Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт прикладной математики и механики

Работа допущена к защите
Руководитель образовательной программы
«Прикладная математика и информатика»
_____ К.Н. Козлов
«_____» _____ 2021 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
РАБОТА БАКАЛАВРА
ЭКСПЕРЕМЕНТЫ С РЕАЛИЗАЦИЕЙ ЯЗЫКА ОХРАНЯЕМЫХ КОМАНД ДЕЙКСТРЫ

по направлению подготовки 01.03.02 Прикладная математика и информатика
Направленность (профиль) 01.03.02_02 Системное программирование

Выполнил
студент гр. 3630102/70201 М.А. Соломатин

Руководитель
проф.,
д-р. техн. наук, ст. науч. сотр. Ф.А. Новиков

Консультант
по нормоконтролю Л.А. Арефьева

Санкт-Петербург
2021

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт прикладной математики и механики

УТВЕРЖДАЮ
Руководитель образовательной программы
«Прикладная математика и информатика»
_____ К.Н. Козлов
«_____» _____ 2021г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы студенту

Соломатину Макару Александровичу гр. 3630102/70201

1. Тема работы: Эксперементы с реализацией языка охраняемых команд Дейкстры.
2. Срок сдачи студентом законченной работы: июнь 2021.
3. Исходные данные по работе:
   - Синтаксическое описания языка охраняемых команд
   - Денотационная семантика языка охраняемых команд
   - Теоретические примеры программ

   Инструментальные средства:
   - Язык программирования Python
   - Генератор лексических и синтаксических анализаторов ANTLR4
   - Библиотеки python-antlr4 и sympy

   Ключевые источники литературы:
   - Дейкстра Э. Дисциплина программирования, 1978.
   - Лавров С.С. Программирование. Математические основы, средства, теория.
   - Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем, 2009.
4. Содержание работы (перечень подлежащих разработке вопросов):
   4.1. Постановка задачи
   4.2. Обзор литературы по теме ВКР

4.3. Исследование программных продуктов
4.4. Разработка интерпретатора и анализатора
4.5. Эксперементы с программами на реализованном языке
4.6. Выводы

5. Дата выдачи задания: 20.12.2020.

Руководитель ВКР _____Ф.А. Новиков

Задание принял к исполнению 20.12.2020

Студент _____М.А. Соломатин

РЕФЕРАТ

На 30 с., 0 рисунков, 0 таблиц, 1 приложение

КЛЮЧЕВЫЕ СЛОВА: РАЗРАБОТКА ИНТЕРПРЕТАТОРА, ЯЗЫК ОХРАНЯЕМЫХ КОМАНД, ФОРМАЛЬНАЯ ВЕРИФИКАЦИЯ ПРОГРАММ, ПОШАГОВОЕ УТОЧНЕНИЕ.

Тема выпускной квалификационной работы: «Эксперементы с реализацией языка охраняемых команд Дейкстры» .

В данной работе изложен подход к доказательству корректности программ с помощью построения их денотационной семантики на основе определенной семантики операторов языка. Для этого рассмотрен и реализован язык охраняемых команд, предложенный Э. Дейкстрой в своем труде «Дисциплина программирования». Реализация языка состоит из интерпретатора программ и программного инструмента, выводящего предусловие программы по заданному постусловию. Рассмотрены примеры доказательства корректности некоторых алгоритмов с помощью этого инструмента. Язык охраняемых команд вместе с разработанными инструментами подходит как для построения корректных алгоритмов, так и для их публикации вместе с доказательством их корректности. Язык охраняемых команд хоть и является удобным для проведения формальных рассуждений, он не снимает с плеч программиста необходимости проектировать программный код, спецификацию программы и ее инварианты. Поэтому разработанный язык носит вспомогательный характер и требует творческих усилий при проектировании корректных программ.

ABSTRACT

30 pages, 0 figures, 0 tables, 1 appendices

KEYWORDS: INTERPRETER DEVELOPMENT, GUARDED COMMAND LANGUAGE, PROGRAM FORMAL VERIFICATION, STEPWISE REFINEMENT.

The subject of the graduate qualification work is «Experiments with the implementation of the Dijkstra's guarded command language».

This work presents an approach to proving the correctness of programs by constructing ther denotational semantics based on defined semantics of language operators. To do this, there was considered and implemented the guarded command

language, proposed by E. Dijkstra in his work «A discipline of programming». The implementation of the language consists of a interpreter and a software tool that derives program precondition for a given postcondition and invariants. Some examples of proving the correctness of algorithms are considered. The guarded command language, alongside with the developed tools, is suitable both for building correct algorithms and for publishing them alongside with their correctness proof. Although the guarded command language is convenient for conducting formal reasoning, it does not remove the need to designb the program specification, program code and it's invariants from the programmer shoulders. The developed language is auxiliary and requires creative efforts when designing correct programs.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные программные системы становятся все более сложными, а требования к их надежности возрастают. Процесс верификации программных систем обычно ограничивается их тестированием, то есть подтверждением их правильности в некоторых частных случаях входных данных. Строгое формальное доказательство программ является крайне трудоемким, поэтому выполняется только в критических местах программного обеспечения, цена ошибки в котором слишком велика. Примером такого программного обеспечения является прошивка бортовых компьютеров ракет и самолетов, военные оборонные системы, прошивка медицинского оборудования.

