer@rpi.edu`
Web site: http://www.rpi.edu/∼brings

110207_0138_HI_USA

Abstract

Logic has long set itself the task of helping humans think clearly. Certain computer programming languages, most prominently the Logo language, have been billed as helping young people become clearer thinkers. It is somewhat doubtful that such languages can succeed in this regard, but at any rate it seems sensible to explore an approach to programming that guarantees an intimate link between the thinking required to program, and the kind of clear thinking that logic has historically sought to cultivate. Accordingly, I have invented a new computer programming language, Reason, one firmly based in the declarative programming paradigm, and specifically aligned with the core skills constituting clear thinking. Reason thus offers the intimate link in question to all who would genuinely use it.

Contents

[*]Selmer is greatly indebted to those who provided insightful comments and objections at E-CAP 2007. More generally, Bringsjord wishes to thank all those who made this conference possible, since the theme driving it (viz., the confluence of computing and philosophy) stands at the heart of Reason.

1 Introduction

Logic has long seen itself as the field seeking to foster clear thinking in human persons. This can of course be readily confirmed by consulting any comprehensive introduction to logic, which invariably offer rationales such as that by study of logic one will be less likely to be hoodwinked by fallacious reasoning.[1] Put in terms of a key distinction that goes back 300 years BC to Aristotle, logic has been viewed as a *prescriptive*, rather than a *descriptive* enterprise: a field charged with explaining, in detail, what representations, and processes over these representations, *ought* to be followed by those aspiring to be clear thinkers.[2] For people with this ambition, logic enables declarative, specifically propositional, content to be expressed in syntactically and semantically rigorous ways, at which point the field can then also specify methods of reasoning to be applied to this formalized content, in order to allow agents to know more, to know why he or she knows more, and to be able to share this knowledge with others.

Unfortunately, only a fraction of people study logic; this is true even if the population in question is restricted to the civilized world. With few exceptions across the globe, pre-college mathematics curricula avoid logic, and the traditional introduction to formal logic is first available, 99 times out of 100, to first-year college students as an elective. Even those majoring in mathematics or philosophy can often obtain these degrees without having to take a formal logic course. On the other hand, at least across the technologized world, computer programming *is* quite often introduced to young students.[3] And not only that, but certain computer programming languages, most prominently the Logo language, have long been billed as helping young people become clearer thinkers. It is somewhat doubtful that such languages can succeed in this regard (for reasons to be briefly discussed below), but at any rate, it seems sensible to explore an approach to programming that *guarantees* an intimate link between the thinking required to program, and the kind of clear thinking that logic has historically sought to cultivate. Accordingly, Bringsjord has invented a new computer programming language standing at the nexus of computing and philosophy: Reason, a language firmly based in the logic-based programming paradigm, and thus one offering the intimate link in question to all who would genuinely use it.

The plan of this paper is as follows. The next section (2) is a barbarically quick introduction to elementary logic, given to ensure that the present paper will be understandable to readers from fields other than logic and philosophy. In section 3 we define what I mean by 'clear thinking,' and provide two so-called *logical illusions* (Johnson-Laird, Legrenzi, Girotto & Legrenzi 2000, Bringsjord & Yang 2003): that is, two examples of difficult problems which those capable of such thinking should be able to answer correctly (at least after they have Reason at their disposal). The next section (4) is devoted to a brief discussion of Logo, and Prolog and logic programming. Logo and Prolog are essential to understanding the motivation for creating Reason. In section 5, Reason itself is introduced in action, as it's used to solve the two problems posed in section 3. The paper ends with brief section on the future of Reason.

[1]See, e.g., the extensive discussion of fallacies in (Copi & Cohen 1997). See also (Barwise & Etchemendy 1999), which has the added benefit of setting out logic in a way designed to reveal it's connection to computer science — a connection central to Reason.

[2]Aristotle's work on logic, devoted to setting out the theory of the syllogism, can be found in his *Organon*, the collection of his logical treatises. This collection, and Aristotle's other main writings, are available in (McKeon 1941). For a nice discussion of how Aristotelean logic is relevant to modern-day logic-based artificial intelligence, see (Glymour 1992).

[3]In the United States, there is an Advanced Placement exam available to any high school student seeking college credit for demonstrated competence in (Java) programming.

2 Elementary Logic in a Nutshell

In logic, declarative statements, or propositions, are represented by formulas in one or more logics, and these logics provide precise machinery for carrying out reasoning. The simplest logics that have provided sufficient raw material for building corresponding programming languages are the propositional calculus, and the predicate calculus (or first-order logic, or just FOL); together, these this pair comprises what is generally called elementary logic. I proceed now to give a very short review of how knowledge is represented and reasoned over in these logics.

