Synth Preview 0.1

(Artificial Intelligence Aspect)

table of contents

1. introduction

Synth is intended to be a web based programming language. It is still at conceptual, but mature stage and although it is planned to be full blown universal programming language, its other side will be enabling easy programming of artificial intelligence (AI) applications. An use of AI in the present moment can vary from solving math, physics and chemistry problems to theorem proving and finding paths of solutions to problems explained in universal manner. Other more general uses like mimicking human interaction and interpreting natural language texts to learn new knowledge are more or less still in infancy state and it is yet to be seen what future brings. What we know by now is the knowledge about fragments of intelligent conclusions that build up complete cognitive process of human behavior. Besides support for these fragments, Synth will try to provide enough freedom to build up more autonomous cognitive processes in the future.

In this paper I will try to show which of fragments that build up intelligent conclusions could be handled by Synth. Full specification of Synth will also include all kinds of extensions needed for exposing mentioned AI fragments to end user (by regular formal programming), but those extensions will not be a matter of this paper which focuses just on AI methods.

2. Synth's view of artificial intelligence

Artificial intelligence in Synth is about having different knowledge systems (math, logic, etc.) which provide solutions to its area of problem embracement. These knowledge systems have different syntactic and semantic forms and Synth provides a model for describing these forms.

Syntax and semantics are clearly distinguished in Synth. So, what is syntax and what is semantics from Synth's point of view? Syntax defines which forms of expressions are valid inside specific knowledge system. Semantics say which conclusions follow from which expressions when specific expressions become valid in specific knowledge instance. We can say that semantics define reasoning about specific syntax elements in some knowledge system. Chapter 3. explains Synth's model of syntactic definition for different knowledge systems while semantic reasoning will be covered in chapter 4.

3. syntax: universal language defining model

Part of Synth's model for knowledge base syntax definition will be borrowed from parser technology. Parser technology is used for defining all kind of textual data and is enough general for us to use it as a base for our syntax definition of knowledge. If you are familiar with parsing technology, Synth's parser will interpret a variant of BNF language, which I'll call IBNF (Inline Bacus Naur Form).

Other than duplicating the same explanation of IBNF, I'll invite you to take a look at my product Moony Parser for its syntax definition. The same language (Moony Grammar is an obsolete name for IBNF) will be used in Synth's knowledge syntax definition, with addition of noting semantic relationships. You can freely disregard paragraphs about low-level javascript calls to parser, as they are not relevant to Synth.

4. semantics: reasoning

As we already said, semantics provides reasoning about syntactic elements in knowledge base. In a way, semantics is similar to math formulas, but in more generalized level. It pairs one expression with the other, stating that one expression can be transformed into the other, while preserving some rules about variables' identities. If expressions are seen as

elements of a set then semantics is ought to say: if *this* element belongs to this set so does *that* element (in math example if ‘a’ equals 2 and a set contains ‘a’ then the set also contains 2).

4.1. math solving system

Let’s bring a definition of simple math expression knowledge system. If we want to define an equality notion, together with basic math formulas, then we can open square brackets right after our syntax definition:

```
Math (
    Expression {
        Sum {
            Fact {
                Pow {
                    Expr (
                        {@WhiteSpace | @Null},
                        Sign {'-' | @Null},
                        Value {
                            Num {@Number} |
                            Var {@Variable} |
                            Bra (Left {'('}, In {@Sum}, Right {')'})
                        },
                        {@WhiteSpace | @Null}
                    ) |
                    PowExpr (Left {@Pow}, In {"^"}, Right {@Expr})
                } |
                MulDiv (Left {@Fact}, In {'*' | '/'}, Right {@Pow})
            } |
            AddSub (Left {@Sum}, In {'+' | '-'}, Right {@Fact})
        }
    },
    Formula (Left {@Sum}, "=", Right {@Sum})
    [
        </a:@Sum, b:@Sum/> a = b |= b = a;

        </a:@Expr, b:@Expr, c:@Expr/> |= a * b + a * c = a * (b + c);
        </a:@Expr, b:@Expr/> |= a^2 - b^2 = (a + b) * (a - b);
        </a:@Expr, b:@Expr/> |= (a + b)^2 = a^2 + 2*a*b + b^2;
        ...
    ]
)
```

As we can see, semantics are attached to wanted variables. In previous example we defined some equalities needed for math calculations and stored them under “Equality” variable. The most important part of previous example is **|=** (read “follows”) sign which states that if expression on its left side is a part of some variable then the expression on the right side is also

a part of the same variable. Left side can be omitted, in which case the right side is always inferred.