Наиболее распространенными методами формальной верификации программ являются *проверка моделей*[0] и доказательство теорем в некотором, соответствующем исследуемому языку *исчислении программ*. Проверка моделей пригодна для использования, когда программная система моделируется несколькими взаимодействующими конечными автоматами, и определяет ее соответствие спецификации – набору формул в темпоральной логике. Существует множество программных реализаций для автоматической проверки моделей: SPIN, NuSMV, BLAST, TLA+. Некоторые из них предлагают специальный язык для построения моделей, а остальные выполняют статический анализ программ для уже имеющихся языков программирования общего назначения. Второй подход основан на аксиоматической семантике языка: для того, чтобы доказать корректность программы, требуется вывести из аксиом формулу специального вида для этой программы. Аксиомы при этом отражают семантику операторов языка. Как и в случае с проверкой моделей, существуют системы для автоматического вывода формул в исчислении программ: Coq, Idris, PVS.

Э. Дейкстра в своей работе «Дисциплина программировния» [0] рассматривает другой подход к созданию программ, в котором доказательство их корректности является одним из главных этапов их построения. Таким образом, корректность программы устанавливается не после построения, а в процессе. С этой целью он предлагает язык программирования и снабжает его множеством примеров.

При построении сложных алгоритмов используется подход, предложенный Н.Виртом в статье «Program Development by Stepwise Refinement»[0]. В методе пошагового уточнения весь процесс создания программы состоит из нескольких уточняющих шагов. Программа представляет собой набор инструкций, и на каждом

шаге уточнения выбирается одна из них и декомпозируется в несколько более детальных инструкций. Весь процесс продолжается до тех пор, пока каждая инструкция не будет являться оператором языка программирования или инструкцией вычислительной машины.

Объектом исследования работы является практическая реализация интерпретатора языка и инструмента для статической верификации кода программ, а также проверка реализованного языка на примерах построения корректных программ.

Предметом исследования работы является реализация языка охраняемых команд Дейкстры и инструмента для построения денотационной семантики программ с помощью заданных инвариантов в программном коде. Язык охраняемых команд должен содержать средства, позволяющие демонстрировать процесс построения программ на этом языке методом пошагового уточнения.

Целью исследований является проверка разработанного языка охраняемых команд на конкретных примерах программ.

Для достижения этой цели необходимо решить следующие задачи:

A. описать язык формально – определить грамматику языка и все его операторы
B. разработать интерпретатор программ на языке охраняемых команд, исполняющий ее и выводящий результат
C. разработать программу-анализатор, позволяющей по исходному тексту программы выводить предусловие, если постусловие и инварианты заданы пользователем
D. провести эксперементы разработанных инструментов на примерах программ

Теоретической основой работы послужили исследования Э. Дейкстры в своей работе «Дисциплина программирования»[0], дополненные результатами, полученными С.С. Лавровым в книге «Математические основы, средства, теория»[0], а также предложенный Н. Виртом[0] метод пошагового уточнения. Практическая часть работы выполнялась на основании примеров, предложенных Э. Дейкстрой.

ГЛАВА 1. ДЕНОТАЦИОННАЯ СЕМАНТИКА

В этой главе рассматривается денотационный способ задания семантики языка программирования путем сопоставления операторам языка математических объектов – множеств состояний и отношений между их характеристическими функциями.

1.1. Семантика языка программирования

Построение семантики языка программирования – формальное описание смысла, придаваемого его элементам – выражениям, предложениям, операторам. Существуют следующие способы определения семантики:

A. *Операционный подход* – смысл программы описывается в терминах реальной или абстрактной вычислительной машины, на которой она исполняется.

B. *Аксиоматический подход* – вводится формальная теория с набором аксиом, отражающим свойства основных конструкций языка, а свойства программ являются теоремами в этой формальной теории.

C. *Денотационный подход* – каждым синтаксическим единицам языка сопоставляется математический объект в виде совокупности множеств и отношений.

Один и тот же язык может иметь сразу несколько формальных описаний семантики, дополняющих друг друга; в таком случае семантика языка описывается с разных сторон.

1.2. Множество состояний

Любая программа, предназначенная для выполнения на вычислительном устройстве, оперирует с определенным набором именованных ячеек памяти – переменных. Совокупность значений переменных на некоторый момент времени называется состоянием программы в этот момент времени. Множеством всех возможных состояний программы является прямое произведение областей значений каждой входящей в него переменной. Однако не все комбинации значений переменных допустимы, поэтому рассматривается не все множество состояний программы, а его подмножество допустимых состояний в разных моментах испол-

нения программы. Программа может изменять свое состояние как изменением значений имеющихся переменных, так и добавлением новых. Экспоненциальный рост числа состояний программы в зависимости от количества переменных дает основание считать множество возможных состояний практически бесконечным.

Для описания подмножеств множества состояний используется формулы исчисления предикатов. Они содержат переменные состояния и истинны на некотором подмножестве множества состояний, которое они характеризуют.

Состояние, в котором программа начинает свою работу, называется *начальным состоянием*. Состояние, в котором программа остается после завершения своей работы, называется *заключительным состоянием*.

1.3. Предусловие и постусловие

Рассмотрим некоторую программу S языка охраняемых команд. Предикат R называется *постусловием* программы S, если он истинен на заключительном состоянии программы. Поскольку конечное состояние зависит от начального состояния, то и постусловие может быть истинно или ложно в зависимости от начального состояния. Предикат $wp(S, R)$ называется *слабейшим предусловием* программы S с постусловием R, если из любого удовлетворяющего ему начального состояния программа обязательно завершит свою работу и останется в конечном состоянии, удовлетворяющем предикату R. При этом под термином *слабейшее* понимается то, что предикат $wp(S, R)$ характеризует наибольшее число начальных состояний, обладающих выше упомянутым свойством. Мы будем рассматривать более широкое множество начальных состояний – таких, для которых завершаемость программы не гарантирована. Предикат $wlp(S, R)$ называется *свободным слабейшим предусловием* программы S с постусловием R, если из любого удовлетворяющего ему начального состояния, если программа завершается, то она остается в состоянии, удовлетворяющем R. Заметим, что формула

$$wlp(S, R) \implies wp(S, R) \tag{1.1}$$

справедлива для всех состояний.

