EasyChair Preprint

№ 9588

Building Intelligent Machines: Logic

Maxim Tarasov

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

January 18, 2023

Building Intelligent Machines: Logic

Maxim Tarasov

www.intelligentmachines.io

Abstract. Swift implementation of Pei Wang's Non-Axiomatic Logic. In his 2013 book, Dr. Wang defines intelligence as "the ability for a system to adapt to its environment and to work with insufficient knowledge and resources." [1] The system he describes is called NARS and it is an attempt at creating artificial general intelligence in the framework of a reasoning system. This paper focuses on the logic of intelligence as described in the book, and is augmented with additional functionalities such as pattern matching provided by miniKanren.

Keywords: Non-Axiomatic Logic · AGI · Intelligent Machines.

1 Overview

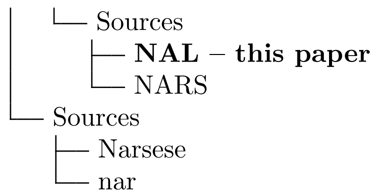
A typical NARS system consists of two parts – logic and control – with the latter dependent on logic. This paper discusses the logic part of the system called NAL and the details of its implementation in Swift [2]. It is assumed that the reader is already familiar with NARS and some of its other implementations [7–11].

In [1], three types of language are used to describe the system: (1) English, used in the book and in this paper to describe the system at the meta-level; (2) Narsese, used for internal representation and external communication; (3) a computer programming language like Java or Prolog, in which the system is implemented. Additionally, it is stated that "these three types of language have different properties, and do not include one another as subsets" [1].

In NARS-Swift [3], we depart from that last notion and embed Narsese in the programming language of the system (Swift) as a DSL or Domain Specific Language, so statements in Swift Narsese dialect are both valid Narsese and valid Swift code. In this way, there are only two languages – the meta-language and the programming language. Our system uses the programming language both for its internal representation and external communication. Section 3 describes the extension to the core system providing the ability to parse strings of text representing Narsese sentences for interoperability and ease of communication with other systems including human users.

1.1 Project structure

NARS-Swift
├── Code.playground



Swift examples from this paper and full project code are available on GitHub [3].

2 Logic

In Narsese, statements represent relations between terms, and inference rules are applied to statements when they share a common term. The simplest type of term is a `word`, a Copula connects two terms to form a `statement`, and you can use a Connector to create a `compound` containing two or more terms (except for certain cases where compounds consist of only one term). In addition to these basic terms, there are also `variable` and `operation` terms.

Basic Narsese terms

$$\langle statement \rangle ::= \langle term \rangle \langle copula \rangle \langle term \rangle$$
$$\langle copula \rangle ::= \rightarrow$$
$$\langle term \rangle ::= \langle word \rangle$$

```swift
enum Term {
    case statement(Term, Copula, Term)
    case symbol(String) // <word>
```

Primary Narsese copulas

$\rightarrow$	Inheritance
$\leftrightarrow$	Similarity
$\Rightarrow$	Implication
$\Leftrightarrow$	Equivalence

```swift
enum Copula {
    case inheritance =  "->"
    case similarity  = "<->"
    case implication =  "=>"
    case equivalence = "<=>"
```

2.1 DSL

Several extensions to the Swift language provide the necessary additions to allow writing Narsese statements like `(bird --> animal)`, which are simultaneously valid Swift code. This Swift Narsese dialect is used in the interface of the system and for internal representation.

```
func --> (s: Term, p: Term) -> { .statement(s, .inheritance, p) }
func <-> (s: Term, p: Term) -> { .statement(s, .similarity,  p) }
func  => (s: Term, p: Term) -> { .statement(s, .implication, p) }
func <=> (s: Term, p: Term) -> { .statement(s, .equivalence, p) }
```

2.2 Rules

Embedding Narsese as a DSL in Swift allows us to demonstrate another benefit of combining these representations. It is now possible to express the inference rules and theorems of NAL directly without any intermediate representations.

