What is the motivation of this paper? What type of problems might this approach solve than other approaches? Practical use of the approach?

The paper primarily aims to demonstrate that, in the context of program synthesis problems based on tree operations, the efficiency of compiler reasoning within a framework that incorporates both user prompts and DSNN (alternatively referred to as "User prompt+DSNN" or "U2N" for User to Neural) is superior to that of a framework relying solely on user prompts, which in turn is superior to an unguided framework.

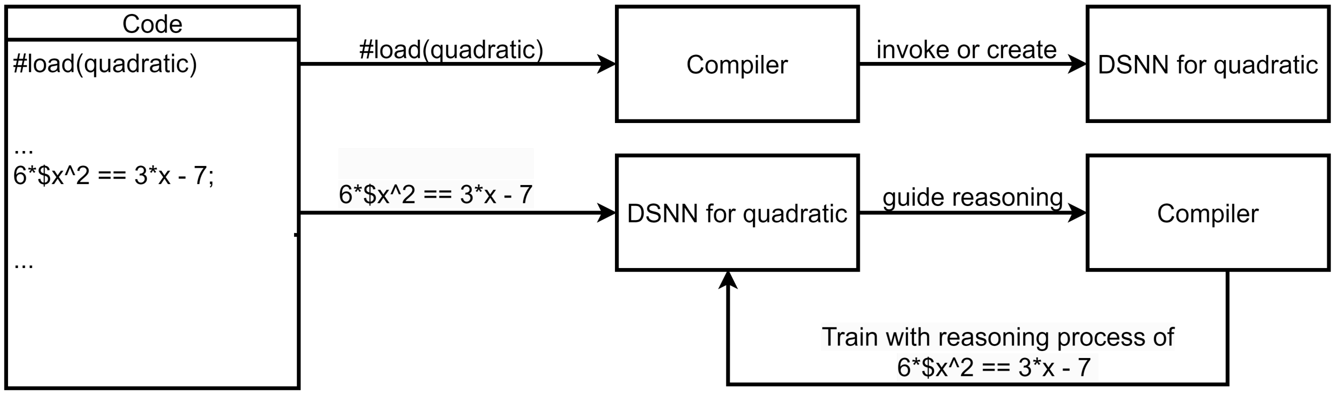
A secondary goal of the paper is to highlight the advantages of the experimental programming language COOL, in comparison to other languages/frameworks. Introduction of secondary purposes is indispensable to the primary purpose.

The paper currently only demonstrates the effectiveness of the (User+Neural) compilation architecture. COOL is still being refined, and the User+Neural compilation structure has not yet been adopted in production activities. We call on more experts to try this novel compiler structure.

Paper structure

The paper first introduces its secondary objectives, which is crucial for understanding the primary objective, before delving into its primary goal, with the experimental section focusing solely on substantiating the primary goal.

What is DSNN (Domain-Specific Neural Network)? And how DSNN works in the compilation framework?



```
This diagram only serves as an example of the DSNN's workflow. The process of using a
DSNN to guide the reasoning for a problem and the process of training the DSNN with
this problem's reasoning are not synchronous. The iterative cycle of application and
learning allows the DSNN to continually refine its ability to assist in problem-
solving, becoming more effective as it accumulates more data and experience.
```

