ДНІПРОПЕТРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА

ЦЕНТР ЗАОЧНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ
КАФЕДРА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

Дипломна робота
за ОКР бакалавра

на тему: «Формальне доведення правильності програм на мові Javascript»

Виконав: студент IV курсу, групи ПК-11-1з

спеціальності 7.04030201 «Інформатика»

Шевцов Леонід Анатолійович

Керівник _____ доц., к. ф.-м. н. Хижа О. Л.
підпис

Рецензент _____ доц., к. ф.-м. н. Пасічник В. А.
підпис

м. Дніпропетровськ - 2015 року

РЕФЕРАТ

Дипломна робота: 42 с., 15 джерел, 4 додатки.

Об'єкт дослідження: програмні засоби для формального доведення правильності програм.

Мета роботи: створення програмного продукту для доведення деяких програм, написаних у імперативному стилі, на прикладі мови Javascript.

Одержані висновки та їх новизна: розроблено модульну програмну систему на мові функціонального програмування Clojure для формулювання та доведення умови правильності програм на підмові Javascript, а також перетворення та доведення логічно-алгебраїчних висловлювань.

Результати досліджень можуть бути застосовані для доведення програм на мові Javascript, для реалізації нових програмних засобів, що працюють з логічними висловлюваннями та їх правильністю, для доробки системи модулями підтримки інших мов програмування.

Перелік ключових слів: ПЕРЕВІРКА ПРАВИЛЬНОСТІ ПРОГРАМ, ФОРМАЛЬНА ПРАВИЛЬНІСТЬ ПРОГРАМ, МОВА НАЙСЛАБКІШОЇ ПЕРЕДУМОВИ, ЛОГІКА ВИСЛОВЛЮВАНЬ, ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ, CLOJURE, JAVASCRIPT.

ABSTRACT

The graduation research of the 4-year student L. Shevtsov (DNU, Center of Correspondence Education and Evening School, Department of Computer Technologies) deals with providing a formal proof of programs written in an imperative subset of the Javascript programming language, by developing a framework to process and prove logical expressions in the Clojure programming language.

The work is interesting to programmers in environments that demand high-quality code that is formally proven to be correct, to students studying formal proofs, to programmers working with logical expressions.

42 pages, 15 references, 4 supplements.

ЗМІСТ

Вступ 7
1. Постановка задачі 8
2. Теоретична база 9
2.1. Поняття про правильність програм 9
2.2. Формальний метод доведення 11
2.3. Мова wP - мова найслабкіших передумов 11
2.4. Доведення тверджень логіки висловлювань 15
2.5. Огляд мови програмування Clojure 17
2.6. Синтаксичний розбір програм 20
3. Огляд існуючих засобів доведення правильності програм 21
4. Розробка системи доведення програм 23
4.1. Структура програми, що доводиться 23
4.2. Логічні анотації 24
4.3 Розгляд алгоритму доведення 25
4.4 Синтаксичний розбір текста 26
4.5. Семантичний розбір 27
4.6. Переклад програми до логічного виразу на мові wP 27
4.7. Приведення твердження до логіки висловлювань 29
4.8. Добудова фактів о нерівностях 31
4.9. Доведення висловлювання методом перебору 33
4.10. Доведення висловлювання методом резолюції 33
4.11. Апарат перетворення алгебраїчних виразів 35
5. Результати 39
Висновки 40
Список використаної літератури 41
Додаток А. граматика мови вхідних текстів 43
Додаток Б. Вхідний текст продукту 46
Додаток В. Тести 68
Додаток Г. Приклади виконання програми 75

ВСТУП

Щодо програмування кажуть, что ми знаходимось на «мисливсько-збиральницькому» етапі його розвитку [8]. Незважаючи на величезний, небачений у інших галузях промисловості рост продуктивності засобів програмування, якість типового програмного продукта набагато гірша ніж якість інших продуктів, що також називають інженерними: будівель, мостів, автомобілів тощо. Кількість помилок, що трапляються у типовому введеному у експлуатацію програмному продукту, у таких продуктах була б неприпустима.

У наш час все більше керувальних відповідальностей перекладаються з апаратних систем на програмні. [8, 10, 12, 13] Коли програма керує двигуном автомобіля або літака або навіть ракети, медичним обладнанням, енергетичними станціями, якість програмного продукту є критичним показником. Тому перевірка правильності програм є невід'ємною частиною розробки, на менш важливою, як власно забезпечення функціоналу, бо який сенс у функціоналу, правильність якого нічим не забезпечена?

Формальне доведення правильності програм - це найбільш надійний спосіб доведення з відомих. Однак до того ж, цей спосіб найбільш важкий, та потребує не тільки математичної підготовки з боку програміста, але й програмних засобів. Для багатьох мов не існує засобів формального доведення, або вони працюють надто повільно.

Ця робота намагається розв'язати обидві проблеми. Завдяки використанню функціонального стилю програмування та мови з доброю підтримкою паралелізації наведений продукт буде швидким. Завдяки модульному підходу продукт можна пристосувати до будь-якої імперативної мови програмування.

1. ПОСТАНОВКА ЗАДАЧІ

Вивчити теоретичні заснови формальної правильності програм, доведення висловлювань пропозиційної логіки, синтаксичного розбіру мов програмування.

Обрати корисну, але обмежену підмножину мови Javascript, яку можна використати для доведення алгоритмів. Задати структуру анотаційних коментарів, що задаватимуть перед- та післяумову програми, а також варіанти та інваріанти циклів. Це ставитиме вхідну мову програми.

Описати граматику вхідної мови та реалізувати парсер з вхідної мови. Написати транслятор з вхідної мови до мови найслабкішої передумови. Задати тезу о правильності програми у вигляді логічного висловлювання. Реалізувати доведення логічних висловлювань за допомогою метода резолюції. Програмну частину виконувати на мові Clojure.

Перевірити правильність програмного продукту, використовуючи модульні тести. Навести пакет тестов, що задовольняє вимоги правильності до продукту, що сам перевіряє правильність інших програм

Забезпечити подальший розвиток системи. Пильнуватись про модульність продукту заради розширюваності, та про функціональність заради подальшої паралелізації.

Обрати декілька алгоритмів для демонстрації продукту, розробити до них анотації, та показати доведення.

2. ТЕОРЕТИЧНА БАЗА

2.1. Поняття про правильність програм

Що взагалі значить твердження, що програма “правильна”? Без удавання до філософії та езотеріки, у практичному сенсі програма називається правильною, якщо вона призводить до очікованого результату. Помітимо, що будь-яка програма може бути правильною тільки в межах її контракту, в межах вимог, що поставив до неї програміст. [1]

Контракт програми існує практично до будь-якої програми. Є різні шляхи задання цього контракту.

Неявний контракт. У такому випадку програма правильна, якщо користувач вважає його правильною. Звісно, перевірка правильності також проходить у неявній формі.

Контракт у формі тестів. Тести - це виконання програми на певному наборі вхідних даних та перевірка результатів виконання. Тести можуть виконуватись у автоматичному режимі, та перевіряти правильність програми після кожних змін.

Основний недолік тестів - вони перевіряють програму на підмножині вхідних даних, часто ця підмножина не перевіряє всі можливі шляхи виконання програми. Існує поняття «крайових випадків» - вхідних даних, на яких програма суттєво змінює поведінку. Якість тестів залежить від того, наскільки добре програміст передбачив ці крайові випадки. Тому навіть у програмних пакетах з добрим покриттям модульними тестами знаходять помилки. У більшості випадків практично неможливо перевірити

правильність програми на всіх вхідних значеннях, та часто це неможливо навіть теоретично.

До того ж, тести мають ту ж алгоритмічну складність, що й програма, що тестується. Якщо програма надто повільна, тести також повільні.

Формальний контракт задає обмеження на вихідні значення програми та на залежність вихідних параметрів від вхідних, у вигляді логічних формул.

Перевірка формального контракту не потребує запуску програми. Вона відноситься до методів статичного аналізу програм. Формальним контрактом можна дійсно покрити всю множину вхідних даних програми. Формальне доведення автоматично покриває крайові випадки та всі гілки програми.

Головним недоліком формального доведення правильності є необхідність додаткового програмування формального контракту. Це потребує від програміста математичної підготовки.

До того ж, на відміну від модульних тестів, формальний контракт потребує додаткового програмного апарату для перевірки програм, та теоретичної бази.

Формальне доведення правильності корисне у системах з надто великою ціною помилки.

У межах цієї роботи розглядається формальне доведення імперативних програм.

2.2. Формальний метод доведення

Формальний метод доведення правильності програм використовується у галузях з високими вимогами до якості програмного забезпечення. Це аерокосмічна промисловість, медицина, автомобільна промисловість, енергетика, тощо. Формальний метод застосовують інженери у таких компаніях, як NASA, Airbus тощо.

Для формального доведення імперативної програми до неї виставляється пара умов - це передумова та післяумова. Передумова накладає обмеження на вхідні дані програми; вона зазначає множину можливих вхідних даних. Післяумова накладає обмеження на вихідний результат програми при будь-яких можливих вхідних даних.

Задачею формального методу є логічне доведення того, що для будь-яких вхідних даних, що задовольняють передумову, вихідні дані будуть задовольнять післяумову.

Треба зазначити, що формальний метод дає якісний результат тільки якщо якісно були задані перед- та післяумова; у тривіальному випадку якщо післяумови та передумови немає, формальний метод може перевірити, що програма закінчується на будь-яких вхідних даних, та в ній немає зациклення.

2.3. Мова wP - мова найслабкіших передумов

Яким чином можливо довести, що післяумова задовольняється? Для цього у 1969 році Хоар [14] сформулював набір перетворень, що роблять з імперативної програми логічне твердження. Цей набір перетворень називається мовою найслабкіших передумов.

У мові найслабкіших передумов гіпотеза про правильність програми має вигляд

$$\{Q\}\ S\ \{R\}$$

де Q - предикат передумови, R - предикат післяумови, та S - команда. Цей запис означає «якщо стан на початку команди задовольняє Q, то команда S закінчується у кінечний час у стані, що задовольняє R».

Мова найслабкішої передумови перетворює цей вираз у вигляд

$$Q \Rightarrow wp(S, R)$$

де wp(S, R) - це вираз для *найслабкішої передумови* для R після S, тобто це найбільш слабка умова на початковий стан перед командою S така, щоб кінцевий стан гарантовано задовольняв R.

Наведемо функції wp для виразів нашої імперативної мови.

Для пустого твердження

$$wp(\text{<noop>}, R) = R$$

Тобто пуста команда має таку ж передумову, як післяумову.

Для присвоєння

$$wp(\text{<var> = <expr>}, R) = R/\text{<var>}/\text{<expr>}/$$

Тобто найслабкіша передумова оператора присвоювання - це післяумова, у якій кожне посилання на змінну заміняється на значення виразу, що присвоюється.

Для послідовності

$$wp(S1; S2; \dots Sn, R) = wp(S1, wp(S2, wp(S3, \dots , wp(Sn, R)\dots)))$$

Тобто передумова для послідовності команд - це рекурсивна передумова для кожної з команд та передумови попередньої команди (при чому команди розглядаються з кінця).

Для умовного оператору (гілки)

wp(IF **P** THEN **S1** ELSE **S2**, **R**) =
P => wp(**S1**, **R**) AND (NOT **P)** => wp(**S2**, **R**)

Тобто для обох можливих значень умови та обох гілок потрібно перевірити відповідну передумову

Для циклу

Цикл - це перша конструкція нашої імперативної мови, що може призвести до нескінченної роботи програми, тобто зациклення, та для циклу потрібно додатково довести, що він закінчується. Найслабкіша передумова для циклу дещо складніша та потребує декількох додаткових означень.

Інваріант циклу - це твердження, що є вірним на початку та наприкінці кожної ітерації циклу. Інваріант спрощує доведення тим, що зазвичай саме з нього випливає післяумова.

Варіант або обмежувальна функція - це числова функція, яка є додатньою та зменшується після кожної ітерації циклу.

Таким чином, найслабкіша передумова має задовольняти інваріант та змешення варіанту:

wp(**INV**, **BOUND**, WHILE(**P**) **S**, **R**) =
INV AND
(**P** AND **INV**) => wp(**S**, **INV**) AND
(**P** AND **INV**) => **BOUND**(n-1) > **BOUND**(n) > 0 AND
(NOT **P** AND **INV**) => **R**

Тут BOUND(i) - це значення обмежувальної функції після і-ї ітерації; на практиці порівнюється значення на початку та кінці ітерації.

