Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Кафедра системного программирования

Моисеенко Евгений Александрович

Реляционная интерпретация многопоточных языков программирования

Магистерская диссертация

Научный руководитель:
к.ф.-м.н. Булычев Д. Ю.

Рецензент:
к.ф.-м.н. Павлов В. А.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY
Software and Administration of Information Systems

Software Engineering

Evgenii Moiseenko

Relational Interpretation of Concurrent Programming Languages

Graduation Thesis

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Vladimir Pavlov

Saint-Petersburg
2018

Оглавление

Введение

Тактовая частота современных процессоров на сегодня подходит к своему теоретическому пределу. По этой причине разработчики аппаратного обеспечения ищут другие способы повысить производительность. Эти поиски привели к появлению сложных многоядерных процессоров, использующих многоуровневую систему кэш-памяти и различные оптимизации, основанные на переупорядочивании инструкций и спекулятивном исполнении.

В полной мере воспользоваться всеми преимуществами многоядерных систем способны только многопоточные программы. Такая программа состоит из множества параллельно исполняющихся потоков, которые могут обмениваться данными через разделяемую память. Использование параллелизма с разделяемой памятью требует от разработчика дополнительных усилий, направленных на обеспечение корректной синхронизации параллельных вычислений. Ситуация осложняется тем, что различные оптимизации, выполняемые компилятором языка программирования и аппаратным обеспечением, могут приводить к неожиданным сценариям поведения многопоточной программы.

$$\begin{array}{l} [\mathtt{x}] := 1; \\ \mathtt{r}_1 := [\mathtt{y}]; \end{array} \;\Big\|\; \begin{array}{l} [\mathtt{y}] := 1; \\ \mathtt{r}_2 := [\mathtt{x}]; \end{array}$$

Рис. 1: Пример программы Store Buffering (SB)

Рассмотрим пример. На рис. 1 показана небольшая программа, называемая Store Buffering (SB). В данной программе два потока выполняют записи в две различные переменные, а затем каждый поток читает из переменной, которую изменял другой поток. Интуитивно, сценарий поведения, в результате которого в обе локальные переменные $\mathtt{r}_1$ и $\mathtt{r}_2$ окажется записано значение `0`, кажется невозможным, так как он не может быть получен в результате чередования отдельных инструкций потоков. Тем не менее, данный сценарий поведения можно наблюдать на современных процессорах, например, на процессорах архитектуры x86. Подобные сценарии поведения многопоточных программ получи-

ли название *слабых сценариев*.

Приведенная в качестве примера программа SB является частью алгоритма Деккера для решения проблемы взаимного исключения [9]. Алгоритм Деккера полагается на тот факт, что чтение разделяемой переменной возвращает последнее записанное значение. Данное предположение может не выполняться при исполнении программы на современных процессорах, допускающих слабые сценарии поведения, что приводит к некорректному поведению наивной реализации алгоритма Деккера.

Таким образом, разработка надежных и корректных многопоточных алгоритмов сложная и нетривиальная задача. По этой причине разработка инструментов для анализа многопоточных программ — актуальная и важная исследовательская задача. Одной из задач анализа многопоточных программ является верификация, то есть проверка, что программа удовлетворяет заданной спецификации. Также интерес представляет задача синтеза кода синхронизации. В данной работе исследуется подход для решения данных задач на базе реляционного языка программирования.

Реляционное программирование — это вид декларативного программирования, в рамках которого программы представляются как набор отношений [4]. Отношения не делают различий между входными и выходными параметрами, благодаря этому одна и та же реляционная программа может быть использована для решения нескольких связанных проблем. Так, например, двухместное отношение $\texttt{sort}^o$, связывающее список с его отсортированной версией, может быть использовано как для сортировки списка, так и для генерации по отсортированному списку всех возможных перестановок.

Среди реляционных программ наибольший интерес представляет реляционный интерпретатор. Такой интерпретатор может быть использован для исполнения программы, проверки, что программа удовлетворяет набору ограничений или для генерации программы, обладающей заданными свойствами [33]. Чтобы воспользоваться преимуществами реляционного интерпретатора разработчику необходимо описать семан-

тику языка в терминах отношений. Таким образом на базе одной реляционной спецификации семантики языка могут быть разработаны различные языковые инструменты.

При разработке реляционных интерпретаторов полезными оказываются такие расширения реляционного программирования, как табличная мемоизация [26, 29] и конструктивное отрицание [5, 18, 25, 27]. Мемоизация позволяет эффективно обходить пространство состояний интерпретатора, а отрицание — проверять, что заданное состояние интерпретатора недостижимо.

В рамках данной работы был реализован реляционный интерпретатор для многопоточного императивного языка программирования. Интерпретатор был разработан на языке OCanren, встроенном в OCaml реляционном языке программирования. В работе демонстрируется, как реляционный интерпретатор может быть применен для верификации многопоточных программ и синтеза синхронизации. Также в ходе работы были разработаны расширения OCanren для поддержки мемоизации и конструктивного отрицания.

Постановка задачи

Целью данной работы является применение идеи реляционной интерпретации к задаче анализа многопоточных программ.

В данной работе были поставлены следующие задачи.

- Разработка реляционного интерпретатора для многопоточных программ.
- Разработка необходимых расширений для языка реляционного программирования OCanren.
- Апробация реляционного интерпретатора.

1. Обзор

В данном разделе будет подробнее рассказано о моделях памяти и способах их формального определения (раздел 1.1), о парадигме реляционного программирования (раздел 1.2) (и в том числе об отрицании и мемоизации в реляционном программировании), а также о языке OCanren (раздел 1.3).

1.1. Модели памяти

Как уже упоминалось во введении, современные компиляторы и аппаратные архитектуры производят множество агрессивных оптимизаций, которые, в контексте многопоточности, могут приводить к слабым поведениям. Для того чтобы абстрагироваться от деталей реализации конкретного компилятора или аппаратной архитектуры обычно рассматривается *модель памяти*.

Модель памяти определяет семантику взаимодействия параллельных потоков с разделяемой памятью. Модель *последовательной консистентности* (sequential consistency, SC) [17] является наиболее простой моделью памяти. В рамках данной модели каждый возможный сценарий поведения программы является результатом некоторого чередования инструкций различных потоков. Современные языки программирования и аппаратные архитектуры предоставляют разработчикам более сложные модели памяти, допускающие слабые сценарии поведения. Такие модели называются *слабыми моделями памяти*. Среди слабых моделей памяти следует отметить модели наиболее распространенных на сегодняшний день архитектур процессоров x86 [34], ARM [21], Power [30], а также модели памяти языков программирования C/C++ [20] и Java [19].

В данной работе будут рассмотрены модели памяти SC, TSO (Total Store Orderin) и SRA (Strong Release-Acquire [16]). Опишем последние две модели подробнее.

Модель памяти TSO разрешает переупорядочивать чтения с более ранними записями (т.е. позволяет буферизировать операции записи).

$$\begin{array}{l} [\mathtt{x}] := 1; \\ \mathtt{r}_1 := [\mathtt{y}]; \end{array} \;\Big\|\; \begin{array}{l} [\mathtt{y}] := 1; \\ \mathtt{r}_2 := [\mathtt{x}]; \end{array}$$

Рис. 2: Пример программы Store Buffering (SB)

Чтобы продемонстрировать отличия данной модели от модели SC, вернемся к рассмотрению программы Store Buffering (рис. 2). Сценарий поведения, в результате которого в обе локальные переменные $\mathtt{r}_1$ и $\mathtt{r}_2$ будет записано значение `0`, невозможен в модели SC. В модели TSO возможен сценарий, при котором запись `[x] := 1` будет переупорядочена с чтением $\mathtt{r}_1$ `:= [y]`, которое прочитает инициализирующее значение `0`. После этого управление перейдет ко второму потоку, и, после записи `[y] := 1`, чтение $\mathtt{r}_2$ `:= [x]` также вернёт значение `0`.

$$\begin{array}{l} [\mathtt{x}] := 1; \\ [\mathtt{f}] := 1; \end{array} \;\Big\|\; \begin{array}{l} \mathtt{r}_1 := [\mathtt{f}]; \\ \mathtt{r}_2 := [\mathtt{x}]; \end{array}$$

Рис. 3: Пример программы Message Passing (MP)

$$\begin{array}{l} [\mathtt{x}] := 1; \\ [\mathtt{f}]_{rel} := 1; \end{array} \;\Big\|\; \begin{array}{l} \mathtt{r}_1 := [\mathtt{f}]_{acq}; \\ \mathtt{r}_2 := [\mathtt{x}]; \end{array}$$

Рис. 4: Пример программы Message Passing (MP+rel+acq)

Модель памяти SRA допускает ещё больше слабых поведений. Рассмотрим вариант программы передачи сообщений (Message Passing, MP, рис. 3). В модели памяти SRA возможна ситуация, при которой в конце исполнения в регистр $\mathtt{r}_1$ будет записано значение `1`, а в $\mathtt{r}_2$ — `0`. Для предотвращения подобных сценариев поведения данная модель предлагает способ синхранизации двух потоков с помощью следующего шаблона: один из потоков должен выполнить *высвобождающую запись* (release-write), а другой поток — *захватывающее чтение* (acquire-read). На рис. 4 представлен вариан программы MP, в котором используются данные операции. Теперь если в регистр $\mathtt{r}_1$ будет прочитано значение `1`, то и в регистр $\mathtt{r}_2$ должно быть прочитано значение `1`.

