ON THE RAPID PROTOTYPING OF A LOGICAL AGENT

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria

JOHANNES KEPLER UNIVERSITY LINZ

An Experiment

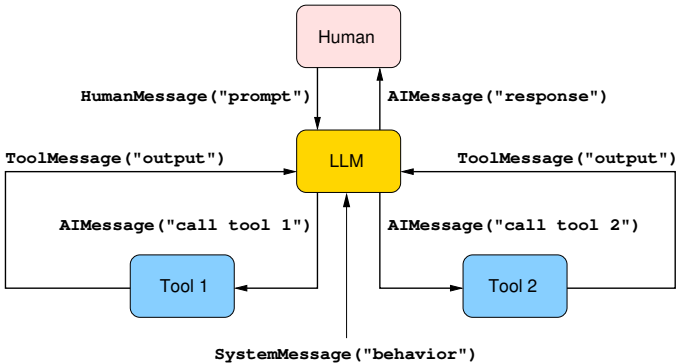
Build a “logical agent” that combines an LLM with a prover.

- Both kinds of software have complimentary strengths:
  - LLM: natural language understanding (based on statistical token prediction).
    - Interacts with humans, follows their instructions, generates artifacts.
  - Prover: sound reasoning (based on a logical calculus).
    - Only claims validity for true statements, generates checkable proofs.
- Development with minimal effort based on two kinds of artifacts:
  - Text documents (written by a human in natural language):
    - System prompt: specification of agent behavior.
    - Tool description: description of prover interface.
    - Language document: definition of prover language.
  - Python code (mostly generated by “vibe coding” with a bit of human adaptation):
    - Agent framework: integration of LLM with external tools via the “React” pattern.
    - Tool wrapper: interface to external prover and downloader.

A blueprint for the “low-code” integration of LLMs with symbolic software.

Principle: The ReAct Approach

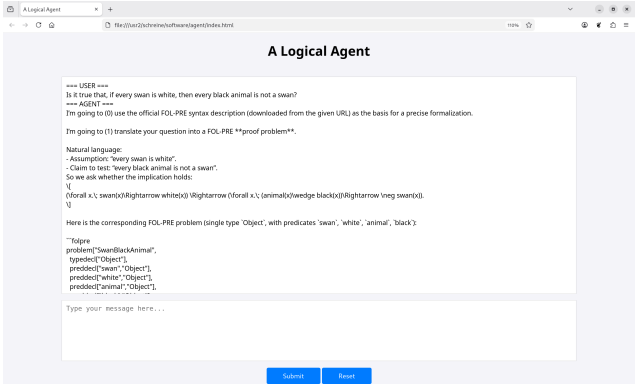
Shunyu Yao, et al: *ReAct: Synergizing Reasoning and Acting in Language Models*, 2022, doi:10.48550/arXiv.2210.03629.



LLM runs in a loop and interacts by messages with external tools and the human.

The Logical Agent: Demo Session

Question: Is it true that, if every swan is white, then no black animal is a swan?



Let the LLM translate the natural language description into a proof problem and have the problem solved by a prover.

The Logical Agent: Demo Session (1)

```
=== USER ===
Is it true that, if every swan is white, then every black animal is not a swan?
=== AGENT ===
I'm going to (0) use the official FOL-PRE syntax description, then (1) translate
your question into a FOL-PRE proof problem, and (2) ask you to confirm the
translation.
## (1) Translation into a FOL-PRE proof problem
Interpretation (tell me if you mean it differently):
- We talk about a single type of objects, **Animal**.
- Predicates:
- ‘Swan(x)‘ means ‘‘x is a swan’’
- ‘White(x)‘ means ‘‘x is white’’
- ‘Black(x)‘ means ‘‘x is black’’
```

The Logical Agent: Demo Session (2)

Your statement: ‘‘If every swan is white, then every black animal is not a swan’’ becomes:
- Assumption: ∀x. Swan(x) ⇒ White(x)
- Conclusion: ∀x. Black(x) ⇒ ¬Swan(x)
- Whole: (∀x. Swan(x) ⇒ White(x)) ⇒ (∀x. Black(x) ⇒ ¬Swan(x))

Here is the corresponding **FOL-PRE** problem:

```
‘‘‘folpre
problem["SwanBlack",
  typedecl["Animal"],
  preddecl["Swan","Animal"],
  preddecl["White","Animal"],
  preddecl["Black","Animal"],
  goal["stmt",
  imp[
    forall["x","Animal",imp[pred["Swan",var["x"]],pred["White",var["x"]]]],
    forall["x","Animal",imp[pred["Black",var["x"]],not[pred["Swan",var["x"]]]]]
  ]]]
‘‘‘
```

