Reasoning in Attempto Controlled English: Mathematical and Functional Extensions

Norbert E. Fuchs
Department of Computational Linguistics
University of Zurich
fuchs@ifi.uzh.ch
http://attempto.ifi.uzh.ch

Abstract

RACE is a first-order reasoner for Attempto Controlled English (ACE). This paper introduces mathematical and functional extensions. It is the third system description of RACE, and also the final one since RACE now covers all ACE constructs that have a representation in first-order logic.

1 Introduction

Attempto Controlled English (ACE)[1] is a logic-based knowledge representation language that uses the syntax of a subset of English. The Attempto Reasoner RACE[2] allows users to show the consistency of an ACE text, to deduce one ACE text from another one, and to answer ACE queries from an ACE text.

Two previous system descriptions (Fuchs, 2012; Fuchs, 2016) of RACE detailed its structure, its functionality, its implementation and its user interfaces, material that will repeated here only to the extent to make this paper self-contained.

This is the third – and final – system description of RACE intended to complete its coverage of ACE. Concretely, RACE has been extended to reason with ACE's mathematical and functional constructs. The mathematical extension offers primarily solutions for arithmetic problems and linear equations. The functional extension allows ACE to directly access Prolog predicates which has a number of important consequences, for example the ability to express recursive algorithms – previously not available – and to operate on ACE's list, set and string constructs.

To avoid a possible misunderstanding, this is not a venture of ACE/RACE into the field of mathematics per se, as realised in the project Naproche[3], or as outlined in this report[4].

Section 2 of this paper recalls general features of RACE. Section 3 presents the mathematical extension. Section 4 motivates and introduces the functional extension. Section 5 concludes with a summary of the presented extensions and with a discussion of their strengths and limitations.

2 General Features of RACE

For the convenience of the reader and to make this paper self-contained the material of this section is partially copied from (Fuchs 2012).

RACE has the following general features:

- RACE offers consistency checking, textual entailment and query answering of ACE texts.
- RACE does not presuppose knowledge of formal logic or theorem proving, does not require users to understand RACE's internal workings, nor does it require users to control the reasoning process.
- All input of RACE is in ACE, all output is in ACE and English.
- Consistency checking: For inconsistent

[1] http://attempto.ifi.uzh.ch/
[2] http://attempto.ifi.uzh.ch/race/
[3] https://korpora-exp.zim.uni-duisburg-essen.de/naproche
[4] https://jiggerwit.files.wordpress.com/2019/06/header.pdf

ACE axioms RACE will list all minimal inconsistent subsets of the axioms.

- Textual entailment and query answering: If the ACE axioms entail the ACE theorems, respectively ACE queries, RACE will list all minimal subsets of the axioms that entail the theorems, respectively queries. Furthermore, there will be substitutions for all occurring query words.
- RACE uses about 100 auxiliary axioms – not expressed in ACE, but in Prolog using ACE's internal representation – to provide domain-independent general knowledge. In spite of their large number, auxiliary axioms have little impact on RACE's performance since they are only called individually and only when needed.
- RACE is implemented as a set of Prolog programs that can be used locally. Furthermore, RACE can be accessed remotely via its web-client[5] or via its web-service[6].

3 Reasoning with Arithmetic, Linear Equations and Quadratic Equations

Reasoning with positive integers that occur as determiners (*2 apples*, *at most 3 apples*) and with positive integers and reals that occur in measurement nouns (*2.25 l of water*) was described in a previous system description (Fuchs, 2012). Not covered, however, was until now reasoning with ACE's arithmetical constructs, which is part of the present system description.

Here is a brief summary of ACE's arithmetical constructs.

ACE offers numbers that syntactically act as nouns. Numbers are positive and negative integers and positive and negative reals. Furthermore, there are arithmetic expressions (*(X ^ 3) ^ 1/2 – 4*Pi*) built with the help of the operators +, –, *, /, ^ from numbers, variables, proper names and parenthesised subexpressions. Arithmetic expressions can evaluate to numbers and thus count as nouns.

ACE's boolean formulas (*X >= 13.4 and X < 20.*) are built from numbers, arithmetic expressions, proper names and variables with the help of the comparison operators =[7], \=, >, >=, < and =<. Boolean formulas syntactically act as sentences.

When reasoning with arithmetic expressions and formulas one encounters four new phenomena.

