UNIVERSITÉ Concordia UNIVERSITY

COMP 6591

Project Report: Contelog

```
GID: 18 = [
    “Prateek Bagora (40156671)”,
    “Shuen Yuan Chen (24627245)”,
    “Nurul Islam (40157122)”
]
```

Submitted To: **Dr. Nematollaah Shiri Varnaamkhaasti**

Submission Date: **June 22, 2021**

Introduction

With the emergence of artificial intelligence (AI) and evolving robotics used in a variety of fields, namely, health, medicine, technology and more, understanding logic and reasoning are becoming more and more important to improve the accuracy of predictions. Ammar's paper on Contelog [1] presents a theory of contexts that defines context representations and operations. Contelog is a conservative extension of Standard Datalog that leverages Datalog's syntax and semantics to enable reasoning in contexts, and allows for change in the context without changing the rules and facts. Evaluation of a Contelog and/or Datalog program consists of either the top-down approach or the bottom up approach. In this project, we propose to implement a bottom-up and semi-naive approach to evaluating Contelog and Datalog programs.

Goals

1. Parser to read and parse Contelog and Datalog programs
2. Check for safety and validity of the parsed programs
3. Bottom-up, semi-naive approach to evaluation of Contelog and Datalog programs parsed
4. Perform successful real-time tests on the system built

System Specifications

The program is divided into three main components: the tokenizer, the parser, and the evaluator. The tokenizer's sole function is to read the characters and create tokens from them. . The parser then identifies the program elements from these tokens and creates corresponding Python objects. Lastly, the evaluator performs a semi-naive bottom-up evaluation of the parsed program to generate all the facts that can be inferred from the program.

This context-aware knowledge base system is built in Python, with the help of the PLY library, and has been tested on the necessary test cases. It consists of the following program files:

1. *elements.py*: All the basic input program components are parsed into objects of the following classes:
    - Predicate: Characterized by name, arguments, context, and type.
    - Fact: Characterized by predicate and type.
    - Rule: Characterized by head, body, and type.
    - Constraint: Characterized by term_x, theta, and term_y.
    - Context: Characterized by name, contextual_predicates, and type.
    - Query: Characterized by predicates and type.

2. *tokenizer.py*: Reads the characters of the input program and identifies the following tokens:
    - `OPEN_CURLY: “{”`
    - `CLOSE_CURLY:  “}”`
    - `OPEN_SQUARE: “[”`
    - `CLOSE_SQUARE: “]”`
    - `OPEN_ROUND: “(”`
    - `CLOSE_ROUND: “)”`
    - `IMPLY: “:-”`
    - `COLON: “:”`
    - `COMMA: “,”`
    - `PERIOD: “.”`
    - `THETA: “!=”, “<=”, “>=”, “<”, “>”, “=”`
    - `QUESTION_MARK: “?”`
    - `ANNOTATION: “@”`
    - `UPPER_NAME: name starting with uppercase.`
    - `LOWER_NAME: name starting with lowercase, digit, or underscore.`

3. *contelog_parser.py*: Identifies the program elements from the tokens and creates corresponding Python objects of types specified in ‘elements.py’.

    For example:

    ```
    ceast = {from : [east], to: [right]}.
    ```

    will be parsed to:

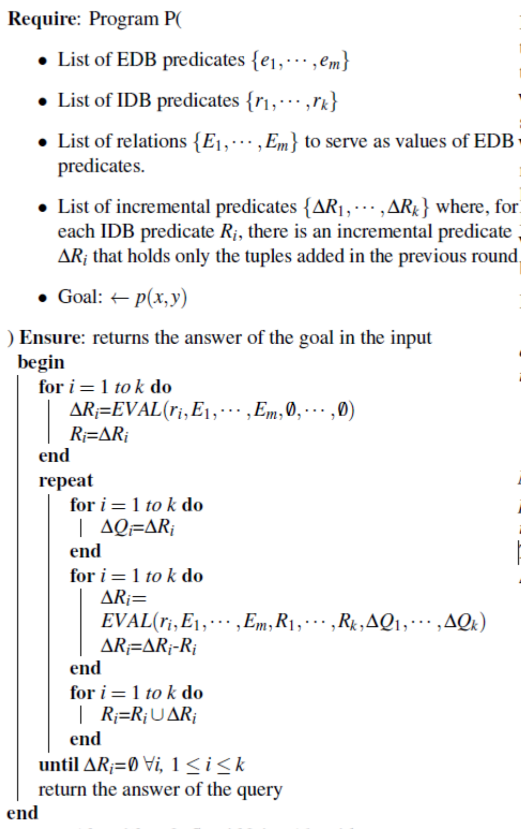
    ```
    Context(name = ‘coeast’, contextual_predicates = predicate_list)
    predicate_list = [
        Predicate(
            name = ‘from’,
            arguments = [‘east’],
            context = ‘ceast’,
            type = ‘contextual_predicate’
        ),
        Predicate(
            name = ‘to’,
            arguments = [‘right’],
            context = ‘ceast’,
            type = ‘contextual_predicate’
        )
    ]
    ```