In the case of both of these systems, and indeed in general when it comes to any logic, three main components are required: one is purely syntactic, one is semantic, and one is metatheoretical in nature. The syntactic component includes specification of the alphabet of a given logical system, the grammar for building well-formed formulas (wffs) from this alphabet, and, more importantly, a proof theory that precisely describes how and when one formula can be inferred from a set of formulas. The semantic component includes a precise account of the conditions under which a formula in a given system is true or false. The metatheoretical component includes theorems, conjectures, and hypotheses concerning the syntactic component, the semantic component, and connections between them. In this paper, we focus on the syntactic side of things. Thorough but refreshingly economical coverage of the formal semantics and metatheory of elementary logic can be found in (Ebbinghaus, Flum & Thomas 1994).

As to the alphabet for propositional logic, it's simply an infinite list

$$p_1, p_2, \ldots, p_n, p_{n+1}, \ldots$$

of propositional variables (according to tradition p_1 is p, p_2 is q, and p_3 is r), and the five familiar truth-functional connectives $\neg, \rightarrow, \leftrightarrow, \wedge, \vee$. The connectives can at least provisionally be read, respectively, as 'not,' 'implies' (or 'if then '), 'if and only if,' 'and,' and 'or.' Given this alphabet, we can construct formulas that carry a considerable amount of information. For example, to say that 'if Alvin loves Bill, then Bill loves Alvin, and so does Katherine' we could write

$$a_l \rightarrow (b_l \wedge k_l)$$

where the propositional variables denote the individuals involved in the obvious way. These propositional variables, as you can see, are each used to represent declarative statements.

We move up to first-order logic when we allow the quantifiers $\exists x$ ('there exists at least one thing x such that ...') and $\forall x$ ('for all x ...'); the first is known as the *existential* quantifier, and the second as the *universal*. We also allow a supply of variables, constants, relations, and function symbols. Using this machinery, the proposition that 'Everyone loves anyone who loves someone' is represented as

$$\forall x \forall y (\exists z Loves(y, z) \rightarrow Loves(x, y))$$

But how does one go about reasoning over these sorts of formulas? This question is answered below when we begin to turn to programs that anticipate those in Reason, and then to Reason programs themselves. In both cases, computer programs are fundamentally chains of deductive reasoning.

But what about a careful account of the *meaning* of formulae in the propositional and predicate calculi?

The precise meaning of the five truth-functional connectives of the propositional calculus is given via truth-tables, which tell us what the value of a statement is, given the truth-values of its components. The simplest truth-table is that for negation, which informs us, unsurprisingly, that if ϕ is T (= TRUE) then $\neg\phi$ is F (= FALSE; see first row below double lines), and if ϕ is F then $\neg\phi$ is T (second row).

ϕ	$\neg\phi$
T	F
F	T

Here are the remaining truth-tables.

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	ψ	$\phi \leftrightarrow \psi$
T	T	T
T	F	F
F	T	F
F	F	T

Notice that the truth-table for disjunction says that when both disjuncts are true, the entire disjunction is true. This is called *inclusive* disjunction. In *exclusive* disjunction, it's one disjunct or another, but not both. This distinction becomes particularly important if one is attempting to symbolize parts of English (or any other *natural language*). It would not do to represent the sentence

George will either win or lose.

as

$$W \vee L,$$

because under the English meaning there is no way both possibilities can be true, whereas by the meaning of $\vee$ it would be possible that W and L are *both* true. One could use $\vee_x$ to denote *exclusive disjunction*, which can be defined through the following truth-table.

ϕ	ψ	$\phi \vee_x \psi$
T	T	F
T	F	T
F	T	T
F	F	F

Given a truth-value assignment v (i.e., an assignment of T or F to each propositional variable p_i), one can say that v "makes true" or "models" or "satisfies" a given formula ϕ; this is standardly written

$$v \models \phi.$$

A formula such that there is some model that satisfies it is said to be *satisfiable.* A formula that cannot be true on any model (e.g., $p \wedge \neg p$) is said to be *unsatisfiable.* Some formulas are true on all models. For example, the formula $((p \vee q) \wedge \neg q) \rightarrow p$ is in this category. Such formulas are said to be *valid* and are sometimes referred to as *validities.* To indicate that a formula ϕ is valid we write

$$\models \phi.$$

Another important semantic notion is *consequence.* An individual formula ϕ is said to be a consequence of a set Φ of formulas provided that all the truth-value assignments on which all of Φ are true is also one on which ϕ is true; this is customarily written

$$\Phi \models \phi.$$

The final concept in the semantic component of the propositional calculus is the concept of consistency: we say that a set Φ of formulas is *semantically consistent* if and only if there is a truth-value assignment on which all of Φ are true. As a check of understanding, the reader may want to satisfy herself that the conjunction of formulas taken from a semantically consistent set must be satisfiable.