Можна побачити, що для кожного цикла у програмі, що доводиться, програміст має вказати належний інваріант та обмежувальну функцію.

Для визову процедури

wp({**Qf**}**F**{**Rf**}, **R**) = **Qf** AND (**Rf** => **R)**

Кожна процедура, що викликається, має у свою чергу мати перед- та післяумову. Тоді перед- та післяумови процедури розглядаються як проміжні умови для загального виразу.

2.4. Доведення тверджень логіки висловлювань

Після трансляції програми до найслабкішої передумови отримуємо гіпотезу о правильності програми, а саме

$$Q \Rightarrow wp(\text{програма}, R)$$

де Q - передумова, R - післяумова.

Все, що лишилось - це довести цю гіпотезу. Обраний для цієї роботи метод доведення працює з твердженнями логики висловлювань. Для доведення тверджень логики висловлювань обрано два підходи.

По-перше, твердження можна доводити перебором всіх можливих комбінацій фактів. Цей підхід добре працює при невеликій кількості фактів, до 15-20 - складність методу перебору експоненційна від кількості фактів. Метод перебору також називають методом таблиці істинності.

Якщо ж фактів більше ніж 15-20, твердження доводиться методом резолюції. Метод резолюції - це відомий метод доведення від зворотнього.

Для метода резолюції твердження зводиться до клаузальної нормальної форми. Клаузальна нормальна форма - це еквівалентне твердження, що є кон’юнкцією диз’юнкцій фактів або їх заперечень.

Для перетворення до клаузальної нормальної форми над твердженням роблять такі перетворення:

1. Замінюють еквівалентність
   A<=>B == A=>B AND B=>A

2. Замінюють імплікацію A=>B == (NOT A) OR B

3. Спускають заперечення до фактів законами де Моргана та подвійного заперечення:
   NOT (A OR B) == (NOT A) AND (NOT B)
   NOT (A AND B) == (NOT застосовуєтьсяA) OR (NOT B)
   NOT NOT A == A

4. Спускають диз'юнкцію законом дистрибутивності:
   A OR (B AND C) == (A OR B) AND (A OR C)

Після цього групуванням кон'юнктів та диз'юнктів отримуємо клаузальну нормальну форму.

Метод резолюції полягає в наступному:

1. Взяти клаузальну форму твердження, оберненого від того, що доводиться.

2. Перебираються всі пари кон'юнктів та до них застосовується правило резолюції:

   A1 OR A2 OR … OR An OR B1 OR … OR Bn) AND
   (NOT A1 OR NOT A2 OR C1 OR … OR Ck) ==
   B1 OR Bn OR … OR C1 OR … OR Ck

   Нові кон'юнкти також додаються до загальної множини

3. Якщо серед кон'юнктів є два, що повністю заперечують один одного, то вся форма еквівалентна хибності.

4. Хибність оберненого твердження доводить істинність твердження, що доводиться.

2.5. Огляд мови програмування Clojure

Clojure - це новітня мова програмування, що була створена у 2007 році Річем Хікі, за впливом таких мов, як Common Lisp, Erlang, Haskell, ML, Prolog, Scheme, Java, Ruby тощо. Мова Clojure була обрана для виконання цієї роботи тому, що вона стоїть на перетині декількох гілок розвитку мов програмування та поєднує цікаві досягнення з ліспу, функціональних мов програмування, світу корпоративних платформ, та актуального у наш час паралельного програмування. [9]

Перше, что потрібно знати про Clojure - це лісп.

```
(print "Hello, world!")
```

Від ліспів Clojure успадкувала потужний апарат макросів, стислий синтаксис та єдине уявлення коду та даних.

Мови родини Лісп часто вважаються цікавими лише в академічному сенсі, бо вони є повільними у виконанні або не мають великий та різноманітний відбір бібліотек. Що робить мову Clojure життєздатною - це те, що вона спирається на віртуальну машину Java, тобто JVM, та глибока інтегрується з нею - наприклад, виклик функції Clojure не є емульованим або інтерпретованим, а прямо перекладається у виклик методу Java. За бажанням у програмах на мові Clojure можна досягти тієї ж самої швидкодії, яка доступна програмам на мові Java.

Як у типовому ліспу, у Clojure функції - це об’єкти першого порядку, їх можна динамічно створювати, передавати параметрами у інші функції, та інше. Clojure - це мова класу лісп-1, тобто простір імен функцій та даних у Clojure один й той же, тобто функції та дані взаємозамінні. У цьому сенсі Clojure є функціональною мовою програмування.

```
; Множина у ролі функції
(#{1 2 3} x)
```

Але функціональність Clojure лежить ще глибше. Фундаментальним рішенням архітектора мови був фокус на роботі з іммутабельними типами даних. Тому всі базові типи даних у Clojure є іммутабельними. Це: списки, вектори, відображення (словники), множини. Базові функції Clojure не змінюють ці структури, а повертають результат у нових екземплярах. (Помітимо, що, як було сказано раніше, за бажанням та потребою можна працювати й зі звичайними, мутабельними масивами та іншими структурами мови Java). Також замість класичних назв функцій лісп, які відомі за свою неочевидність, стандартна бібліотека Clojure застосовує назви з лексикону функціональних мов - map, reduce, filter тощо.

У досвідченого програміста можуть виникнути сумніви щодо практичності копіювання структур даних, як по часу виконання, так і по витратам оперативної пам’яти. Ці проблеми були відомі архітектору мови, та їм протистоїть декілька особливостей реалізації. По-перше, оскільки жодна зі структур не може бути змінена, то при створенні нових структур можна використати частини старих (так званий structural sharing). Наприклад: додавання елементу до списку не будує суцільно новий список, а будує деяку структуру, що містить новий елемент та посилання на старий список, зберігаючи старий список для тих функцій, що мають на нього пряме посилання. По-друге, більшість стандартних функцій є лінивими, та нові елементи створюються тільки тоді, коли вони потрібні, що зберігає і час обчислення, і пам’ять.

```
; лінiва версiя швидкого сортування Хоара
(defn sort-parts [work]
  (lazy-seq
    (loop [[part & parts] work]
      (if-let [[pivot & xs] (seq part)]
        (let [smaller? #(< % pivot)]
          (recur (list*
                   (filter smaller? xs)
                   pivot
                   (remove smaller? xs)
                   parts)))
        (when-let [[x & parts] parts]
          (cons x (sort-parts parts)))))))
```

Сукупність іммутабельних структур та зручної стандартної бібліотеки для роботи з ними дає можливість занотувати алгоритми з лаконічністю та експресивністю, невідомими мовам з родини C.

Змінні у мові Clojure також є іммутабельними, тобто вони є посиланнями на значення, та таке посилання може бути змінено, спрямоване на інше іммутабельне значення. Взагалі у програмах на мові Clojure змінні не є обов’язковими, частіше застосовують константні посилання у межах функції.

Завдяки іммутабельним структурам, Clojure має зручну систему підтримки паралельних обчислень, що реалізована на рівні самої мови. Для паралельних обчислень у Clojure передбачено *атоми, агенти, та транзакції*.

Атом - це змінна, значення якої розділено між потоками обчислень. Оскільки єдина операція зміни над атомом - це повна його заміна новим значенням, то цілісність даних у межах атома гарантована.

Агент - це функція з чергою задач, що виконується для кожного набору аргументів з черги. Таким чином можно легко відвантажувати деякі частини обчислень до асинхронного потоку.

Транзакція - це відокремлена частина коду, що читає та змінює деякі спеціально зазначені посилання. Значення зчитуються на початку тразакції. Якщо наприкінці транзакції виявляється, що поки транзакція була відкрита, значення посилань були змінені іншим потоком, транзакція перезапускається, вже з новими значеннями посилань. (Помітимо, що це спирається на те, що код у транзакції на має побічних дій.) Транзакції диригують асинхронні дії над спільним ресурсом.

2.6. Синтаксичний розбір програм

Задача синтаксичного розбору різноманітних мов програмування є добре розв’язаною на наш час, та зазвичай для розбору потрібно лише задати правильну граматику до існуючого синтаксичного аналізатора.

Для цієї работи був обрано синтаксичний аналізатор ANTLR. ANTLR приймає на вхід опис граматики та генерує набір класів Java, що можна застосувати для розбіру програм на заданій мові. Завдяки зручної інтеграції Clojure та Java ті ж самі класи можна було підключити до основного коду продукту на мові Clojure.

Граматика - це набір правил, що описують всі можливі речення мови. Для потреб цієї роботи загальна структура програми описується через набор команд, які в свою чергу описуються через алгебраїчні вирази, ключові слова та примитиви (див. додаток).

3. ОГЛЯД ІСНУЮЧИХ ЗАСОБІВ ДОВЕДЕННЯ ПРАВИЛЬНОСТІ ПРОГРАМ

Формальні методи доведення правильності програм застосовуються багатьма компаніями. Існують як системи, що перевіряють правильність виконання машинного коду, спираючись на модель центрального процесору. так і системи, що перевіряють код високого рівня. Є системи, що перевіряють код на наявність семантичних помилок таких, як незвільнені посилання, індекси масивів поза межами, та ніколи не відвідувані гілки - навіть без явно зазначених анотацій до програми.

З розглянутих засобів доведення можу виділити Frama-C - систему статичного аналізу програм на C, що включає до себе і систему перевірки перед- та післяумови; вбудовані засоби перевірки правильності мови ACL2; системи Microsoft SLAM та Microsoft Z3, що використовуються для перевірки апаратних драйверів; систему CAVEAT, що розроблена у Франції для ядерних реакторів; систему Astree для перевірки на відсутність помилок часу виконання.

Мова Javascript не звертає велику увагу з боку спеціалістів з формального доведення правильності програм. Це можна пояснити відсутністю чітко зазначеного стандарту мови [12], а також її динамічним розвитком - наступна версія Javascript, ECMAScript 6, містить суттєві відмінності навіть у базовому синтаксисі мови.

Щодо засобів статичного аналізу - для мови Javascript існує парсер та інтерпретатор на мові Haskell, що називається LambdaJS [12], але він на зачіплює питання доведення правильності.

Отже, при розробці продукту довелось спиратись на подібні засоби для інших мов, наприклад мові ACL2 [15], а також анотацій до псевдомови, використаній у [1].

Для доведення логічних висловлювань на мові Clojure існує програмний пакет core.logic [5]. Цей пакет добре підходить для логічного програмування, розв'язку логічних рівнянь та задовольняння реляційних схем. Але при доведенні програм він виявився незручним. До того ж, на вхід core.logic має бути подана клаузальна нормальна форма висловлювання. Під час розробки продукту виявилось, що перетворення гіпотези о правильності до клаузальної нормальної форми - це більша частина доведення, тому замість написання адаптеру до core.logic було написано свій апарат доведення.

Щодо необхідності створення нової системи можна зазначити наступне. По-перше, написання системи формального доведення на мові Clojure із застосуванням функціонального стилю програмування, дає можливість у майбутньому легко зробити процес доведення мультипотоковим або навіть розподіленим. Швидкодія доведення є одним з слабких місць формальних систем, та розподілення праці на декілька ЕОМ є доцільним та актуальним на наш час.

По-друге, модульний підхід робить окремі частини продукту корисними на самоті. Апарат доведення виразів можна застосовувати для доведення будь-чого. Апарат перетворення алгебраїчних виразів стане до нагоди у прикладній математиці.

По-третє, ще на ранньому етапі розробки було вирішено робити систему якнайменш залежною від мови вхідної програми. Написанням додаткового синтаксичного модуля можна підключити до системи інші імперативні мови.

4. РОЗРОБКА СИСТЕМИ ДОВЕДЕННЯ ПРОГРАМ

Програмний продукт на вхід отримує вхідний текст програми, що доводиться, а на виході видає одне з трьох повідомлень - “PROVED”, коли програма доведена, “FAILED TO PROVE”, якщо продукт не може довести програму за нестачею обмежень, або “ERROR”, якщо вхідна програма не відповідає вимогам до структури та синтаксису.

Крім того, програмний продукт виводить повідомлення щодо роботи алгоритму у файл журналу або до терміналу.

4.1. Структура програми, що доводиться

У межах цієї роботи розглядаються програми, написані на деякій підмножині мови Javascript - кожна програма, що може бути доведеною, є коректною програмою Javascript, але не навпаки.

Основне обмеження - програма не може визначати ані викликати функції, включаючи вбудовані функції Javascript. Все тіло програми вважається єдиною імперативною послідовністю команд. На вхід програма отримує деякі змінні, визначені глобально, а результатом виконання програми вважаються значення деяких інших глобальних змінних.