- While previously RACE fundamentally relied on unification, i.e. on the syntactic matching of logical atoms, numerical expressions – like those in the formula *100/50 + 8 = 4 + 6* – cannot simply be unified but must be numerically evaluated before being tested.
- While previously the order of processing did not matter, the evaluation of expressions – as in *A is B + C. C is D - 1. B is 2. D is 3.* – must be delayed until all constituents have a value.
- Even after evaluation remain problems of relating formulas, as can be seen in the deduction *X=1 |– X>0.*
- As in standard logic, arithmetical contradictions can involve negation, as for instance in *A is 1. A is not 1.* But there are new forms of contradictions not involving negation, for example *A is 1. A is 2.* or simply *1=2.*

While it is possible to solve the problems associated with these phenomena in RACE's implementation language SWI Prolog, this would amount to duplicating functionality that is available off-the-shelf in the form of mathematical frameworks. Because it efficiently copes with the four phenomena above, because it does not require changes of ACE's syntax, because it is highly efficient, and because of its simple integration, RACE uses SWI Prolog's library clpqr[8] that provides constraint logic programming over rationals and reals, and also logical entailment.

Following are three simple examples that show a range of possible applications of RACE's mathematical extension.

A Banking Problem. A capital *C* is invested in a bank at an interest rate *I* for the duration of *D* years while the bank charges a yearly fee *F*. Then the approximate final balance is

*C *(1+I)^D - F*(1+I)^(D - 1) - F(1+I)*(D - 1)*

disregarding higher powers of *I* in the last term to get a closed expression. This leads to a small

[5] http://attempto.ifi.uzh.ch/race/

[6] http://attempto.ifi.uzh.ch/ws/race/racews.perl

[7] Note: RACE accepts the copula *is* as a synonym for the comparison operator =.

[8] https://www.swi-prolog.org/man/clpqr.html

error that will be ignored here. Given the ACE axioms

The capital C is 1000.00.
The interest I is 0.005.
The yearly fee F is 12.00.
The duration D is 10.
*The balance of the account is C *(1+I)^D - F*(1+I)^(D - 1) - F*(1+I)*(D - 1).*

and the ACE query

What is the balance of the account?

RACE arrives at the balance of 930.05.

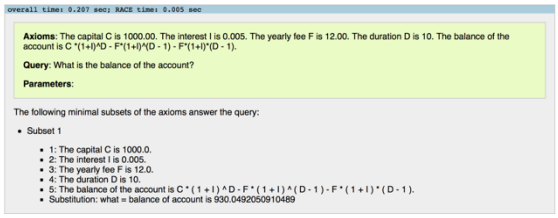

Figure 1: A Banking Problem

The result (Figure 1) is presented as a screenshot of the output window of RACE's web-interface. This window contains the ACE axioms, the ACE query, the subset of the axioms needed to answer the query and the substitution of the query word *what*, i.e. the actual numerical result. The entry "parameters" is used for testing.

A Word Problem. Many word problems are dressed-up arithmetic problems. Here is an example:

A farmer has some cows and some ducks. Altogether he has 100 animals with 260 feet. How many cows and how many ducks does the farmer have?

To solve this word problem, one needs some background information, namely

Every cow is an animal. Every duck is an animal. No cow is a duck. Every cow has 4 feet. Every duck has 2 feet.

Even with this background information the problem cannot yet be solved because a problem-solving strategy is needed. Schwitter (2012) demonstrated how such a strategy could be devised for the Marathon Puzzle that should determine the arrival order of a group of runners. Schwitter's strategy consists of formulating the puzzle in the controlled natural language PENG, and then translating the PENG text into an answer set program (ASP) that is submitted to an ASP solver. Though Schwitter's strategy is elegant and efficient, it is specific to the Marathon Puzzle and cannot be immediately generalised.

Instead, I suggest for word problems that are hidden arithmetic problems a strategy that is perhaps less elegant, but efficient and more general, namely to manually derive from the text the – often linear – equations, thereby taking into account the explicit or implicit background information.

Here is the ACE version of the farmer-cow-duck problem, expressed as three axioms and two queries. Note that the axioms implicitly incorporate the complete background information.

A farmer has a number X of some cows and has a number Y of some ducks.

X+Y=100.