1.1.1. Операционные семантики

Чтобы рассуждать о многопоточных программах необходим формальный математический аппарат. При определении моделей памяти как правило используется один из двух способов: задание операционной семантики или задание аксиоматической семантики. Операционная семантика определяет каждый сценарий поведения многопоточной программы как трассу некоторой абстрактной машины. Аксиоматическая семантика ставит в соответствие каждому сценарию поведения граф, в котором вершинам соответствуют события, такие как чтения или записи разделяемых переменных, а ребра задают некоторые отношения на множестве событий.

Для задания операционной семантики могут быть использованы системы помеченных переходов (СПП). Формально, система помеченных переходов это четвёрка (S, I, L, R), где S — это множество состояний, $I \subseteq S$ — множество начальных состояний, L — множество меток, а $R \subseteq L \times S \times S$ — отношение перехода. Вместо $(l, s, s') \in R$ для удобства обычно пишут $s \xrightarrow{l} s'$. Трассой, соединяющей состояния s и s' будем называть последовательность переходов $\{l_i\}_{i=1}^{n}$, такую что $\exists s_1, ..., s_{n-1}. s \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} ... \xrightarrow{l_n} s'$. Будем говорить, что состояние s' достижимо из s ($s \rightarrow^* s'$) если существует трасса, соединяющая s и s'. Введём также определение *инварианта* СПП. Рассмотрим одноместный предикат $Safe$, заданный на множестве состояний S. Будем говорить, что СПП удовлетворяет инварианту $Safe$, если в каждом состоянии, достижимом из начального, выполняется $Safe$, т.е. $\forall s.(\exists s_i \in I.\ s_i \rightarrow^* s) \implies Safe(s)$.

1.2. Реляционное программирование

Реляционная программирование — это парадигма программирования, в которой основной абстракцией являются отношения. Преимущества данного подхода в его декларативности и гибкости. Как мы увидим далее, одно и тоже отношение может быть использовано для решения нескольких задач.

Реляционная парадигма наследует многие идеи от парадигм *логи-*

ческого прогаммирования (logic programming) и *программирования в ограничениях* (constraint programming). В частности, большое влияние на данную парадигму оказали языки семейства Prolog [22, 28, 29]. В основе реляционного программирования лежит идея, что метод SLD-резолюции, который является основой логических языков, может быть представлен как монада и встроен в функциональный язык программирования [2]. Одной из первых реализацей этой идеи стал язык программирования MiniKanren [4, 10, 11], встроенный в язык Scheme. Ядро MiniKanren затем было встроено в качестве DSL в различные языки программирования, в том числе Clojure, Racket, Python и другие. Язык OCanren [15] является версией MiniKanren, встроенной в язык OCaml. Его отличительной чертой является статическая типизация. Везде далее мы будем говорить о реляционном программировании в контексте языка OCanren.

```
let append^o xs ys zs =
  ((xs ≡ []) ∧ (ys ≡ zs))
  ∨
  fresh (x xs' zs')
    (xs ≡ x::xs') ∧
    (zs ≡ x::zs') ∧
    (append^o xs' ys zs')
```

Листинг 1: Определение отношения *append*^{o}

Рассмотрим пример реляционной программы. На листинге 1 определено отношение `append`^{o}, связывающее списки `xs` и `ys` с их конкатенацией `zs`. Обратим внимание, что отношения, в отличии от функций, не делают различий между входными и выходными параметрами. Отношение является лишь декларативной спецификацией некоторого множества.

Для того, чтобы с помощью спецификации отношения `append`^{o} выполнить, например, конкатенацию двух конкретных списков необходимо сформировать *запрос*. При запросе пользователь передаёт отношению некоторые аргументы, а на месте других оставляет *свободные переменные*. Ответом на запрос является либо ответ `success` вместе со

```
run * (append^o [1] [2] q) ⇝ success {
  q = [1; 2]
}

run * (append^o q r [1; 2]) ⇝ success {
  q = [], r = [1; 2];
  q = [1], r = [2];
  q = [1; 2], r = [];
}

run * (append^o [] [q] []) ⇝ failure {}

run * (append^o [] q q) ⇝ success {
  q = _0
}
```

Листинг 2: Примеры запросов к append^o

списком (потенциально бесконечным) подстановок для свободных переменных, на которых выполняется отношение, либо ответ `failure`, если отношение не выполняется на переданных аргументах. На листинге 2 показаны примеры запросов к отношению `append`^{o}: первый запрос выполняет конкатенацию двух списков, второй генерирует все возможные разбиения списка на пару списков, результатом конкатенации которых является исходный список, третий запрос проверяет, что конкатенация пустого списка и списка из одного элемента равна пустому списку (данный запрос заканчивается неудачей `failure`), наконец четвертый запрос проверяет, что конкатенация пустого списка с любым другим списком не изменяет список. Свободная переменная $\_0$ в ответе на последний запрос означает, что на её место может быть подставлено любое значение соответствующего типа.

В определении отношения `append`^{o} можно видеть использование основных примитивов реляционного языка программирования: ограничения эквивалентности $\equiv$, связок $\wedge$ и $\vee$, а также конструкции `fresh`. Связки $\wedge$ и $\vee$ служат для объединения нескольких отношений и имеют интуитивно понятную семантику конъюнкции и дизъюнкции. Кон-

струкция `fresh` служит для введения новых *свежих* переменных. В примере `append`^{o} с помощью `fresh` во втором дизъюнкте были введены три новые переменные `x`, `xs'`, `zs'`, которые затем использовались для того, чтобы декомпозировать списки `xs` и `zs` на голову и хвост.

1.2.1. Отрицание

В языке OCanren имеется ещё один примитив, ранее не упомянутый. Это бинарное отношение $\not\equiv$, накладывающее на термы *ограничение неэквивалентности*. С помощью данного примитива можно выразить, например, отношение `remove`^{o}, связывающее терм x, список xs и список ys, равный списку xs из которого было удалено первое вхождение элемента, равного x (листинг 3).

```
let removeᵒ x xs ys =
  ((xs ≡ []) ∧ (ys ≡ []))
  ∨
  fresh (y xs')
    (x ≡ y) ∧
    (xs ≡ y::xs') ∧
    (ys ≡ xs')
  ∨
  fresh (y xs' ys')
    (x ≢ y) ∧
    (xs ≡ y::xs') ∧
    (ys ≡ y::ys') ∧
    (removeᵒ x xs' ys')
```

Листинг 3: Определение отношения $remove^o$

Отношение $\not\equiv$ представляет пользователю очень ограниченную форму отрицания, которой зачастую оказывается недостаточно. Например, при попытке обобщить отношение `remove`^{o} таким образом, чтобы оно принимало произвольный предикат `p`^{o} и связывало список `xs` со списком `ys`, равным списку `xs`, в котором удалено первое вхождение элемен-

та x удовлетворяющего предикату, необходимо использовать отрицание предиката $\mathtt{p}^o$ (листинг 4).

```
let remove^o p^o xs ys =
  ((xs ≡ []) ∧ (ys ≡ []))
  ∨
  fresh (x xs')
    (p^o x) ∧
    (xs ≡ x::xs') ∧
    (ys ≡ xs') ∧

  ∨
  fresh (x xs' ys')
    (¬p^o x)
    (xs ≡ x::xs') ∧
    (ys ≡ x::ys') ∧
    (remove^o p^o xs' ys')
```

Листинг 4: Обобщение отношения $remove^o$

Одним из способов добавления отрицания в реляционное программирование является метод "отрицание как неудача" (Negation as a failure) [6]. Суть данного метода заключается в следующем: при запросе к отрицанию отношения сначала делается запрос к его положительной версии. Если запрос завершается неудачей, то полагается, что его отрицание успешно. Иначе, если запрос завершается успехом, его отрицание считается неудачей. К сожалению, данный подход обладает существенным недостатком: применение отрицания как неудачи к отношению с аргументами, содержащими свободные логические переменные, некорректно (unsound).

