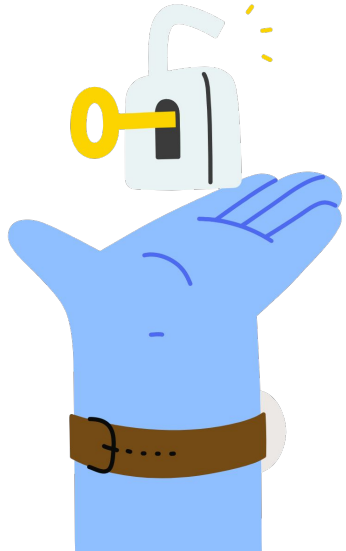
Creation of a modular proof assistant engine for a logic e-tutor

Jakub Dakowski, Aleksandra Draszewska, Barbara Adamska,
Dominika Juszczak, Łukasz Abramowicz, Robert Szymański

Adam Mickiewicz University in Poznań (Poland) · *tiny.one/larch*

Conference on Artificial Intelligence and Theorem Proving 2021



Larch is a ***modular*** system created in the ***POP paradigm*** which ***assists*** students in making ***formal proofs***.

Plugin Oriented Programming (POP)

Plugin Oriented Programming is a new programming paradigm in which the codebase splits into **modular**, **interchangeable** and **independent plugin subsystems** with a **central hub**.

It lays emphasis on **plugins**, **modularity**, **namespaces** and **isolation of development and testing**. Developing large codebases often requires a shift to more modular structure over time – POP allows you to create a project that is modular and pluggable from the very beginning.

Macey, T., Hatch, T. (2019). ***Making Complex Software Fun And Flexible With Plugin Oriented Programming***. Podcast .__init__. https://pythonpodcast.com/plugin-oriented-programming-episode-240/.

Two main groups of *Larch* users

Learners

Contributors

Larch is made in Python

This lowers the entry threshold
and provides better development
opportunities.
Python is also reflective, which is important.

PROBLEM

There is no *pattern matching*
mechanism in Python.

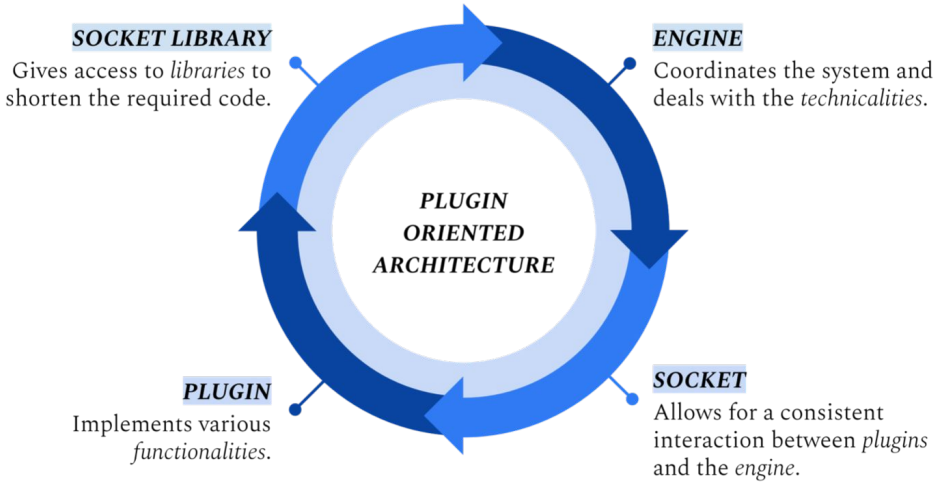
SOLUTIONS

- Sentence class with methods implementing most useful mechanisms.
- Structural Pattern Matching is coming to Python (Moisset, 2020).

Moisset , D. F. (2020). ***PEP 636 – Structural Pattern Matching: Tutorial.*** Python.org. https://www.python.org/dev/peps/pep-0636/.

Implementation of a *POP* system

- Built-in module `importlib` enables the creation of a system inspired by Macey and Hatch (2019).
- Template files are used to define the shape of a socket.
- Exception handling is done mostly by the engine.
- Included libraries allow for a more concise code.



Macey, T., Hatch, T. (2019). ***Making Complex Software Fun And Flexible With Plugin Oriented Programming***. Podcast .__init__. https://pythonpodcast.com/plugin-oriented-programming-episode-240/.