And now, what about the semantic side of first-order logic?

Unfortunately, the formal semantics of FOL gets quite a bit more tricky than the truth table-based scheme sufficient for the propositional level. The central concept is that in FOL formulas are said to be true (or false) on *models*; that some formula ϕ is true on a model is often written as $\mathcal{M} \models \phi$. (This is often read, "$\mathcal{I}$ satisfies, or models, ϕ.") For example, the formula $\forall x \exists y Gyx$ might mean, on the standard model for arithmetic, that for every natural number n, there is a natural number m such that $m > n$. In this case, the *domain* is the set of natural numbers, that is, $\mathbf{N}$; and G symbolizes 'greater than.' Much more could of course be said about the formal semantics (or *model theory*) for FOL — but this is an advanced topic beyond the scope of the present, brief treatment. For a fuller but still-succinct discussion using the traditional notation of model theory see (Ebbinghaus et al. 1994).

3 What is Clear Thinking?

The concept of clear thinking, at least to a significant degree, can be operationally defined with help from psychology of reasoning; specifically with help from, first, a distinction between two empirically confirmed modes of reasoning: context-dependent reasoning, versus context-*in*dependent reasoning; and, two, from a particular class of stimuli used in experiments to show that exceedingly few people can engage in the latter mode. The class of stimuli are what have been called *logical illusions*. We now proceed to explain the distinction and the class.

3.1 Context-Dependent v. Context-Indepedent Reasoning

In an wide-ranging paper in *Behavioral and Brain Sciences* that draws upon empirical data accumulated over more than half a century, Stanovich & West (2000) explain that there are two dichotomous systems for thinking at play in the human mind: what they call System 1 and System 2. Reasoning performed on the basis of System 1 thinking is bound to concrete contexts and is prone to error; reasoning on the basis of System 2 cognition "abstracts complex situations into canonical representations that are stripped of context" (Stanovich & West 2000, p. 662), and when such reasoning is mastered, the human is armed with powerful techniques that can be used to handle the increasingly abstract challenges of the modern, symbol-driven marketplace. System 1 reasoning is context-dependent, and System 2 reasoning is context-independent. We now explain the difference in more detail.

Psychologists have devised many tasks to illuminate the distinction between these two modes of reasoning (without always realizing, it must be granted, that that was what they were doing). One such problem is the Wason Selection Task (Wason 1966), which runs as follows. Suppose that you are dealt four cards out of a larger deck, where each card in the deck has a digit from 1 to 9 on one side, and a capital Roman letter on the other. Here is what appears to you when the four cards are dealt out on a table in front of you:

$$\boxed{\text{E}} \quad \boxed{\text{K}} \quad \boxed{4} \quad \boxed{7}$$

Now, your task is to pick just the card or cards you would turn over to try your best at determining whether the following rule is true:

(R_1) If a card has a vowel on one side, then it has an even number on the other side.

Less than 5% of the educated adult population can solve this problem (but, predictably, trained mathematicians and logicians are rarely fooled) This result has been repeatedly replicated over the past 15 years, with subjects ranging from 7th grade students to illustrious members of the Academy; see (Bringsjord, Bringsjord & Noel 1998). About 30% of subjects do turn over the E card, but that isn't enough: the 7 card must be turned over as well. The reason why is as follows. The rule in question is a so-called **conditional** in formal logic, that is, a proposition having an if-then form, which is often symbolized as $\phi \rightarrow \psi$, where the Greek letters here are variables ranging over formulas from some logical system. As the truth-tables routinely taught to young pre-12 math students make clear (e.g., see Chapter 1 of Bumby, Klutch, Collins & Egbers 1995), a conditional is false if and only if its antecedent, ϕ, is true, while its consequent, ψ, is false; it's true in the remaining three permutations. So, if the E card has an odd number on the other side, (R_1) is overthrown. However, if the 7 card has a vowel on the other side, this too would be a case sufficient to refute (R_1). The other cards are entirely irrelevant, and flipping them serves no purpose whatsoever, and is thus profligate.

This is the abstract, context-independent version of the task. But now let's see what happens when some context-*de*pendent reasoning is triggered in you, for there is incontrovertible evidence that *if the task in question is concretized*, System 1 reasoning can get the job done (Ashcraft 1994). For example, suppose one changes rule (R_1) to this rule:

(R_2) If an envelope is sealed for mailing, it must carry a 20 cent stamp on it.