Inference rules of Narsese

$J_2 \backslash J_1$	$M \to P \ \langle f_1, c_1 \rangle$	$P \to M \ \langle f_1, c_1 \rangle$
$S \to M \ \langle f_2, c_2 \rangle$	$S \to P \ \langle F_{\text{ded}} \rangle$ $P \to S \ \langle F'_{\text{exe}} \rangle$	$S \to P \ \langle F_{\text{abd}} \rangle$ $P \to S \ \langle F'_{\text{abd}} \rangle$
$M \to S \ \langle f_2, c_2 \rangle$	$S \to P \ \langle F_{\text{ind}} \rangle$ $P \to S \ \langle F'_{\text{ind}} \rangle$	$S \to P \ \langle F_{\text{exe}} \rangle$ $P \to S \ \langle F'_{\text{ded}} \rangle$

```
case .deduction:
    return [(M --> P, S --> M, S --> P, tf),
            (P --> M, M --> S, P --> S, tfi)]
case .induction:
    return [(M --> P, M --> S, S --> P, tf),
            (M --> P, M --> S, P --> S, tfi)]
```

2.3 Inference

During inference, additional extensions transform Narsese into logic terms, and the solver from miniKanren [6] produces a set of substitutions matching the rule's pattern. To accomplish this, the solver uses a form of unification to help decide which rules apply to any two statements. Later, we reverse the process to obtain Narsese statements from logic terms.

miniKanren is a relational programming language designed to be small and embeddable, and in NARS-Swift, we use Dimitri Racordon's implementation [4]. It works by giving the solver a `LogicGoal` for which it returns valid substitutions.

```
extension Term {
    func logic() -> LogicTerm {
        switch self {
        case .symbol:
            return self
        case .statement(let s, let c, let p):
            return List.cons(c, [s, p].toList())

    func from(logic: LogicTerm) -> Term {
        if let term = logic as? Term {
            return term
        }
        if case .cons(let head, let tail) = logic as? List {
            if let copula = head as? Copula { // statement
                let terms = process(list: tail)
                return .statement(terms[0], copula, terms[1])

    func logicReasoning(_ t: Term) -> Term? {
        var result = t

        let g1: LogicGoal = (p1.logic() ≡ j1.statement.logic())
        let g2: LogicGoal = (p2.logic() ≡ j2.statement.logic())

        let substitution = solve(g1 && g2).makeIterator().next()

        for item in substitution {
            result.replace(termName: item.LogicVariable.name,
                                term: .from(logic: item.LogicTerm))
```

3 Parsing

For external communication, it is often convenient to express Narsese as a string of text. While technically not part of the core system, this functionality is highly desirable and it is implemented as part of NARS+ [1], extending the system's capabilities. A third-party library Covfefe [5] by Palle Klewitz translates Narsese grammar defined in Backus-Naur Form (see Fig. 1) into an Abstract Syntax Tree (AST) which we then convert to Narsese data structures.

```
extension Term {
    init(s: String, parser: Narsese) throws {
        let ast = try parser.parse(s)

        func convert(tree: SyntaxTree) throws -> Term {
            switch tree {
            case .node(let key, let children):
                switch key.name {
```

```
case "statement":
    let s = try convert(tree: children[0])
    let p = try convert(tree: children[4])
    let c = String(s[children[2].leafs.first!])
    let copula = Copula(rawValue: c)!
    return .statement(s, copula, p)
case "term":
    return try convert(tree: children.first!)
```