Таким образом каждая программа S однозначно определяет некоторую функцию $wlp(S, R)$, вычисляющую свободное слабейшее предусловие этой программы по заданному постусловию. Функция $wlp(S, R)$ также называется преобразователем предикатов для программы S.

Вычисление свободного слабейшего предусловия для программы на языке охраняемых команд является одной из задач настоящей работы.

1.4. Выводы

Описана денотационная семантика языка программирования с помощью сопоставления каждой программе преобразователя предикатов. Получение семантики программы позволяет формально доказать корректность составленной программы. Однако процесс построения денотационной семантики может быть затруднительным, как будет видно из следующей главы.

ГЛАВА 2. ОПИСАНИЕ ЯЗЫКА

В этой главе изложено синтаксическое описание и семантика языка охраняемых команд. Поэтапно определяются используемые в нем операторы и понятие охраняемой команды. В следующих определениях предикатами T и F обозначены предикаты, истинные и ложные соответственно на всех возможных состояниях программы.

2.1. Оператор «skip»

Оператор *skip* обладает следующей семантикой:

$$wp(skip, R) = R \tag{2.1}$$

Он оставляет программму в том же состоянии, какое было и перед ее выполнением. Иначе, он не производит никаких вычислений и не оказывает эффекта на состояние программы.

2.2. Оператор «abort»

Другим очевидным примером семантики является

$$wp(abort, R) = F \tag{2.2}$$

Оператор, обладающей такой семантикой, называется оператором *abort*. Он не может привести к успешному завершению ни при каком начальном состоянии.

Заметим, что не может быть оператора, обладающего семантикой:

$$wp(abort, R) = T \tag{2.3}$$

Действительно, если бы такой оператор существовал, то из состояния, удовлетворяющего предикату F, он бы должен был завершиться в состоянии, удовлетворющем предикату T. Однако не существует начального состояния, удовлетворяющего F.

2.3. Оператор присваивания

Оператор присваивания связывает значение некоторого выражения E с переменной x и записывается в виде:

$$x := E \tag{2.4}$$

Пусть R – постусловие, а предикат $R_{E \to x}$ получен из R подстановкой выражения E вместо всех вхождений переменной x. Тогда

$$wp(x := E, R) = R_{E \to x} \tag{2.5}$$

2.3.1. Множественное присваивание

Естественным расширением оператора присваивания является *оператор множественного присваивания*, который записывается в виде:

$$x_1, x_2 \ldots, x_n := E_1, E_2 \ldots, E_n \tag{2.6}$$

Он вычисляет значения выражений $E_i,\ i = 1,2 \ldots, n$ и связывает соответственно переменные $x_1, x_2 \ldots, x_n$ с их значениями. Тогда

$$wp(x_1, x_2 \ldots, x_n := E_1, E_2 \ldots, E_n, R) = R_{E_1 \to x_1, E_2 \to x_2 \ldots, E_3 \to x_3} \tag{2.7}$$

Предикат $R_{E_1 \to x_1, E_2 \to x_2 \ldots, E_3 \to x_3}$ получается из предиката R подстановкой выражений $E_1, E_2 \ldots, E_3$ вместо всех вхождений переменных $x_1, x_2 \ldots, x_n$ соответственно.

2.3.2. Литералы и типы выражений

В разработанном языке охраняемых команд есть выражения двух типов: численные и логические. Вычисление численного выражения производит вещественное число, а вычисление логического выражения производит истинностное значение: *True* или *False*. Численное выражение является:

A. Либо целочисленным или вещественным литералом;
B. Либо идентификатором переменной;
C. Либо суммой, разностью, произведением или частным двух численных выражений;

Логическое выражение является:

A. Либо литералом True или False;
B. Либо конъюнкцией, дизъюнкцией, отрицанием, импликацией логических выражений;
C. Либо сравнением двух численных выражений;

Для любых выражений допустимо заключать подвыражения в скобки для повышения приоритета вычислений операндов выражения.

2.4. Композиция операторов

До этого момента рассматривались только однооператорные программы. Рассмотрим возможность составлять программы из нескольких операторов. Композиция операторов – программа вида

$$S1; S2$$

При композиции операторов программа обладает следующей семантикой:

- Выполняется оператор S1
- Если выполнение оператора S1 завершилось, то выполняется оператор S2 с начальным состоянием, равным конечному состоянию оператора S1

Сопоставим композиции операторов ее преобразователь предикатов. Пусть R – постусловие композиции операторов $S_1; S_2$. Это значит, что выполнение оператора S_2 должно оставить программу в состоянии, удовлетворющем предикату R. Тогда предусловием оператора S_2 будет являться $wp(S_2,R)$. Это то состояние, в котором должна находиться программа после выполнения оператора S_1. Соответствующее предусловие оператора S_1 есть $wp(S_1, wp(S_2, R)))$. Итак, получили преобразователь предикатов для композиции операторов:

$$wp(S_1; S_2, R) = wp(S_1, wp(S_2, R)) \tag{2.8}$$

В практическом смысле это означает, что для получения предусловия композиции двух операторов необходимо последовательно получить предусловия

каждого оператора начиная с последнего; этот процесс называется *обратным выводом*.

Несложным образом, определяя трехместную композицию операторов $S_1; S_2; S_3$ как $S_1; (S_2; S_3)$ получим следующий преобразователь предикатов для нее

$$wp(S_1; S_2; S_3, R) = wp(S_1, wp(S_2, wp(S_3, R))) \tag{2.9}$$

Аналогичным образом строится преобразователь предикатов для n-местной композиции $S_1; S_2; \ldots; S_n$.