And now suppose one presents four envelopes to you (keeping in mind that these envelopes, like our cards, have a front and back, only one side of which will be visible if the envelopes are "dealt" out onto a table in front of you), viz.,

sealed envelope	unsealed envelope	env. w/ 20 cent stamp	env. w/ 15 cent stamp

Suppose as well that you are told something analogous to what subjects were told in the abstract version of the task, namely, that they should turn over just those envelopes needed to check whether (R_2) is being followed. Suddenly the results are quite different: Most subjects choose the sealed envelope (to see if it has a 20 cent stamp on the other side), *and* this time they choose the envelope with the 15 cent stamp (to see if it is sealed for mailing).

3.2 The King-Ace Problem

Now we come to a logical illusion, the King-Ace Problem. As we present the problem, it's a slight variant[4] of a puzzle introduced by Johnson-Laird (1997). Here it is:

> Assume that the following is true:
>
> 'If there is a king in the hand, then there is an ace in the hand,' or 'If there is not a king in the hand, then there is an ace in the hand,' — but not both of these if-thens are true.
>
> What can you infer from this assumption? Please provide a careful justification for your answer.

Your are encouraged to record your own answer. We return to this problem later, when using Reason to solve it. But please note that the correct answer to the problem is not 'There is an ace in the hand,' but rather the (counterintuitive!) proposition that there *isn't* an ace in the hand. If Reason is on the right track, use of it will help students see that this is the right answer.

[4]The variation arises from disambiguating Johnson-Laird's 's or else s'' as 'either s or s', but not both.'

3.3 The Wine Drinker Problem

Now let us consider a second logical illusion, an interesting puzzle devised by Johnson-Laird & Savary (1995) that has the same general form as Aristotle's syllogisms:

> Suppose:
>
> - All the Frenchmen in the restaurant are gourmets.
> - Some of the gourmets are wine drinkers.
>
> Does it follow that some of the Frenchmen are wine drinkers? Please provide a careful justification for your answer.

We will return to this problem later, when using Reason to solve it. But note for now that the correct answer is 'No.' Reason will itself provide a justification for this negative answer.

4 The Logo Programming Language; Logic Programming

When youth learn to program by using Logo,[5] by far the programming language most used in the States to teach programming in grades 6–12, almost without exception, they produce instructions designed drive a turtle through some sequence of states. For example, the procedure

```
to square
repeat 4 [forward 50 right 90]
end
```

causes the turtle to draw a square. In a second, more sophisticated mode of programming, the Logo programmer can process lists in ways generally similar to those available to the Lisp programmer. Ever since a seminal paper by Black, Swan & Schwartz (1988), it has been known that while students who program in the second way do seem to thereby develop some clearer thinking skills, the improvement is quite slight, the cognitive distance from processing lists to better logical reasoning is great, and hence *transfer* from the first activity to the second is very problematic.[6] In an intelligent reaction to this transfer challenge, Black et al. make a move that is quite interesting from the perspective of our own objective, and the language Bringsjord has built to meet it: viz., they consider whether teaching Prolog[7] might be a better strategy for cultivating in those who learn it a significant gain in clear thinking. Unfortunately, there are five fatal problems plaguing the narrow logic programming paradigm of which Prolog is a concretization. Here's the quintet, each member of which, as shall soon be seen, is overcome by Reason:

1. Logic programming is based on a fragment of full first-order logic: its inexpressive. Human reasoning, as is well-known, not only encompasses full first-order logic (and hence on this score alone exceeds Prolog), but also modal logic, deontic logic, and so on.
2. Logic programming is "lazy." By this we mean that the programmer doesn't herself construct an argument or proof; nor for that matter does she create a model or countermodel.
3. While you can issue queries in Prolog, all you can get back are assignments to variables, not the proofs that justify these assignments.

[5]http://el.media.mit.eduLogo-foundation/logo/programming.html

[6]In particular, it turns out that making a transition from "plug-and-chug" mathematics to being able to produce proofs is a very difficult one for students to achieve. See (Moore 1994). For further negative data in the case of Logo, see, e.g., (Louden 1989, **?**).

[7]There is of course insufficient space to provide a tutorial on Prolog. We assume readers to be familiar with at least the fundamentals. A classic introduction to Prolog is (Clocksin & Mellish 2003).

4. In addition, Prolog can't return models or counter-models.

5. Finally, as to deductive reasoning, Prolog locks those who program in it into the rule of inference known as *resolution*.[8] Resolution is not used by humans in the business of carrying out clear deductive thinking. Logic and mathematics, instead, are carried out in what is called *natural deduction* (which is why this is the form of deduction almost invariably taught in philosophy and mathematics).