Для того чтобы убедиться в этом рассмотрим пример. Код на листинге 5 определяет тип данных список и два предиката $\mathtt{nil}^o$ и $\mathtt{cons}^o$ для проверки, является ли аргумент экземпляром типа с конструктором `Nil` или `Cons`. Листинг также демонстрирует пример применения отрицания как неудачи. Выполняется запрос к коньюнкции двух пре-

дикатов. Первый коньюнкт применяет предикат $\texttt{cons}^o$ к переменной, а второй применет отрицание $\texttt{nil}^o$. Запрос возвращает ожидаемый результат — свободная переменная связалась с экземпляром типа `list` с конструктором `Cons` и произвольными аргументами конструктора. Но если поменять порядок коньюнктов, запрос вернет результат "неудача". Такой неожиданный результат связан с тем, что в данном случае первым происходит вызов отрицания предиката $\neg\texttt{nil}^o\ \texttt{q}$ со свободной переменной `q`. Отрицание выполняет вызов $\texttt{nil}^o\ \texttt{q}$, который заканчивается успехом, так как переменная `q` свободна и, следовательно, может быть связана с конструктором `Nil`. Из успеха $\texttt{nil}^o\ \texttt{q}$, отрицание как неудача заключает, что вызов $\neg\texttt{nil}^o\ \texttt{q}$ должен окончиться неудачей.

```
type α list = Nil | Cons of α * α list

let nil^o x =
  (x ≡ Nil)

let cons^o x =
  fresh (hd tl)
    (x ≡ Cons hd tl)

run * (λ q → cons^o q ∧ ¬nil^o q) ⇝ success {
  q = Cons (_.0, _.1);
}

run * (λ q → ¬nil^o q ∧ cons^o q) ⇝ failure {}
```

Листинг 5: Пример использования отрицания как неудачи

Конструктивное отрицание [5, 18, 25, 27] — это метод отрицания в реляционном программировании, который обобщает метод "отрицания как неудачи".

Проблема отрицания как неудачи в том, что данный метод никак не использует информацию, полученную при выполнении отрицаемого отношения. Рассматривается только успех или неудача запроса в

целом. При конструктивном отрицании на основе списка ответов, полученных при исполнении запроса, строится список ограничений. Немного упростив, можно сказать, что в каждом ответе меняются местами отношения эквивалентности и ограничения неэквивалентности, а также конъюнкции и дизъюнкции.

Конструктивное отрицание применимо не ко всем реляционным программам. Оператор отрицания может быть применен только к запросу, имеющему конечное количество ответов (в противном случае невозможно построить множество ограничений). Ещё одним ограничением является то, что семантика конструктивного отрицания определена только на подклассе реляционных программ, так называемых *стратифицированных* программах.

```
let A x y = ¬B x y
let B x y = x ≡ y
```

Рис. 5: Пример стратифицированной программы

```
let A x y = ¬B x y
let B x y = A x y
```

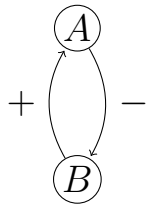

Рис. 6: Пример нестратифицированной программы

Для того, чтобы дать определение стратифицированной реляционной программы, свяжем с каждой программой ориентированный граф. Вершинами в данном графе являются отношения. Между двумя вершинами A и B есть ребро $A \to B$, если отношение B встречается в определении отношения A. Если B встречается в определении A в позитивной форме (т.е. без отрицания), то будем помечать ребро меткой $+$, иначе меткой $-$. Реляционная программа является стратифицированной, если в соответствующем графе нет циклов, содержащих хотя

бы одно ребро с меткой $-$. На рисунке 5 приводится пример стратифицированной программы, а на рисунке 6 — нестратифицированной программы.

Граф стратифицированной программы может быть разбит на компоненты связности по отношению связности по ребрам с меткой $+$. Данные компоненты называются *стратами*. Страты соединены ребрами с меткой $-$. Определение стратифицированной программы гарантирует, что между стратами нет циклических зависимостей.

Применительно к реляционному интерпретатору, основанному на СПП (раздел 1.1.1), отрицание позволяет проверять недостижимость некоторого состояния. В рамках данной работы было реализовано расширение языка OCanren для поддержки метода конструктивного отрицания (раздел 2.1).

1.2.2. Табличная мемоизация

Табличная мемоизация — это метод оптимизации реляционных программ, заключающийся в переиспользовании ответов на подзапросы. Данный метод способен ускорить исполнение запросов к рекурсивным отношениям. Кроме того, применение мемозации к некоторым отношениям позволяет запросам к данным отношениям завершаться за конечное время, в то время как аналогичные запросы к немемоизированной версии отношения не завершаются.

Рассмотрим классический пример. На рисунке 7 слева показано отношение `edge`^{o}, задающее пары связанных вершин в графе, и отношение `reachable`^{o}, задающее отношение достижимости на графе. Справа показано изображение графа, соответствующего заданному отношением `edge`^{o}. Запрос `run * (reachable`^{o} `''a'' q)` к немемоизированной версии отношения завершается успехом и возвращает бесконечный список ответов `{q = ''a''; q = ''b''; q = ''d''; q = ''a''; q = ''b''; q = ''d''; ...}`, в то время как этот же запрос к мемоизированной версии возвращает конечный список ответов `{q = ''a''; q = ''b''; q = ''d''; }`.

Системы помеченных переходов (раздел 1.1.1) также задают граф, а трасса соответствует некоторому пути в графе. При попытке опре-

```
let edge^o x y =
  (x, y) ≡ ("a", "b")  ∨
  (x, y) ≡ ("b", "a")  ∨
  (x, y) ≡ ("b", "d")

let reachable^o x y =
  (x ≡ y)  ∨
  fresh (z)
    (edge^o x z)  ∧
    (reachable^o z y)
```

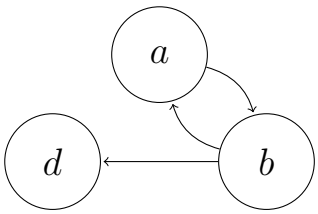

Рис. 7: Определение отношения достижимости на графе

делить отношение достижимости на СПП можно натолкнуться на проблемы аналогичные тем, что возникают при определении отношения `reachable`^{o}. Таким образом, при написании реляционных интерпретаторов на основе СПП наличие поддержки табличной мемоизации в реляционном языке становится критически важным.

Табличный метод мемоизации был реализован в нескольких версиях языка Prolog [28, 29]. В работе [4] представлена реализация данного метода для языка MiniKanren, втроенного в Scheme. В рамках данный работы эта реализация была улучшена (добавлена поддержка ограничений неэквивалентности и использованы более эффективные структуры данных) и встроена в язык OCanren типобезопасным образом.

1.3. Реализация языка OCanren

Для понимания дальнейшего содержания работы, в частности, реализации расширений OCanren, необходимо рассмотреть основные идеи, лежащие в основе реализации этого языка.

Главная идея заключается в том, чтобы встроить интерпретатор реляционного языка в функциональный язык с помощью использования монады с выбором (MonadPlus) [2, 15]. OCanren использует реализацию монады с выбором, основанную на ленивых потоках (Streams).

При исполнении запроса происходит поиск ответов. Использование

монады с выбором позволяет закодировать чередующийся поиск с возвратом (interleaving search). Данный вид поиска является полным в отличии от применяемого в языках семейства Prolog поиска в глубину (depth-first search).

Каждая ветка поиска поддерживает текущее состояние, которое кодирует множество ограничений, накопленных в данной ветке. Состояние кодирует систему консистентных (т.е. выполнимых) ограничений. Если при добавлении очередного ограничения в состояние система становится неконсистентной, данная ветка поиска может быть отсечена.

Выполнимость ограничений эквивалентности может быть проверена с помощью алгоритма унификации [1]. Алгоритм унификации либо приводит выполнимое ограничение $t \equiv u$ в решенную форму (solved form), либо детектирует невыполнимость. Решенная форма для ограничений эквивалентности это подстановка σ, т.е. частичная функция, которая ставит в соответствие логическим переменным термы $\sigma :: V \to T$. Применение данной подстановки к термам делает их синтаксически эквивалентными: $\sigma(t) = \sigma(u)$.

Например, для термов `Cons(1, `$_0$`)` и `Cons(`$_1$`, Nil)` алгоритм унификации возвращает подстановку $\sigma = \{\_0 \mapsto \texttt{Nil};\ \_1 \mapsto \texttt{1}\}$. Для термов `Cons(1, `$_0$`)` и `Nil` такой подстановки не существует, и алгоритм унификации возвращает $\bot$.