Заметим, что композиция операторов является ассоциативной:

– $wp(S_1; (S_2; S_3), R) = wp(S_1, wp(S_2; S_3, R)) = wp(S_1, wp(S_2, wp(S_3, R)))$
– $wp((S_1; S_2); S_3, R) = wp(S_1; S_2, wp(S_3, R)) = wp(S_1, wp(S_2, wp(S_3, R)))$

2.5. Охраняемая команда

Понятие охраняемой команды является центральным в языке охраняемых команд. Охраняемой командой называется конструкция вида

$$B \to S \tag{2.10}$$

где B – логическое выражение, называемое *предохранителем* или *сторожевым условием*, а S – оператор или композиция операторов.

Команда S может быть выполнена только в том случае, если логическое выражение-предохранитель B имеет истинное значение на текущем состоянии программы, т.е. на состоянии, в котором программа приступила к выполнению охраняемой команды.

Сама охраняемая команда не является оператором языка, но служит составной часть для *условного оператора* или оператора *цикла*.

2.6. Условный оператор

Условный оператор записывается в виде одной или нескольких охраняемых команд, заключенных в лексемы if и fi:

$$\textbf{if } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ if} \tag{2.11}$$

Условный оператор обладает следующей семантикой:

A. Если ни один из предохранителей $B_1, B_2, \ldots, B_n$ не имеет истинное значение на текущем состоянии программы, то выполнение условного оператора не приводит к завершению программы. В этом случае он эквивалентен оператору *abort*.

B. Пусть предохранители $B_{i_1}, B_{i_2}, \ldots, B_{i_m}$, где $\{i_1, i_2, \ldots, i_m\}$ – размещение множества $\{1, 2, \ldots, n\}$, имеют истинное значение на начальном состоянии условного оператора. Тогда случайным образом выбирается к исполнению оператор из множества $\{S_{i_1}, S_{i_2}, \ldots, S_{i_m}\}$.

Таким образом, условный оператор является *недетерминированным*: его выполнение с одним и тем же начальным состоянием может приводить к различным конечным состояниям. Недетерминированности выбора охраняемой команды для исполнения не возникает в случае, когда все предохранители попарно исключают друг друга. Детерминированный вариант условного оператора выбирает первую команду, чей предохранитель имеет истинное значение.

Построим преобразователь предикатов, соответствующей условному оператору. Пусть задано постусловие R условного оператора

$$S_{if} = \textbf{if } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ if} \tag{2.12}$$

Для завершения выполнения оператора необходимо, чтобы хотя бы один предохранитель был истинен на начальном состоянии, откуда приходим к следующему предикату:

$$B_1 \vee B_2 \vee \ldots \vee B_n \tag{2.13}$$

Помимо этого, у каждой команды, у который предохранитель истинен, конечное состояние должно удовлетворять R. Поэтому получаем следующее предусловие для условного оператора:

$$wp(S_{if}, R) = (B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (\forall i : B_i \Rightarrow wp(S_i, R)) \tag{2.14}$$

В практическом смысле это означает, что для нахождения предусловия условного оператора необходимо вычислить предусловия всех охраняемых команд, и подставить их в формулу (2.14).

2.7. Оператор цикла

Оператор цикла записывается в виде одной или нескольких охраняемых команд, заключенных между лексемами $do \ldots od$:

$$\textbf{do } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ od} \tag{2.15}$$

Семантика оператора цикла следующая:

A. Если ни один из предохранителей $B_1, B_2, \ldots, B_n$ не имеет истинного значения на текущем состоянии программы, то оператор цикла завершает свою работу

B. Пусть предохранители $B_{i_1}, B_{i_2}, \ldots, B_{i_m}$, где $\{i_1, i_2, \ldots, i_m\}$ – размещение множества $\{1, 2, \ldots, n\}$, имеют истинное значение на начальном состоянии оператора цикла. Тогда случайным образом выбирается к исполнению оператор из множества $\{S_{i_1}, S_{i_2}, \ldots, S_{i_m}\}$. Затем заново вычисляются истинностные значения всех предохранителей на новом состоянии программы, полученном после выполнения одной из охраняемых команд, и выполняется переход к предыдущему шагу.

Соответствующий преобразователь предикатов выражается через преобразователь предикатов для условного оператора, имеющего те же охраняемые команды.

Пусть

$$S_{do} = \textbf{do } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ od} \tag{2.16}$$

$$S_{if} = \textbf{if } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ if} \tag{2.17}$$

Дадим рекурсивное определение предикатов $H_0(R),\ k = 1, 2 \ldots$:

$$H_0(R) = R \wedge \neg(\exists j : 1 \leqslant j \leqslant n : B_j) \tag{2.18}$$

Для любого $k > 0$:

$$H_k(R) = wp(S_{if}, H_{k-1}(R)) \vee H_0(R) \tag{2.19}$$

Тогда преобразователь предикатов для S_{do} определяется как

$$wp(S_{do}, R) = (\exists k : k \geqslant 0 : H_k(R)) \tag{2.20}$$

Предикат $H_k(R)$ является слабейшим предусловием того, что оператор цикла S_{do} завершится не более чем за k выборок.

Формула (2.17) означает предусловие, в котором должна находиться программа в случае, если все предохранители ложны. В этом случае не производится вычисление ни одной охраняемой команды.

Формула (2.18) при $k > 0$ имеет два дизъюнктивных слагаемых: либо все предохранители оказались ложны, и это произошло меньше чем за k выборок, либо есть истинные предохранители, и тогда выполняются действия, семантически эквивалентные однократному выполнению оператора S_{if}. После этой выборки программа должна оказаться в таком состоянии, чтобы далее потребовалось не более чем $k - 1$ выборка охраняемой команды для завершения цикла в состоянии, удовлетворющем R.

Формальное определение семантики оператора цикла не имеет большого практического смысла, так как заблаговременно определить количество итераций цикла k представляется невозможным.