*4*X+2*Y=260.*

/-

What is a number of some cows? What is a number of some ducks?

Submitting these axioms and queries to RACE we get the result (Figure 2) that the farmer has 70 ducks and 30 cows. Note that by design the two queries are answered separately since there are different substitutions of the two occurrences of the query word *what*.

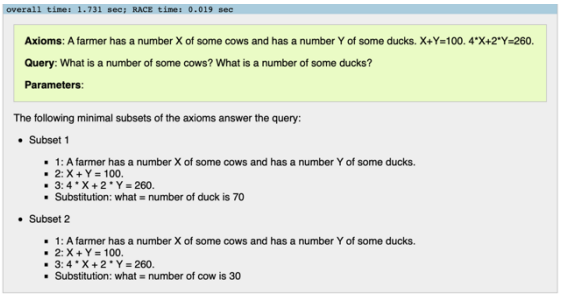

Figure 2: Farmer, Cows and Ducks

Quadratic Equations. Quadratic equations occur in many problems[9], specifically in physics and geometry.

[9] https://en.wikipedia.org/wiki/Quadratic_equation

Unfortunately, clpqr cannot solve quadratic equations directly. However, replacing the quadratic equation *X^2 + P * X + Q = 0* by its two solutions *X = -P/2 ± √ (P^2/4 - Q)* eliminates this stumbling block.

As an example, here is the quadratic equation for the ubiquitous Golden Ratio[10]:

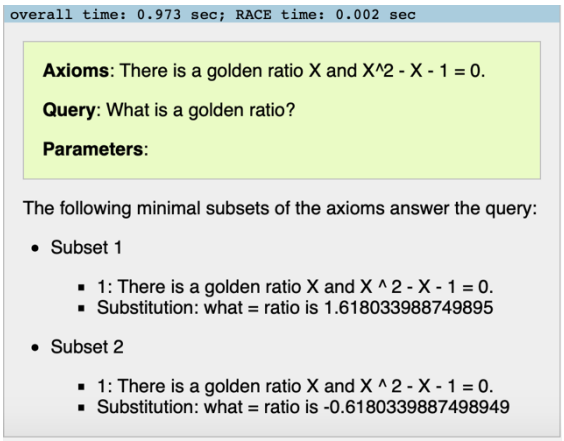

Figure 1: Quadratic Equation for Golden Ratio

Like most quadratic equations this one has two solutions (Figure 3). Only the positive solution pertains to the golden ratio.

The current implementation for quadratic equations has the following syntactic restrictions: The quadratic term has no coefficient, the terms P and Q are integers, integer fractions or reals. Since ACE does not know complex numbers, quadratic equations with complex solutions are flagged with an error message.

Replacing quadratic equations by their solutions solves an important problem, yet in a way that cannot easily be generalised. A mathematical framework more powerful than clpqr could possibly avoid this problem.

4 Extending RACE by a Functional Notation

For most of its intended applications ACE does not need a functional notation, and hence does not provide one. RACE, however, needs a functional notation to extend its reasoning capabilities. Exploiting ACE's list structure, I devised a functional notation in the form *["functor", argument1, argument2, ...]* that allows RACE to call the Prolog predicate *functor(argument1, argument2, ...)*. Note that the functor is expressed as a string to be accepted by the ACE parser without having an entry in ACE's lexicon.

This simple extension has three beneficial consequences. RACE can

- call any built-in or user-defined Prolog predicate,
- make use of recursive algorithms that cannot be implemented otherwise,
- operate on ACE's list, set and string constructs.

Here is a simple example of list operations using three built-in Prolog predicates combined with RACE's arithmetic:

There is a list L of ["append", [1,2,3], [4,5,6], L]
and there is a maximum M1 of ["max_list", L, M1]
and there is a minimum M2 of ["min_list", L, M2]
and there is a result R and R = M1 + M2.

/-

What is a result?

Submitting this example to RACE we get 7 as the result (Figure 4).

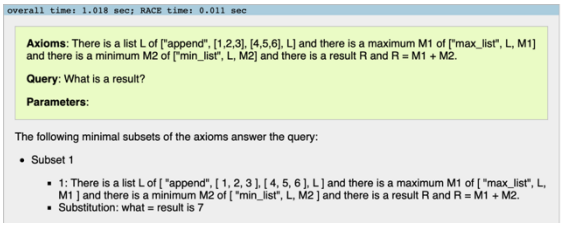

Figure 4: Using Prolog's List Predicates

Figure 5 shows an example of the user-defined Prolog predicate *gcd/3* that implements Euclid's recursive algorithm for the greatest common divisor. In this case a function name is explicitly introduced.