Псевдокод процедуры унификации представлен в алгоритме 1. Процедура унификации принимает на вход текущую подстановку для переменных σ и список ограничений $t \equiv u \in L$. Для проверки ограничения $t \equiv u$ на вход унификации подаются $\sigma = \varnothing$ и $L = \{t \equiv u\}$. Результатом работы процедуры является подстановка для переменных σ', если ограничение выполнимо, и $\bot$ иначе.

Разберём, как работает данный алгоритм. Если список L пуст — процедура оканчивает работу, возвращая текущую подстановку. Иначе недетерменированно выбирается одно из ограничений $t \equiv u$. К термам t и u применяется текущая подстановка и выполняется разбор случаев. Если в корне двух термов находятся конструкторы (строки 7-11) и эти конструкторы равны, то в L добавляются новые ограничения на

Алгоритм 1 Алгоритм унификации

```
1:  procedure UNIFY(σ, L)
2:      if L = ∅ then
3:          return σ
4:      pick (t ≡ u) ∈ L
5:      L := L \ (t ≡ u)
6:      switch σ(t), σ(u) do
7:          case C_i^n(t_1, ..., t_n), C_j^m(u_1, ..., u_m)
8:              if i = j then
9:                  L := {t_1 ≡ u_1, ... , t_n ≡ u_n} ∪ L
10:             else
11:                 return ⊥
12:         case x, y when var(x) and var(y)
13:             σ := {x ↦ y} ∪ σ
14:         case x, t = C^n(t_1, ..., t_n) when var(x)
15:             if x ∈ Vars(t) then
16:                 return ⊥
17:             else
18:                 σ := {x ↦ t} ∪ σ
19:     return Unify(U, E, σ, L)
20:
```

равенство аргументов конструкторов. Если конструкторы различны — алгоритм оканчивается неудачей (возвращается $\perp$). Если оба терма являются переменными (строки 12-13), в подстановку добавляются новое связывание (переменная x связывается с y). Наконец, рассматривается случай, когда один из термов является переменной, а в корне другого находится конструктор (строки 14-18). Выполняется проверка, входит ли переменная x в терм t (*occurs-check*). В случае если это так алгоритм заключает, что ограничения невыполнимы. Иначе переменная x связывается с t в подстановке. После разбора всех случаев выполняется рекурсивный вызов процедуры унификации на обновленном множестве L и подстановке σ.

Проверка ограничения $t \not\equiv u$ также выполняется при помощи алгоритма унификации. Чтобы проверить выполнимость $t \not\equiv u$ на вход подается $\sigma = \varnothing$, $L = \{t \equiv u\}$ и проверяется, что в результате процедура возвращает либо $\perp$, либо непустую подстановку $\sigma \neq \varnothing$.

2. Расширения OCanren

Далее приводится описание расширений языка OCanren, реализованных в ходе данной работы.

2.1. Конструктивное отрицание

В данном разделе будет представлена реализация расширения OCanren для поддержки метода конструктивного отрицания. Сперва будет представлена семантика конструктивного отрицания (раздел 2.1.1). Будет показано, что для конструктивного отрицания необходимо добавить в язык поддержку двух новых видов ограничений $\forall \overline{x}.t \not\equiv u$ и $\forall \overline{x} \exists \overline{y}.t \equiv u$. В разделах 2.1.2 и 2.1.3 будут представлены алгоритмы для проверки выполнимости данных ограничений.

2.1.1. Семантика конструктивного отрицания

Для того, чтобы определить семантику конструктивного отрицания будем рассматривать процесс ответа на запрос как процедуру трансляции в логическую формулу [25]. Предположим, что определение отношения содержит только "положительные" вхождения других отношений и не содержит отношений неэквивалентности (будем называть такую программу *положительной*). В таком случае, множество ответов на запрос к такой программе соответствует формуле 1.

$$E_+ = \exists\ \overline{x}. \bigvee_{i=1}^{N} \bigwedge_{j=1}^{M_i} t_{ij} \equiv u_{ij} \tag{1}$$

Таким образом, множество ответов на запрос к отрицанию данного отношения должно соответствовать отрицанию формулы 1 (показано в формуле 2). Заметим, что формула 2 содержит ограничения неэквивалентности на термы, которые могут содержать универсально квантифицированные переменные, т.е. ограничения вида $\forall \overline{x}.t \not\equiv u$. Данные ограничения сводятся к обычным ограничениям неэквивалентности $t \not\equiv u$ в случае когда список универсально квантифицированных переменных $\overline{x}$

пуст.

$$\neg E_+ = \forall\ \overline{x}. \bigwedge_{i=1}^{N} \bigvee_{j=1}^{M_i} t_{ij} \not\equiv u_{ij} \tag{2}$$

Перейдем к рассмотрению отношений, которые также могут содержать как "положительные" так и "отрицательные" вхождения других отношений. Будем рассматривать только стратифицированные программы (раздел 1.2.1). В этом случае, можно разбить реляционную программу на страты, выполнить топологическую сортировку страт и строить ответ на запрос поэтапно, начиная с последней страты в порядке сортировки.

По построению, последней страте соответствует некоторое "положительное" отношение, т.е. в определении данного отношения не встречаются "отрицательные" вхождения других отношений. Это значит, что ответу на запрос к данному отношению соответствует логическая формула 1. Рассмотрим отрицание данной формулы (формула 2) и подставим его на место всех "отрицательных" вхождений данного отношения в предыдущей страте. После трансляции предыдущей страты получим формулу 3.

$$E = \exists\ \overline{x}. \bigvee_{i=1}^{N} ((\bigwedge_{j=1}^{M_i} t_{ij} \equiv u_{ij}) \wedge (\forall \overline{y}. \bigwedge_{k=1}^{L_i} \bigvee_{l=1}^{Q_k} p_{kl} \not\equiv u_{kl})) \tag{3}$$

В результате отрицания формулы 3 получим формулу 4.

$$\neg E = \forall\ \overline{x}. \bigwedge_{i=1}^{N} ((\bigvee_{j=1}^{M_i} t_{ij} \not\equiv u_{ij}) \vee (\exists \overline{y}. \bigvee_{k=1}^{L_i} \bigwedge_{l=1}^{Q_i} p_{kl} \equiv u_{kl})) \tag{4}$$

Формула 4 содержит отношения эквивалентности на термах, которые могут содержать свободные переменные, а также как экзистенциально так и универсально квантифицированные переменные, т.е. ограничения вида $\forall\overline{x}\exists\overline{y}.t \equiv u$. Отношения эквивалентности данного вида не могут быть решены обычным алгоритмом унификации. Тем не менее, существует расширение алгоритма унификации, которое позволяет решать подобные ограничения [18]. Более подробное описание данного

алгоритма унификации приведено в разделе 2.1.3, здесь лишь отметим, что данный алгоритм позволяет получить по формуле вида $\forall\overline{x}\exists\overline{y}.t \equiv u$ формулу вида $\exists y'.t' \equiv u'$. Таким образом, формулу 4 можно переписать, избавившись от ограничений вида $\forall\overline{x}\exists\overline{y}.t \equiv u$, как показано в формуле 5.

$$\neg E = \bigwedge_{i=1}^{N}((\exists\overline{y'}.\bigvee_{k=1}^{L'_i}\bigwedge_{l=1}^{Q'_i} p'_{kl} \equiv u'_{kl}) \vee \forall\ \overline{x}.(\bigvee_{j=1}^{M_i} t_{ij} \not\equiv u_{ij})) \tag{5}$$

Раскрыв внешнюю конъюнкцию в данной формуле по правилу дистрибутивности, получим формулу, аналогичную по форме формуле 3. Данную формулу можно подставить в предыдущую в порядке сортировки страту и повторить процесс. Итеративно повторив этот процесс для всех страт, получим в итоге формулу, соответствующую исходному запросу.

2.1.2. Решение ограничений $\forall\overline{x}.t \not\equiv u$

Вернемся к рассмотрению ограничений вида $\forall\overline{x}.t \not\equiv u$. Необходимо предоставить процедуру, которая сможет проверить выполнимость данных ограничений. Для того чтобы предоставить такой алгоритм, мы обобщили алгоритм для проверки выполнимости обычных ограничений неэквивалентности $t \not\equiv u$.