2.7.1. Инвариант цикла

Если удается найти предикат, не изменяющий свою истинность после выполнения оператора S_{if}, и при этом истинный до начала исполнения оператора S_{do}, более того достаточно сильный, чтобы из него следовало постусловие цикла, то преобразователь предикатов для оператора цикла можно определить гораздо проще, чем в формуле (2.19). Данные рассуждения следуют из *основной теоремы для оператора цикла*.

Теорема. Пусть оператор цикла S_{do} и предикат P таковы, что предикат

$$(P \wedge (\exists i : 1 \leqslant i \leqslant n : B_i)) \Rightarrow wp(S_{if}, P) \tag{2.21}$$

справедлив для всех состояний. Тогда для оператора цикла S_{do} предикат

$$(P \wedge wp(S_{do}, T)) \Rightarrow wp(S_{do}, P \wedge \neg(i : 1 \leqslant i \leqslant n : B_j)) \tag{2.22}$$

где T – тавтология, также справедлив для всех состояний.

Предикат $w(S_{do}, T)$ характеризует начальные состояний, для которых оператор цикла завершает свою работу. Основная теорема для оператора цикла позволяет считать предикат $P \wedge wp(S_{do}, T)$ предусловием цикла, если:

A. Удается доказать, что цикл завершается. Другими словами, удается найти предикат $wp(S_{do}, T)$
B. Предикат $P \wedge \neg(i : 1 \leqslant i \leqslant n : B_j)$ истинен для всех состояний

2.8. Макросы

Многие алгоритмы удобно строить разбивая исходную задачу на подзадачи, а их, в свою очередь на еще более мелкие подзадачи и так далее. Такой метод проектирования алгоритмов и программного обеспечения носит название *метода пошагового уточнения*. Для того, чтобы использовать этот метод, удобным инструментом внутри языка являются макросы.

В настоящей реализации языка охраняемых команд введено понятие макросов с параметрами. Такие макросы могут быть объявлены и определены отдельно от программы, а затем встроены в место ее вызова с заменой формальных параметров макроса, указанных в определении этого макроса на фактически предоставленные параметры при вызове. Макросы являются синтаксическими и работают с абстрактным синтаксисом, а не исходным текстом. Вызов макроса с параметрами является отдельным оператором языка.

Например макрос, складывающий два числа x и y объявляется следующим образом:

```
add(x,y) :=
    z := x + y
```

Оператор вызова макроса:

```
x := 2; add(x, 2 * x)
```

Программа будет иметь конечное состояние $x = 2; z = 6$

2.9. Задание постусловий и инвариантов

Поскольку исследуется вывод предусловия исходя из заданного пользователем постусловие, необходима возможность его указывать либо вне программного текста, либо внутри него. Было решено встроить постусловие и инварианты цикла в сам программный текст. Таким образом, вместе с операторами программы, которые будут исполнены при интерпретации, содержится и спецификация алгоритма в виде его постусловия и инвариантов.

Постусловие объявляется как логическое выражение, содержащее переменные состояния программы. В этом случае программа принимает вид:

```
S1; S2; ...; Sn {<постусловие>}
```

Каждый оператор цикла содержит свой инвариант, который также является логическим выражением:

```
{<инвариант>} do ... od
```

2.10. Выводы

Описаны операторы языка охраняемых команд и определена их семантика путем сопоставления им некоторого преобразователя предикатов – на множестве предикатов. В приложении приведено описание конкретного синтаксиса языка в расширенной форме Бэкуса-Наура и иллюстрированы синтаксичекие диаграммы Вирта.

ГЛАВА 3. РАЗРАБОТКА ИНТЕРПРЕТАТОРА И АНАЛИЗАТОРА

3.1. Генерация лексического и синтаксического анализатора

Для разработки интерпретатора языка охраняемых команд дейкстры, необходимо разработать:

A. лексический анализатор, разбивающий исходный набор символов программы на лексемы
B. синтаксический анализатор, разбивающий полученный набор лексем на синтаксические единицы и строящий дерево разбора программы
C. обход дерева разбора и выполнение необходимых семантических операций в его узлах

Существует два подхода к разработке лексического и синтаксического анализатора: создание их вручную, или автоматическая генерация. При разработке интерпретатора языка охраняемых команд был выбран второй подход, а в качестве инструмента для генерации анализаторов был использован ANTLR4[0] и библиотека python-antlr4[0] для языка программирования Python.

Генератор ANTLR4 использует грамматику языка, заданную в специальном формате, похожем на РБНФ. Грамматика ANTLR4 изложена в приложении Приложении 1.

Синтаксический анализатор получает на вход исходный текст программы и выполняет две задачи:

A. Проверяет, что текст программы составлен правильно в соответствии с грамматикой языка
B. Строит дерево разбора программы в соответствии с правилами вывода грамматики

3.2. Интерпретация программы

Интерпретация программы выполняется путем обхода ее дерева разбора. Корневой узел этого дерева представляет из себя всю программу. Его дочерними узлами являются операторы, упорядоченные в порядке их нахождения в исходной программе.

Например, для программы $x := 2; y := 3; z := x + y$, дочерними узлами корневого узла являются три оператора присваивания. Узлы разных операторов имеют специфичные для них семантические операции. Далее изложены все семантические операции, выполняемые для каждого типа оператора.

3.2.1. Оператор присваивания

Оператор присваивания содержит два важных дочерних узла: имя переменной, стоящее слева от знака присваивания, и выражение, стоящее справа. Для вычисления выражений используется стек выражений, на который кладутся результаты выполнения арифметических или логических операций, а снимаются с него перед их вычислением. Выражение справа от знака присваивания вычисляется, и его значение оказывается на вершине стека. Связь между именем переменной и ее значением сохраняется в отдельном словаре.