4. *contelog.py*:
    - Checks for rule safety and validity.
        - All facts need to be ground.
        - All variables in the rule head must also appear in the rule body.
        - All variables occurring in a constraint must also occur in the arguments of some other predicate of the same rule body.

    - Evaluates the input program using the Python objects obtained from the parser. The evaluation steps are discussed in the next section

Bottom-Up, Semi-Naive Algorithm

The bottom-up, semi-naive algorithm consists of starting with facts, and substituting them into corresponding atoms (predicate, constraint with or without contextual information) from the body. Once all the predicates in the body have been substituted, the rule head may be inferred from the facts that were substituted in. This process is repeated until no more new facts can be derived. Below is a depiction of the algorithm used for reference in more detail, from [2]:



Require: Program P(

- List of EDB predicates $\{e_1, \cdots, e_m\}$
- List of IDB predicates $\{r_1, \cdots, r_k\}$
- List of relations $\{E_1, \cdots, E_m\}$ to serve as values of EDB predicates.
- List of incremental predicates $\{\Delta R_1, \cdots, \Delta R_k\}$ where, for each IDB predicate R_i, there is an incremental predicate ΔR_i that holds only the tuples added in the previous round
- Goal: $\leftarrow p(x, y)$

) **Ensure**: returns the answer of the goal in the input

```
begin
    for $i = 1$ to $k$ do
        $\Delta R_i$=$EVAL(r_i, E_1, \cdots, E_m, \emptyset, \cdots, \emptyset)$
        $R_i$=$\Delta R_i$
    end
    repeat
        for $i = 1$ to $k$ do
            $\Delta Q_i$=$\Delta R_i$
        end
        for $i = 1$ to $k$ do
            $\Delta R_i$=
            $EVAL(r_i, E_1, \cdots, E_m, R_1, \cdots, R_k, \Delta Q_1, \cdots, \Delta Q_k)$
            $\Delta R_i$=$\Delta R_i$-$R_i$
        end
        for $i = 1$ to $k$ do
            $R_i$=$R_i \cup \Delta R_i$
        end
    until $\Delta R_i$=$\emptyset$ $\forall i$, $1 \leq i \leq k$
    return the answer of the query
end
```

Algorithm 2: Semi Naive Algorithm

Evaluation

Preprocessing

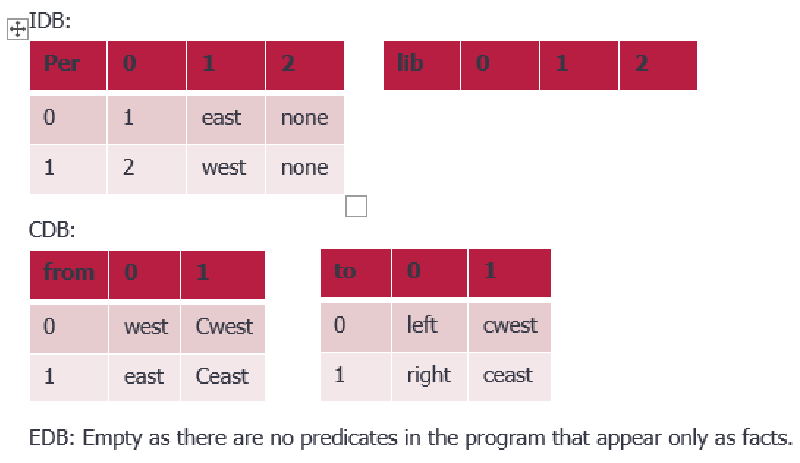
Our implementation relies on Pandas DataFrames for the evaluation. So, we first needed to define DataFrames for all the predicates in the program. We partitioned these DataFrames into three parts, which are:

- Extensional Database (EDB): Predicates appearing in facts only.
- Intensional Database (IDB): Predicates appearing in a rule head.
- Context Database (CDB): Predicates appearing in context definitions.