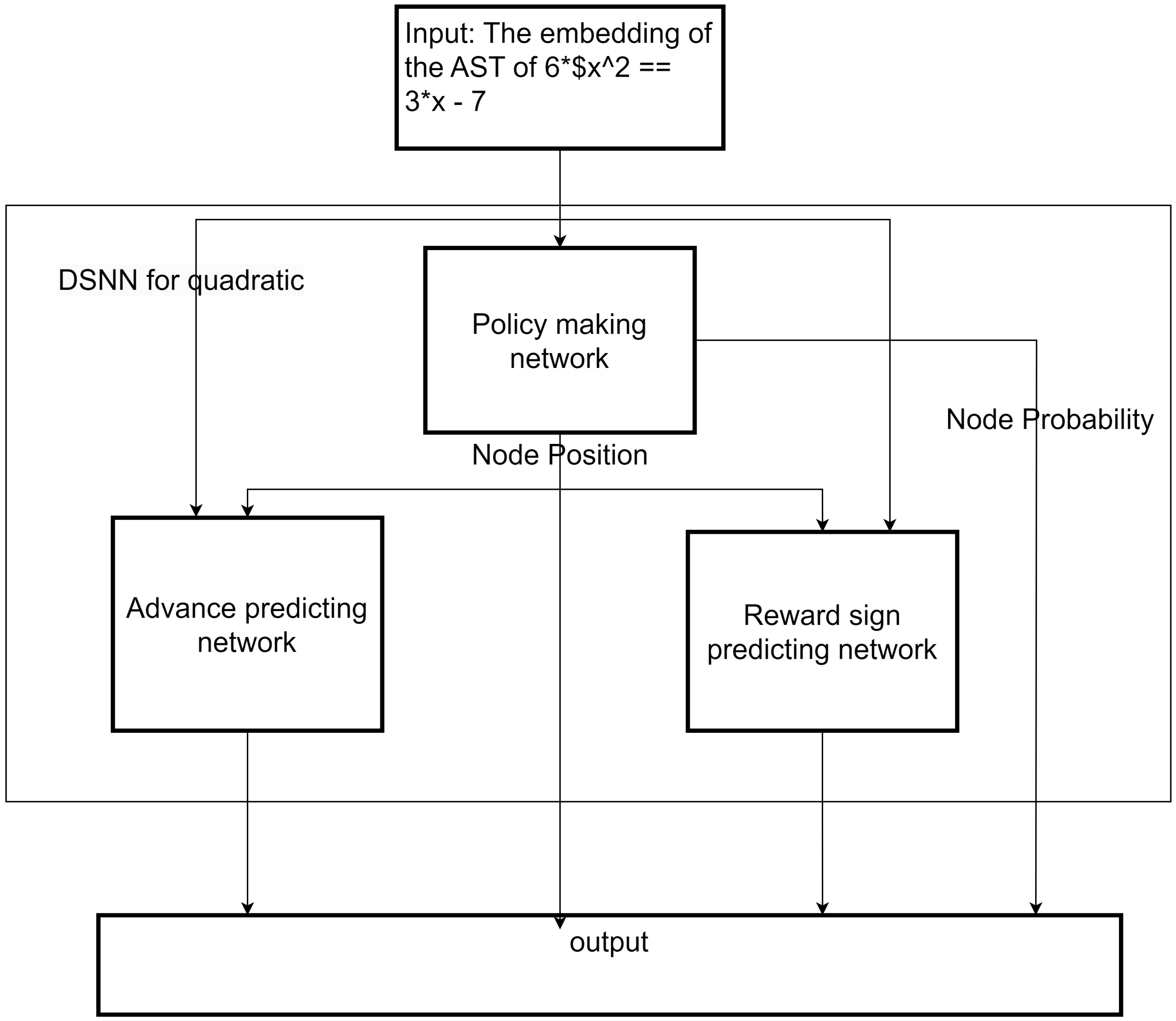
A DSNN (Domain-Specific Neural Network) is designed based on the application of neural networks within Domain-Specific Languages (DSLs). Some efforts integrate neural networks with logic programming languages specific to certain fields to enhance these languages' reasoning capabilities.

COOL aims to be a General-Purpose tool that encapsulates the tools needed to handle specific problem types into a class or library file, referred to as a Knowledge Domain. However, considering the impracticality of integrating a large, general-purpose neural network model directly into the compiler, COOL opts to bind individual neural network models with individual Knowledge Domains instead.

From the compiler's perspective, when a user needs to address a particular type of problem and calls upon relevant tools, they would load a library or inherit a class through specific syntax during programming. When the compiler encounters such syntax in compilation, it calls upon a specific neural network for reasoning and optimization within the current code scope. This neural network is referred to as a DSNN.

The DSNN approach allows for a more focused and efficient integration of neural network models for enhancing the programming language's capabilities, specifically tailored to the needs of the problem domain being addressed. This strategy leverages the strengths of neural networks in prediction within the constraints and context of domain-specific logic, providing a powerful tool for optimizing and reasoning about code in ways that traditional programming languages and compilers may not be able to achieve on their own.