Example of a template file

```python
import typing as tp
import plugins.Output.__utils__ as utils

SOCKET = 'Output'
VERSION = '0.0.1'

def get_readable(sentence: utils.Sentence) -> str:
    """Returns a readable version of the sentence

    :param sentence: Transcribed sentence
    :type sentence: Sentence
    :return: Transcribed string
    :rtype: str
    """
    pass

def write_tree(tree: utils.PrintedProofNode) -> list[str]:
    """
    Returns a tree/table representation of the whole proof
    """
    pass
```

Sockets in Larch

Defining a notation
Lexicon socket

```
Lex = Lexicon()

# Multiple patterns can be assigned to a token
Lex['and'] = 'oraz', 'and', 'r^', '&'
Lex['or'] = 'lub', 'or', '|', 'v'
Lex['imp'] = 'imp', '->'

# Patterns can be activated only in certain conditions
with use_language('predicate'):
    Lex['forall'] = 'forall', '/\\', 'A'
    Lex['exists'] = 'exists', '\\/', 'E'

# Lexicon can generate new tokens. Typically these will be the ones which are used the most.

# The program can however search for unique tokens
with find_new():
    Lex['sentvar'] = RegEx(r'[a-z]')

# Some tokens can be omitted from this mechanism
with no_generation():
    Lex['sentvar'] = RegEx(r'\w+')
```

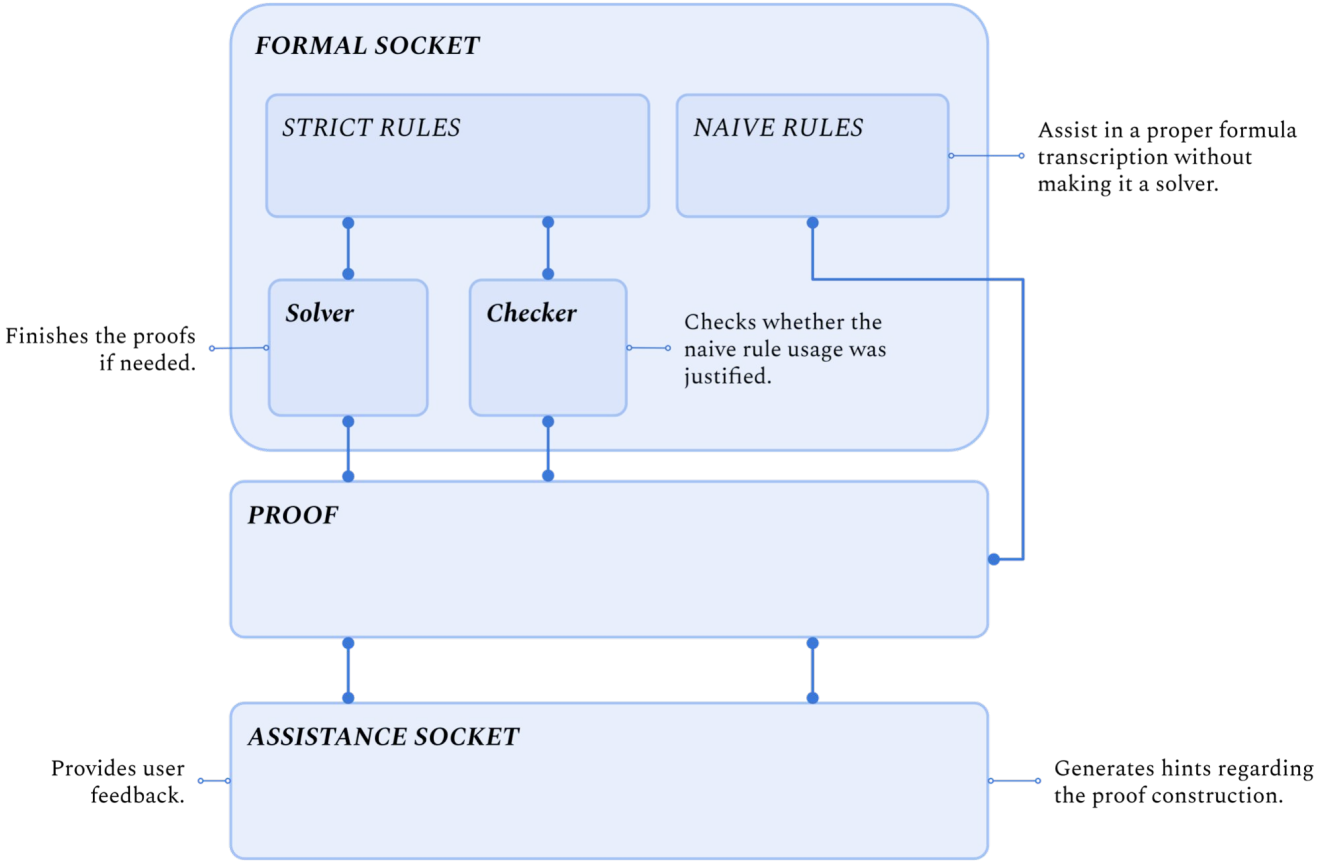
Defining new rules
Formal socket

```
# Rules are objects
double_not = utils.Rule(
    name='double not',
    symbolic="~~A / A",
    docs="Your docs go here",
    reusable=False,
    children=(RULES['true and'],)
)

# Easily connect functions to proper rules
@double_not.setStrict
def strict_doublenot(sentence: Sentence):
    return utils.reduce_prefix(utils.reduce_prefix(utils.empty_creator(sentence), 'not'), 'not')

# Additional premises which need to be entered by the user can be defined
SentenceID = ParameterContext(int, 'sentenceID', 'Sentence Number', 'The number of the sentence in this branch')

@double_not.setNaive
def naive_doublenot(branch: list[Sentence], sentenceID: SentenceID):
    return utils.reduce_prefix(utils.reduce_prefix(utils.empty_creator(branch[sentenceID]), 'not'), 'not')
```