5 The Reason Programming Language

5.1 Reason in the Context of the Four Paradigms of Computer Programming

There are four programming paradigms: *procedural*, reflected, e.g., in Turing machines themselves, and in various "minimalist" languages like those seen in foundational computer science texts (e.g., Davis & Weyuker 1994, Pascal, etc.); *functional*, reflected, e.g., in Scheme, ML, and purely functional Common Lisp (Shapiro 1992, Abelson & Sussman 1996); *object-oriented*; and declarative, reflected, albeit weakly, in Prolog (Clocksin & Mellish 2003). Reason is in, but is an extension of, the declarative paradigm.[9]

Reason programs are specifically extensions and generalizations of the long-established concept of a *logic program* in computer science (succinctly presented, e.g., in the chapter "Logic Programming" in Ebbinghaus et al. 1994).

5.2 Proofs as Programs through a Simple Denotational Proof Language

In order to introduce Reason itself, we first introduce the syntax within it currently used to allow the programmer to build purported proofs, and to then evaluate these proofs to see if they produce the output (i.e., the desired theorem). This syntax is based on the easy-to-understand type-α denotational proof language NDL invented by Konstantine Arkoudas (for background see Arkoudas 2000, Bringsjord, Arkoudas & Bello 2006) that corresponds for the most part to systems of Fitch-style natural deduction often taught in logic and philosophy. Fitch-style natural deduction was first presented in 1934 by two thinkers working independently to offer a format designed to capture human mathematical reasoning as it was and is expressed by real human beings: Gentzen (1935) and Jaskowski (1934). Streamlining of the formalism was carried out by Fitch (1952). The hallmark of this sort of deduction is that assumptions are made (and then discharged) in order to allow reasoning of the sort that human reasoners engage in.

Now here is a simple deduction in NDL, commented to make it easy to follow. This deduction, upon evaluation, produces a theorem that Newell and Simon's Logic Theorist, to great fanfare (because here was a machine doing what "smart" humans did), was able to muster at the dawn of AI in 1956, at the original Dartmouth AI conference.

```
// Here is the theorem to be proved,
// Logic Theorist's ``claim to fame'':
// (p ==> q) ==> (~q ==> ~p)
```

[8] All of resolution can essentially be collapsed into the one rule that from $p \vee q$ and $\neg p$ one can infer q.

[9] As is well known, in theory, any Turing-computable function can be implemented through code written in any Turing-complete programming language. There is nothing in principle precluding the possibility of writing a program in assembly language that, at a higher level of abstraction, processes information in accordance with inference in many of the logical systems that Reason allows its programmers to work in. (In fact, as is well-known, the other direction is routine, as it occurs when a high-level computer program in, say, Prolog, is compiled to produce corresponding to low-level code; assembly language, for example.) However, the mindset of a programmer working in some particular programming language that falls into one of the four paradigms is clearly the focus of the present discussion, and Turing-completeness can safely be left aside.

```
Relations p:0, q:0. // Here we declare that we have two
                    // propositional variables, p and q.
                    // They are defined as 0-ary relations.

// Now for the argument.  First, the antecedent (p ==> q)
// is assumed, and then, for contradiction, the antecedent
// (~q) of the consequent (~q ==> ~p).
assume p ==> q
   assume ~q
      suppose-absurd p
          begin
            modus-ponens p ==> q, p;
            absurd q, ~q
          end
```

If, upon evaluation, the desired theorem is produced, the program is successful. In the present case, sure enough, after the code is evaluated, one receives this back:

```
Theorem: (p ==> q) ==> (~q ==> ~p)
```

Now let us move up to programs written in first-order logic, by introducing quantification. As you will recall, this entails that we now have at our disposal the quantifiers $\exists x$ ('there exists at least one thing x such that ...') and $\forall x$ ('for all x ...'). In addition, there is now a supply of variables, constants, relations, and function symbols; these were discussed above. What follows is a simple NDL deduction at the level of first-order logic that illuminates a number of the concepts introduced to this point. The code in this case, upon evaluation, yields the theorem that Tom loves Mary, given certain helpful information. It is important to note that both the answer and the justification have been assembled, and that the justification, since it is natural deduction, corresponds to the kinds of arguments often given by human beings.

```
Constants mary, tom.  // Two constants announced.

Relations Loves:2. // This concludes the simple signature, which
                   // here declares Loves to be a two-place relation.

// That Mary loves Tom is asserted:
assert Loves(mary, tom).

// 'Loves' is a symmetric relation, and this is asserted:
assert (forall x (forall y (Loves(x, y) ==> Loves(y, x)))).

//Now the evaluable deduction proper can be written:
suppose-absurd ~Loves(tom, mary)
   begin
      specialize (forall x (forall y (Loves(x, y) ==> Loves(y, x)))) with mary;
      specialize (forall y (Loves(mary, y) ==> Loves(y, mary))) with tom;
      Loves(tom,mary) BY modus-ponens Loves(mary, tom) ==> Loves(tom, mary), Loves(mary, tom);
end;
Loves(tom,mary) BY double-negation ~~Loves(tom,mary)
```

When this program is evaluated, one receives the desired result back: `Theorem: Loves(tom,mary)`. Once again, it is important to note that both the answer and the justification have been assembled, and that the justification, since it is natural deduction, corresponds to the kinds of proofs often given by human beings.