3.2.2. Условный оператор

Условный оператор содержит один или несколько узлов, соответствующих охраняемым командам. Узел охраняемой команды, в свою очередь, содержит сторожевое условие, представляемое логическим выражением, и список операторов, охраняемых этим предохранителем.

При посещении узла условного оператора, поэтапно выполняются следующие шаги:

A. Вычисляются логические значения предохранителей всех охраняемых команд, содержащихся в условном операторе.
B. Если среди них не находится ни одного истинного значения, программа завершает свою работу, поскольку в этом случае условный оператор не завершается успешно.
C. Среди охраняемых команд, чей предохранитель истинен, случайным образом выбирается одна из них. Случайность выбора получается путем генерирования псевдослучайного целого числа.
D. Вызывается процедура посещения узла списка операторов, соответствующего выбранной охраняемой команде.

3.2.3. Оператор цикла

Оператор цикла не отличается по своей синтаксической структуре от условного оператора, за исключением лексем do и od, обрамляющих охраняемые команды. При его посещении, поэтапно выполняются следующие шаги:

A. Вычисляются логические значения предохранителей всех охраняемых команд, содержащихся в условном операторе.
B. Если среди них не находится ни одного истинного значения, посещение узла завершается.
C. Среди охраняемых команд, чей предохранитель истинен, случайным образом выбирается одна из них. Случайность выбора получается путем генерирования псевдослучайного целого числа.
D. Вызывается процедура посещения узла списка операторов, соответствующего выбранной охраняемой команде.
E. Выполняется переход к шагу 1.

3.2.4. Оператор вызова макроса

Узел оператор вызова макроса имеет следующие важные дочерние узлы: имя макроса и список аргументов, передаваемых макросу. При посещении этого узла:

A. Выполняется поиск списка операторов, соответствующего этому имени макроса. Такое соответствие сохраняется в словаре при объявлении макросов.

B. Создается копия этого списка операторов и заменяются все вхождения формальных параметров макроса, указанных в его объявлении, на фактически переданные аргументы.
C. Вызывается процедура посещения узла полученного списка операторов.

3.3. Вывод предусловия

Для вывода предусловия используется тот же лексический и синтаксический анализатор, что и для интерпретации программы. Однако семантические операции в узлах дерева разбора совершенно другие.

Вся программа состоит из списка операторов и заданного в конце программы желаемого постусловия. Это постусловие необходимо задавать именно для обратного вывода, но совершенно не обязательно для интерпретации. Если интерпретатор встретит постусловие в конце программы, он проверит истинность этого постусловия на полученном конечном состоянии программы.

Для обратного вывода необходимо посещать узлы операторов не в прямом порядке, а в обратном. Таким образом, постусловие "протаскивается"от конца программы к началу, видоизменяясь при встрече с каждым оператором. Далее описаны те семантические операции, которые выполняются при посещении каждого типа оператора.

3.3.1. Оператор присваивания

При посещении узла оператора присваивания $x := E$, совершаются следующие действия:

A. Со стека предикатов снимается последний R
B. В этом предикате все вхождения переменной x заменяются на выражение E
C. Полученный предикат R' кладется обратно на стек

3.3.2. Условный оператор

Для преобразования предиката через условный оператор

$$\textbf{if } B_1 \to S_1 \mid B_2 \to S_2 \mid \ldots \mid B_n \to S_n \textbf{ if} \tag{3.1}$$

сначала со стека предикатов снимается послений R. Необходимо вычислить все предикаты $wp(S_i, R), 0 \leqslant i \leqslant n$. Для вычисления $wp(S_i, R)$:

A. На стек кладется предикат R

B. Вызывается процедура посещения узла S_i

C. Со стека снимается предикат $wp(S_i, R)$

После того, как все предикаты $wp(S_i, R), 0 \leqslant i \leqslant n$ вычислены, на стек кладется предикат

$$(B_1 \vee B_2 \vee \ldots \vee B_n) \wedge (B_1 \Rightarrow wp(S_1, R)) \wedge (B_2 \Rightarrow wp(S_2, R)) \wedge \ldots \wedge (B_n \Rightarrow wp(S_n, R)) \quad (3.2)$$

3.3.3. Оператор цикла

Для вывода предусловия каждый цикл программы должен быть снабжен своим инвариантом P. При посещении узла оператора цикла, верхней на стеке предикат R заменяется на указанный инвариант. Однако, на инвариант наложено ограничение, обсужденное в главе 2. Поэтому при выводе предусловия пользователю предлагается вывести самостоятельно формулу:

$$\mathrm{P} \wedge \neg B_1 \neg B_2 \neg \ldots \neg B_n \Rightarrow R \quad (3.3)$$

Таким образом, вывод предусловия становится полуавтоматическим и требует ручного вывода формул, которые не могут быть выведены автоматически в исчислении предикатов. Такая ситуация возникает, например, когда в инварианте использованы функции, свойства которых неизвестны при обратном выводе.

3.3.4. Оператор вызова макроса

Поскольку макросы работают на уровне синтаксиса языка, вызов макроса всего лишь заменяется вызовом процедуры посещения узла списка операторов это макроса.

3.3.5. Упрощение предикатов

Изложенный способ трансформации предикатов, хоть и являяется удобным для машинной автоматизации, совершенно не пригоден для чтения результирующего предусловия человеком. Дело в том, что с каждым условным оператором или

оператором цикла предикат сильно увеличивается в соответствии с формулой (3.2). Поэтому необходимо сокращать получающиеся предикаты, пользуясь правилами вывода исчисления предикатов.