Прежде, чем переходить к описанию алгоритма, заметим, что ограничение $\forall\overline{x}.t \not\equiv u$ не выполнимо тогда и только тогда, когда существует подстановка для переменных $\overline{x}$, применение которой к термам t и u делает их синтаксически эквивалентными (формуа 6). Таким образом, для опровержения выполнимости $\forall\overline{x}.t \not\equiv u$ достаточно предъявить такую подстановку.

$$\forall\overline{x}.t \not\equiv u \Leftrightarrow \neg\exists x.t \equiv u \tag{6}$$

Теперь обобщим алгоритм проверки выполнимости ограничений неэквивалентности таким образом, чтобы он также пытался построить данную подстановку. Напомним, что алгоритм для проверки ограничения

$t \not\equiv u$ выполняет унификацию термов t и u, т.е. возвращает либо подстановку $\sigma : \sigma(t) = \sigma(u)$, либо $\perp$ если термы не унифицируются. Если термы унифицируются в пустой подстановке $\sigma = \varnothing$, то ограничение не выполнимо, иначе оно выполнимо.

Для того, чтобы проверять ограничения вида $\forall\overline{x}.t \not\equiv u$, достаточно лишь слегка модифицировать алгоритм. В случае, если унификация термов t и u вернула непустую подстановку σ, необходимо проверить, связывает ли эта подставновка только универсально квантифицированные переменные $\overline{x}$. Если это так, то мы доказали невыполнимость ограничения и предъявили контрпример — подстановку σ, такую что $Dom(\sigma) \subseteq \overline{x}$ и $\sigma(t) = \sigma(u)$. Иначе ограничение выполнимо.

2.1.3. Решение ограничений $\forall\overline{x}\exists\overline{y}.t \equiv u$

Перейдем к рассмотрению ограничений $\forall\overline{x}\exists\overline{y}.t \equiv u$. Для проверки выполнимости данных ограничений мы воспользуемся модификацией алгоритма унификации [18]. Модифицированный алгоритм позволяет обрабатывать встречающиеся в термах универсально и экзистенциально квантифицированные переменные при условии, что квантор всеобщности предшествует квантору существования (это как раз случай ограничений $\forall\overline{x}\exists\overline{y}.t \equiv u$). Заметим, что помимо квантифицированных переменных в термах также могут присутствовать свободные переменные.

Алгоритм 2 содержит псевдокод модифицированной процедуры унификации. Процедура принимает на вход множество универсально квантифицированных переменные U, множество экзистенциально квантифицированных переменных E, текущую подстановку для переменных σ и список ограничений $t \equiv u \in L$. Переменные $x \notin U \cup E$ считаются свободными. Для проверки решения ограничения $\forall\overline{x}\exists\overline{y}.t \equiv u$ на вход унификации подаются $U = \overline{x}$, $E = \overline{y}$, $\sigma = \varnothing$ и $L = \{t \equiv u\}$. Результатом работы процедуры является подстановка для переменных σ', если ограничение выполнимо, и $\perp$ иначе.

Как можно видеть, алгоритм 2 очень похож на стандартный алгоритм унификации. Рассмотрим главные отличия.

Если оба терма являются переменными (строки 12-21) рассматри-

Алгоритм 2 Расширенный алгоритм унификации

```
1: procedure UNIFY(U, E, σ, L)
2:     if L = ∅ then
3:         return σ
4:     pick (t ≡ u) ∈ L
5:     L := L \ (t ≡ u)
6:     switch σ(t), σ(u) do
7:         case C_i^n(t_1, ..., t_n), C_j^m(u_1, ..., u_m)
8:             if i = j then
9:                 L := {t_1 ≡ u_1, ... , t_n ≡ u_n} ∪ L
10:            else
11:                return ⊥
12:        case x, y when var(x) and var(y)
13:            if x ∈ U and y ∈ U then
14:                return ⊥
15:            else if x ∈ E then
16:                σ := {x ↦ y} ∪ σ
17:            else if y ∈ E then
18:                σ := {y ↦ x} ∪ σ   x ∈ U   or y ∈ U
19:                return ⊥
20:            else
21:                σ := {x ↦ y} ∪ σ
22:        case x, t = C^n(t_1, ..., t_n) when var(x)
23:            if x ∈ U or x ∈ Vars(t) then
24:                return ⊥
25:            else if x ∈ E then
26:                σ := {x ↦ t} ∪ σ
27:            else if Vars(t) ∩ U = ∅ then
28:                z_1, ..., z_n — свежие свободные переменные
29:                σ := {x ↦ C^n(z_1, ..., z_n)} ∪ σ
30:                L := {z_1 ≡ t_1, ..., z_n ≡ t_n} ∪ L
31:            else
32:                return ⊥
33:    return Unify(U, E, σ, L)
34:
```

вается несколько случаев. Если обе переменные универсально квантифицированны, алгоритм возвращает $\bot$. Если одна из переменных экзистенциально квантифицированна, в подстановку добавляется связывание для этой переменной. Иначе одна из переменных является свободной. Если вторая переменная при этом универсально квантифицирована, то нужно вернуть $\bot$. Если и вторая переменная свободна, в подстановку добавляется новое связывание для одной из переменных (неважно, x или y).

Если один из термов является переменной, а в корне другого терма конструктор, вновь рассматривается несколько случаев (строки 22-32). Если переменная универсально квантифицирована или она встречается в терме t (occurs check), алгоритм возвращает $\bot$. Если переменная экзистенциально квантифицирована, в подстановку добавляется связывание для неё. Если переменная свободна и терм не содержит универсально квантифицированных переменных, то алгоритм создает n новых свободных переменных $z_1, \ldots, z_n$ по числу аргументов конструктора C^n. В подстановке переменная связывается с термом $C^n(z_1, \ldots, z_n)$, а в множество ограничений L добавляются ограничения эквивалентности для новых переменных и соответствующих подтермов t.

Обратим внимание, что в получившейся подстановке σ' нет связываний для унивесально квантифицированных переменных, а все свободные переменные связаны с термами, не содержащими унивесально квантифицированные переменные. Это значит, что в результате работы алгоритма мы свели ограничение $\forall\overline{x}\exists\overline{y}.t \equiv u$ к набору ограничений $\exists\overline{y'}.x \equiv t$, которые выполнимы по построению.

2.2. Табличная мемоизация

Как уже было сказано в разделе 1.2.2, идея метода табличной мемоизации заключается в сохранении и переиспользовании ответов на запрос. Для этого, с каждым мемоизированным отношением связывается таблица, в которой ключом является запрос (т.е. набор переданных аргументов отношения), а значением — список ответов на данный за-

прос. Если в ходе исполнения программы вновь будет сделан запрос, эквивалентный некоторому более раннему, ответы на данный запрос будут извлечены из таблицы.