What is the inner structure of the DSNN? How it works?


Input: The embedding of the AST of 6*$x^2 == 3*x - 7
DSNN for quadratic
Policy making network
Node Position
Node Probability
Advance predicting network
Reward sign predicting network
output

The structure of the DSNN, as shown in the diagram, includes a Policy Making Network, an Advance Predicting Network, and a Reward Sign Predicting Network. Here's how these components interact:

- **Policy Making Network:** This part of the DSNN takes the problem's Abstract Syntax Tree (AST) and predicts the probability that each node will be operated on as the root node in tree operations. It produces two key outputs: the most probable node `position` and the predicted `probability` of node modification .

- **Advance Predicting Network and Reward Sign Predicting Network:** Using the node position determined by the Policy Making Network, these networks generate two variables, `advance` and `reward sign`.
  - **Position:** Constraints the rule to be consistent with a specific subtree pattern in the problem's AST.

- **Advance:** Constraints the reasoning progress to advance a stage as the user prompt (pcp).
- **Reward Sign:** Constraints the sign (positive or negative) of reward value of the rule provided by the user.
- ``

```
//The (2,0,0,0) is the user prompt (pcp)
//The position of '2' indicates the reasoning stage where this function can be called, and it is constrained by the "advance". If "advance" is 'true' and the reasoning process is at 1, only the functions of stage greater than 1 can get DSNN's reward and this function can't.
//The number '2' indicates the reward value defined by user, and is constrained by the "reward sign". If "reward sign" is '+', this function can get the reward from DSNN. Vice versa, cannot.
       V^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
expr:@(2,0,0,0)
{(ne(0):#?a + ne(0):#?b)^2} //The rule's AST pattern, or the function name, is constraint by "position"
{
        return:a^2+2*a*b+b^2;
}


...
//Expanding squared terms
expr:@(2,0,0,0){(ne(0):#?a + ne(0):#?b)^2}{
        return:a^2+2*a*b+b^2;
}
...
//Removing parentheses
expr:@(0,1,0,0,0){#?a-(#?b+#?c)}{
        return:a-b-c;
}
...
//Subtraction to addition
expr:@(0,0,0.5,0,0){ne(0):#a-immediate:b}{
        new:tmp = 0-b;
        return:a+tmp;
}
...
//Extracting constant coefficient
```

```
expr:@(0,0,0,-4,0){$a*$b}{
        return:b*a;
}
...
```

When reasoning, only those rules that satisfy **ALL** three constraints can get the reward of DSNN. In doing so, it obtains rewards from both the user prompt and the DSNN (calculated as k *probability* accuracy), aligning the problem-solving process with the guidance provided by the DSNN and user input.

The DSNN chooses to use a series-parallel predictive structure of **(1) problem --> constraints of rule, probability** instead of **(2) problem, rule --> probability** or **(3) problem --> solution, probability** for two reasons:

1. **Overhead Reduction:** If the DSNN had to evaluate each rule individually against the problem, it would introduce significant overhead, especially when there are a large number of rules to consider. By focusing on predicting constraints and the probability that a rule should be applied, the DSNN streamlines the decision-making process. This approach filters out a small subset of rules based on the predicted constraints, which are likely to be relevant to the problem at hand, thereby reducing the computational burden.