До того ж, з типів даних Javascript програми можуть використати тільки цілі числа - числа з плаваючою комою, об’єкти, масиви, символьні рядки не підтримуються.

Приклад програми для доведення:

```
// Програма обчислює цілу частину та залишок від ділення X на Y
/*
  PRE:
      x >= 0;
      y > 0
  POST:
      x == q * y + r;
      r < y;
      q >= 0;
      r >= 0;
*/

r = x;
q = 0;

/* INV: x == q * y + r
   BOUND: r+1
 */
while (y<=r) {
    r = r-y;
    q = 1+q;
}
```

4.2. Логічні анотації

Для уможливлення доведення програма має бути доповнена анотаціями, що зазначають умови на змінні у критичних точках виконання.

Звісно, будь-яка програма, що доводиться, повинна мати перед- та післяумову, що записуються у спеціальному коментарі на початку програми.

Крім перед- та післяумов для кожного з циклів програми має бути визначений інваріант (умова, що незмінно набуває значення “істина” після кожної

ітерації) та варіант, або обмежувальна функція (невід'ємна функція, що монотонно зменшується після кожної ітерації).

Всі ці умови записуються у вигляді алгебраїчних виразів Javascript. Для зручності окрім операторів Javascript можна використати такі оператори, як: оператор імплікації =>, оператор логічної еквівалентності <=>.

4.3 Розгляд алгоритму доведення

Отже, визначивши структуру вхідних даних, розглянемо шлях від вхідного тексту до результату доведення.

Доведення програми є ланцюгом перетворень, що виконуються над вхідним текстом. Кожний з етапів, у свою чергу, перетворює дані з одної форми до іншої.

- Синтаксичний розбір текста

- Семантичний аналіз, перетворення до внутрішньої структури

```
(defn parse-program-from-filename [filename]
  (try
    (-> filename
        slurp
        parse
        parse-tree->program)
    (catch FileNotFoundException _
      (log/errorf "File not found: %s" filename))
    (catch ParseCancellationException _
      (log/errorf "Failed to parse: %s" filename))))
```

- Переклад програми до логічного висловлювання на мові wP

- Доведення висловлювання за використанням метода резолюції

```
(defn prove-program [program]
  (let [correctness-hypothesis (program-correctness-hypothesis program)

        _ (log/spyf "Program correctness hypothesis:\n%s"
                    correctness-hypothesis)

        [comparison-axioms factualized-hypothesis]
          (factualize-comparisons correctness-hypothesis)

        _ (log/spyf "After reducing comparisons into facts:\n%s"
                    (simplify-expression factualized-hypothesis))
        _ (log/spyf "Axioms about comparisons:\n%s"
                    (s/join "\n" (map str comparison-axioms)))]

    (resolution-prover comparison-axioms factualized-hypothesis)))

(defn -main [filename & args]
  (if-let [program (parse-program-from-filename filename)]
    (println (if (prove-program program) "Proved" "Disproved"))
    (println "Error")))
```

4.4 Синтаксичний розбір текста

Для синтаксичного розбору було застосовано генератор компіляторів ANTLR. Була написана вхідна граматика для мови доведень (підмножини Javascript, доповненої анотаціями) (див. додаток А, та за допомогою ANTLR граматика перетворена до системи класів Java, які в свою чергу перетворюють вхідний код програм до лексичного дерева.

Отже, на виході синтаксичного розбору маємо лексичне дерево, що відповідає вхідному тексту. У випадку, якщо текст не співпадає з граматикою мови, розбір повертає ошибку та подальші етапи доведення не виконуються.

4.5. Семантичний розбір

Семантичний аналіз - це перетворення лексичного дерева до більш зручної та наочної структури. У процесі розробки я виявив, що без етапу семантичного аналізу робота з лексичним деревом напрямки призводить до складного та крихкого коду.

Програма у внутрішньому уявленні має вигляд:

```
{:precondition expression, :postcondition expression, :commands sequence-command}
```

Під час семантичного аналізу у програмі формуються перші алгебраїчні вирази, у тому числі перед- та післяумови.

4.6. Переклад програми до логічного виразу на мові wP

Для перетворення програми до логічного виразу алгоритм починає з післяумови (що є логічним виразом) та “загортає” її як матрьошку у відповідну структуру для кожної з команд програми.

```
(defn command-wp [postcondition command]
  (let [c-type (:type command)
        c-params (:params command)]
    (case c-type
      :noop postcondition
      :sequence (reduce command-wp postcondition (reverse c-params))
      :assign (let [[identifier value] c-params]
                (replace-identifier-in-expression identifier
                                                  value
                                                  postcondition))

      :if
        (let [[predicate if-command else-command] c-params]
          (conditional-wp predicate if-command else-command postcondition))
```

```
      :while
        (let [[predicate loop-command invariant bound] c-params]
          (loop-wp (hash command)
                   predicate
                   loop-command
                   invariant
                   bound
                   postcondition)))))
```

Для присвоєнь:

```
(defn replace-identifier-in-expression
  [identifier value construct]
  (cond
    (expr? construct)
    (expr-map
      (partial replace-identifier-in-expression identifier value) construct)

    (identifier? construct)
    (if (= construct identifier)
      value
      construct)

    :else construct))
```

Для умовного оператору:

```
(defn conditional-wp [predicate if-command else-command postcondition]
  (expr :and
        (expr :implies
              predicate
              (command-wp postcondition if-command))
        (expr :implies
              (expr :not predicate)
              (command-wp postcondition else-command))))
```

Для циклу:

```
(defn loop-wp
  [loop-id loop-condition loop-command invariant bound postcondition]
  (let [bound-variable (str "bound_" loop-id)]
```

```
(log/spyf "Loop condition %s"
          (reduce (partial expr :and)
                  [invariant
                   [:implies [:and loop-condition invariant]
                             (command-wp invariant loop-command)]
                   [:implies [:and loop-condition invariant]
                             [:> bound 0]]
                   [:implies [:and loop-condition invariant]
                             (replace-identifier-in-expression
                               bound-variable
                               bound
                               (command-wp (expr :< bound bound-variable)
                                           loop-command))]
                   [:implies [:and [:not loop-condition] invariant]
                             postcondition]])))
```

Після перебору всіх команд з останньої по першу, отримуємо найслабкішу передумову програми. Щоб записати твердження про правильність програми, тепер достатньо поєднати вхідну передумову та найслабкішу передумову знаком імплікації.

4.7. Приведення твердження до логіки висловлювань

На останньому етапі твердження має вигляд дерева, кожною вершиною якого є оператор, а спадками - операнди. Нагорі дерева містяться логічні оператори, але атомами для логічних операторів є алгебраїчні відношення: рівності та нерівності. У свою чергу рівності та нерівності звертаються до арифметичних виразів. У межах цієї роботи вирази можуть використати операції додавання, віднімання, та множення.

(AND (== X 1) (IMPLIES (> Y 0) (> (+ X Y) 1)))

Є декілька підходів до доведення виразів, предикатами яких є алгебраїчні відношення. Обраний підхід зводить відношення до логічних фактів ще до

початку процесу доведення. Для цього кожне відношення приводиться до обраного канонічного виду, а саме:

1. всі доданки переносяться в ліву частину вираза, вираз приводиться до багаточлену;

2. доданки впорядковуються по ступеню;

3. доданки нормалізуються: діляться на множник при найпершому доданку;

4. скалярний доданок, якщо такий є, переноситься до правої частини.

5. нерівності зі знаками >=, <= та != виражаються через >, <, = для спощення подальших етапів (наприклад, x>=y перекладається до NOT (x<y)).

6. після канонізації вираз замінюється на логічну змінну.

Канонізація забезпечує те, що еквіваленті вирази замінюються на одну й ту ж змінну, наприклад:

$$x - y > 0$$
$$x > y$$
$$y+1 < x+1$$

тощо замінюються на змінну «-y+x>0». Семантичний зміст виразу при цьому втрачається, тобто подальший метод доведення розглядає змінну як незалежну, що набуває значення «істина» або «хибність». Наприклад, з точки зору доведення змінні «x>0» та «x<0» можуть водночас бути істинні.

4.8. Добудова фактів о нерівностях

У більшості випадків втрата цього змісту робить доведення неможливим. Тому під час заміни нерівностей на змінні висловлювання добудовується залежностями між всіми можливими нерівностями.

Наприклад, якщо у висловлювання трапились вирази x>0 та x<0, вираз добудується імплікаціями

«x>0» => NOT «x<0»
«x<0» => NOT «x>0»

(нагадаю, що на цьому етапі нерівності вже мають вигляд логічних змінних)

До того ж, якщо також трапився вираз «x>5», залежність з ним також додається до імплікацій:

NOT «x>0» => NOT «x>5»
«x<0» => NOT «x>5»
«x>5» => «x>0»
«x>5» => NOT «x<0»

Разом з цими імплікаціями метод може перевіряти складні проблемі інтервального характеру. Наприклад, без імплікацій вираз

«x<0» AND «x>5»

не є хибним, але з добудовою

«x<0» AND «x>5» AND («x<0» => NOT «x>5») AND («x>5» => NOT «x<0»)

вираз легко доводиться звичайним методом резолюції або будь-яким іншим.

Той самий підхід можна використати та й в інших випадках для впровадження додаткового змісту до метода резолюції.

У випадку, якщо для доведення висловлювання не вистачає залежності між деякими змінними, їх можна вручну додавати до передумови програми.

```
(defn comparison-implications
  "Input:
   2 comparisons: lside ox x, and lside oy y, where ox,oy - one of [>, <],
   x, y - scalars
   Output:
   if there is an implication from lside ox x being true or false,
   to lside oy y being true or false, then
   returns list of such implications, otherwise empty list
   "
  [lside [[ox x] [oy y]]]
  (let [exprx (str lside (name ox) x)
        expry (str lside (name oy) y)]
    (remove false?
            (case [ox oy]
              [:> :<] [(and (>= x y) (expr :implies exprx [:not expry]))
                       (and (< x y) (expr :implies [:not exprx] expry))]
              [:> :>] [(and (> x y) (expr :implies exprx expry))
                       (and (< x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :<] [(and (< x y) (expr :implies exprx expry))
                       (and (> x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :>] [(and (<= x y) (expr :implies exprx [:not expry]))
                       (and (> x y) (expr :implies [:not exprx] expry))]
              [:> :==] [(and (>= x y) (expr :implies exprx [:not expry]))
                        (and (< x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :==] [(and (<= x y) (expr :implies exprx [:not expry]))
                        (and (> x y) (expr :implies [:not exprx] [:not expry]))]
              [:== :==] [(and (not= x y) (expr :implies exprx [:not expry]))]
              [:== :>] [(and (> x y) (expr :implies exprx expry))
                        (and (<= x y) (expr :implies exprx [:not expry]))]
              [:== :<] [(and (< x y) (expr :implies exprx expry))
                        (and (>= x y) (expr :implies exprx [:not expry]))]
              ))))
```

4.9. Доведення висловлювання методом перебору

```
(defn evaluation-method
  ([expression identifiers bindings]

  (if-let [identifier (first identifiers)]
    (every? true?
            (map #(evaluation-method
                   (replace-identifier-in-expression identifier % expression)
                   (rest identifiers)
                   (assoc bindings identifier %))
                 #{true false}))
    (or (false? (evaluate-constants expression))
        (and (log/spyf "Counter-example: %s" bindings) false))))

  ([expression identifiers] (evaluation-method expression identifiers {})))
```

4.10. Доведення висловлювання методом резолюції

Нарешті, маємо логічне твердження першого порядку, яке доводимо класичним методом резолюції, а саме:

- формулюємо антитезу
- приводимо до клаузальної нормальної форми
- розбиваємо на кон’юнкти
- доки немає пустого кон’юнкту, обираємо два кон’юнкту та застосуємо правило резолюції
- якщо невикористаних кон’юнктів немає - висновок “неможливо довести”
- якщо отримали пустий кон’юнкт - висновок “доведено”

```
(defn derive-resolutions [clauses]
  (clause-set (for [c1 clauses c2 clauses :when (not= c1 c2)]
    (let [[yes1 no1] c1
          [yes2 no2] c2
          all-yes (set/union yes1 yes2)
          all-no (set/union no1 no2)]
      (with-meta
        [(set/difference all-yes all-no) (set/difference all-no all-yes)]
        {:parents [c1 c2]})))))

(defn empty-clause? [[yes no]]
  (and (empty? yes) (empty? no)))

(defn resolution-method [clauses iteration]
  (if (> iteration 1000)
    :exceeded-iteration-limit
    (if-let [best-clause
             (log-clause "Best derived clause"
                         (first (set/difference (derive-resolutions clauses)
                                                clauses)))]
      (if (empty-clause? best-clause)
        :proved
        (recur (conj clauses best-clause) (inc iteration)))
      (do
        (log/debug "Failed to disprove: " (clauses->str clauses))
        :failed-to-prove))))
```

У метода резолюції є дуже важливий недолік - його складність росте експоненційно з кожним оператором AND, що вкладений у оператор OR - бо кожен такий оператор буде розкладений у два за законом дистрибутивності.