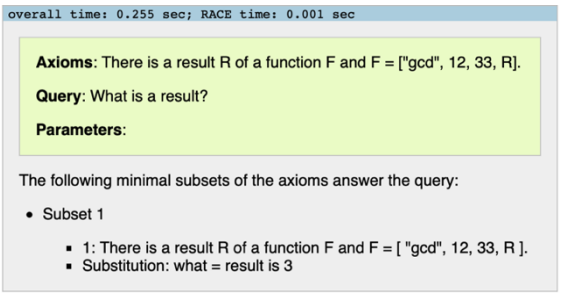

Figure 5: Euclid's Recursive GCD Algorithm

[10] https://en.wikipedia.org/wiki/Golden_ratio

Besides the three beneficial consequences listed above, the functional notation has another practical application. Understanding Prolog programs can pose problems for people not familiar with this language, as it happened recently in the context of a psychology project (private communication, 2021). The psychologist in question had developed a set of Prolog programs to solve statistical problems, but could not help to notice the lack of understanding of his colleagues. The functional notation would have enabled him to incorporate calls to these Prolog programs into ACE texts that his colleagues could understand without knowing much of Prolog.

5 Conclusions

I added to the Attempto Reasoner RACE mathematical and functional extensions and illustrated them by small examples.

RACE's mathematical extension does not raise any questions besides being limited to some extent by the restrictions of the chosen mathematical framework clpqr.

RACE's functional extension, however, raises two issues. First issue: Though the list notation is accepted ACE syntax, the explicit calls of Prolog predicates in an ACE text could be considered a violation of the intention of the Attempto project to hide formality from its users. Second issue: RACE relies on auxiliary Prolog axioms that add domain-independent general knowledge to the domain-specific knowledge of the given ACE axioms (Fuchs, 2012). This reliance on Prolog is increased by the functional extension that allows us to directly call Prolog predicates. Since Prolog has the power of the Turing machine, RACE could in principle deduce any conclusion from the axioms. As a consequence, besides the usual questions of the correctness and the completeness of the reasoning process a further question arises, namely the relevance. What should RACE actually deduce? For instance, RACE's auxiliary axioms enable the deduction *John's cat purrs.* |- *John has a cat.* Is this deduction – that has no logical justification, but is based solely on common sense – acceptable? The answer depends not only on the application domain, but also on the expectations and intuitions of the users, and this – as my experience has amply shown – may be highly debatable.

RACE now covers all language constructs of ACE with the exception of imperative sentences that do not play a role in logical deduction, and two constructs that have no – or at least no generally accepted – logical representation, namely the unconventional modal operators for recommendation (*should*) and admissibility (*may*) that were provisionally introduced into ACE to cover the medical jargon of Clinical Practice Guidelines (Shiffman et al., 2009).

Acknowledgments

Many thanks go to my colleague Uta Schwertel for her essential contributions in the initial phase of the development of RACE. The constructive comments of the reviewers of a previous version of this paper are gratefully accepted. Finally, I would like to thank the Department of Computational Linguistics, University of Zurich, for its hospitality.

References

Norbert E. Fuchs. 2012. *First-Order Reasoning for Attempto Controlled English.* In *Proc. of the Second International Workshop on Controlled Natural Language (CNL 2010)*. Marettimo, Italy.

Norbert E. Fuchs. 2016. *Reasoning in Attempto Controlled English: Non-monotonicity.* In *Proc. of the Fifth International Workshop on Controlled Natural Language (CNL 2016)*. Aberdeen, UK.

Richard N. Shiffman, George Michel, Michael Krauthammer, Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. 2009. *Writing Clinical Practice Guidelines in Controlled Natural Language*. In *Proc. of the Workshop on Controlled Natural Language (CNL 2009),* Marettimo, Italy.

Rolf Schwitter. 2012. *Answer Set Programming via Controlled Natural Language Processing*. In *Proc. Third International Workshop on Controlled Natural Language (CNL 2012)*. Marettimo, Italy.