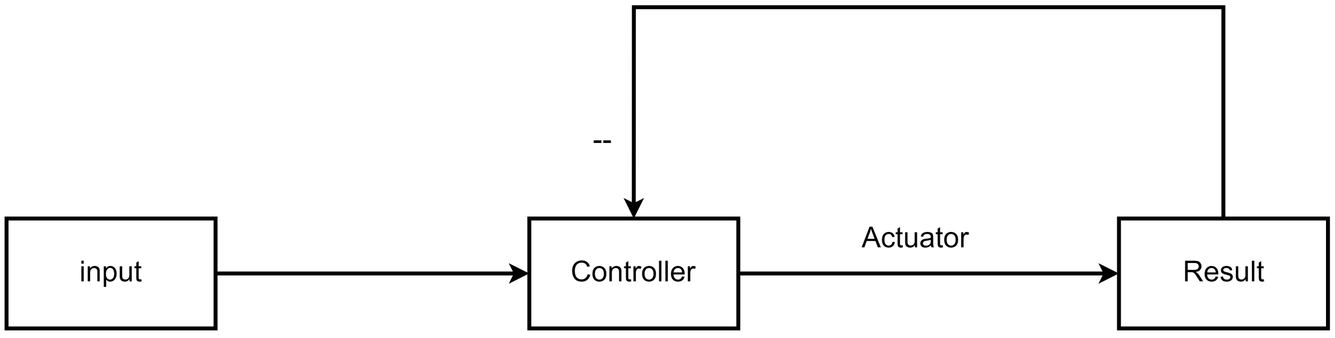
2. **Verification Function:** The architecture of having three neural networks working in a series-parallel configuration serves as a form of cross-verification. Each network outputs a variable—position, advance, and reward sign—that acts as a constraint for the rule to be applied. If any of the networks is poorly trained or performing suboptimally, the probability of producing an invalid combination of constraints is increased. This, in turn, lowers the DSNN's weight in the decision-making process.

By designing the DSNN this way, the system not only becomes more efficient in processing but also more robust. This is also the basis for the U2N mechanism to work.

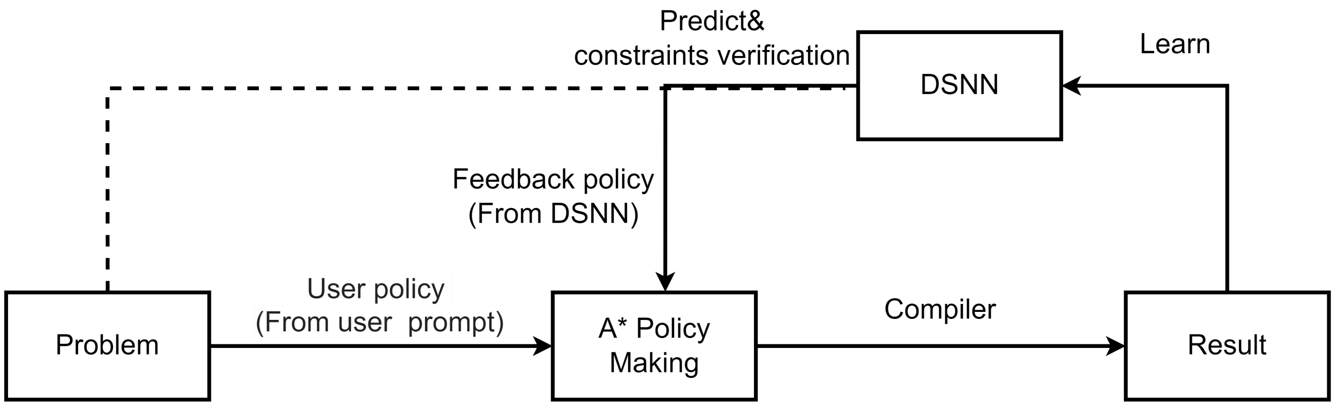
How user guidance and neural guidance work together? What is U2N mechanism? What are its advantages?

User and neural guidance collaborate under the U2N (User to Neural) mechanism. U2N utilizes the DSNN to adjust the decision-making process during the user-guided code synthesis process. As the DSNN becomes more proficient with increased training, its contribution to decision-making in code synthesis grows, gradually shifting the decision-making power from the user to the neural network.

The design of the U2N mechanism is based on the concept of a negative feedback control loop from control theory. This is depicted in the following diagrams, where the result is fed back into the input to add a compensating signal, achieving better control and enhancing the system's robustness.



feedback negative control loop



U2N mechanism

In practice, it was found that when the DSNN is in its early stages and not fully trained, its involvement in the feedback loop introduces a lot of noise into the decision-making process. Therefore, it is essential that the decision-making weight of the neural network increases as its predictive ability improves, which is the essence of U2N.