4.11. Апарат перетворення алгебраїчних виразів

Додатково хочу розглянути апарат роботи з алгебраїчними виразами, що був створений під час розробки програмного продукту та суттєво спростив багато етапів алгоритму.

У системі алгебраїчні вирази мають форму

(оператор [параметри])

де параметри - це масив з інших алгебраїчних виразів, ідентифікаторів, чисел. Для зручності нові вирази можна створювати функцією expr:

```
(expr :and [:> "x" 0] [:< "y" 3])
```

Отже, функція expr може створювати вирази будь-якого ступіню вкладеності.

Під час роботи вирази дуже часто потрібно перетворювати до еквівалентної форми. Наприклад, приведення до кон’юнктивної нормальної форми - це результат двох перетворень:

- зведення операції NOT до логічних атомів за допомогою законів де Моргана:

  NOT (A AND B) <=> NOT A OR NOT B
  NOT (A OR B) <=> NOT A AND NOT B

- “розгортання” операції OR за законом дистрибутивності.

  A OR (B AND C) <=> (A OR B) AND (A OR C)

Ще один приклад - спрощення виразів за допомогою таких тотожностей, як

NOT (NOT A) <=> A

тощо.

Виникла ідея використати подібну математичну форму запису для завдання перетворень. Система функцій для перетворень виявилась дуже потужною та застосовується скрізь продукту для спрощення та приведення як логічних, так і аріфметичних виразів.

```
(algebra/expr-match [:+ "x" 1] (algebra/expr :+ "a" 1))
=> {"x" "a"}

(algebra/expr-match [:+ "x" 1] (algebra/expr :+ "a" 2))
=> false

(algebra/expr-match [:and "x" true] (algebra/expr :and "a" true))
=> {"x" "a"}

(algebra/expr-transform [:not [:and "x" "y"]]
                        [:or [:not "x"] [:not "y"]]
                        (algebra/expr :not [:and "a" "b"]))
=> (algebra/expr :or [:not "a"] [:not "b"])

(algebra/expr-transform [:not [:not "x"]]
                        "x"
                        (algebra/expr :not [:not [:not [:not "x"]]]))
=> "x"))
```

Завдяки цій зручній системі реалізована система спрощення логічних виразів. Пропоную звернути увагу на те, як стисло та наочно задані правила спрощення, та як легко їх застосувати, або додати нові.

```
(defn- simplify-expression-once [expression]
  (let [simplification-rules {; Logic rules
                              [:implies true "x"]      "x"
                              [:implies "x" true]      true
```

```
                              [:implies false "x"]     true
                              [:implies "x" false]     [:not "x"]
                              [:not [:not "x"]]        "x"
                              [:or "x" false]          "x"
                              [:or "x" true]           true
                              [:and "x" false]         false
                              [:and "x" true]          "x"
                              [:or "x" "x"]            "x"
                              [:and "x" "x"]           "x"
                              [:or "x" [:not "x"]]     true
                              [:and "x" [:not "x"]]    false
                              [:and "x" [:or "x" "y"]] "x"
                              [:or "x" [:and "x" "y"]] "x"
                              ; Arithmetic rules
                              [:+ "x" 0]               "x"
                              [:* "x" 0]               0
                              [:* "x" 1]               "x"
                              [:+ [:- "x" "y"] "y"]    "x"
                              [:- "x" "x"]             0
                              [:- "x" 0]               "x"
                              [:/ "x" 1]               "x"
                              [:/ 0 "x"]               0
                              [:== "x" "x"]            true
                              [:<= "x" "x"]            true
                              [:>= "x" "x"]            true
                              [:!= "x" "x"]            false
                              [:< "x" "x"]             false
                              [:> "x" "x"]             false
                              }]
    (reduce
      (fn [expression [p-from p-to]] (expr-transform p-from p-to expression))
      expression simplification-rules)))
```

Для повноцінного спрощення необхідно також замінять константні вирази на константи. Для цього виконується обхід дерева виразу.

```
(defn evaluate-constants
  "Find constant sub-expressions in expression and evaluate them"
  [expression]
  (let [expression-operators {:and     #(and %1 %2)
                              :or      #(or %1 %2)
                              :not     not
```

```
                              :implies #(or (not %1) %2)
                              :+       +
                              :-       -
                              :*       *
                              :/       /
                              :==      =
                              :!=      not=
                              :>       >
                              :<       <
                              :>=      >=
                              :<=      <=}]
    (if (expr? expression)
      (let [with-evaluated-params (expr-map evaluate-constants expression)]
        (if (every? primitive? (:params with-evaluated-params))
          (let [operator-fn
                  (get expression-operators (:operator with-evaluated-params))
                params (:params with-evaluated-params)]
            (apply operator-fn params))
          with-evaluated-params))
        expression)))
```

Об’єднавши спрощення та виконання, маємо повний алгоритм спрощення.

```
(defn simplify-expression [expression]
  (let [simplified (-> expression simplify-expression-once evaluate-constants)]
    (if (not= expression simplified)
      (recur simplified)
      simplified)))
```

5. РЕЗУЛЬТАТИ

Вивчено теорію формальної правильності програм, логіки висловлювань, синтаксичного розбору програм.

Сформульовано підмножину мови Javascript для доведення деяких програм, та дороблено анотаціями. Для мови розроблена граматика та синтаксичний аналізатор.

Реалізовано транслятор з вхідної мови до логічного висловлювання через мову найслабкішої передумови. Реалізовано систему перетворення та доведення логічних висловлювань.

Розроблено тестовий пакет, що перевіряє правильність вище зазначених транслятора та системи. За його допомогою забезпечено високу якість програмного продукту.

Спроектовано архітектуру продукту, що може розширюватись. Запропоновано точки додавання нових мов програмування, нових можливостей мов, та методів доведення. Увесь продукт реалізовано у функціональному стилі заради легкої паралелізації.

Обрано набор демонстраційних алгоритмів та продемонстровано роботу системи з цими алгоритмами.

ВИСНОВКИ

Запропонований продукт тільки починає розкривати питання формального доведення правильності програм. Більшість сучасних ідіоматичних засобів програмування Javascript не підтримуються системою доведення - наприклад, функції та об’єкти. У подальшому хотілось би доробити продукт у напрямку підтримки цих засобів.

До того ж, щоб продукт був корисним для практичних задач рядового програміста, потрібно розробити анотації для функцій стандартної бібліотеки Javascript.

Інший напрямок розвитку - це розробка анотацій та парсерів для інших популярних мов програмування, та створення першого у світі формального прувера-поліглота.

Третій напрямок розвитку - це екстракція апарату перетворення та доведення логічних виразів у власний програмний пакет, корисний у доведенні висловлювань у інших галузях програмування.

Четвертий напрямок розвитку - розвиток розподіленої системи доведення, що могла б доводити більші за розміром програми з використанням актуальної на наш час хмарної архітектури, автоматично виділяючи апаратні ресурси для складних обчислень.

Отже, розроблений програмний продукт може бути відправною точкою для різноманітних наступних досліджень та робіт.

У майбутньому планується випуск продукту у формі з відкритим вхідним кодом та пропонування співпраці іншим діячам предметної області.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Gries D. The Science of Programming / — Нью-Йорк: Springer-Verlag New York Inc., 1981. — 716 с.

2. Fogus M. The Joy of Clojure, Second Edition / Fogus M., Houser C. — Шелтер-Айленд: Manning Publications Co., 2014 — 1076 с.

3. Parr T. ANTLR 4 Documentation [електрон. текст. дан.] - Режим доступу: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation - Заголовок з екрану

4. Marick B. Midje wiki [електрон. текст. дан.] - Режим доступу: https://github.com/marick/Midje/wiki - Заголовок з екрану

5. Nolen D. core.logic wiki [електрон. текст. дан.] - Режим доступу: https://github.com/clojure/core.logic/wiki - Заголовок з екрану

6. Kao E. Introduction to Logic [електрон. текст. дан.] / Computer Science Department, Stanford University - Режим доступу: http://logic.stanford.edu/intrologic/chapters/cover.html - Заголовок з екрану

7. Aho A. Foundations of Computer Science: C Edition / Aho A., Ullman J. - Principles of Computer Science Series - W.H. Freeman, 1994. - 786 с. - гл. 14

8. Fishman C. They Write The Right Stuff [електрон. бюл.] / Fast Company Magazine, 1996 - Режим доступу: http://www.fastcompany.com/28121/they-write-right-stuff

9. Hickey R. Clojure Rationale [електрон. текст. дан.] / - Режим доступу: http://clojure.org/rationale - Заголовок з екрану

10. Wyatt J. B. Software disasters [електрон. текст. дан.] / - Режим доступу: http://jbwyatt.com/disasters.html - Заголовок з екрану

11. Krishnamurthi S. The Essence of JavaScript [електрон. бюл.] / The Brown PLT Blog, Brown University, Providence, 2011 - Режим доступу: http://blog.brownplt.org/2011/09/29/js-essence.html - Заголовок з екрану

12. Ganssle J. Perfect Software [електрон. бюл.] / EETimes India, 2011 - Режим доступу: http://forum.eetindia.co.in/BLOG_ARTICLE_6832.HTM - Заголовок з екрану

13. General Principles of Software Validation; Final Guidance for Industry and FDA Staff / U.S. Department of Health and Human Services, Food and Drug Administration, Center for Devices and Radiological Health, Center for Biologics Evaluation and Research, 2002 - Режим доступу: http://www.fda.gov/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm085281.htm#_Toc517237951 - Заголовок з екрану

14. Hoare C. An Axiomatic Basis for Computer Programming / The Queen's University of Belfast, Northern Ireland - Communications of the ACM, vol. 12, no. 10, Oct. 1969

15. Kaufmann M. A Brief ACL2 Tutorial [електрон. текст. дан.] / Matt Kaufmann, Advanced Micro Devices, Inc., J Strother Moore, Department of Computer Sciences, University of Texas at Austin - Режим доступу: http://www.cs.utexas.edu/users/kaufmann/tutorial/rev3.html - Заголовок з екрану

ДОДАТОК А. ГРАМАТИКА МОВИ ВХІДНИХ ТЕКСТІВ

Граматика у форматі ANTLR v4:

```
grammar ImperativeLanguage;

@header { package thesis; }

provingStructure: assertionComment commands EOF;

assertionComment: '/*' preconditions postconditions '*/';

preconditions: 'PRE:' predicates;

postconditions: 'POST:' predicates;

predicates: predicate ( SEMICOLON predicate)* SEMICOLON?;

commands: command ( SEMICOLON command)* SEMICOLON?;

loopComment: '/*' invariant  boundFunction '*/';

invariant: 'INV:' predicate;
boundFunction: 'BOUND:' expression;

command:
  identifier '=' expression # assignmentCommand
  | '{' commands  '}' # sequenceCommand
  | 'if' '(' predicate ')' command # conditionalCommand
  | 'if' '(' predicate ')' command 'else' command # fullConditionalCommand
  | loopComment 'while' '(' predicate ')' command # loopCommand
  ;

// Expression grammar

expression:
  expression MULT_OPERATOR expression # multExpression
  | expression SUM_OPERATOR expression # sumExpression
  | atom # atomExpression
  ;
```

```
atom: negAtom | numericConstant | identifier | parensExpression;

negAtom: NEG_OPERATOR atom;

parensExpression: '(' expression ')';

// Predicate grammar

predicate:
  predicate AND_OPERATOR predicate # andPredicate
  | predicate OR_OPERATOR predicate # orPredicate
  | predicate IMPLIES_OPERATOR predicate # orPredicate
  | predicateAtom # atomPredicate
  ;

notAtom: NOT_OPERATOR predicateAtom;

predicateAtom: notAtom | booleanConstant | comparison | parensPredicate;

comparison:
  expression COMPARISON_OPERATOR expression;

parensPredicate: '(' predicate ')';

// Primitives

booleanConstant: 'true' | 'false';

numericConstant: NUMBER;

identifier: ID;

COMPARISON_OPERATOR: '<' | '>' | '==' | '>=' | '<=' | '!=';

MULT_OPERATOR: MULTIPLY | DIVIDE;
MULTIPLY: '*';
DIVIDE: '/';

SUM_OPERATOR: PLUS | MINUS;
PLUS: '+';
MINUS: '-';

NEG_OPERATOR: MINUS;
```

```
AND_OPERATOR: '&&';
OR_OPERATOR: '||';
NOT_OPERATOR: '!';
IMPLIES_OPERATOR: '=>';

NUMBER: '-'?[0-9]+;

ID: [a-zA-Z_][a-zA-Z_0-9]*;

SEMICOLON: ';';

WS: ('//'  ~( '\r' | '\n' )*  | [\n\r\b\t\f ]+) -> skip;
```