So far we have conceived of programs as proof-like entities. But what about the semantic side? What about models? Moving beyond NDL, in Reason, programs can be written to produce, and to manipulate, models. In addition, while in NDL the full cognitive burden is borne by the programmer, Reason can be queried about whether certain claims are provable. In addition, in Reason, the programmer can set the degree to which the system is intelligent on a session-by-session basis. This last property of Reason gives rise to the concept that the system can be set to be "oracular" at a certain level. That is, Reason can function as an oracle up to a pre-set limit. One common limit is propositional inference, and the idea here is to allow Reason to be able to prove on its own anything that requires only reasoning at the level of the propositional calculus.

5.3 Cracking King-Ace and Wine Drinker with Reason

5.3.1 Cracking the King-Ace Problem with Reason

In this example, the selected logic to be used with Reason is standard first-order logic as described to above, with the specifics that reasoning is deductive and Fitch-style. The system is assumed to have oracular power at the level of propositional reasoning, that is, the programmer can ask Reason itself to prove things as long as only reasoning at the level of the propositional calculus is requested. This request is signified by use of `prop`.

To save space, we assume that the programmer has made these selections through prior interaction with Reason. Now, given the following two propositions, is there an ace in the hand? Or is it the other way around?

F1 If there is a king in the hand, then there is an ace in the hand; or: if there isn't a king in the hand, then there is an ace in the hand.

F2 Not both of the if-thens in F1 are true.

In this case, we want to write a Reason program that produces the correct answer, which is "There is not an ace in the hand." We also want to obtain certification of a proof of this answer as additional output from our program.

We can obtain what we want by first declaring our symbol set, which in this case consists in simply declaring two propositional variables, `K` (for 'There is a king in the hand') and `A` (for 'There is an ace in the hand'). Next, in order to establish what is known, we present the facts to Reason. Note that Reason responds by saying that the relevant things are known, and added to a knowledge base (`KB1`).

```
> (known F1 KB1 (or (if K A) (if (not K) A)))
F1 KNOWN
F1 ADDED TO KB1

> (known F2 KB1
   (not (and (if K A) (if (not K) A))))

F2 KNOWN
F2 ADDED TO KB1
```

Next, we present the following partial proof to Reason. (It's a *partial* proof because the system itself is called upon to infer that the negation of a conditional entails a conjunction of the antecedent and the negated consequent. More precisely, from `(not (if P Q))` it follows that `(and P (not Q))`.)

```
(proof P1 KB1
  demorgan F2;
  assume (not (if K A))
    begin
      (and K (not A)) by prop on (not (if K A));
      right-and K, (not A)
    end
  assume (not (if (not K) A))
    begin
      (and (not K) (not A) by prop on (not (if (not K) A)));
      right-and (not K), (not A)
    end
  proof-by-cases (or (not (if K A)) (not (if (not K) A))),
                     (if (not (if K A)) (not A)),
                     (if (not (if (not K) A)) (not A)))
```

When this proof is evaluated, Reason responds with:

```
PROOF P1 VERIFIED
ADDITIONAL KNOWNS ADDED TO KB1:
THEOREM: (not A)
```

I make the perhaps not unreasonable claim that anyone who takes the time to construct and evaluate this program (or for that matter any reader who takes the time to study it carefully to see why `(not A)` is provable) doesn't succumb to the logical illusion in question any longer. Now we can proceed to issue an additional query:

```
> (provable? A)

NO
DISPLAY-COUNTERMODEL OFF
```

If the flag for countermodeling was on, Reason would display a truth-table showing that `A` can be false while F1 and F2 are true.

5.3.2 Cracking the Wine Drinker Problem with Reason

Recall the three relevant statements, in English:

F3 All the Frenchmen in the restaurant are gourmets.

F4 Some of the gourmets are wine drinkers.

F5 Some of the Frenchmen in the restaurant are wine drinkers.