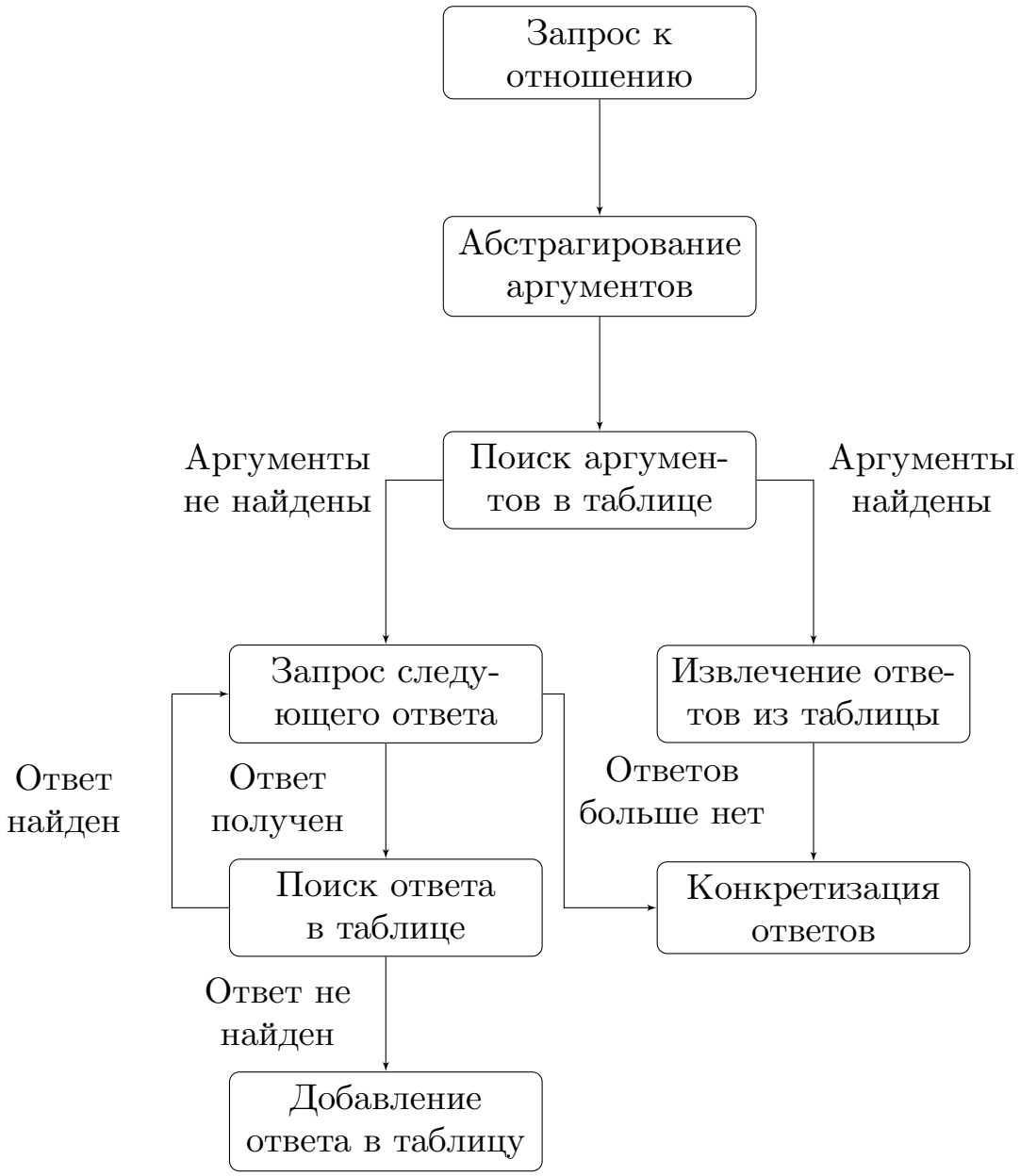

Рис. 8: Блок-схема алгоритма табличной мемоизации

Возникает вопрос, какие запросы считать эквивалентными? Мы полагаем, что два запроса эквиваленты, если наборы их аргументов *альфа-эквиваленты*, т.е. синтаксически равны с точностью до переименования переменных. Более формально: $t =_\alpha u \Leftrightarrow \exists\sigma.\sigma(t) = \sigma(u) \wedge Dom(\sigma) \subseteq V$.

Табличная мемоизация добавляет в OCanren два новых примитива — `tabled` и `tabledrec`, которые позволяют получить мемоизированную

версию отношения типобезопасным образом. Примитив `tabled` используется для мемоизации нерекурсивных отношений. При необходимости применить мемоизацию к рекурсивному отношению, пользователю необходимо немного модифицировать определение отношения. Требуется абстрагироваться от рекурсивного вызова, сделав его первым параметром отношения. После этого необходимо применить к полученному отношению примитив `tabledrec`. Можно видеть, что `tabledrec` является своего рода комбинатором неподвижной точки.

```
let edge^o x y =
  (x ≡ 'a') ∧ (y ≡ 'b')
  ∨
  (x ≡ 'b') ∧ (y ≡ 'a')
  ∨
  (x ≡ 'b') ∧ (y ≡ 'd')

let edge^o_tabled = tabled edge^o
```

Рис. 9: Пример мемоизации нерекурсивного отношения

```
let reachable^o_norec reachable^o_rec x y =
  (edge^o x y)
  ∨
  fresh (z)
    (edge^o x z) ∧ (rec^o z y)

let reachable^o_tabled = tabledrec reachable^o
```

Рис. 10: Пример мемоизации рекурсивного отношения

Схема алгоритма табличной мемоизации представлена на рис. 8.

При выполнении запроса к отношению сначала выполняется *абстракция* аргументов. Абстракция аргументов позволяет уменьшить размер таблицы для хранения аргументов и списка ответов — несколько запросов с различными аргументами потенциально могут быть абстрагированы до одного запроса. Сильная абстракция может привести

к множеству повторяющихся вычислений на этапе конкретизации (подробнее о конкретизации ниже). Кроме того, сильная абстракция может привести к тому, что запросы к рекурсивным отношениям не смогут завершиться за конечное время. Таким образом, при абстракции необходим разумный компромисс между экономией памяти (чем сильнее абстракция, тем меньше запросов храниться в таблице) и временем работы и терменируемостью запросов. В нашей реализации при абстракции аргументов отбрасывается информация об ограничениях неэквивалентности (по соображениям простоты реализации, следуя работе [26]).

Полученный после абстракции список аргументов используется как ключ для поиска записи об эквивалентном запросе в таблице. В качестве структуры данных для хранения множества используется хеш-таблица. Перед вычислением хеша логического терма выполняется переименование всех переменных в порядке обхода терма в глубину. Это необходимо для того, чтобы альфа-эквивалентые термы имели одинаковый хеш.

В случае, когда запись не была найдена в таблице, необходимо выполнить запрос к немемоизированной версии отношения, чтобы получить ответы. Для каждого полученного ответа выполняется проверка, не был ли данный ответ добавлен в таблицу ранее. Если такой ответ встречается впервые, он добавляется в таблицу.

В случае, когда запись была найдена, т.е. запрос к отношению с соответствующими аргументами уже был выполнен ранее, необходимо извлечь полученные ранее ответы из таблицы.

Перед возвращением списка ответов необходимо выполнить их конкретизацию. Напомним, что табличный алгоритм начинается с абстракции аргументов вызова. Абстракция может удалить некоторую информацию об аргументах, чтобы привести их к более общему виду. Другими словами, полученные ответы являются ответами не на исходный запрос, а на его абстрагированную версию. Чтобы получить ответ на исходный запрос необходимо "скомбинировать" информацию об аргументах, потерянную на стадии абстракции и полученные ответы. Заметим, что возможна ситуация, при которой некоторые ответы окажутся от-

брошены как не удовлетворяющие исходному запросу. Таким образом, хотя информации об ограничениях неэквивалентности была отброшена на стадии абстракции, на этапе конкретизации происходит проверка совместимости ограничений с полученными ответами.

3. Реляционный интерпретатор

В данном разделе описан реляционный интерпретатор для императивного языка программирования с многопоточностью. Для задания семантики императивного языка мы используем системы помеченных переходов. Сначала мы определим основные типы данных, необходимые для СПП. Затем мы покажем, как имея отношение перехода step^o построить отношение достижимости на СПП reachable^o. При помощи данного отношения будет определено отношение eval^o для вычисления программы, а также отношение invariant^o для проверки инвариантов программы.

Для того, чтобы поддержать несколько моделей памяти, мы будем представлять систему переходов как пару двух подсистем — подсистемы потоков и подсистемы памяти. Подсистема потоков отвечает за хранение локальных переменных потока и его команд, а также за планирование исполнения (т.е. выбор потока для исполнения следующей команды). Подсистема памяти описывает эффекты обращений к общей памяти, такие как чтения и записи разделямых переменных. Таким образом, изменяя реализации подсистемы памяти, можно получить интерпретаторы, работающие с различными моделями памяти.

3.1. Реляционные системы помеченных переходов

Напомним, что система помеченных переходов (СПП) — это четвёрка (S, I, L, R), где S — это множество состояний, $I \subseteq S$ — множество начальных состояний, L — множество меток, а $R \subseteq L \times S \times S$ — отношение перехода. СПП тривиальным образом кодируются в реляционную программу. Необходимо определить типы данных `State` и `Label`, а также отношение перехода $\texttt{step}^o$ (листинг 6).

Далее можно определить отношение достижимости $\texttt{reachable}^o$ (листинг 7), принимающее предикат p^o и связывающее состояние `s` с некоторым достижимым состоянием s', на котором выполняется p^o. Для того, чтобы избежать проблем, упомянутых в разделе 1.2.2, необходимо применить табличную мемоизацию (раздел 2.2).

```
type state

type label

val step^o :: label × state × state
```

Листинг 6: Основные компоненты реляционной СПП

```
let reachable^o p^o s s' =
  let reachable^o_norec reachable^o_rec s s'' =
    ((p^o s) ∧ (s ≡ s'')) ∨
    fresh (s')
      (step^o s s') ∧
      (reachable^o_rec s' s'')
  in
  let reachable^o_tabled =
    tabledrec_2 reachable^o_norec
  in
  reachable^o_tabled s s'
```

Листинг 7: Определение отношения `reachable`^{o} для СПП

Передав в качестве p^o отношение `terminal`^{o}, задающее множество терминальных состояний СПП, получим отношение `eval`^{o} (листинг 8), связывающее состояние `s` с достижимым терминальным состоянием `s'`.