Для упрощения предикатов используется система компьютерной алгебры в виде библиотеки SymPy[0] языка Python. Для ее использования, при посещении узлов дерева разобра выражений используется специфические для sympy объекты, тем самым строится объект выражения sympy. Такой объект может быть автоматически упрощен, используя правила вывода пропозициональной логики и формальной арифметики.

3.4. Функции в предикатах

В постусловии программы, как и в инвариантах циклов, могут присутствовать вызовы функции с некоторым именем как одна из альтернатив выражения. Например, следующая программа содержит функцию add внутри постусловия:

```
z := x + y
{z == add(x, y)}
```

Очевидным образом будет выведено предусловие

$$x + y == add(x, y) \tag{3.4}$$

Это равенство должно быть истинным для любых переменных состояния x и y, однако поскольку заранее о свойствах функции add ничего не известно, сократить предусловие в $True$ не удается. Данный механизм является мощным инструментом при формальной проверке корректности алгоритма, поскольку таким образом может быть введена произвольная функция с неизвестными свойствами и указано постусловие, ее содержащие, а после вывода предусловия его необходимо рассмотреть, опираясь на известные свойства введенной функции. Общезначимость полученного предусловия не обязательна, как в примере ???, может быть получено некоторое ограничение на множество начальных состояний.

3.5. Выводы

Спроектирован и реализован язык охраняемых команд, содержащий основные конструкции структурного программирования и при этом не является строго

детерминированным. Для него реализован вспомогательный инструмент, который позволяет в полуавтоматическом режиме строить семантику разработанной программы. Однако значительная часть разработки алгоритма и доказательства его корректности остается на плечах его создателя, потому что параллельно с написанием кода алгоритма необходимо:

A. Указать постусловие, отражающее желаемый результат работы программы
B. Для каждого цикла программы вычислить его инвариант
C. Доказать, пользуясь свойствами введенных в предикаты функций, что из инвариантов циклов следует их постусловие, а также упростить полученное предусловие пользуясь теми же свойстами.

ГЛАВА 4. ПРИМЕРЫ ПРОГРАММ

Для эксперементального вывода корректных программ, а также обратного вывода семантики программы по ее исходному тексту, использованы примеры, приведенные Э. Дейкстрой[0] и С.С. Лавровым[0].

4.1. Факториал числа

Следующий алгоритм вычисляет факториал числа N:

```
n, f := N, 1;

{factorial(n) * f == factorial(N) & n >= 1}
do n > 1 -> f, n := f * n, n - 1 od

{f == factorial(N)}
```

Программа состоит из инициализации переменных n и f и цикла, умножающего f на числа от 2 до N. Инвариант цикла

$$P = (n! \cdot f == N! \wedge n \geqslant 1) \tag{4.1}$$

обеспечивает истинность постусловия по окончании этого цикла. Действительно, после окончания работы цикла состояние программы удовлетворяет

предикату $P \wedge n <= 1$. Подставляя P, получаем

$$n! \cdot f == N! \wedge n \geqslant 1 \wedge n \leqslant 1 \vdash n! \cdot f == N! \wedge n == 1 \vdash f == N! \tag{4.2}$$

Выведенное предусловие:

$$N \geqslant 1 \tag{4.3}$$

Для функции $factorial$, введенной в постусловии, необходимо показать, что

$$n == 1 \wedge factorial(N) == f \cdot factorial(n) \vdash f == factorial(N) \tag{4.4}$$

Тогда денотационная семантика программы будет построена, и корректность программы будет формально доказана. Формула (4.4) выводится очевидным образом и не требует знания о свойствах функции $factorial$, поэтому разработанный инструмент выведет ее автоматически.

4.2. Максимум двух чисел

Следующий алгоритм вычисляет максимум из двух чисел a и b:

```
if a > b -> m := a
|  b >= a -> m := b
fi

{(m >= a) & (m >= b) & (m == a || m == b)}
```

Постусловие алгоритма является определением максимума двух чисел; условный оператор преобразует его в тождественно истинный предикат.

4.3. Алгоритм Евклида

Следующая программа вычисляет наибольший общий делитель двух целых чисел X и Y: `{x == gcd(X, Y)}`

```
x, y := X, Y;

{gcd(x, y) == gcd(X, Y) & x >= 0 & y >= 0}
do x > y -> x := x - y
|  x < y -> y := y - x
od
```

Чтобы убедиться в том, что из инварианта

$$P = gcd(x, y) == gcd(X, Y) \wedge x \geqslant 0 \wedge y \geqslant 0 \tag{4.5}$$

и условия окончания цикла $x == y$ следует постусловие $x == gcd(X, Y)$, необходимо воспользоваться свойством наибольшего общего делителя:

$$(x, y) = |x| \text{ при } x = y \text{ и } x, y \geqslant 0 \tag{4.6}$$

Разработанный инструмент для вывода предусловия предлагает вывести эту формулу самостоятельно для окончательной верификации алгоритма.

4.4. Возведение в степень

Следующий алгоритм возводит число X в степень Y:

```
x, y, z := X, Y, 1;

{z * pow(x, y) == pow(X, Y)}
do y != 0 -> y, z := y - 1, z * x od

{z == pow(X, Y)}
```

Когда цикл завершает свою работу выполнено условие $y = 0$, поэтому из инварианта цикла непосредственно следует постусловие.

4.5. Сортировка четырех чисел

Необходимо построить алгоритм, который для заданных чисел $Q1, Q2, Q3, Q4$ обеспечивает в конце своей работы истинность постусловия:

$$R = (q1, q2, q3, q4 \text{ – перестановка чисел } Q1, Q2, Q3, Q4) \wedge (q1 \leqslant q2 \leqslant q3 \leqslant q4) \tag{4.7}$$

Поскольку язык не поддерживает оператора проверки того факта, что четыре заданных числа являются перестановкой других четырех заданных чисел, для этого используются абстрактные функции внутри постусловия и инвариантов.