Subvariables involved in expressions have to be noted between **</** and **/>** pair at the beginning of the line. Each subvariable has to have a type. Types prevent accepting i.e. chemical expression, as a valid variable content in i.e. math system.

Previous example also shows an interesting property of Synth to describe semantics of knowledge base in very syntax of the knowledge base, meaning that we deal with a parser inside parser. Everything inside semantic brackets is parsed by rules from syntax braces, enriched by reasoning notation.

Before we dive into solving math expressions we have to do one more thing: we have to attach equality definitions to each of `@Sum`, `@Fact`, `@Pow` and `@Expr` variable. When we do this, each fragment of once instanced `@Sum`, `@Fact`, `@Pow` or `@Expr` expression would besides its granted value have automatically inferred values also. It is like an expression subtree, where each node have one or multiple, but equal expressions, inferred for each sub-node. Once we get this expression tree, we can walk through its partial multiple stages, finding the final result by some weighting function. Let's see how this final touch looks like in our example:

```
Math (
    Expression {
        Sum {
            Fact {
                Pow {
                    Expr (
                        {@WhiteSpace | @Null},
                        Sign {'-' | @Null},
                        Value {
                            Num {@Number} |
                            Var {@Variable} |
                            Bra (Left {'('}, In {@Sum}, Right {')'})
                        },
                        {@WhiteSpace | @Null}
                    )
                    [
                        </a:@Expr/> a |= (a);
                        </a:@Expr/> (a) |= a
                    ] |
                    PowExpr (Left {@Pow}, In {"^"}, Right {@Expr})
                }
                [
                    </a:@Pow/> a |= (a);
                    </a:@Pow/> (a) |= a
                ] |
                MulDiv (Left {@Fact}, In {'*' | '/'}, Right {@Pow})
            }
```

```
            [
                </a:@Fact/> a |= (a);
                </a:@Fact/> (a) |= a
            ] |
            AddSub (Left {@Sum}, In {'+' | '-'}, Right {@Fact})
        }
        [
            </a:@Sum, b:@Sum/> @Formula {a = b} && a |= b;

            </a:@Sum/>         a |= (a);
            </a:@Sum/>         (a) |= a
        ]
    },
    Formula (Left {@Sum}, "=", Right {@Sum})
    [
        </a:@Sum, b:@Sum/> a = b |= b = a;

        </a:@Expr, b:@Expr, c:@Expr/> |= a * b + a * c = a * (b + c);
        </a:@Expr, b:@Expr/> |= a^2 - b^2 = (a + b) * (a - b);
        </a:@Expr, b:@Expr/> |= (a + b)^2 = a^2 + 2*a*b + b^2;
        ...
    ]
)
```

Each of variables: `@Sum`, `@Fact`, `@Pow` and `@Expr` have their own semantic reasoners that tell Synth how to populate expression tree upon evaluation. `@Formula` variable serves to hold our math formulas.

First new thing we can see in these semantic lines is an use of `&&` operator `@Sum` semantic brackets:

```
</a:@Sum, b:@Sum/> @Formula {a = b} && a |= b;
```

`&&` simply concatenates multiple expressions for which we want all of them to be matched to infer the right side of “follows” operator. Second new thing is reaching inference formulas outside of current scope in noted line. This is done by stating outer variable (`@Formula` in our case), following by wanted expression inside curly braces.

Let’s consider the first semantic expression at “Sum” variable:

```
</a:@Sum, b:@Sum/> @Formula {a = b} && a |= b
```

It says: conclude “`b`“ when variable “Formula” contains “`a = b`“ and “a“ is an element of “Sum” variable. This is our mechanism for formula application to each “Sum” subnode that appears inside any math expression. The other two semantic expressions from “Sum variable” are:

```
</a:@Sum/>         a |= (a);
```

```
</a:@Sum/>          (a) |= a
```

They serve as braces introduction and elimination. Variables “Fact”, “Pow” and “Expr” contain somewhat the same code, but without formulas application. I.e. variable Fact contains:

```
</a:@Fact/> a |= (a);
</a:@Fact/> (a) |= a
```

The first line introduces braces to any “Fact” expression. Purpose of this line is to convert “a” from “Fact” to “Sum”. This conversion is done through parsing “(a)” after which “a” converts to “Sum” subchoice. After conversion, all semantics from `@Sum` applies to braced `@Fact` expression, including our formula application semantics. That conversion is a mechanism by which we can automatically populate any (not only “Sum”) sub-expression with equalities that follow from formulas.