EDB, IDB, and CDB are maintained as dictionaries of DataFrames with predicate name as the key. Taking the following program as an example:

```
cwest = {from : [west], to : [left]}.
ceast = {from : [east], to : [right]}.
per(1, east).
per(2, west).
per(X, Y)@C :- per(X, Y), from(Y)@C.
lib(X, Y)@C :- per(X, Z)@C, to(Y)@C.
```

The output of preprocessing would be the following DataFrames, where last column always contains the context information:



IDB:

Per	0	1	2
0	1	east	none
1	2	west	none

lib	0	1	2

CDB:

from	0	1
0	west	Cwest
1	east	Ceast

to	0	1
0	left	cwest
1	right	ceast

EDB: Empty as there are no predicates in the program that appear only as facts.

Basic Operations Required for Rule Processing

To implement the evaluation algorithm, we first need to define the basic operations required on the DataFrames for evaluating the rules. These operations look at the Python objects and corresponding DataFrames simultaneously. A rule is evaluated as a sequence of the following steps:

- Evaluating a predicate in the rule body
    1. Create a copy of the DataFrame corresponding to the predicate.
    2. Rename DataFrame's columns according to the predicate arguments and context.
    3. Find all headings having some constant as its value (starting with lowercase or integers). Filter these columns' values for these constants.

- Evaluating the rule body
    1. Find the common column headings among the DataFrames evaluated for the predicates in the rule body.
    2. If at least one common heading is found among these DataFrames, perform an inner join over these DataFrames. If not, then perform cross product over these DataFrames.

- Evaluating the constraints in the rule body
    1. Apply the constraint operation on the DataFrame evaluated over all the ordinary predicates present in the rule body. The comparison implied by the constraint is performed over the columns of the DataFrame having the headings same as the arguments of the constraint.

- Inferring the rule head
    1. For all constants present in rule head's arguments, add a constant column to the DataFrame obtained from the rule body.
    2. Select all columns corresponding to rule head's arguments and context from the resulting DataFrame.

Evaluation Algorithm

Based on the rule processing procedure defined above, the program evaluation works as follows:

```
IDB_old = empty
IDB_new = empty
IDB_delta = IDB

while count_rows(IDB_delta) != 0:
    for each rule in the program:
         for each combination of IDB predicates present in rule body, taken either
         from IDB_old or IDB_delta, except the combination taking all the predicates
         from IDB_old:
            for each predicate in the combination:
                evaluate and join the predicate to the combination_result
            append the records from combination_result to result
        for each CDB predicate in the rule body:
            evaluate and join the predicate with the result
        for each IDB predicate in the rule body:
             evaluate and join the predicate with the result
        apply constraints to the result
         evaluate the rule head over the result and append the facts derived to
         IDB_new
    IDB_old = merge(IDB_old, IDB_delta)
    IDB_delta = set_difference(IDB_new, IDB_old)
    IDB_new = empty


return IDB_old
```

Sample Program:

```
%Contexts
c1 = {from : [east], to : [left]}.
c2 = {from : [west], to : [right]}.

%Facts
p(john, east).
p(rose, west).

%Rules
q(X, Y)@C :- p(X, Y), from(Y)@C.
side(X, Z)@C :- q(X, Y)@C, to(Z)@C.
```

Sample output:

All inferences from the program:

```
q(john, east)@c1

q(rose, west)@c2

side(john, left)@c1

side(rose, right)@c2
```

Conclusion

In conclusion, we have successfully implemented a version of the bottom-up, semi-naive evaluation of simple Contelog programs. Tests were performed for safety checks (rule, fact and constraint safety) and programs obtained from www.contelog.com (test_*.clg and demo_test_*.clg) yielded correct results.

The main challenges that we have met are:

- time constraint - having only 10-11 days to design, implement, and test.
- distribution of tasks - everyone being on a different schedule, and one member residing in India, we still managed to hold meetings and discuss readings needed, the project implementation, and meeting with the TA (Ammar) to show him our preliminary results.
- more extensive tests were not performed on our system due to time constraint, though we have managed to successfully yield correct results on many samples (see test files included) from www.contelog.com and from our own testing.

References

1. Alsaig, Ammar; Alagar, Vangalur; Nematollaah, Shiri. “Contelog: A declarative language for modeling and reasoning with contextual knowledge”. *Knowledge-Based Systems*, vol. 207, no. 106403, pp. 1-13, http://www.selsevier.com/locate/knosys
2. Yaacoub, Antoun; Awada, Ali. “Information Flow in Datalog using Very Naive and Semi Naive Bottom-Up Evaluation Techniques.” *International Journal of Computer Science: Theory and Application*, vol. 5, no. 2, 2016, pp. 35-48, https://core.ac.uk/download/pdf/228424644.pdf. Accessed 17 06 2021.