$$
\begin{array}{rcl}
\langle \textit{sentence} \rangle & ::= & \langle \textit{judgment} \rangle \mid \langle \textit{goal} \rangle \mid \langle \textit{question} \rangle \\
\langle \textit{judgment} \rangle & ::= & [\langle \textit{tense} \rangle]\langle \textit{statement} \rangle.\ \langle \textit{truth-value} \rangle \\
\langle \textit{goal} \rangle & ::= & \langle \textit{statement} \rangle!\ \langle \textit{desire-value} \rangle \\
\langle \textit{question} \rangle & ::= & [\langle \textit{tense} \rangle]\langle \textit{statement} \rangle? \mid \langle \textit{statement} \rangle ¿ \\
\langle \textit{statement} \rangle & ::= & (\langle \textit{term} \rangle\ \langle \textit{copula} \rangle\ \langle \textit{term} \rangle) \mid \langle \textit{term} \rangle \\
 & & \mid (\neg\ \langle \textit{statement} \rangle) \\
 & & \mid (\wedge\ \langle \textit{statement} \rangle\ \langle \textit{statement} \rangle^{+}) \\
 & & \mid (\vee\ \langle \textit{statement} \rangle\ \langle \textit{statement} \rangle^{+}) \\
 & & \mid (\ ,\ \langle \textit{statement} \rangle\ \langle \textit{statement} \rangle^{+}) \\
 & & \mid (\ ;\ \langle \textit{statement} \rangle\ \langle \textit{statement} \rangle^{+}) \\
 & & \mid (\Uparrow \langle \textit{word} \rangle\ \langle \textit{term} \rangle^{*}) \\
\langle \textit{copula} \rangle & ::= & \rightarrow \mid \leftrightarrow \mid \Rightarrow \mid \Leftrightarrow \\
 & & \mid \circ\!\!\rightarrow \mid \rightarrow\!\!\circ \mid \circ\!\!\rightarrow\!\!\circ \\
 & & \mid /\!\!\Rightarrow \mid \backslash\!\!\Rightarrow \mid |\!\!\Rightarrow \mid /\!\!\Leftrightarrow \mid |\!\!\Leftrightarrow \\
\langle \textit{tense} \rangle & ::= & /\!\!\Rightarrow \mid \backslash\!\!\Rightarrow \mid |\!\!\Rightarrow \\
\langle \textit{term} \rangle & ::= & \langle \textit{word} \rangle \mid \langle \textit{variable} \rangle \mid \langle \textit{statement} \rangle \\
 & & \mid \{\langle \textit{term} \rangle^{+}\} \mid [\langle \textit{term} \rangle^{+}] \\
 & & \mid (\cap\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^{+}) \\
 & & \mid (\cup\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^{+}) \\
 & & \mid (-\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle) \\
 & & \mid (\ominus\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle) \\
 & & \mid (\times\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^{+}) \\
 & & \mid (/\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^{*} \diamond \langle \textit{term} \rangle^{*}) \\
 & & \mid (\backslash\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^{*} \diamond \langle \textit{term} \rangle^{*}) \\
\langle \textit{variable} \rangle & ::= & \langle \textit{independent-variable} \rangle \\
 & & \mid \langle \textit{dependent-variable} \rangle \\
 & & \mid \langle \textit{query-variable} \rangle \\
\langle \textit{independent-variable} \rangle & ::= & \# \langle \textit{word} \rangle \\
\langle \textit{dependent-variable} \rangle & ::= & \#\,[\langle \textit{word} \rangle(\langle \textit{independent-variable} \rangle^{*})] \\
\langle \textit{query-variable} \rangle & ::= & ?\,[\langle \textit{word} \rangle]
\end{array}
$$

Fig. 1. Narsese grammar.

4 Discussion

In this paper, we described the Swift implementation of Non-Axiomatic Logic, which is just one part of the reasoning system called NARS. The other is the control mechanism, which is the subject of ongoing research. We demonstrated how embedding Narsese in the programming language of the system can simplify the implementation, allowing for near one-to-one correspondence of computer code to the meta-language description of the logic.

NARS-Swift is only one possible implementation of a NARS-like system (some examples in the bibliography), and our primary focus for this project is on simplicity and modularity. Currently, it can handle basic inference, and you can find examples in the project's repository. Additional work is planned[1] to implement all of the inference and meta-rules of NAL, but the basic building blocks are all there.

As already mentioned, the control strategy is the next big open question. You can find the beginnings of one implementation in the repository. Similar to the logic, the control part should be simple and modular. One of the benefits of extracting the logic into a separate module is that we can now experiment with different attention and control strategies and compare them with each other.

[1] Contributors are welcome (see project's GitHub).

References

1. Wang, P.: Non-Axiomatic Logic: A Model Of Intelligent Reasoning. World Scientific Publishing Co. Pte. Ltd. (2013)
2. Swift Programming Language, https://www.swift.org
3. NARS-Swift, https://github.com/maxeeem/NARS-Swift
4. SwiftKanren by Dimitri Racordon, https://github.com/kyouko-taiga/SwiftKanren
5. Covfefe by Palle Klewitz, https://github.com/palle-k/Covfefe
6. miniKanren, http://minikanren.org
7. OpenNARS, https://github.com/opennars/opennars
8. ONA, https://github.com/opennars/OpenNARS-for-Applications
9. ALANN2018, https://github.com/opennars/ALANN2018
10. Narjure, https://github.com/opennars/Narjure
11. NARS-Python, https://github.com/ccrock4t/NARS-Python