To speed the exposition, let us assume that the Reason programmer has asserted these into knowledgebase `KB2`, using the expected infix syntax of first-order logic, so that, for example, F3 becomes

```
(forall x (if (Frenchman x) (Gourmet x)))
```

In addition, let us suppose that Reason can once again operate in oracular fashion at the level of propositional reasoning, that the flag for displaying countermodels has been activated, and that we have introduced the constants `object-1` and `object-2` to Reason for this session. Given this, please study the following interaction.

```
> (proof P2 KB2
    begin
      assume (and (Frenchman object-1) (Gourmet object-1) (not (Wine-drinker object-1)));
      assume (and (Gourmet object-2) (In-restaurant object-2) (Wine-drinker object-2));
      not-provable (F3 F4 F5) => (some x (and (In-restaurant x) (Wine-drinker x)))
    end)
PROOF P2 VERIFIED
DISPLAY-COUNTERMODEL?

> Y
```

In the situation the programmer has imagined, all Frenchmen are gourmets, and there exists someone who is a wine-drinker and a gourmet. This ensures that both the first two statements are true. But it's *not* true that there exists someone who is both a Frenchman and a wine drinker. This means that F5 is false; more generally, it means that F5 isn't a deductive consequence of the conjunction of F3 and F4.

When countermodels are rendered in visual form, they can be more quickly grasped. Figure 1 shows such a countermodel relevant to the present case.

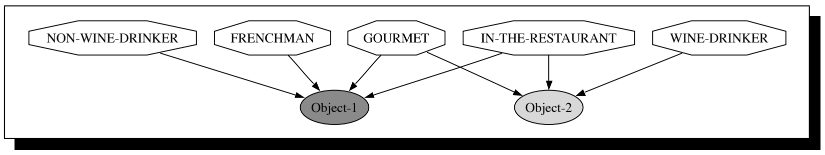

Figure 1: Visual Countermodel in Wine Drinker Puzzle (provided via a grammar and program written by Andrew Shilliday and Joshua Taylor that Reason can call)

6 The Future

The full and fully stable implementation of Reason is not yet complete. Fortunately, this implementation is not far off, as it is aided by the fact that this implementation is to a high degree meta-programming over computational building blocks that have been provided by others.[10] For example, resolution-based deduction is computed by the automated theorem provers Vampire (Voronkov 1995) and Otter (Wos 1996, Wos, Overbeek, e. Lusk & Boyle 1992) (and others as well), while natural deduction-style automated theorem proving is provided by the Oscar system invented by the philosopher John Pollock (Pollock 1989, Pollock 1995). As to automated model finding for first-order logic, a number of mature, readily available systems now compute this relation as well, for example Paradox and Mace (Claessen & Sorensson 2003). At the propositional level, truth-value assignments are automatically found by many SAT solvers (e.g., see Kautz & Selman 1999).

[10]For a discussion of meta-programming in the logic programming field, see (Bringsjord & Ferrucci 1998).

While it seems sensible to strive for teaching clear thinking via programming languages that, by their very nature, are more intimately connected to the formalisms and processes that (from the perspective of logic, anyway) constitute clear thinking, noting this is not sufficient, obviously. One needs to empirically test determinate hypotheses. We need, specifically, to test the hypothesis that students who learn to program in Reason will as a result show themselves to be able, to a higher degree, to solve the kind of problems that are resistant to context-dependent reasoning. Accordingly, empirical studies of the sort we have carried out for other systems (e.g., Rinella, Bringsjord & Yang 2001, Bringsjord et al. 1998) are being planned for Reason.

References

Abelson, H. & Sussman, G. (1996), *Structure and Interpretation of Computer Programs (2nd Edition)*, MIT Press, Cambridge, MA.

Arkoudas, K. (2000), Denotational Proof Languages, PhD thesis, MIT.

Ashcraft, M. (1994), *Human Memory and Cognition*, HarperCollins, New York, NY.

Barwise, J. & Etchemendy, J. (1999), *Language, Proof, and Logic*, Seven Bridges, New York, NY.

Black, J., Swan, K. & Schwartz, D. (1988), 'Developing thinking skills with computers', *Teachers College Record* **89**(3), 384–407.

Bringsjord, S., Arkoudas, K. & Bello, P. (2006), 'Toward a general logicist methodology for engineering ethically correct robots', *IEEE Intelligent Systems* **21**(4), 38–44.

Bringsjord, S., Bringsjord, E. & Noel, R. (1998), In defense of logical minds, *in* 'Proceedings of the 20^{th} Annual Conference of the Cognitive Science Society', Lawrence Erlbaum, Mahwah, NJ, pp. 173–178.

Bringsjord, S. & Ferrucci, D. (1998), 'Logic and artificial intelligence: Divorced, still married, separated...?', *Minds and Machines* **8**, 273–308.