Two primary methods are used:

1. **Reward Scaling with Model Accuracy:** By multiplying the rewards provided by the DSNN with the model's accuracy and setting an accuracy threshold, the amplitude of noise can be reduced.

2. **Splitting DSNN into Multiple Coupled Networks:** By creating a series-parallel structure of multiple neural networks, not only is the size of individual models reduced, but the outputs of the models require cross-verification, which filters out some of the noise.

The experimental results confirmed that the U2N mechanism, through the DSNN's negative feedback, improves the compiler's reasoning ability beyond what is achieved with User Prompt only. Moreover, it does not adversely affect the reasoning process when the DSNN is not sufficiently trained with examples. This design balances the innovative approach of neural guidance with the reliability of user prompts, helping to achieve higher-quality code synthesis without the risk of untrained neural models.

Why the language COOL is designed to be functional?

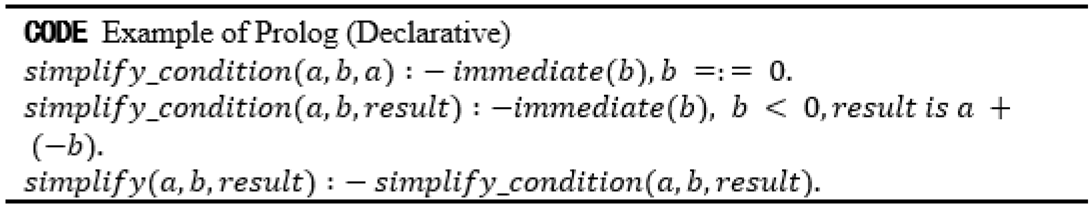
Thank you for your interest in this innovative programming language. The reason for choosing declarative over functional is that functional is more concise and easier to read when expressing complex rule logic.

The most important advantage of using functions to represent rules or facts is that they are more concise and readable when dealing with complex logic in rules and facts, although the code is still declarative in nature.

Here is an is a simple simplification rule:

When ‘a’ is an unknown number and ‘b’ is an immediate number, if ‘b’ is zero, simplify ‘a-b’ to ‘a’, if ‘b’ is not zero, simplify to ‘a+(-b)’.

1. Written in a purely declarative style, it is not easy to quickly understand the purpose of this set of rules:



CODE Example of Prolog (Declarative)

```
simplify_condition(a, b, a) : − immediate(b), b =:= 0.
simplify_condition(a, b, result) : −immediate(b), b < 0, result is a +
(−b).
simplify(a, b, result) : − simplify_condition(a, b, result).
```

2. When written in a function-like style, the processing logic can be understood relatively easily:

CODE Example of Prolog's code (Function-like style)

```
simplify(a + b, result): −
    immediate(b),
    (b == 0
        −> result = a
    ;  b < 0
        −> tmp is 0 − b,
        result = a + tmp
    ).
```

3. In COOL, the style is further functionalized. In addition to explicitly adopting keywords common in GPLs (C++, Java, etc.), it is also allowed to present constraints on variables and rules/facts in a manner similar to variable attributes.

Furthermore, expressions are allowed to be used directly as function names, which improves the expressivity of rules and saves part of the work of naming rules/facts.

CODE Example of COOL's code (Functional style)

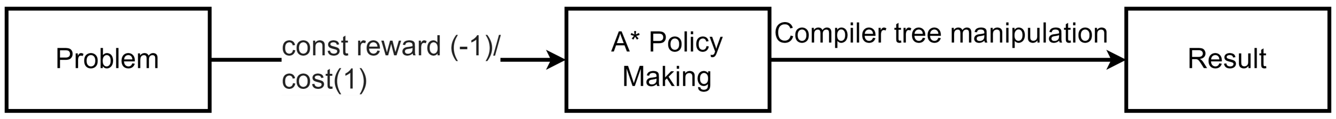
```
expr: @(2,2,2,2,2){$a − immediate: b}{
    if(b == 0){
        return: a;
    }else{
        new: tmp = 0 − b;
        return: a + tmp;
    }
}
```

Is the baseline of the language BFS?