Программа принимает следующий вид:

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;

{isPermutation(q1, q2, q3, q4, Q1, Q2, Q3, Q4) == true}
do q1 > q2 -> q1, q2 := q2, q1
|  q2 > q3 -> q2, q3 := q3, q2
|  q3 > q4 -> q3, q4 := q4, q3
od

{
  isPermutation(q1, q2, q3, q4, Q1, Q2, Q3, Q4) == true &
  q1 <= q2 &
  q2 <= q3 &
  q3 <= q4
}
```

Понятно, что инвариант, утверждающий то, что числа $q1, q2, q3, q4$ являются перестановкой чисел $Q1, Q2, Q3, Q4$, был истинен до цикла, и остается истинным после каждой итерации обмена значениями двух из этих чисел. Из инварианта и отрицания предохранителей непосредственно следует постусловие алгоритма.

4.6. Выводы

В этой главе были рассмотрены несколько простых примеров алгоритмов, записанных на языке охраняемых команд. Из примеров становится ясно, что наиболее трудоемким занятием при проектировании корректных программ является нахождение инвариантов циклов.

ЗАКЛЮЧЕНИЕ

В ходе данной работы реализован язык охраняемых команд Дейкстры:

A. Описана денотационная семантика языка охраняемых команд путем сопоставления каждым базовым операторам языка своего преобразователя предикатов
B. Разработан интерпретатор языка охраняемых команд, позволяющий исполнять программы и выводить конечное состояние в виде набора переменных и их значений
C. Реализован вывод денотационной семантики программы с помощью заданных пользователем инвариантов и желаемого постусловия программы. Результатом вывода является предусловие, предъявляемое к начальному состоянию программы, и истинность которого гарантирует истинность постусловия в конечном состоянии программы. Для линейно ветвлящихся программ, не включающих операторы цикла, вывод предусловия происходит в автоматическом режиме, так как не требует указания инвариантов. Инварианты указываются в программном коде рядом с циклами, к которым они относятся. Постусловие указывается в конце кода программы.
D. Рассмотрены несколько примеров программ на языке охраняемых команд, корректность которых проверена с помощью разработанных инструментов.

Язык охраняемых команд является удобным средством для реализации подхода создания программ, корректных по построению, который описан Э. Дейкстрой в работе «Дисциплина программирования». Данный подход рассмотрен как альтернатива другим способам формальной верификации программ, таким как проверка моделей или автоматическое доказательство теорем, однако он может использоваться и совместно с ними. Такой подход значительно увеличивает время разработки программ и выдвигает требование к квалификации программиста, поэтому он не может использоваться повсеместно. Разработанный инструментарий может быть вспомогательным средством для реализации этого подхода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

0. *Дейкстра Э.* Дисциплина программирования. — 1-е изд. — Москва: МИР, 1978. — 265 с.

0. *Карпов Ю. Г.* Model checking. Верификация параллельных и распределенных программных систем. — 1-е изд. — СПб: БХВ-Петербург, 2010. — 560 с.

0. *Лавров С. С.* Программирование. Математические основы, средства, теория. — СПб: БХВ-Петербург, 2002. — 320 с.

0. *Новиков Ф.* Дискретная математика для программистов. — 3-е изд. — СПб: Питер, 2009. — 384 с.

0. ANTLR. — URL: https://www.antlr.org/ (visited on 17.06.2021).

0. python-antlr4. — URL: https://github.com/antlr/antlr4/blob/master/doc/python-target.md (visited on 17.06.2021).

0. SymPy computer algebra system. — URL: https://www.sympy.org/ (visited on 17.06.2021).

0. *Wirth N.* Program Development by Stepwise Refinement. Vol. 14. — 4th ed. — Association for Computing Machinery, Inc., 1971. — P. 221–227.

Приложение 1

Грамматическое описание языка

Далее представлена грамматика языка охраняемых команд в расширенной форме Бэкуса-Наура. Метасимволы РБНФ выделены красным цветом, а терминальные цепочки самого языка охраняемых команд выделены синим цветом.

```
<программа> ::= <список операторов> <список макро-функций>
<список операторов> ::= <оператор> { ; <оператор> }
<оператор> ::= <оператор присваивания> | <условный оператор> |
    <оператор цикла> | <вызов макро-функции>
<оператор присваивания> ::= <идентификатор> := <выражение> |
    <идентификатор> <оператор присваивания> <выражение>
<цифра> ::= 0 | ... | 9
<число> ::= <цифра> {<цифра>} [ . <цифра> <цифра> ]
<буква> ::= _ | a | ... | z | A | ... | Z
<идентификатор> ::= <буква> {<буква> | <цифра>}
<выражение> ::= <идентификатор> | <число> | True | False |
    (<выражение>) | -<выражение> | ~<выражение> |
    <выражение> + <выражение> | <выражение> - <выражение> |
    <выражение> * <выражение> | <выражение> / <выражение> |
    <выражение> & <выражение> | <выражение> | <выражение> |
    <выражение> == <выражение> | <выражение> != <выражение> |
    <выражение> > <выражение> | <выражение> >= <выражение> |
    <выражение> < <выражение> | <выражение> <= <выражение> |
    <выражение> >> <выражение>
<охраняемая команда> ::= <выражение> -> <список операторов>
<список охраняемых команд> ::= <охраняемая команда>
    { | <охраняемая команда>}
<условный оператор> ::= if <список охраняемых команд> fi
<оператор цикла> ::= do <список охраняемых команд> od
<список макросов> ::= {<определение макро-функции>}
<определение макросов> ::= <идентификатор>
    ( [<идентификатор> {, <идентификатор>}] ) := <список операторов>
```

```
<вызов макроса> ::= <идентификатор>
    ( [<выражение> {, <выражение>}] )
```