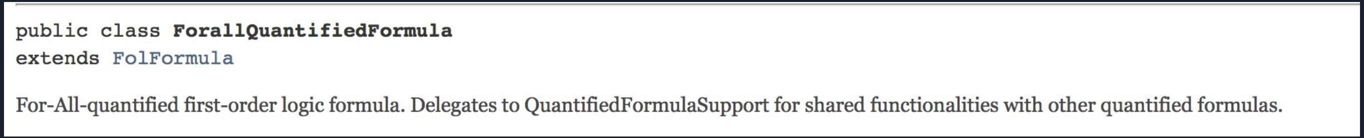
4TB3 P0 Extension with Quantifiers

COMP SCI 4TB3 Project- Group 11
Andriy Yuzva, Daniel Nova, Parsa Abadi
April 21, 2022

The Goal


```
public class ForallQuantifiedFormula
extends FolFormula
```

For-All-quantified first-order logic formula. Delegates to QuantifiedFormulaSupport for shared functionalities with other quantified formulas.

- Quantification such as for all and for some are know as universal and existential quantifiers accordingly.
- Languages such as Python can have this implemented in various ways, one way is to utilize external libraries that already exist to design OOP version for quantifiers, Propositional logic, conditional logic, etc. For java for example there are libraries such as ‘TweetyProject.
- By having this computational power we can address this focused section of logic much easier by providing the language being used with algorithmic and implementation details for quantification.
- We can also manually without using any libraries create such mechanism for different languages.
- We manually extended the P0 language with quantified expressions.
- Extensions added are:
    - Universal quantifier
    - Array builder
    - Set builder
    - Existential quantifier

Functional Requirements

Requires parsing and code generation for expressions of the following format:

- sorted := all i ∈ 0 .. N - 1 • a[i] ≤ a[i + 1]   or   sorted := ∀ i ∈ 0 .. N - 1 • a[i] ≤ a[i + 1]
- squares := [i ∈ 0 .. N - 1 • i × i]   or   squares := [i × i for i ∈ 0 .. N - 1]
- odds := {i ∈ 0 .. N - 1 | i mod 2 = 1 • i}   or   odds := {i for i ∈ 0 .. N - 1 if i mod 2 = 1}
- found := ∃ i ∈ 0 .. N - 1 • a[i] = x   or   found := some i ∈ 0 .. N - 1 • a[i] = x

Changes to the P0 Grammar

```
expression ::= simpleExpression
        {("=" | "≠" | "<" | "≤" | ">" | "≥" | "∈" | "⊆" | "⊇") simpleExpression} | quantifiedExpression

quantifiedExpression ::=
        ("all" | "some" | "∀" | "∃") identRange "•" expression |
        "[" { identRange "•" expression | expression "for" identRange } "]" |
        "{" { identRange "|" expression "•" expression | expression "for" identRange "if" expression } "}"

identRange ::= ident "∈" expression ".." expression
```

Changes to Scanner

- The scanner had to be modified to support:
  - `all`, ` ∀ ` - Universal quantifier symbols
  - ` ∃ `, `some` - Existential quantifier symbols
  - `•`, `for` - Quantifier expression symbol
  - `|`, `if` - Quantifier condition symbol
  - `{`, `}` - Set builder notation (curly braces)

Changes to Parser

- The parser was extended with the following grammar rules:

identRange ::= ident "∈" expression ".." expression

quantifiedExpression ::= ("all" | "some" | "∀" | "∃") identRange "•" expression |

"[" { identRange "•" expression | expression "for" identRange } "]" |

"{" { identRange "|" expression "•" expression | expression "for" identRange "if" expression } "}"

- The implementation requires backtracking, to determine the order of supplied arguments and to differentiate from existing notation, namely set definitions like {1, 2, 3}
- Backtracking was accomplished by
    - Recording the position of the scanner before the quantified expression is parsed
    - Checking weather a `•` or `for` is encountered first to determine the format of the expression
    - Rolling back the scanner and parsing in the order which was determined

Changes to Code Generator

- Expressions generate to loops instead of a sequence of instructions (unrolled loop)
- Expressions were considered to be generated as function calls, instead are generated as code of current calling function
- Iterated variable declaration and instantiation was considered to be done internally, instead the user is expected to declare iterated variables.
- Code generation was done in two or three code generation functions calls as follows:

Universal, Existential and Array Builder

- Upper half of the loop
- Generated internal expression code
- Lower half of the loop

Set Builder

- Upper half of the loop
- Generated condition code
- Upper half of decision IF statement
- Generated internal expression code
- Lower half of the loop

Testing & Documentation

- Manual testing for robustness and consistency
- Unit testing for future validation
    - SCTest.ipynb
    - P0ParsingTest.ipynb
    - P0TypeCheckingTest.ipynb
    - CGWatTest.ipynb
- Markdown for general descriptions
- In code comments for further explanation

Statistics

File	SC.ipynb	P0.ipynb	CGWat.ipynb	Total
Lines of Code	7	212	158	377
Lines of Documentation	2	16	36	54
Total Lines	9	228	194	431
Test Cases	1	18	8	27

Limitations

- Iteration range bounds have to be constants, to allow for verifying size at compile time, mainly for array assignments.
- Iterator variables have to be declared locally before using them in quantifier expressions.
- Set builder is limited to generating 32-bit bitsets.

Challenges

- Ensuring that the code we write doesn’t interfere with other existing code
- Generating the code, sometimes in 3 segments
- Backtracking to determine the order of operators and to process rules with same literals but different meaning
- Influence of existing code generation on implementation of the extension