Наконец, можно также определить отношение `invariant`^{o} (листинг 9), предназначенное для проверки инвариантов программ. Данное отношение принимает в качестве аргументов одноместное отношение `safe`^{o}, задающее множество безопасных состояний (т.е. тех состояний, в которых инвариант выполняется), а также начальное состояние `s`. Отношение `invariant`^{o} выполняется если все состояния, достижимые из данного, являются безопасными. Иначе отношение нарушается. Обратим внимание, что в определении `invariant`^{o} фигурирует применение оператора отрицания. Важно, что оператор использует метод конструктивного отрицания (раздел 1.2.1). Применение метода “отрицание как неудача” некорректно в случае, когда начальное состояние содержит свободные переменные.

```
let eval^o t t' = reachable^o terminal^o t t'
```

Листинг 8: Определение отношения `eval`^{o} для СПП

```
let invariant^o safe^o t = ¬ (
  fresh (t')
    (reachableo^o ¬safe^o t t')
)
```

Листинг 9: Определение отношения `invariant`^{o} для СПП

3.2. Описание языка

Мы реализовали минималистичный императивный язык программирования, способный моделировать поведение многопоточных программ. Грамматика языка представлена на рисунке 11. Язык состоит из выражений и набора стандартных для императивных языков операторов, в том числе оператора присваивания, условного оператора, цикла с предусловием и цикла с постусловием.

Отметим некоторые особенности языка. Язык различает множество локальных переменных Reg и множество разделяемых переменных Loc. Операция чтения разделяемой переменной относится к операторам, а не выражениям, так как чтение такой переменной может модифицировать внутренее состояние памяти. Также в языке присутствует операция атомарного сравнения с обменом (Compare-And-Swap, CAS). Данная операция атомарно выполняет чтение значения разделяемой переменной, сравнивает её с ожидаемым значением (второй аргумент CAS). Если значения совпали, то CAS заменяет значение переменной на желаемой (третий аргумент) и возвращает `true`, иначе значение не изменяется и команда возвращает `false`.

По аналогии с языком C/C++11 все операции с разделяемыми переменными аннотированы модификатором доступа (MO). Модификатор доступа определяет набор ограничений, накладываемых на возможность переупорядочивания инструкций в слабых моделях памяти. На значениях типа MO определено отношение порядка $mo_1 \sqsubseteq mo_2$, которое означает, что mo_2 является более строгим модификатором и наклады-

$$
\begin{array}{rcl}
Val & = & \mathbb{Z} \\
Reg & = & \{r_1, r_2, r_3, \dots\} \\
Loc & = & \{x, y, z, \dots\} \\
Uop & = & - \\
Bop & = & + \mid - \mid * \mid / \\
MO & = & rlx \mid rel \mid acq \mid relacq \mid sc \\
Expr & = & Val \mid Reg \\
 & \mid & Uop\ Expr \\
 & \mid & Expr\ Bop\ Expr \\
Stmt & = & Reg := Expr \\
 & \mid & Reg := [Loc]_{MO} \\
 & \mid & [Loc]_{MO} := Expr \\
 & \mid & Reg := CAS_{(mo_1, mo_2)}(Loc, Expr, Expr) \\
 & \mid & \textbf{skip} \\
 & \mid & Stmt;\ Stmt \\
 & \mid & \textbf{if}\ Expr\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt\ \textbf{fi} \\
 & \mid & \textbf{while}\ Expr\ \textbf{do}\ Stmt\ \textbf{od} \\
 & \mid & \textbf{repeat}\ Stmt\ \textbf{until}\ Expr
\end{array}
$$

Рис. 11: Синтаксис императивного языка

вает больше ограничений на переупорядочивание инструкций. Как и в C/C++11 мы различаем три уровня доступа для операций чтения $rlx \sqsubseteq acq \sqsubseteq sc$; три уровня доступа для операций записи $rlx \sqsubseteq rel \sqsubseteq sc$; и три уровня доступа для операций чтения-модификации-записи $rlx \sqsubseteq relacq \sqsubseteq sc$. Заметим, что операция CAS проаннатирована двумя модификаторами доступа, первый из которых применяется в случае неудачи, второй в случае успеха.

3.3. Подсистема потоков

Опишем подсистему потоков. Состояние подсистемы потоков состоит из списка потоков, каждый из которых содержит код программы данного потока, а также хранилище локальных переменных (рис. 12).

$$\begin{aligned} RegStorage &= Reg \to Val \\ Thread &= Stmt \times RegStorage \\ ThreadSystem &= Tid \to Thread \end{aligned}$$

Рис. 12: Описание подсистемы потоков

$$\frac{s(e) = v}{\langle r := e, s\rangle \xrightarrow{\epsilon} \langle \mathbf{skip}, s[r \mapsto v]\rangle}\ \text{Assign}$$

$$\frac{}{\langle r := [x]_{mo}, s\rangle \xrightarrow{R(x,mo,v)} \langle \mathbf{skip}, s[r \mapsto v]\rangle}\ \text{Load} \qquad \frac{s(e) = v}{\langle [x]_{mo} := e, s\rangle \xrightarrow{W(x,mo,v)} \langle \mathbf{skip}, s\rangle}\ \text{Store}$$

$$\frac{s(e_r) = v_r \qquad s(e_w) = v_w}{\langle r := \mathbf{CAS}_{mo_1,mo_2}(x, e_r, e_w), s\rangle \xrightarrow{RMW(x,mo_2,v_r,v_w)} \langle \mathbf{skip}, s[r \mapsto 1]\rangle}\ \text{CAS-Success}$$

$$\frac{s(e_r) = v_r \qquad v \neq v_r}{\langle r := \mathbf{CAS}_{mo_1,mo_2}(x, e_r, e_w), s\rangle \xrightarrow{R(x,mo_1,v)} \langle \mathbf{skip}, s[r \mapsto 0]\rangle}\ \text{CAS-Fail}$$

$$\frac{}{\langle \mathbf{skip}; p, s\rangle \xrightarrow{\epsilon} \langle p, s\rangle}\ \text{Skip} \qquad \frac{\langle p_1, s\rangle \xrightarrow{\epsilon} \langle p_1', s'\rangle}{\langle p_1; p_2, s\rangle \xrightarrow{\epsilon} \langle p_1'; p2, s'\rangle}\ \text{Seq}$$

$$\frac{s(e) \neq 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, s\rangle \xrightarrow{\epsilon} \langle p_1, s\rangle}\ \text{If-True} \qquad \frac{s(e) = 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2, s\rangle \xrightarrow{\epsilon} \langle p_2, s\rangle}\ \text{If-False}$$

$$\frac{}{\langle \mathbf{while}\ e\ \mathbf{do}\ p\ \mathbf{od}, s\rangle \xrightarrow{\epsilon} \langle \mathbf{if}\ e\ \mathbf{then}\ (p; \mathbf{while}\ e\ \mathbf{do}\ p\ \mathbf{od})\ \mathbf{else}\ \mathbf{skip}, s\rangle}\ \text{While}$$

$$\frac{}{\langle \mathbf{repeat}\ p\ \mathbf{until}\ e, s\rangle \xrightarrow{\epsilon} \langle p; \mathbf{while}\ !e\ \mathbf{do}\ p\ \mathbf{od}, s\rangle}\ \text{Repeat}$$

Рис. 13: Отношение перехода для потока

Отношение перехода для одного потока связывает два последовательных состояния потока и метку действия. Допускаются следующие действия: $R(x, mo, v)$ — чтение из разделяемой переменной x значения

$$\frac{\langle p, s\rangle \xrightarrow{a} \langle p', s'\rangle}{t[i \mapsto \langle p, s\rangle] \xrightarrow{i:a} t[i \mapsto \langle p', s'\rangle]}$$

Рис. 14: Отношение перехода для подсистемы потоков

v с модификатором доступа mo; $W(x, mo, v)$ — запись в разделяемую переменную x значения v с модификатором доступа mo; $RMW(x, mo, v_r, v_w)$ (Read-Modify-Write) — операция чтения-модификации-записи в разделяемую переменную x значения v_w с ожидаемым значением v_r и модификатором доступа mo. Отношение перехода для одного потока определяется набором правил, представленных на рис. 13. Отношение перехода для подсистемы потоков недетерменированно выбирает пару из идентификатора и потока и выполняет переход для выбранного потока (рис.14).

3.4. Подсистема памяти

Подсистема памяти определяет эффекты обращений к разделяемой памяти. Каждой модели памяти соответвует своя подсистема. В данном разделе мы опишем реализованные нами подсистемы для трёх моделей памяти: SC, TSO и SRA.