ДОДАТОК Б. ВХІДНИЙ ТЕКСТ ПРОДУКТУ

project.clj

```
(defproject thesis "0.1.0-SNAPSHOT"
            :description "FIXME: write description"
            :url "http://example.com/FIXME"
            :license {:name "Eclipse Public License"
                      :url  "http://www.eclipse.org/legal/epl-v10.html"}
            :dependencies [[org.clojure/clojure "1.6.0"]
                           [org.antlr/antlr4 "4.5"]
                           [log4j/log4j "1.2.17"]
                           [org.clojure/tools.logging "0.3.1"]]
            :main ^:skip-aot thesis.core
            :target-path "target/%s"
            :profiles {:uberjar {:aot :all}
                       :dev     {:dependencies [[midje "1.6.3"]]}})
```

core.clj

```
(ns thesis.core
  (:gen-class)
  (:require [thesis.parser :refer [parse]]
            [thesis.program-semantics :refer [parse-tree->program]]
            [thesis.wp :refer [program-correctness-hypothesis]]
            [thesis.comparison-semantics :refer [factualize-comparisons]]
            [thesis.prover :refer [prover]]
            [thesis.simplify :refer [simplify-expression]]
            [clojure.tools.logging :as log]
            [clojure.string :as s])
  (:import (org.antlr.v4.runtime.misc ParseCancellationException)
           (java.io FileNotFoundException)))

(defn parse-program-from-filename [filename]
  (try
    (if-let [parse-tree (-> filename slurp parse)]
      (parse-tree->program parse-tree)
```

```
      (log/errorf "Failed to parse: %s" filename))
    (catch FileNotFoundException _
      (log/errorf "File not found: %s" filename))))

(defn prove-program [program]
  (let [correctness-hypothesis (program-correctness-hypothesis program)

        _ (log/spyf "Program correctness hypothesis:\n%s"
                    correctness-hypothesis)

        [comparison-axioms factualized-hypothesis]
          (factualize-comparisons correctness-hypothesis)

        _ (log/spyf "After reducing comparisons into facts:\n%s"
                    factualized-hypothesis)
        _ (log/spyf "Simplified:\n%s"
                    (simplify-expression factualized-hypothesis))
        _ (log/spyf "Axioms about comparisons:\n%s"
                    (s/join "\n" (map str comparison-axioms)))]
    (prover comparison-axioms factualized-hypothesis)))

(defn -main [filename & args]
  (if-let [program (parse-program-from-filename filename)]
    (println (prove-program program))
    (println "Error")))
```

parser.clj

```
(ns thesis.parser
  (:require [clojure.string :as s])
  (:import (org.antlr.v4.runtime ANTLRInputStream CommonTokenStream
                                 BailErrorStrategy)
           (org.antlr.v4.runtime.tree ParseTree)
           (thesis ImperativeLanguageLexer ImperativeLanguageParser)))

(defn- antlr-parse
  "Invokes ANTLR parser to produce a raw parse structure from source code"
  [source-code]
  (let [lexer (ImperativeLanguageLexer. (ANTLRInputStream. source-code))
        tokens (CommonTokenStream. lexer)
```

```
        parser (ImperativeLanguageParser. tokens)
        proving-structure (.provingStructure parser)]
    (if (zero? (.getNumberOfSyntaxErrors parser))
      proving-structure
      nil)))

(defn- context-keyword
  "For object of class Something$SomeParseContext, return :some-parse"
  [^ParseTree context]
  (keyword (.toLowerCase (s/replace (s/replace (str (.getClass context))
                                               #"^.+\$(.+)Context$" "$1")
                                    #"([a-z])([A-Z])" "$1-$2"))))

(defn zap-semicolons
  "Remove all semicolons from context sequence"
  [parse-seq]
  (remove #{";"} parse-seq))

(defn- context->seq
  "Convert a parse context into a nested list"
  [^ParseTree context]
  (if (zero? (.getChildCount context))
    (.getText context)
    (cons (context-keyword context)
          (zap-semicolons (for [child-index (range 0 (.getChildCount context))]
            (context->seq (.getChild context child-index)))))))

(defn parse
  "Produce a parse tree from source code"
  [source-code]
  (if-let [proving-structure (-> source-code antlr-parse)]
    (context->seq proving-structure)
    nil))
```

program-semantics.clj

```
(ns thesis.program-semantics
  (:require [thesis.algebra :refer [expr]]))

(defrecord Command [type params])
```

```
(defn- find-child
  "Find a child seq with a given name inside the context seq
  For example: find (:plus 2 3) in (:multiply 4 (:plus 2 3))"
  [parse-seq child-name]
  (first (filter #(= (first %) child-name) (next parse-seq))))

(defn- conjoin-predicates
  "Given a list of predicates, produce a conjunction of all of them"
  [predicates]
  (reduce (fn [& predicate-pair] (apply expr :and predicate-pair)) predicates))


(defn translate-expression
  "Convert expression from the parse tree structure to a structure for further
  processing"
  [expression-tree]
  (let [e-type (first expression-tree)
        e-params (next expression-tree)
        tr translate-expression]
    (case e-type
      :and-predicate
        (let [[x _and y] e-params]
          (expr :and (tr x) (tr y)))
      :or-predicate
        (let [[x _or y] e-params]
          (expr :or (tr x) (tr y)))
      :implies-predicate
        (let [[x _implies y] e-params]
          (expr :implies (tr x) (tr y)))
      (:mult-expression :sum-expression :comparison)
        (let [[x operator y] e-params]
          (expr (keyword operator) (tr x) (tr y)))

      :not-atom
        (let [[_not x] e-params]
          (expr :not (tr x)))
      (:parens-predicate :parens-expression)
        (let [[_paren expression _paren] e-params]
          (tr expression))

      :identifier
        (first e-params)
      :numeric-constant
        (Integer/parseInt (first e-params))
```

```
      :boolean-constant
        (Boolean/parseBoolean (first e-params))

      ; pass-through nodes
      (:atom-predicate :atom-expression :atom :predicate-atom)
        (tr (first e-params)))))

(defn translate-command
  [command-tree]
  (let [c-type (first command-tree)
        c-params (next command-tree)]
    (case c-type
      :empty-command (->Command :noop [])

      :assignment-command
        (let [[[_identifier identifier] _equals expression] c-params]
          (->Command :assign [identifier (translate-expression expression)]))

      :sequence-command
        (let [[_bracket [_commands & commands] _bracket] c-params]
          (if (== 1 (count commands))
            (translate-command (first commands))
            (->Command :sequence (map translate-command commands))))

      :conditional-command
        (let [[_if _paren predicate _paren command] c-params]
          (->Command :if [(translate-expression predicate)
                          (translate-command command)
                          (->Command :noop [])]))

      :full-conditional-command
        (let [[_if _paren predicate _paren if-command _else else-command]
              c-params]
          (->Command :if [(translate-expression predicate)
                          (translate-command if-command)
                          (translate-command else-command)]))

      :loop-command
        (let [[[_loop-comment
                _star
                [_invariant _inv invariant]
                [_bound _bound bound-function] _star
                ]
               _while
```

```
                _paren
                predicate
                _paren
                command]
              c-params]
            (->Command :while [(translate-expression predicate)
                               (translate-command command)
                               (translate-expression invariant)
                               (translate-expression bound-function)])))))

(defn- extract-annotation [assertion-comment key-name]
  (-> (find-child (find-child assertion-comment key-name) :predicates)
      next
      (#(map translate-expression %))
      conjoin-predicates))

(defn parse-tree->program
  "Convert parse tree into a semantic structure"
  [parse-tree]
  (let [assertion-comment
          (find-child parse-tree :assertion-comment)
        [precondition postcondition]
          (map (partial extract-annotation assertion-comment)
               [:preconditions :postconditions])
        commands
          (next (find-child parse-tree :commands))]
    {:precondition precondition,
     :postcondition postcondition,
     :commands (->Command :sequence (map translate-command commands))}))
```

wp.clj

```
(ns thesis.wp
  (:require [thesis.algebra :refer [expr]]
            [thesis.identifiers :refer [replace-identifier-in-expression]]
            [clojure.tools.logging :as log]))

(declare command-wp)

(defn conditional-wp [predicate if-command else-command postcondition]
  (expr :and
```

```
        (expr :implies
              predicate
              (command-wp postcondition if-command))
        (expr :implies
              (expr :not predicate)
              (command-wp postcondition else-command))))

(defn loop-wp
  [loop-id loop-condition loop-command invariant bound postcondition]
  (let [bound-variable (str "bound_" loop-id)]
    (log/spyf "Loop condition %s"
              (reduce (partial expr :and)
                      [invariant
                       [:implies [:and loop-condition invariant]
                                 (command-wp invariant loop-command)]
                       [:implies [:and loop-condition invariant]
                                 [:> bound 0]]
                       [:implies [:and loop-condition invariant]
                                 (replace-identifier-in-expression
                                   bound-variable
                                   bound
                                   (command-wp (expr :< bound bound-variable)
                                               loop-command))]
                       [:implies [:and [:not loop-condition] invariant]
                                 postcondition]])))))

(defn command-wp [postcondition command]
  (let [c-type (:type command)
        c-params (:params command)]
    (case c-type
      :noop postcondition
      :sequence (reduce command-wp postcondition (reverse c-params))
      :assign (let [[identifier value] c-params]
                (replace-identifier-in-expression identifier
                                                  value
                                                  postcondition))

      :if
        (let [[predicate if-command else-command] c-params]
          (conditional-wp predicate if-command else-command postcondition))

      :while
        (let [[predicate loop-command invariant bound] c-params]
          (loop-wp (hash command)
```

```
                    predicate
                    loop-command
                    invariant
                    bound
                    postcondition)))))

(defn weakest-precondition [program]
  (command-wp (:postcondition program) (:commands program)))

(defn program-correctness-hypothesis
  "Returns hypothesis that the program is correct, that is:
  precondition => weakest-predicate(commands, postcondition)"
  [program]
  (expr :implies (:precondition program) (weakest-precondition program)))
```