The second line from `@Fact` semantics serves as braces elimination when a type of “a” is equal or contained in `@Fact`. Variables `@Pow` and `@Expr` contain semantics analogous to `@Fact` variable.

Getting all permutations inside one expression in Synth will be companioned by a history mechanism that shows which expressions followed from which others. Finding a solution would be done by weighting each permutation for its complexity, picking up the most simplest one, then backtracking it to original expression. Reversed path of backtracking would be a procedure of solution to original expression, all cleanly reachable from Synth programming language without need for outer low-level calls.

With previous math definition we are ready to solve a subset of math expressions restricted by entered formulas. We can now i.e. instantiate variable `@Math` on imaginary place in imaginary code by asserting:

```
@Math [
    @Expression [(x + 1)^2 - y^2]
]
```

Asserted place’s variable `@Expression` will automatically be also populated with element:

```
((x + 1) + y) * ((x + 1) - y)
```

Explained tree-based approach of permuting expressions allows us to avoid searching of subexpressions after which we would replace them one by one by new inferred subexpressions. Instead of that approach, permuting will be wired inside each node variable. It will be a feature that brings a new AI method to desktop.

4.2. logic reasoning

Just to test the Synth theory, lets build a logic reasoner also. It would look like this:

```
Logic {
    abstract OptWS {@WhiteSpace | @Null} |
    ImplExp {
        AndOrExp {
            NotExp {
                Exp (
                    @OptWS,
                    Value {
                        @Variable |
                        Bra (Left {'('}, In {@Logic}, Right {')'})
                    },
                    @OptWS
                )
                [
                    </a:@Exp/> a |= (a);
                    </a:@Exp/> (a) |= a
                ] |
                Not (@OptWS, '¬', Value {@Exp})
            }
            [
                </a:@NotExp/> a |= (a);
                </a:@NotExp/> (a) |= a
            ] |
            AndOr (Left {@AndOrExp}, In {'&' | '|'}, Right {@NotExp})
        }
        [
            </a:@AndOrExp/> a |= (a);
            </a:@AndOrExp/> (a) |= a
        ] |
        Impl (Left {@AndOrExp}, In {'->', '<->'}, Right {@ImplExp})
    }
    [
        </a:@ImplExp/> a |= (a);
        </a:@ImplExp/> (a) |= a;
    ]
}
[
    </p:@Exp, q:@Exp/>          p -> q && p |= q;
    </p:@Exp, q:@Exp/>          p -> q && p -> ¬ q |= ¬ p;
    </p:@Exp, r:@Exp/>          ¬ p |= p -> r;
    </p:@Exp/>                  ¬ (¬ p) |= p;
    </p:@Exp, q:@Exp/>          p && q |= p & q;
    </p:@Exp, q:@Exp/>          p & q |= p;
    </p:@Exp, q:@Exp/>          p & q |= q;
```

```
    </p:@Exp, q:@Exp/>          p |= p | q;
    </p:@Exp, q:@Exp/>          q |= p | q;
    </p:@Exp, q:@Exp, r:@Exp/> p | q && p -> r && q -> r |= r;
    </p:@Exp, q:@Exp/>          p -> q && q -> p |= p <-> q;
    </p:@Exp, q:@Exp/>          p <-> q |= p -> q;
    </p:@Exp, q:@Exp/>          p <-> q |= q -> p
]
```

We used the same “brace” mechanism like we did in our math example. Upon asserting a logic expression like:

```
@Logic [
    a -> b;
    b -> c;
    a
]
```

the “Logic” variable should also automatically contain expression “c”. We can think of our “Logic” variable as a placeholder which after invoking with starting set of expressions contains all conclusions that follow from that expressions. This is what is called theorem proving. It is finding a path from assumptions (starting set) to final formula (inferred) that we want to prove it is true.