The Logical Agent: Demo Session (3)

```
## (2) Is this translation correct?
Two quick checks for meaning:
1. Do you want “black animal” to mean ‘Black(x)‘ (a predicate), as above?
2. Do you assume anything like “black implies not white” (i.e., ∀x. Black(x) ⇒
¬White(x))? I did **not** assume that.
Tell me “yes, correct” or specify changes, and I’ll proceed to (4) run an
automatic prover on the confirmed formalization and show its full output.
=== USER ===
Please add the assumption and then show me the proof problem again.
=== AGENT ===
I’m going to (1) extend the translation by adding the extra assumption you
requested (that black implies not white), and then (2) show you the updated
FOL-PRE proof problem and ask you to confirm it.

## (1) Added assumption
Added axiom:
- ∀x:Animal. Black(x) ⇒ ¬White(x)
```

The Logical Agent: Demo Session (4)

```
## Updated FOL-PRE proof problem
‘‘‘folpre
problem["SwanBlack",
  ...
  axiom["BlackNotWhite",
    forall["x","Animal",
      imp[pred["Black",var["x"]],not[pred["White",var["x"]]]]]],
...
]]]]
‘‘‘
## (2) Is this updated translation correct?
If yes, I will (4) run the automatic prover on exactly this problem and show you
the full prover output.
=== USER ===
Okay, start the proof.
=== AGENT ===
I’m going to (4) call an automatic prover on the confirmed FOL-PRE proof problem
and then present the full prover output.
```

The Logical Agent: Demo Session (5)

```
## (4) Automatic prover result
Proof attempt was **successful**.
Full output produced by the prover:
‘‘‘
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
---------------------------------
...
Goal 3:[stmt.2] ⊤ ∧ ¬Swan(_x§)
...
SUCCESS: the proof problem has been solved (3 clause applications, 13 ms).
...
‘‘‘
I’m going to (5) offer to run an interactive prover session on the *same*
problem so you can inspect the proof steps. Do you want me to start the
interactive prover?
```

Another Logical Question (1)

Consider the following assumptions:

1. If superman were able and willing to prevent evil, he would do so.
2. If superman were unable to prevent evil, he would be impotent.
3. If superman were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.

Is it true, that under these assumptions superman does not exist?

Use in your formulation only predicates, not constants or functions.

Again, from a natural language description...

Another Logical Question (2)

```
problem["superman",
  typedecl["Being"],
  preddecl["S", "Being"], preddecl["A", "Being"], preddecl["W", "Being"],
  preddecl["P", "Being"], preddecl["I", "Being"], preddecl["M", "Being"],
  axiom["a1", forall["x", "Being", imp[pred["S", var["x"]], imp[and[pred["A",
    var["x"]], pred["W", var["x"]]], pred["P", var["x"]]]]]],
  axiom["a2", forall["x", "Being", imp[pred["S", var["x"]], imp[not[pred["A",
    var["x"]]], pred["I", var["x"]]]]]],
  axiom["a3", forall["x", "Being", imp[pred["S", var["x"]], imp[not[pred["W",
    var["x"]]], pred["M", var["x"]]]]]],
  axiom["a4", forall["x", "Being", imp[pred["S", var["x"]], not[pred["P",
    var["x"]]]]]],
  axiom["a5", forall["x", "Being", imp[pred["S", var["x"]], and[not[pred["I",
    var["x"]]], not[pred["M", var["x"]]]]]]],
  goal["g", not[exists["x", "Being", pred["S", var["x"]]]]]
]
```

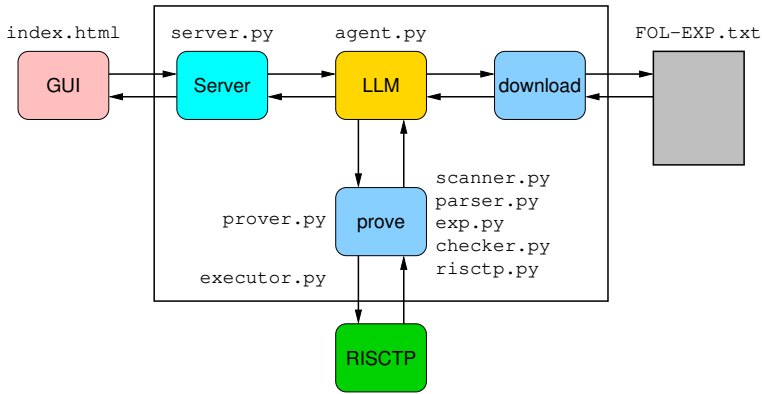
. . . to a formal proof problem . . .

Another Logical Question (3)

```
RISC Theorem Proving Interface 1.8.5 (February 20, 2026)
https://www.risc.jku.at/research/formal/software/RISCTP
(C) 2022-, Research Institute for Symbolic Computation (RISC)
This is free software distributed under the terms of the GNU GPL.
Execute "RISCTP -h" to see the available command line options.
------------------------------------------------------------------
Reading file /tmp/tmpoxddao0y...
=== proof method 'meson': model elimination, subgoal-oriented
Goal 1:[g] ⊤ ∧ ¬S(_x§)
Iteration 1 (proof depth 1)... 12 clause applications.
Iteration 2 (proof depth 2)... 30 clause applications.
Iteration 3 (proof depth 3)... 61 clause applications.
SUCCESS: the proof problem has been solved (61 clause applications, 14 ms).
===
SUCCESS termination (28 ms).
```

. . . that can be solved by a prover.