FORMAL SOCKET
STRICT RULES
NAIVE RULES
Assist in a proper formula transcription without making it a solver.
Solver
Checker
Finishes the proofs if needed.
Checks whether the naive rule usage was justified.
PROOF
ASSISTANCE SOCKET
Provides user feedback.
Generates hints regarding the proof construction.

Hint generation and mistake correction

Assistant socket

```python
# Hints

def hint_command(proof: Union[Proof, None]) -> Union[list[str], None]:
    try:
        mistakes = proof.check()
    except EngineError:
        mistakes = []
    if mistakes:
        return mistake_check(mistakes[0])
    moves = proof.copy().solve()
    if moves:
        return [articles['main']['rule'], f"Is it possible to perform any operation on
<code>{moves[0].get_premisses()['sentenceID']}</code>"]
    else:
        return ['The proof is finished!']

# ...

# Mistake correction

def mistake_userule(mistake: UserMistake) -> Union[list[str], None]:
    pass

def mistake_check(mistake: UserMistake) -> Union[list[str], None]:
    pass

def mistake_syntax(mistake: UserMistake) -> Union[list[str], None]:
    pass
```

Discussion

The (near) future of *Larch*

By the end of September we want *Larch* to be **publicly available as a downloadable local web app**. If we succeed, three months later ***Larch* will assist teaching logic** for cognitive science students at our faculty.

This will allow us to **empirically assess the effectiveness** of using *Larch* as an e-tutor for teaching the analytic tableaux method.

It is also important to **improve the hint generation algorithms**. This also will be aided by empirical data.

Achievements

It appears that it is possible to use plugin system to **encompass multiple proof systems in one software**.

Similar methods can be used to create **environments for other applications** both in Logic and in AI.

Currently, both plugins and the engine can be **recycled for other purposes** (such as being *classic* software libraries).

Discussion

Future works

The **limitations of this approach aren't well known**, as this paradigm is just being created. It is worth to check, whether using it affects **application performance** and **security**.

Such design creates a need for **diverse socket libraries** that can be reused by other plugins.

The **socket architecture** also should be improved to allow more elasticity.

The plugin management system could also be **enriched with certain features**.

Vision

In the future this paradigm might encourage the development of **a universal architecture of logic and AI software**.

This might promote **compatibility** between different libraries which would allow **almost seamless interchangeability and code reusability.**

Sources

David Beazley. Writing parsers and compilers with ply. PyCon'07, 2007.

c0fec0de. anytree, Dec 2019.

Christa Cody, Behrooz Mostafavi, and Tiffany Barnes. Investigation of the influence of hint type on problem solving behavior in a logic proof tutor. In International Conference on Artificial Intelligence in Education, pages 58–62. Springer, 2018. https://link.springer.com/chapter/10.1007/978-3-319-93846-2_11.

Cristiano Galafassi, Fabiane FP Galafassi, Eliseo B Reategui, and Rosa M Vicari. Evologic: Intelligent tutoring system to teach logic. In Brazilian Conference on Intelligent Systems, pages 110–121. Springer, 2020. https://link.springer.com/chapter/10.1007/978-3-030-61377-8_8.

Jacob M Howe. Two loop detection mechanisms: a comparison. In International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, pages 188–200. Springer, 1997.

Antonia Huertas. Ten years of computer-based tutors for teaching logic 2000-2010: Lessons learned. In International Congress on Tools for Teaching Logic, pages 131–140. Springer, 2011. https://link.springer.com/chapter/10.1007/978-3-642-21350-2_16.

Tobias Macey and Thomas Hatch. Making complex software fun and flexible with plugin oriented programming. Podcast.__init__, 2019. https://www.pythonpodcast.com/plugin-oriented-programming-episode-240/.

Moisset , D. F. (2020). PEP 636 – Structural Pattern Matching: Tutorial. Python.org. https://www.python.org/dev/peps/pep-0636/.

Raymond M. Smullyan. First-order logic. Dover, 1995.

Adam Tauber. exrex, Jun 2017.