4.3. induction and genetic algorithms

Induction algorithm is easy to implement, but it is expensive in means of processing power because of its property of combinatorial explosion. Induction is all about imagining new rules then checking if they hold for every relevant data record in our database. This imagining of new rules is done by systematic combining formulas’ fragments, starting from the simplest one, striving to more complex ones.

Construction of formulas is done by combining syntactic elements, gradually growing starting formula in each step. Each new step bases new formulas combinations on previous formulas, complicating previous formula by some amount of a magnitude. As the entire process suffer from combinatorial explosion, genetic algorithms are about speeding up the process by rejecting formula combinations that have low level of compliance with checked data. The more the level of compliance is, the more probability is that we are on the right path to reach compliance of 100%. When compliance of our combined formula reaches 100%, we have induced a new formula that holds on all of our data.

To show the power of induction algorithm we will consider inductive finding of rules for propositional logic. Let the following be a definition of propositional logic without inference rules (which we will find out later):

```
Logic {
```

```
    abstract OptWS {@WhiteSpace | @Null} |
    ImplExp {
        AndOrExp {
            NotExp {
                Exp (
                    @OptWS,
                    Value {
                        'T' |
                        'F' |
                        @Variable \ {'T' | 'F'} |
                        Bra (Left {'('}, In {@Logic}, Right {')'})
                    },
                    @OptWS
                ) |
                Not (@OptWS, '¬', Value {@Exp})
            } |
            AndOr (Left {@AndOrExp}, In {'&' | '|'}, Right {@NotExp})
        } |
        Impl (Left {@AndOrExp}, In {'->', '<->'}, Right {@ImplExp})
    }
}
[
    </a:@Exp, b:@Exp/> a -> b && b -> a |= a <-> b;

    ¬ T -> F;
    ¬ F -> T;

    T & T -> T;
    T & F -> F;
    F & T -> F;
    F & F -> F;

    T | T -> T;
    T | F -> T;
    F | T -> T;
    F | F -> F
]
```

In this example we introduced a new type of semantic expressions, namely granted semantic elements. They are written without “follows” sign and can contain subvariables (remember “</” and “/>” ?), but subvariables aren’t used in our example. Granted semantic elements will be our data on which we’ll check newly composed formulas.

Now we want to induce some formulas that hold for our elements. It is done in the following way:

1. We start by combining syntactic elements, one by one, and asserting subvariables on all possible places in any combination of their presence or absence (many combinations). Each asserted place would have a differently named subvariable.

2. After constructing each expression we have to isolate only granted semantic elements that succeed to parse against constructed expression.
3. For those elements which pass parsing we check equality of parsed content of different subvariables. If the content of two or more subvariables is the same, the subvariables are getting the same name and are considered to be the same notion. If we have the same variable at multiple places, we can celebrate because we just induced a new formula that holds on our data.

This way new formulas will show up and with given enough time, among them will be all axioms that build up propositional logic. Some of formulas will be:

```
¬ (¬ a) <-> a
a & b <-> ¬ (¬ a | ¬ b)
a | b <-> ¬ (¬ a & ¬ b)
(a | b) & (¬ a | c) <-> b | c
```

Let's consider genetic aspect of induction. As we said, induction is a slow process because of combinatorial explosion. To speed it up, we can reject further developing of some combinations if they do not parse against any granted expression in certain amount of percentage. Instead of rejecting, we can also give smaller developing priority to formulas with small percentage and focus more on those with larger percentage, which will give us more complete algorithm because there might be a slight chance that less suited formula will succeed in some of future combinations.

Induction can be applied to numerous fields. It is not hard to imagine finding physics formulas from a table of empirical numeric data. With explained method we can inductively find patterns in any field where we have enough data to reason about. Note that incomplete set of data is a subject to more erroneous inductive conclusions as results can not be enough tested on exemplars.

As Synth is planned to be a complete programmed language, it will be possible to implement any specific genetic algorithm through mechanisms that are not described in this paper.

5. conclusion

We saw some of methods that can be used to achieve automated reasoning. What is not shown here, but will be included in Synth programming language, is dealing with state machine. It would answer questions like: if A is a state machine and its current state is A_x, what actions are required to get the machine in state A_y?

Methods of AI can have numerous uses, some of which are shown in this paper. Complete autonomous thinking and acting machine is still an open question and that is where can Synth hop in: to enable creation of such a machine. Until that achievement, we can cover some of AI fragments and try to pass our knowledge about AI to the next generations.