algebra.clj

```
(ns
  thesis.algebra
  "An Expression is any kind of logical expression, it consists of an operator
  and its params, which in turn may be Expression, string identifiers, or
  numeric and boolean primitive constants"
  (:require [clojure.string :as s]))

(defrecord Expression
  [operator params]

  Object
  (toString [this]
    (str "("
         (-> this :operator name s/upper-case)
         " "
         (s/join " " (map str (:params this)))
         ")")))

(defmethod print-method Expression [expr ^java.io.Writer writer]
  (.write writer (str expr)))

(defn expr? [expression] (isa? (class expression) Expression))

(defn expr
```

```
  "Simplified expression constructor. Accepts nested structure of
  operator1 [operator2 param1 param2] param3 etc"
  [operator & params]
  {:pre [(keyword? operator) (not (empty? params))]}
  (let [realize-param-fn (fn [param] (if (and (not (expr? param)) (coll? param))
                                       (apply expr param)
                                       param))
        realized-params (map realize-param-fn params)]
    (->Expression operator realized-params)))

(defn expr-map
  "Apply fun to every parameter of expression"
  [fun expression]
  (apply expr (:operator expression) (map fun (:params expression))))

(defn expr-atom?
  [expression]
  (not (expr? expression)))

(defn logical-expr?
  [predicate]
  (and (expr? predicate) (#{:and :or :not :implies} (:operator predicate))))

(defn logical-atom?
  [predicate]
  (not (logical-expr? predicate)))

(defn identifier?
  [identifier]
  (string? identifier))

(defn primitive?
  [predicate]
  (or (number? predicate) (true? predicate) (false? predicate)))

(defn expr-commutative?
  [expression]
  (#{:and :or :+ :* :==} (:operator expression)))

(defn merge-bindings [bindings-seq]
  (let [merged-bindings (apply merge-with #(when (= %1 %2) %1) bindings-seq)]
    (and (not-any? #(nil? (fnext %)) merged-bindings) merged-bindings)))

(defn expr-match
```

```
  "Attempts to match expression by a pattern.
  Pattern is a structure of [operator param1 [operator param2 param3]...
  String params in pattern are matched against any sub-expression, identifier
  or primitive
  Boolean and numeric params in pattern are matched by equality.
  Returns a map of string params to their values, or false if there is no match"
  [pattern expression]
  (if (identifier? pattern)
    {pattern expression}
    (if (primitive? pattern)
      (and (= expression pattern) {})
      (and (expr? expression)
           (let [operator (first pattern) params (next pattern)]
             (and (= operator (:operator expression))
                  (let [forward-param-bindings
                          (map expr-match params (:params expression))
                        forward-match
                          (not-any? false? forward-param-bindings)
                        reverse-param-bindings
                          (and (not forward-match)
                               (expr-commutative? expression)
                               (map expr-match
                                    params
                                    (reverse (:params expression))))
                        reverse-match
                          (and reverse-param-bindings
                               (not-any? false? reverse-param-bindings))
                        param-bindings
                          (or (and forward-match forward-param-bindings)
                              (and reverse-match reverse-param-bindings))]
                    (and param-bindings (merge-bindings param-bindings)))))))))

(defn expr-construct
  "Constructs expression from a pattern. For pattern format see expr-match
  String params in pattern are replaced by the values specified in
  param-bindings"
  [param-bindings pattern]
  (cond
    (primitive? pattern)
      pattern
    (identifier? pattern)
      (or (get param-bindings pattern)
          (throw (Exception. (str "Undefined expression param " pattern))))
    (coll? pattern)
```

```
      (let [operator (first pattern)
            params (map (partial expr-construct param-bindings) (next pattern))]
        (apply expr operator params))))


(defn expr-transform
  "Replaces parts of expression that match pattern-from, with expressions
  constructed from pattern-to and the matched param bindings"
  [pattern-from pattern-to expression]
  (if (expr? expression)
    (if-let [param-bindings (expr-match pattern-from expression)]
      (recur pattern-from pattern-to (expr-construct param-bindings pattern-to))
      (let [expr-with-transformed-params
              (expr-map (partial expr-transform pattern-from pattern-to)
                        expression)]
        (if-let [param-bindings
                 (expr-match pattern-from expr-with-transformed-params)]
          (recur pattern-from pattern-to
                 (expr-construct param-bindings pattern-to))
          expr-with-transformed-params)))
    expression))

(defn expr-clauses
  "Given expression subexpr-A <clause-operator> subexpr-B <clause-operator> ...,
  produces a seq of (subexpr-A subexpr-B ... )"
  [clause-operator expression]
  (if (or (expr-atom? expression) (not= clause-operator (:operator expression)))
    [expression]
    (mapcat (partial expr-clauses clause-operator) (:params expression))))
```

polynomial.clj

```
(ns thesis.polynomial
  (:require [thesis.algebra :refer [expr-transform expr-clauses identifier?]]
            [clojure.string :as s]))

;; implementation of clausal-polynomial form

(defn differences-to-sums
  "Convert expression A - B to A + (-1 * B) for further normalization"
  [expression]
```

```
  (expr-transform [:- "a" "b"] [:+ "a" [:* -1 "b"]] expression))

(defn distribute-products
  "Convert A * (B + C) to A*B + A*C"
  [expression]
  (let [transformed (expr-transform [:* "a" [:+ "b" "c"]]
                                    [:+ [:* "a" "b"] [:* "a" "c"]]
                                    expression)]
    (if (not= expression transformed)
      (recur transformed)
      transformed)))

; TODO maybe (/ a const) = (* a 1/const)
(defn move-division-outward
  [predicate]
  (let [transformed
        (->> predicate
           (expr-transform [:+ [:/ "a" "b"] [:/ "c" "d"]]
                           [:/ [:+ [:* "a" "d"] [:* "c" "b"]] [:* "b" "d"]])
           (expr-transform [:* "a" [:/ "b" "c"]] [:/ [:* "a" "b"] "c"])
           (expr-transform [:+ "a" [:/ "b" "c"]] [:/ [:+ [:* "a" "c"] "b"] "c"])
           (expr-transform [:/ "a" [:/ "b" "c"]] [:/ "a" [:* "b" "c"]])
           (expr-transform [:/ [:/ "a" "b"] "c"] [:/ "a" [:* "b" "c"]])
           )]
    (if (not= predicate transformed)
      (recur transformed)
      transformed)))


(defn polynomial-clauses [normalized-polynomial]
  (map (partial expr-clauses :*) (expr-clauses :+ normalized-polynomial)))

(defn join-similar-clauses
  "Returns a normalized hash of (sorted variable list) -> numeric factor"
  [polynomial-terms]
  (reduce (fn [term-factors new-term]
            (let [factor-key
                    (s/join "*" (reverse (sort (filter identifier? new-term))))
                  factor (reduce * 1 (filter number? new-term))
                  existing-factor (get term-factors factor-key 0)]
              (assoc term-factors factor-key (+ existing-factor factor))))
          (sorted-map-by #(compare %2 %1)) polynomial-terms))

(defn remove-zeroes-from-cpf [terms]
```

```
  (into {} (remove (fn [[t-vars t-factor]] (zero? t-factor)) terms)))

(defn clausal-polynomial-form
  "Given an algebraic expression, produce sorted map of
  (variable multiplier) => (constant multiplier)"
  [expression]
  (-> expression
      differences-to-sums
      distribute-products
      polynomial-clauses
      join-similar-clauses
      remove-zeroes-from-cpf))

;; implementation of normalize-cpf

(defn multiply-cpf [factor terms]
  (into {} (map (fn [[t-vars t-factor]] [t-vars (/ t-factor factor)]) terms)))

(defn normalize-cpf
  "Given CPF, return
  [factor-for-max-power-clause clauses-with-max-power-clause-equal-to-1]"
  [clausal-polynomial-form]
  (let [lex-first-factor (first (vals clausal-polynomial-form))]
    [lex-first-factor (multiply-cpf lex-first-factor clausal-polynomial-form)]))

;; implemetation of separate-scalar

(defn separate-scalar
  "Given clausal polynomial form, returns
  [cpf-without-scalar-factor scalar-factor-or-0]"
  [cpf]
  [(dissoc cpf "") (get cpf "" 0)])

;; implementation of cpf->str

(defn addend->str [[addend-variables addend-factor]]
  (if (= "" addend-variables)
    (str addend-factor)
    (case addend-factor
      1 addend-variables
      -1 (str "-" addend-variables)
      (str addend-factor "*" addend-variables))))

(defn join-addends [addend1 addend2]
```

```
  (if (= \- (first addend2))
    (str addend1 addend2)
    (str addend1 "+" addend2)))

(defn cpf->str [clausal-polynomial-form]
  (if (empty? clausal-polynomial-form)
    "0"
    (reduce join-addends (map addend->str clausal-polynomial-form))))
```

comparison-semantics.clj

```
(ns thesis.comparison-semantics
  (:require [clojure.string :as s]
            [clojure.set :as set]
            [thesis.algebra :refer [expr expr-transform expr-construct
                                    expr-clauses expr? identifier?]]
            [thesis.polynomial :refer [clausal-polynomial-form normalize-cpf
                                       cpf->str separate-scalar]]
            [clojure.tools.logging :as log]))

(defn comparison? [expr] (-> expr :operator #{:> :< :>= :<= :!= :==}))

(defn make-comparison-one-sided
  "Convert A _sign_ B to (A - B) _sign_ 0"
  [comparison]
  (let [[lside rside] (:params comparison)]
    (if (and (number? rside) (zero? rside))
      comparison
      (expr (:operator comparison) (expr :+ lside (expr :* -1 rside)) 0))))

(def flipped-operator
  {:>  :<
   :>= :<=
   :<  :>
   :<= :>=
   :== :==
   :!= :!=})

(def operators-pattern
  {:>  [[">"]    "l>r"]
   :<  [["<"]    "l<r"]
```

```
   :<= [[">"]    [:not "l>r"]]
   :>= [["<"]    [:not "l<r"]]
   :== [["=="]  "l==r"]
   :!= [["=="] [:not "l==r"]]
   })

(defn record-comparison [bindings comparison]
  (let [one-sided (make-comparison-one-sided comparison)]
    (if-let [cpf (-> one-sided :params first clausal-polynomial-form not-empty)]
      (let [[normalizing-factor normalized-clauses] (normalize-cpf cpf)
            normalized-operator (if (pos? normalizing-factor)
                                  (:operator one-sided)
                                  (get flipped-operator (:operator one-sided)))
            [non-scalar scalar] (separate-scalar normalized-clauses)
            rside (* -1 scalar)]
        (if (empty? non-scalar)                              ; if left side empty
          [bindings (expr normalized-operator 0 rside)]
          (let [string-lside (cpf->str non-scalar)
                [operators pattern] (get operators-pattern normalized-operator)
                expr-params
                  (into {} (map #(vector (str "l" % "r")
                                         (str string-lside % rside))
                                operators))
                new-expression (expr-construct expr-params pattern)
                new-bindings
                  {non-scalar
                   (into #{} (map #(vector (keyword %) rside) operators))}]
            [(merge-with set/union bindings new-bindings) new-expression])))
      ; if left side is empty (0), expression is true if operator allows
      ; equality (because right side is 0)
      [bindings (expr (:operator one-sided) 0 0)])))


;; Implementation of augment-hypothesis-with-comparison-implications

(defn comparison-implications
  "Input:
   2 comparisons: lside ox x, and lside oy y, where ox,oy - one of [>, <],
   x, y - scalars
   Output:
   if there is an implication from lside ox x being true or false,
   to lside oy y being true or false, then
   returns list of such implications, otherwise empty list
   "
```

```
  [lside [[ox x] [oy y]]]
  (let [exprx (str lside (name ox) x)
        expry (str lside (name oy) y)]
    (remove false?
            (case [ox oy]
              [:> :<] [(and (>= x y) (expr :implies exprx [:not expry]))
                       (and (< x y) (expr :implies [:not exprx] expry))]
              [:> :>] [(and (> x y) (expr :implies exprx expry))
                       (and (< x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :<] [(and (< x y) (expr :implies exprx expry))
                       (and (> x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :>] [(and (<= x y) (expr :implies exprx [:not expry]))
                       (and (> x y) (expr :implies [:not exprx] expry))]
              [:> :==] [(and (>= x y) (expr :implies exprx [:not expry]))
                        (and (< x y) (expr :implies [:not exprx] [:not expry]))]
              [:< :==] [(and (<= x y) (expr :implies exprx [:not expry]))
                        (and (> x y) (expr :implies [:not exprx] [:not expry]))]
              [:== :==] [(and (not= x y) (expr :implies exprx [:not expry]))]
              [:== :>] [(and (> x y) (expr :implies exprx expry))
                        (and (<= x y) (expr :implies exprx [:not expry]))]
              [:== :<] [(and (< x y) (expr :implies exprx expry))
                        (and (>= x y) (expr :implies exprx [:not expry]))]
              ))))

(defn binding-implications
  "Given one binding produced by extract-comparisons, build a collection of
  axioms about the comparisons mentioned in the binding"
  [[clauses comparisons]]
  (let [string-clause
          (cpf->str clauses)
        permutations
          (for [c1 comparisons c2 comparisons :when (not= c1 c2)] [c1 c2])]
    (mapcat (partial comparison-implications string-clause) permutations)))

(defn extract-comparisons
  [bindings expression]
  (if (expr? expression)
    (if (comparison? expression)
      (record-comparison bindings expression)
      (let [[bindings params]
            (reduce (fn [[bindings params] param]
                      (let [[new-bindings new-param]
                            (extract-comparisons bindings param)]
                        [new-bindings (conj params new-param)]))
```

```
                    [bindings []]
                    (:params expression))]
        [bindings (apply expr (:operator expression) params)]
        ))
    [bindings expression]))

(defn factualize-comparisons [expression]
  (let [[bindings expression] (extract-comparisons {} expression)]
    [(mapcat binding-implications bindings) expression]))
```