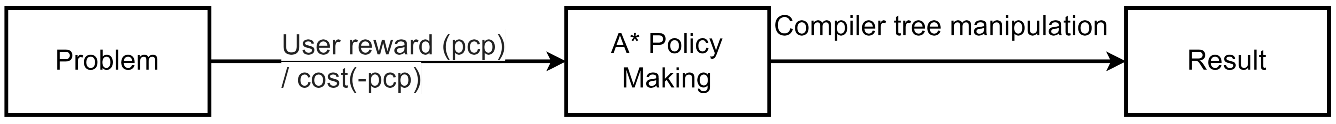
NO. In this paper, stating that the BFS algorithm is used as a baseline and then compared with the user-guided and Neural+user-guided algorithms is a misrepresentation.

In this experiment, COOL actually employs the A *algorithm. Since the experiment aims to demonstrate the effectiveness of DSNN, it's crucial to control for extraneous variables by ensuring consistency in the algorithms used across all groups. A* algorithm, as a classic heuristic optimization approach, easily represents situations without user prompts or neural network corrections through its heuristic function, making it highly suitable for validating this issue.

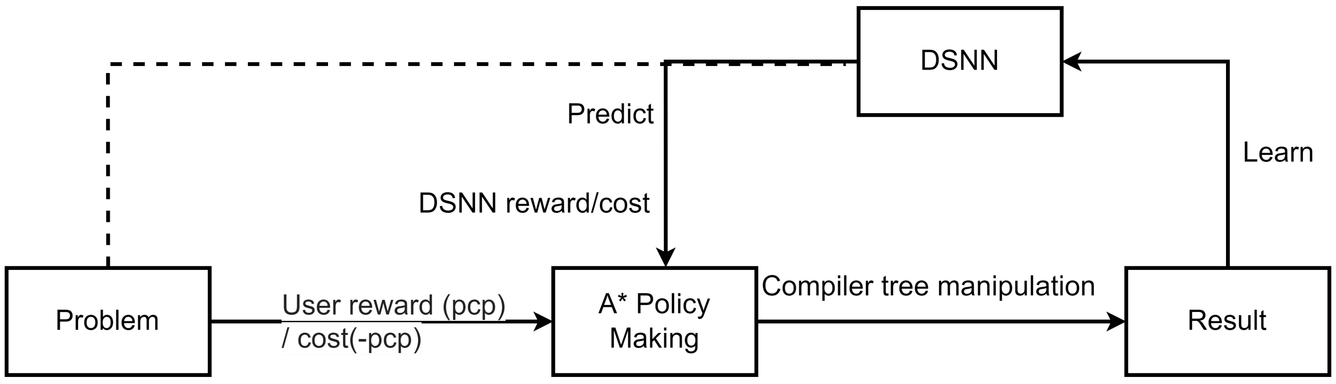
Therefore, the differences between groups lie only in the heuristic function's return values (the group without dsnn or user returns -1/-1, the group with user prompts but without nn returns pcp value, and the group with both user prompts and dsnn returns pcp+nn predicted value). It's just that BFS happens to represent the complexity of the A* algorithm in scenarios without any prompts (no heuristic).



(a) the group without dsnn or user prompt/ no heuristic



(a) the group with user prompt only



(c) the group with user prompt and dsnn

We apologize for any confusion caused here. The independent variable of this experiment is the compilation system structures 1) whether or not there are user prompts, 2) in the presence of user prompts, whether DSNN is involved, and 3) the number of compiler inference problems (affecting the number of examples for the DSNN to learn).

Is the benchmark used in the experiment reasonable?

The measurement goal of this paper is not the ability to solve quadratic equations, but the efficiency of tree operations. Tree operations are crucial for measuring reasoning ability during the code synthesis process; fewer tree operations indicate higher reasoning efficiency.