Bringsjord, S. & Yang, Y. (2003), Logical illusions and the welcome psychologism of logicist artificial intelligence, *in* D. Jacquette, ed., 'Philosophy, Psychology, and Psychologism: Critical and Historical Essays on the Psychological Turn in Philosophy', Kluwer, Dordrecht, The Netherlands, pp. 289–312.

Bumby, Klutch, Collins & Egbers (1995), *Integrated Mathematics Course 1*, Glencoe/McGraw Hill, New York, NY.

Claessen, K. & Sorensson, N. (2003), New techniques that improve Mace-style model finding, *in* 'Model Computation: Principles, Algorithms, Applications (Cade-19 Workshop)', Miami, Florida.

Clocksin, W. & Mellish, C. (2003), *Programming in Prolog (Using the ISO Standard; 5th Edition)*, Springer, New York, NY.

Copi, I. & Cohen, C. (1997), *Introduction to Logic*, Prentice-Hall, Englewood Cliffs, NJ. This is the tenth edition of the book.

Davis, M., Sigal, R. & Weyuker, E. (1994), *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, Academic Press, New York, NY.

Ebbinghaus, H. D., Flum, J. & Thomas, W. (1994), *Mathematical Logic (second edition)*, Springer-Verlag, New York, NY.

Fitch, F. (1952), *Symbolic Logic: An Introduction*, Ronald Press, New York, NY.

Gentzen, G. (1935), 'Untersuchungen über das logische Schlieben I', *Mathematische Zeitschrift* **39**, 176–210.

Glymour, C. (1992), *Thinking Things Through*, MIT Press, Cambridge, MA.

Jaskowski, S. (1934), 'On the rules of suppositions in formal logic', *Studia Logica* **1**.

Johnson-Laird, P. (1997), 'Rules and illusions: A criticial study of Rips's *The Psychology of Proof*', *Minds and Machines* **7**(3), 387–407.

Johnson-Laird, P. N., Legrenzi, P., Girotto, V. & Legrenzi, M. S. (2000), 'Illusions in reasoning about consistency', *Science* **288**, 531–532.

Johnson-Laird, P. & Savary, F. (1995), How to make the impossible seem probable, *in* M. Gaskell & W. Marslen-Wilson, eds, 'Proceedings of the 17th Annual Conference of the Cognitive Science Society', Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 381–384.

Kautz, H. & Selman, B. (1999), Unifying SAT-based and graph-based planning, *in* J. Minker, ed., 'Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14– 16, 1999', Computer Science Department, University of Maryland, College Park, Maryland.
URL: *citeseer.ist.psu.edu/kautz99unifying.html*

Louden, K. (1989), 'Logo as a prelude to lisp: Some surprising results', *ACM SIGCSE Bulletin* **21**(3).

McKeon, R., ed. (1941), *The Basic Works of Aristotle*, Random House, New York, NY.

Moore, R. C. (1994), 'Making the transition to formal proof', *Educational Studies in Mathematics* **27.3**, 249–266.

Pea, R. D. (1987), 'Logo programming and problem solving: Children's experiences with logic'.

Pollock, J. (1989), *How to Build a Person: A Prolegomenon*, MIT Press, Cambridge, MA.

Pollock, J. (1995), *Cognitive Carpentry: A Blueprint for How to Build a Person*, MIT Press, Cambridge, MA.

Rinella, K., Bringsjord, S. & Yang, Y. (2001), Efficacious logic instruction: People are not irremediably poor deductive reasoners, *in* J. D. Moore & K. Stenning, eds, 'Proceedings of the Twenty-Third Annual Conference of the Cognitive Science Society', Lawrence Erlbaum Associates, Mahwah, NJ, pp. 851–856.

Shapiro, S. (1992), *Common Lisp: An Interactive Approach*, W. H. Freeman, New York, NY.

Stanovich, K. E. & West, R. F. (2000), 'Individual differences in reasoning: Implications for the rationality debate', *Behavioral and Brain Sciences* **23**(5), 645–665.

Voronkov, A. (1995), 'The anatomy of vampire: Implementing bottom-up procedures with code trees', *Journal of Automated Reasoning* **15**(2).

Wason, P. (1966), Reasoning, *in* 'New Horizons in Psychology', Penguin, Hammondsworth, UK.

Wos, L. (1996), *The Automation of Reasoning: An Experimenter's Notebook with* OTTER *Tutorial*, Academic Press, San Diego, CA.

Wos, L., Overbeek, R., e. Lusk & Boyle, J. (1992), *Automated Reasoning: Introduction and Applications*, McGraw Hill, New York, NY.