The Logical Agent: Architecture



LLM (guided by a system prompt) is connected (via the “LangChain” framework) to a “prove” tool and a “download” tool (both documented by tool descriptions).

Text Artifact: The System Prompt

```
You are a helpful assistant that answers logical questions in the following way
(inform the user before every step what you are going to do):
0. You download the description of the FOL-PRE syntax from
   https://www.risc.jku.at/people/schreine/FOL-PRE.txt and read it carefully.
1. You translate the question into a proof problem in the FOL-PRE syntax
   (make sure it is really in this syntax, do not just guess).
2. You ask the user whether this translation is correct.
3. If the translation is not correct, you follow the suggestions of the user
   to come up with a correct translation; then go back to step 2.
4. Once the translation is correct, you call an automatic prover with the proof
   problem and present its result (state whether the proof attempt was successful
   and show the full output produced by the prover).
5. Finally you also offer to to call an interactive prover on the problem
   for investigating the successful proof or the unsuccessful proof attempt.
   Call the interactive prover with the same translation that you used
   for the automatic prover.
Do not try to answer a logical question in any other way.
```

Guide the LLM to only perform the language translation, not the reasoning.

Text Artifact: The Tool Description

```
@tool
def prove(fol_text: str, interactive: bool=False) -> tuple[bool,str]:
    """
    This function applies a prover to a proof problem, either interactively or
    automatically (depending on the value of the second parameter). The first
    parameter is the text of the proof problem in the FOL-PRE syntax
    described at https://www.risc.jku.at/people/schreine/FOL-PRE.txt.
    In the automatic mode, the function returns 'True' if and if the the proof
    could be completed successfully, together with the output produced by the
    prover. In the interactive mode, the function always returns False and
    the empty string (the function blocks until the session has terminated).
    """
    return prover.prove(fol_text, interactive)
```

The prover interface utilizes the formal language FOL-PRE (specifically created for the purpose of the experiment).

Text Artifact: The Language Documentation

```
The Language of Logic in the Notation FOL-PRE:
From Natural Language to Formal Expressions
===========================================
We describe the language of first-order logic (also called "first-order
predicate logic" or just "predicate logic"). This language allows to denote
values and to express statements about values. We describe this language both
formally and informally, namely by giving formal expressions which we
subsequently interpret in natural language. This also demonstrates how we can
conversely translate statements in natural language to formal expressions.

We give the formal expressions in a variant of "prefix notation" that we call "FOL-PRE".
Whenever we refer in the following to "formal expressions", we mean "FOL-PRE" expressions.

As an example, the formal expression

  forall["x","A",imp[fun["p",var["x"]],exists["y","B",fun["q",var["x"],var["y"]]]]]

can be read as "for every value x of type A, if x satisfies property p, then
there exists some value y of type B, such that x is in relation q to y".
...
```

A document with about 1,100 lines of text that describes FOL-PRE in a (systematic but) semi-formal way with examples of natural language translations.

Code Artifacts

- The GUI:
  - `index.html`: the web page (140 lines, 95% vibe coded)
  - `server.py`: the service handler (30 lines, 95% vibe coded)
- The Agent:
  - `agent.py`: the LangChain agent (150 lines, 90% vibe coded)
- FOL-PRE:
  - `exp.py`: the abstract syntax trees (50 lines, 100% vibe coded)
  - `scanner.py`: the scanner (70 lines, 100% vibe coded)
  - `parser.py`: the parser (180 lines, 100% vibe coded)
  - `checker.py`: the type checker (230 lines, 0% vibe coded)
- RISCTP:
  - `risctp.py`: the translator (220 lines, 0% vibe coded)
  - `prover.py`: the prover interface (90 lines, 10% vibe coded)
  - `executor.py`: the process executor (80 lines, 100% vibe coded)

Apart from the type-checker for FOL-PRE and its translation to the actual RISCTP language, most of the code was generated by “vibe coding” (using Google Gemini) with some code adaptations.

Conclusions

- Off-the-shelf (commercial) LLMs worked surprisingly well.
  - OpenAI GPT, Google Gemini, Anthropic Claude.
  - Mostly (not always) followed human instructions, generated correct translations, provided helpful suggestions.
- Tool integration was quite simple (for the workflow considered).
  - React pattern implemented via LangChain.
- Vibe coding was able to generate most of the boilerplate code.
  - Web service and client, agent skeleton, abstract syntax tree generation.
- We could mostly focus on writing natural language documents.
  - Specification of agent behavior, tool descriptions, language definitions.

Future: refining workflows (separating syntax/type checking from proving), integrating other tools, using open LLMs, dealing with stateful artifacts, . . .

Wolfgang Schreiner. *Building a Logical Agent with LangChain . . . and Quite Some Vibe Coding*, 2026, doi:10.35011/risc.26-02.