prover.clj

```
(ns thesis.prover
  (:require [clojure.tools.logging :as log]
            [clojure.string :as s]
            [thesis.algebra :refer [expr]]
            [thesis.clausal-normal-form :refer [clausal-normal-form clause-set]]
            [thesis.simplify :refer [evaluate-constants simplify-expression]]
             [thesis.identifiers :refer [replace-identifier-in-expression identifiers-
in-expression]]
            [clojure.set :as set]))

; https://en.wikipedia.org/wiki/Resolution_(logic)

(def EVALUATION-METHOD-VARIABLE-LIMIT 20) ; up to 1000000 evaluations

(defn derive-resolutions [clauses]
  (clause-set (for [c1 clauses c2 clauses :when (not= c1 c2)]
    (let [[yes1 no1] c1
          [yes2 no2] c2
          all-yes (set/union yes1 yes2)
          all-no (set/union no1 no2)]
      (with-meta
        [(set/difference all-yes all-no) (set/difference all-no all-yes)]
        {:parents [c1 c2]})))))

(defn clause->str [[yes-disjuncts no-disjuncts]]
  (if (every? empty? [yes-disjuncts no-disjuncts])
    "FALSE"
    (s/join " OR " (concat yes-disjuncts
```

```
                          (map (partial str "NOT ") no-disjuncts)))))

(defn clauses->str [cnf]
  (s/join "\nAND\n" (map clause->str cnf)))

(defn empty-clause? [[yes no]]
  (and (empty? yes) (empty? no)))

(defn log-clause [label clause]
  (log/debugf (str label ": %s") (clause->str clause))
  (when-let [[c1 c2] (get (meta clause) :parents)]
    (log/debugf "From: %s  ~AND~  %s" (clause->str c1) (clause->str c2)))
  clause)

(defn resolution-method [clauses iteration]
  (if (> iteration 1000)
    :exceeded-iteration-limit
    (if-let [best-clause
             (log-clause "Best derived clause"
                         (first (set/difference (derive-resolutions clauses)
                                                clauses)))]
      (if (empty-clause? best-clause)
        :proved
        (recur (conj clauses best-clause) (inc iteration))
        )
      (do
        (log/debug "Failed to disprove: " (clauses->str clauses))
        :failed-to-prove)
      )))

(defn trivial-solution [[yes no]]
  (cond
    ; Take note - trivial solution deals with the counter example
    (and (= yes #{true}) (empty? no)) :disproved
    (and (= yes #{false}) (empty? no)) :proved
    :default :failed-to-prove))

(defn evaluation-method
  ([expression identifiers bindings]

  (if-let [identifier (first identifiers)]
    (every? true?
            (map #(evaluation-method
                   (replace-identifier-in-expression identifier % expression)
```

```
                    (rest identifiers)
                    (assoc bindings identifier %))
                  #{true false}))
    (or (false? (evaluate-constants expression))
        (and (log/spyf "Counter-example: %s" bindings) false))))

  ([expression identifiers] (evaluation-method expression identifiers {})))

(defn prover [facts hypothesis]
  {:post (#{:proved :disproved :failed-to-prove} %)}
  (let [counter-hypothesis (expr :not hypothesis)

        _ (log/spyf "Simplified counter hypothesis: %s"
                    (simplify-expression counter-hypothesis))

        provable-statement (simplify-expression
                            (reduce #(expr :and %1 %2)
                                    counter-hypothesis
                                    facts))
        identifiers (identifiers-in-expression provable-statement)]
    (if (< (count identifiers) EVALUATION-METHOD-VARIABLE-LIMIT)
      (do
        (log/spyf "Using evaluation method for %d variables"
                  (count identifiers))
        (log/spyf "Provable statement %s"
                  provable-statement)
        (if (evaluation-method provable-statement identifiers)
          :proved
          :failed-to-prove))
      (do
        (log/spyf "Using resolution method for %d variables"
                  (count identifiers))
        (let [cnf (clausal-normal-form provable-statement)
              _ (log/spyf "Initial clauses: %s" (clauses->str cnf))]
          (if (= 1 (count cnf))
            (trivial-solution (first cnf))
            (resolution-method cnf 0)))))))
```

simplify.clj

```
(ns thesis.simplify
  (:require [thesis.algebra :refer [expr expr-map expr-transform expr?
                                    primitive?]]))

; Reduces a 500-char expression into a 120-char expression through the power
; of logic and arithmetic equalities and evaluation of constant expressions

; (str (-> "division.js"
;          slurp parse parse-tree->program weakest-predicate
;          conjunctive-normal-form))
; (AND (== x (+ x (* y 0))) (AND (AND (AND (AND (OR (OR (>= y x) (!= x (+ x (* y
; 0)))) (== x (+ (* (+ 1 0) y) (- x y)))) (OR (OR (>= y x) (!= x (+ x (* y 0))))
; (< (- x y) y))) (OR (OR (>= y x) (!= x (+ x (* y 0)))) (>= (+ 1 0) 0))) (OR
; (OR (>= y x) (!= x (+ x (* y 0)))) (>= (- x y) 0))) (AND (AND (AND (OR (OR
; (< y x) (!= x (+ x (* y 0)))) (== x (+ (* 0 y) x))) (OR (OR (< y x) (!= x (+ x
; (* y 0)))) (< x y))) (OR (OR (< y x) (!= x (+ x (* y 0)))) (>= 0 0))) (OR (OR
; (< y x) (!= x (+ x (* y 0)))) (>= x 0)))))

; (str (-> "division.js"
;          slurp parse parse-tree->program weakest-predicate
;          conjunctive-normal-form simplify-expression))
; (AND (AND (OR (>= y x) (< (- x y) y)) (OR (>= y x) (>= (- x y) 0))) (AND (OR
; (< y x) (< x y)) (OR (< y x) (>= x 0))))

(defn- simplify-expression-once [expression]
  (let [simplification-rules {; Logic rules
                              [:implies true "x"]      "x"
                              [:implies "x" true]      true
                              [:implies false "x"]     true
                              [:implies "x" false]     [:not "x"]
                              [:not [:not "x"]]        "x"
                              [:or "x" false]          "x"
                              [:or "x" true]           true
                              [:and "x" false]         false
                              [:and "x" true]          "x"
                              [:or "x" "x"]            "x"
                              [:and "x" "x"]           "x"
                              [:or "x" [:not "x"]]     true
                              [:and "x" [:not "x"]]    false
                              [:and "x" [:or "x" "y"]] "x"
```

```
                              [:or "x" [:and "x" "y"]] "x"
                              ; Arithmetic rules
                              [:+ "x" 0]               "x"
                              [:* "x" 0]               0
                              [:* "x" 1]               "x"
                              [:+ [:- "x" "y"] "y"]    "x"
                              [:- "x" "x"]             0
                              [:- "x" 0]               "x"
                              [:/ "x" 1]               "x"
                              [:/ 0 "x"]               0
                              [:== "x" "x"]            true
                              [:<= "x" "x"]            true
                              [:>= "x" "x"]            true
                              [:!= "x" "x"]            false
                              [:< "x" "x"]             false
                              [:> "x" "x"]             false
                              }]
    (reduce
      (fn [expression [p-from p-to]] (expr-transform p-from p-to expression))
      expression simplification-rules)))

(defn evaluate-constants
  "Find constant sub-expressions in expression and evaluate them"
  [expression]
  (let [expression-operators {:and     #(and %1 %2)
                              :or      #(or %1 %2)
                              :not     not
                              :implies #(or (not %1) %2)
                              :+       +
                              :-       -
                              :*       *
                              :/       /
                              :==      =
                              :!=      not=
                              :>       >
                              :<       <
                              :>=      >=
                              :<=      <=}]
    (if (expr? expression)
      (let [with-evaluated-params (expr-map evaluate-constants expression)]
        (if (every? primitive? (:params with-evaluated-params))
          (let [operator-fn
                  (get expression-operators (:operator with-evaluated-params))
                params (:params with-evaluated-params)]
```

```
            (apply operator-fn params))
          with-evaluated-params))
      expression)))

(defn simplify-expression [expression]
  (let [simplified (-> expression simplify-expression-once evaluate-constants)]
    (if (not= expression simplified)
      (recur simplified)
      simplified)))
```

identifiers.clj

```
(ns thesis.identifiers
  (:require [thesis.algebra :refer [expr? expr-map identifier?]]))

(defn replace-identifier-in-expression
  [identifier value construct]
  (cond
    (expr? construct)
    (expr-map
      (partial replace-identifier-in-expression identifier value) construct)

    (identifier? construct)
    (if (= construct identifier)
      value
      construct)

    :else construct))

(defn identifiers-in-expression
  [expression]
  (cond
    (identifier? expression)
    [expression]

    (expr? expression)
    (distinct (mapcat identifiers-in-expression (:params expression)))

    :else
    []))
```

ДОДАТОК В. ТЕСТИ

Тести для цього програмного продукту виконані за допомогою середовища тестування Midje.

t_algebra.clj

```
(ns thesis.t-algebra
  (:use midje.sweet)
  (:require [thesis.algebra :as algebra]))

(fact
  "expr builds expressions"
  (algebra/expr :+ 1 2)
  => (algebra/->Expression :+ [1 2])

  (algebra/expr :+ 1 [:- 2 3])
  => (algebra/->Expression :+ [1 (algebra/->Expression :- [2 3])])

  (algebra/expr :+ 1 (algebra/->Expression :- [2 3]))
  => (algebra/->Expression :+ [1 (algebra/->Expression :- [2 3])]))

(facts
  "about expr-map"
  (fact
    "expr-map applies a function to all of an expression's parameters"
    (algebra/expr-map #(+ 2 %) (algebra/expr :+ 1 2))
    => (algebra/expr :+ 3 4)))

(facts
  "about expr-match"
  (fact
    "expr-match matches expressions and returns the map of matching values"
    (algebra/expr-match [:not "x"] (algebra/expr :not 1))
    => {"x" 1}

    (algebra/expr-match [:and "x" "y"] (algebra/expr :and "a" "b"))
    => {"x" "a", "y" "b"}
```

```
  (algebra/expr-match [:not "x"] (algebra/expr :and "x" "y"))
  => falsey

  (algebra/expr-match [:and "x" "y"] (algebra/expr :or "a" "b"))
  => falsey)

(fact
  "expr-match matches nested expressions"
  (algebra/expr-match [:not [:and "x" "y"]]
                      (algebra/expr :not '(:and "a" "b")))
  => {"x" "a", "y" "b"})

(fact
  "expr-match matches placeholders to non-primitive values"
  (algebra/expr-match [:not "z"] (algebra/expr :not '(:and "a" "b")))
  => {"z" (algebra/->Expression :and ["a" "b"])})

(future-fact
  "expr-match can handle commutative sub-expressions in different orientation"
  ; Right now each sub-expression is matched separately
  ; Need to search and back-track instead.
  ; For example - from x OR y = a OR b we bind x=a, y=b,
  ; but we could also bind x=b, y=a
  ; which would then let x => y = b => a
  (algebra/expr-match [:and [:or "x" "y"] [:implies "x" "y"]]
                      (algebra/expr :and [:or "a" "b"] [:implies "b" "a"]))
  => {"x" "b", "y" "a"}
  )

(fact
  "for commutative expressions, expr-match matches in reverse, too"
  (algebra/expr-match [:and "x" [:not "y"]]
                      (algebra/expr :and [:not "a"] "b"))
  => {"x" "b", "y" "a"})

(fact
  "for non-commutative expressions, expr-match does NOT match in reverse"
  (algebra/expr-match [:implies "x" [:not "y"]]
                      (algebra/expr :implies [:not "a"] "b"))
  => falsey)

(fact
  "expr-match matches booleans and numbers as their actual value"
```

```
    (algebra/expr-match [:+ "x" 1] (algebra/expr :+ "a" 1))
    => {"x" "a"}

    (algebra/expr-match [:+ "x" 1] (algebra/expr :+ "a" 2))
    => falsey

    (algebra/expr-match [:and "x" true] (algebra/expr :and "a" true))
    => {"x" "a"}

    (algebra/expr-match [:and "x" true] (algebra/expr :and "a" false))
    => falsey))

(facts
  "about expr-construct"
  (fact
    "expr-construct builds expressions from a pattern and a set of bindings"
    (algebra/expr-construct {"x" "a", "y" "b"} [:and "x" [:not "y"]])
    => (algebra/expr :and "a" [:not "b"])))

(facts
  "about expr-transform"
  (fact
    "transforms expressions"
     (algebra/expr-transform [:not [:and "x" "y"]]
                             [:or [:not "x"] [:not "y"]]
                             (algebra/expr :not [:and "a" "b"]))
     => (algebra/expr :or [:not "a"] [:not "b"])

     (algebra/expr-transform [:not [:not "x"]]
                             "x"
                             (algebra/expr :not [:not "x"]))
     => "x")

  (fact
    "transforms parts of expressions"
    (algebra/expr-transform [:not [:not "x"]]
                            "x"
                            (algebra/expr :and [:not [:not "x"]]
                                               [:not [:not "y"]]))
    => (algebra/expr :and "x" "y")

    (algebra/expr-transform [:not [:not "x"]]
                            "x"
                            (algebra/expr :not [:not [:not "x"]])))
```

```
    => (algebra/expr :not "x")

    (algebra/expr-transform [:not [:not "x"]]
                            "x"
                            (algebra/expr :not [:not [:not [:not "x"]]]))
    => "x"))

(fact
  "expr-clauses groups expression by an operator"
  (algebra/expr-clauses :* (algebra/expr :* [:+ 1 2] [:* [:+ 3 4] [:+ 5 6]]))
  => [(algebra/expr :+ 1 2) (algebra/expr :+ 3 4) (algebra/expr :+ 5 6)])

(fact
  "expr-clauses with just one expression will return its params"
  (algebra/expr-clauses :* (algebra/expr :* 1 2))
  => [1 2]
  )

(fact
  "top-level expression for expr-clauses must match clause-operator, or the
  result will have only one clause"
  (algebra/expr-clauses :* (algebra/expr :+ [:+ 1 2] [:* [:+ 3 4] [:+ 5 6]]))
  => [ (algebra/expr :+ [:+ 1 2] [:* [:+ 3 4] [:+ 5 6]]) ])
```