It should be clarified that **this experiment has nothing to do with whether there are mature solutions for quadratic equations, nor is it to demonstrate that COOL can be used to simplify polynomials**. It's purely a task to measure reasoning efficiency by counting tree operations, as explicitly stated in our Experiment section.
What we value is the efficiency of compilers with different structures in transforming quadratic equations into standard forms according to certain rules:

```
//In the file "quadratic.cos"
//This is a library file that stores rule functions and fact functions required to solve a type of problem (quadratics). This is also called a knowledge domain. The compiler will create a DSNN to bind to the library when it is first used.
//General Transformations for Simplification
expr:@(0,1,1,1,1){#a*a}{
        return:a^2;
}
...
//Moving Terms
expr:@(5,0,0,0,0){$a==$?b}{
        return:a-b==0;
}
...
//Expanding squared terms
expr:@(2,0,0,0){(ne(0):#?a + ne(0):#?b)^2}{
        return:a^2+2*a*b+b^2;
}
...
//Removing parentheses
expr:@(0,1,0,0,0){#?a-(#?b+#?c)}{
        return:a-b-c;
}
...
//Subtraction to addition
expr:@(0,0,0.5,0,0){ne(0):#a-immediate:b}{
        new:tmp = 0-b;
        return:a+tmp;
}
...
//Extracting constant coefficient
expr:@(0,0,0,-4,0){$a*$b}{
        return:b*a;
}
...
//Combining like terms and arranging by descending powers
expr:@(0,0,0,0,2){immediate:a*$x+immediate:b*$x}{
        new:tmp = a+b;
        return:tmp*x;
}
...
//Matching and applying the quadratic formula
@(0,0,0,0,0,10){a*$x^2+b*x+c==0}{
        if(b^2-4*a*c<0){
                x="null";
        }
        else {
                new:x1=(-b+(b^2-4*a*c)^0.5)/(2*a);
                new:x2=(-b-(b^2-4*a*c)^0.5)/(2*a);
                x={x1,x2};
        }
}
...
```

When the compiler parses a user's request to utilize a library to solve a problem—indicated by `#load(quadratic)` in the following snippet—it activates the DSNN bound to that library to enhance the compiler itself's reasoning capabilities. Additionally, after solving the problem, the system collects examples to further train the DSNN. This mechanism demonstrates a dynamic interaction between the programming environment and the neural network model.

```
#load(quadratic) //The DSNN bound with "quadratic.cos" is invoked.
new:x=1;
...
6*$x^2 == 3*x - 7;
x-->"#FILE(SCREEN)";
($x - 6)*(x + 3) == x;
x-->"#FILE(SCREEN)";
$x^2 + 4*x == 5 + 2*x^2;
x-->"#FILE(SCREEN)";
...
```

The technique of solving polynomials, exemplified by quadratic equations, is very mature. However, the reason for choosing it as the experimental subject is operations on quadratic equations cover most basic operators and various operational laws. Therefore, the tree operations involved are sufficiently diverse and complex. Since most of the tree operations used in transforming a general form of quadratic equation code into a solvable standard form can also be applied to other mathematical problems, these benchmarks are representative and generalizable enough.

If you think the problem of converting quadratic equations to standard form is not representative for all kinds of problems, we can supplement the data with experiments on trigonometric functions and family relationship reasoning on page nine to enhance the experiment's representativeness in symbolic reasoning and relational reasoning, which are two important directions in logical reasoning research.

We will also provide more experimental metrics, such as time spent on inference. Although it has not been fully optimized, in the optimal state, calling DSNN only takes about 5% more time than not calling it.