t_simplify.clj

```
(ns thesis.t-simplify
  (:use midje.sweet)
  (:require [thesis.algebra :refer [expr]]
            [thesis.simplify :as simplify]))

(facts
  "about simplify-expression"

  (fact
    "simplify-expressions evaluates nested constant expressions and reduces
    logical equalities"
    (simplify/simplify-expression (expr :and "x" [:> [:+ 2 3] 1]))
    => "x")

  (fact
```

```
    "simplify-expressions can eliminate variable dependencies using
    logical equalities"
    (simplify/simplify-expression (expr :and true [:and true [:or true "y"]]))
    => true))
```

t_polynomial.clj

```
(ns thesis.t-polynomial
  (:use midje.sweet)
  (:require [thesis.polynomial :as subj]
            [thesis.algebra :refer [expr]]))

(fact
  "clausal polynomial form produces CPF"
  (subj/clausal-polynomial-form (expr :+ "x" [:* 2 [:- "x" "y"]]))
  => {"y" -2 "x" 3}

  (subj/clausal-polynomial-form (expr :* "x" [:* 2 [:- "x" "y"]]))
  => {"y*x" -2 "x*x" 2}

  (subj/clausal-polynomial-form (expr :* [:+ "x" 1] [:+ "x" 1]))
  => {"x*x" 1, "x" 2, "" 1}
  )

(let [cpf (sorted-map-by #(compare %2 %1) "y*x" -2 "x*x" 2)]
  (fact
    "normalize-cpf normalizes CPF"
    (subj/normalize-cpf cpf)
    => [-2 {"y*x" 1 "x*x" -1}]
    ))

(let [cpf (sorted-map-by #(compare %2 %1) "x*x" -1, "x" -2, "" 1)]
  (fact
    "separate-scalar separates scalar"
    (subj/separate-scalar cpf)
    => [{"x*x" -1 "x" -2} 1]
    )

  (fact
    "cpf->str stringifies CPF"
    (subj/cpf->str cpf)
```

```
    => "-x*x-2*x+1"
    ))

(facts
  "about addend->str"

  (subj/addend->str ["x*x" 1])
  => "x*x"

  (subj/addend->str ["x*x" -1])
  => "-x*x"

  (subj/addend->str ["x*x" 2])
  => "2*x*x"

  (subj/addend->str ["x*x" -2])
  => "-2*x*x"

  (subj/addend->str ["" 1])
  => "1"
  )
```

t_comparison_semantics.clj

```
(ns thesis.t-comparison-semantics
  (:use midje.sweet)
  (:require [thesis.comparison-semantics :as subj]
            [thesis.algebra :refer [expr]]))

(fact
  "make-comparisons-one-sided carries everything to one side of the comparison"
  (subj/make-comparison-one-sided (expr :> "x" "y"))
  => (expr :> [:+ "x" [:* -1 "y"]] 0)
  )

(facts
  "about record-comparison"

  (subj/record-comparison {} (expr :> "x" 1))
  => [{{"x" 1} #{[:> 1]}} "x>1"]
```

```
  (subj/record-comparison {} (expr :!= "x" 1))
  => [{{"x" 1} #{[:== 1]}} (expr :not "x==1")]
  )

(facts
  "about extract-comparisons"

  (subj/extract-comparisons {} (expr :and [:> "x" 1] [:< [:* 2 "y"] 0]))
  => [{{"y" 1} #{[:< 0]} {"x" 1} #{[:> 1]}} (expr :and "x>1" "y<0")]
  )
```

t_prover.clj

```
(ns thesis.t-prover
  (:use midje.sweet)
  (:require [thesis.prover :as subj]
            [thesis.algebra :refer [expr]]))

(fact
  "prover can prove truth"
  (subj/resolution-prover [] true)
  => :proved
  )
```

ДОДАТОК Г. ПРИКЛАДИ ВИКОНАННЯ ПРОГРАМИ

Доведення пустої програми - empty.js

```
/*
  PRE: a == 1
  POST: a == 1
 */

// noop
a = a;
```

```
>java -jar prover.jar empty.js

Program correctness hypothesis:
(IMPLIES (== a 1) (== a 1))
After reducing comparisons into facts:
(IMPLIES a==1 a==1)
Simplified:
(IMPLIES a==1 a==1)
Axioms about comparisons:

Simplified counter hypothesis: (NOT (IMPLIES a==1 a==1))
Using evaluation method for 1 variables
Provable statement (NOT (IMPLIES a==1 a==1))
:proved
```

Доведення пустої програми, що є хибною - empty-fail.js

```
/*
  PRE: a == 1
  POST: a == 2
 */

// noop
a = a;
```

```
>java -jar prover.jar empty-fail.js

Program correctness hypothesis:
(IMPLIES (== a 1) (== a 2))
After reducing comparisons into facts:
(IMPLIES a==1 a==2)
Simplified:
(IMPLIES a==1 a==2)
Axioms about comparisons:
(IMPLIES a==2 (NOT a==1))
(IMPLIES a==1 (NOT a==2))
Simplified counter hypothesis: (NOT (IMPLIES a==1 a==2))
Using evaluation method for 2 variables
Provable statement (AND (AND (NOT (IMPLIES a==1 a==2)) (IMPLIES a==2 (NOT a==1))) (IM-
PLIES a==1 (NOT a==2)))
Counter-example: {"a==2" false, "a==1" true}
:failed-to-prove
```

Доведення присвоєння - assignment.js

```
/*
  PRE: a==1
  POST: b==1
 */

b = a;

>java -jar prover.jar assignment.js

Program correctness hypothesis:
(IMPLIES (== a 1) (== a 1))
After reducing comparisons into facts:
(IMPLIES a==1 a==1)
Simplified:
(IMPLIES a==1 a==1)
Axioms about comparisons:

Simplified counter hypothesis: (NOT (IMPLIES a==1 a==1))
Using evaluation method for 1 variables
Provable statement (NOT (IMPLIES a==1 a==1))
:proved
```

Доведення програми с гілкою - branch.js

```
// Find abs(x)

/*
  PRE: true
  POST: abs >= 0; abs*abs == x*x
 */

if (x >= 0) {
  abs = x
} else {
  abs = 0-x
}
```

>java -jar prover.jar branch.js

```
Program correctness hypothesis:
(IMPLIES true (AND (IMPLIES (>= x 0) (AND (>= x 0) (== (* x x) (* x x)))) (IMPLIES (NOT (>= x 0)) (AND (>= (- 0 x) 0) (== (* (- 0 x) (- 0 x)) (* x x))))))
After reducing comparisons into facts:
(IMPLIES true (AND (IMPLIES (NOT x<0) (AND (NOT x<0) (== 0 0))) (IMPLIES (NOT (NOT x<0)) (AND (NOT x>0) (== 0 0)))))
Simplified:
(AND (IMPLIES (NOT x<0) (NOT x<0)) (IMPLIES x<0 (NOT x>0)))
Axioms about comparisons:
(IMPLIES x<0 (NOT x>0))
(IMPLIES x>0 (NOT x<0))
Simplified counter hypothesis: (NOT (AND (IMPLIES (NOT x<0) (NOT x<0)) (IMPLIES x<0 (NOT x>0))))
Using evaluation method for 2 variables
Provable statement (AND (AND (NOT (AND (IMPLIES (NOT x<0) (NOT x<0)) (IMPLIES x<0 (NOT x>0)))) (IMPLIES x<0 (NOT x>0))) (IMPLIES x>0 (NOT x<0)))
:proved
```

Доведення програми з циклом - division.js

```
/*
  PRE:
      x >= 0;
      y > 0
  POST:
      x == q * y + r;
      r < y;
      q >= 0;
      r >= 0;
*/

r = x;
q = 0;

/* INV: x == q * y + r
   BOUND: r+1
 */
while (y<=r) {
    r = r-y;
    q = 1+q;
}
```

>java -jar prover.jar division.js

```
Loop condition (AND (AND (AND (AND (== x (+ (* q y) r)) (IMPLIES (AND (<= y r) (== x (+ (* q y) r))) (== x (+ (* (+ 1 q) y) (- r y))))) (IMPLIES (AND (<= y r) (== x (+ (* q y) r))) (> (+ r 1) 0))) (IMPLIES (AND (<= y r) (== x (+ (* q y) r))) (< (+ (- r y) 1) (+ r 1)))) (IMPLIES (AND (NOT (<= y r)) (== x (+ (* q y) r))) (AND (AND (AND (== x (+ (* q y) r)) (< r y)) (>= q 0)) (>= r 0))))
Program correctness hypothesis:
(IMPLIES (AND (>= x 0) (> y 0)) (AND (AND (AND (AND (== x (+ (* 0 y) x)) (IMPLIES (AND (<= y x) (== x (+ (* 0 y) x))) (== x (+ (* (+ 1 0) y) (- x y))))) (IMPLIES (AND (<= y x) (== x (+ (* 0 y) x))) (> (+ x 1) 0))) (IMPLIES (AND (<= y x) (== x (+ (* 0 y) x))) (< (+ (- x y) 1) (+ x 1)))) (IMPLIES (AND (NOT (<= y x)) (== x (+ (* 0 y) x))) (AND (AND (AND (== x (+ (* 0 y) x)) (< x y)) (>= 0 0)) (>= x 0)))))
After reducing comparisons into facts:
(IMPLIES (AND (NOT x<0) y>0) (AND (AND (AND (AND (== 0 0) (IMPLIES (AND (NOT y-x>0) (== 0 0)) (== 0 0))) (IMPLIES (AND (NOT y-x>0) (== 0 0)) x>-1)) (IMPLIES (AND (NOT y-x>0) (== 0 0)) y>0)) (IMPLIES (AND (NOT (NOT y-x>0)) (== 0 0)) (AND (AND (AND (== 0 0) y-x>0) (>= 0 0)) (NOT x<0)))))
```

```
Simplified:
(IMPLIES (AND (NOT x<0) y>0) (AND (AND (IMPLIES (NOT y-x>0) x>-1) (IMPLIES (NOT y-x>0)
y>0)) (IMPLIES y-x>0 (AND y-x>0 (NOT x<0)))))
Axioms about comparisons:
(IMPLIES (NOT x<0) x>-1)
(IMPLIES (NOT x>-1) x<0)
Simplified counter hypothesis: (NOT (IMPLIES (AND (NOT x<0) y>0) (AND (AND (IMPLIES
(NOT y-x>0) x>-1) (IMPLIES (NOT y-x>0) y>0)) (IMPLIES y-x>0 (AND y-x>0 (NOT x<0))))))
Using evaluation method for 4 variables
Provable statement (AND (AND (NOT (IMPLIES (AND (NOT x<0) y>0) (AND (AND (IMPLIES (NOT
y-x>0) x>-1) (IMPLIES (NOT y-x>0) y>0)) (IMPLIES y-x>0 (AND y-x>0 (NOT x<0)))))) (IM-
PLIES (NOT x<0) x>-1)) (IMPLIES (NOT x>-1) x<0))
:proved
```