Here are the code snippets for family relationship reasoning. The original purpose of our experiments on multiple different types of problems is to study the collaboration and interference issues of DSNNs in different knowledge fields acting in the same scope.

```
//In the file "family.cos"
//This is the knowledge domain for inferring family relationships.
//Elders
expr:@(1){(a) is (b)s father}{
        return:  (a) is male && (a) is (b)s parent;
}
...
//Peers
expr:@(1){(a) is (b)s brother}{
        symbol:c;
        return: (a) is male && (a) is (b)s sibling&&(c) is (a)s parent &&(c) is
(b)s parent;
}
...

//Descendants
expr:@(1){(a) is (b)s son}{
        return:(a) is male && (b) is (a)s parent;
}
...
//Implicit relationships
```

```
expr:@(-2,-2,-2){(a) is (b)s sibling}{
        if(this expr. exist subexpr{(b) is (a)s sibling} == false){
                return:(b) is (a)s sibling && (a) is (b)s sibling;
        }else{
                abort;
        }
}
...
//Combining duplicate relationships
expr:@(1,1,1){#a&&a}{
        return:a;
}
...
//Relationship querying
expr:@(0,0,10){(a) is (b)s ($relation)}{
        //immediate family
        //father & mother
        while(this expr. exist subexpr{(a) is (b)s parent}){
                if(this expr. exist subexpr{(a) is male}){
                        return:relation = "father";
                }
                if(this expr. exist subexpr{(a) is female}){
                        return:relation = "mother";
                }
        }
    ...
}
...
```

```
#load(family)
new:relation = "";
...
new:Ashley = "Ashley";
new:Nicholas = "Nicholas";
new:Lillian = "Lillian";
(Nicholas) is (Ashley)s son && (Lillian) is (Ashley)s daughter &&
        (Nicholas) is (Lillian)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

Why the experimental comparison between COOL and other programming languages, such as SyGus, was not conducted?

The decision not to compare COOL with other programming languages like SyGus in the experiment was made to control extraneous variables, such as the type of programming language and algorithms used. Additionally, the method COOL uses to collect examples is greatly different from the known example collection methods from PBE systems, making direct experimental comparison impractical.

Despite this, comparisons between COOL and similar programming languages (or frameworks) are made in other sections of the paper and we point out its advantages or the disadvantages it addresses:

- **General-Purpose:** COOL is intended as a general-purpose Logic Programming language rather than a Domain Specific Language. Many declarative languages, which often serve as DSLs, are typically only adept at solving one type of problem or more niche issues. In contrast, COOL's ability to handle a broader range of problem types is a significant advantage over other imperative languages. Examples given include solving quadratic equations, trigonometric functions, family relationship reasoning, container operations, and object-oriented programming. These examples cover symbolic reasoning, relational reasoning, and various programming paradigms, along with logic-guided function derivation not mentioned in the paper. This versatility supports the first contribution 1 highlighted in the text.

- **Use of DSNN for Program Synthesis:** DSNN (Domain-Specific Neural Networks) is used for guiding program synthesis, transforming declarative programs that require reasoning into imperative programs. By binding DSNN to specific classes or libraries and managing its lifecycle through the compiler, COOL transcends the limitations of a DSL to achieve general-purpose functionality.

- **No Need to Provide Training Examples:** Code synthesis is closely related to PBE systems, and PBE systems rely heavily on examples to guide the reasoning process, the effectiveness of a PBE system hinges on acquiring sufficiently representative examples. PBE systems is not good at reasoning through complex logic, especially for problems with multiple branches and boundaries, as this requires users to provide examples for various scenarios, which can be costly and sometimes require auxiliary programs for example generation. This process can be more challenging than writing the code directly. In contrast, DSNN examples are supplied by the compiler, eliminating the need for manual example preparation.

Why we didn't choose and extend an existing (language/ compiling) system? Why those system cannot be extended to address these limitation?

- **From the Design Perspective of Programming Languages:** COOL primarily extends from a combination of PROLOG and C++, utilizing vectors as function prefixes to control the reasoning process, a syntax similar to what can be found in Problog. The integration of logic programming with object-oriented programming also follows numerous precedents.

- **From the Reasoning/Compiling Perspective:** COOL's compiler attempts to combine the advantages of User-Guided Reasoning and Neural-Guided Reasoning while avoiding their respective disadvantages. COOL integrates PBE systems internally (invisible to the user, thus eliminating the need for users to provide examples) and incorporates user prompts directly into the programming syntax (determining user prompts during programming, thereby eliminating the need for repeated interactions with the user during reasoning).

- **From the Implementation Perspective:** While COOL's syntax and compilation systems are extensions and integrations of existing systems, they were not developed by extending a specific code project. This is mainly because no suitable project for extension was found. Existing projects were either experimental in nature or had their interfaces highly optimized, making extension less feasible than developing a new system.

Broken reference and typos

Mainly because we were too hasty when compressing the content of the article. We will correct these mentioned and unmentioned errors in future versions.

Paper reproduction

The experimental materials are in the same folder as this file.