3.4.1. Модель памяти SC

Модель памяти SC [17] является наиболее простой. Состояние данной системы это просто функция, ставящая в соответствие переменным их значения (рис. 15).

$$Memory_{SC} \;=\; Loc \to Val$$

Рис. 15: Описание подсистемы памяти SC

Отношение перехода представлено на рис. 16. Чтение возвращает значение переменной в текущем состоянии, а запись обновляет значение переменной. Для атомарного чтения-модификации также выполня-

ется проверка, что ожидаемое значение равно значению переменной в текущем состоянии.

$$\frac{m(x) = v}{m \xrightarrow{i:R(x,sc,v)} m}\ \text{Load} \qquad \frac{}{m \xrightarrow{i:W(x,sc,v)} m[x \mapsto v]}\ \text{Store}$$

$$\frac{m(x) = v_r}{m \xrightarrow{i:RMW(x,sc,v_r,v_w)} m[x \mapsto v_w]}\ \text{Read-Modify-Write}$$

Рис. 16: Отношение перехода для подсистемы памяти SC

3.4.2. Модель памяти TSO

Модель памяти TSO [34] разрешает переупорядочивать чтения с более ранними записями. Для того, чтобы моделировать данный эффект, как правило, состояние системы представляют как пару из хранилища переменных и списка локальных для потоков буфферов записи (рис. 17). Буфер записи поддерживает интерфейс FIFO очереди: $enq(b, x, v)$ добавляет новую пару (x, v) в конец очереди, $deq(b)$ возвращает первую в очереди пару и остаток очереди, $get(b, x)$ возвращает значение наиболее ранней в очереди записи переменной x или $\perp$ если в очереди нет записей для переменной x. Пустая очередь обозначается как ϵ.

$$\begin{aligned} StoreBuffer &= (Loc \times Val)^* \\ Storage &= Loc \to Val \\ Memory_{TSO} &= Storage \times (Tid \to StoreBuffer) \end{aligned}$$

Рис. 17: Описание подсистемы памяти TSO

Рассмотрим отношение перехода для подсистемы памяти TSO (рис. 18). Если в буфере записи текущего потока ожидают записи переменной x, чтение возвращает значение наиболее ранней буфферезированной записи. Иначе чтение происходит напрямую из основной памяти. Записи попадают в основную память через буфер. В любой момент подсистема может протолкнуть наиболее раннюю запись из буфера какого-либо

потока в основную память. Операции чтения и записи с модификатором доступа *sc* опустошают буфер, проталкивая все его записи в основную память. Операции чтения-модификации-записи выполняются только если буфер текущего потока пуст.

$$\frac{get(\beta(i), x) = \bot \qquad m(x) = v \qquad mo \sqsubseteq acq}{\langle m, \beta\rangle \xrightarrow{i:R(x,mo,v)} \langle m, \beta\rangle} \text{Read-Memory}$$

$$\frac{get(\beta(i), x) = v \qquad mo \sqsubseteq acq}{\langle m, \beta\rangle \xrightarrow{i:R(x,mo,v)} \langle m, \beta\rangle} \text{Read-Buffered}$$

$$\frac{mo \sqsubseteq rel}{\langle m, \beta\rangle \xrightarrow{i:W(x,mo,v)} \langle m, \beta[i \mapsto enq(\beta(i), x, v)]\rangle} \text{Write}$$

$$\frac{(v, b) = deq(\beta(i), x)}{\langle m, \beta\rangle \xrightarrow{i:\epsilon} \langle m[x \mapsto v], \beta[i \mapsto b]\rangle} \text{Propagate}$$

$$\frac{\beta(i) = \{(x_i, v_i)\}_{i=1}^{n} \qquad m' = m[x_1 \mapsto v_1] \dots [x_n \mapsto v_n] \qquad m'(x) = v}{\langle m, \beta\rangle \xrightarrow{i:R(x,sc,v)} \langle m', \beta[i \mapsto \epsilon]\rangle} \text{Read-SC}$$

$$\frac{\beta(i) = \{(x_i, v_i)\}_{i=1}^{n} \qquad m' = m[x_1 \mapsto v_1] \dots [x_n \mapsto v_n][x \mapsto v]}{\langle m, \beta\rangle \xrightarrow{i:W(x,sc,v)} \langle m', \beta[i \mapsto \epsilon]\rangle} \text{Write-SC}$$

$$\frac{m(x) = v_r \qquad \beta(i) = \epsilon \qquad mo \sqsubseteq sc}{\langle m, \beta\rangle \xrightarrow{i:RMW(x,mo,v_r,v_w)} \langle m[x \mapsto v_w], \beta\rangle} \text{Read-Modify-Write}$$

Рис. 18: Отношение перехода для подсистемы памяти TSO

3.4.3. Модель памяти SRA

Модель памяти SRA [16] является наиболее слабой и наиболее сложной из реализованных в данной работе. В данной модели каждый поток имеет *фронт видимости*: подмножество всех записей, доступных для чтения. Фронт видимости потока может монотонно увеличиваться по ходу исполнения программы. Также данная модель позволяет двум потокам синхронизировать свои фронты с помощью следующего шаблона: один из потоков должен выполнить *высвобождающую запись* (release-write), а другой поток — *захватывающее чтение* (acquire-read).

$$
\begin{aligned}
Timestamp &= \mathbb{N} \\
ViewFront &= Loc \to Timestamp \\
ThreadFront &= ViewFront_{cur} \times ViewFront_{acq} \times ViewFront_{rel} \\
History &= (Loc \times Timestamp \times Value \times ViewFront)^* \\
Memory_{SRA} &= History \times (Tid \to ThreadFront) \times ViewFront_{sc}
\end{aligned}
$$

Рис. 19: Описание подсистемы памяти SRA

Для определения подсистемы памяти SRA мы воспользовались идеями работ [16, 24, 31].

Состояние модели памяти SRA состоит из трёх компонент: набора локальных для потока фронтов видимости, истории и глобального фронта последвательной консистентности.

История состоит из множества сообщений, т.е. четвёрок вида (x, t, v, σ), которые далее будут обозначаться как $x@t = (v, \sigma)$. Сообщение соответствует некоторой записи в разделяемую переменную x значения v. На множестве сообщений задано отношение порядка, которое полностью упорядочивает все записи в одну разделяемую переменную. Данное отношение поддерживается при помощи присваивания каждому сообщению метки времени t. Кроме того, каждое сообщение несёт с собой фронт видимости, т.е. функцию из локации в метку времени (подробности далее). Таким образом, история хранит все записи, когда-либо сделанные программой.

Каждый поток имеет три фронта видимости. *Базовый фронт* (cur) определяет, какие сообщения видны потоку. Поток не может прочитать сообщение с меткой времени меньше, чем метка в его базовом фронте. *Фронт захвата* (acq) и *фронт высвобождения* (rel), вместе с фронтом, хранящемся в каждом сообщении, необходимы для реализации операций захватывающего чтения (acquire-read) и высвобождающей записи (release-write). Глобальный фронт последовательной консистентности (sc) необходим для реализации последовательно консистентных операций чтения и записи.

Отношение перехода для подсистемы SRA достаточно сложно [1] (рис 20).

[1]Заметим некоторые особенности определения отношения перехода для SRA. Правила чтения/за-

$$\frac{\begin{array}{ccc} T = \psi(i) & & T'.rel = T.rel \\ \sigma_{seen} = T.cur & t_{seen} \leq t_r & T'.acq = T.acq \sqcup \sigma \\ t_{seen} = \sigma_{seen}(x) & \langle x@t_r = (v, \sigma)\rangle \in h & T'.cur = \sigma_{seen}[x \mapsto t_r] \end{array}}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:R(x,rlx,v)} \langle h, \psi[i \mapsto T'], \sigma_{sc}\rangle}\ \text{Read-Relaxed}$$

$$\frac{\begin{array}{ccc} T = \psi(i) & & T'.rel = \sigma \\ \sigma = T.rel[x \mapsto t_w] & & T'.acq = T.acq \\ t_w = max_{ts}(h, x) + 1 & h' = \langle x@t_w = (v, \sigma)\rangle \cup h & T'.cur = T.cur[x \mapsto t_w] \end{array}}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:W(x,rlx,v)} \langle h', \psi[i \mapsto T'], \sigma_{sc}\rangle}\ \text{Write-Relaxed}$$

$$\frac{\sigma_{seen} = T.cur \sqcup T.acq \qquad \ldots}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:R(x,acq,v)} \langle h, \psi[i \mapsto T'], \sigma_{sc}\rangle}\ \text{Read-Acquire}$$

$$\frac{\sigma = T.cur \sqcup T.rel[x \mapsto t_w] \qquad \ldots}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:W(x,rel,v)} \langle h', \psi[i \mapsto T'], \sigma_{sc}\rangle}\ \text{Write-Release}$$

$$\frac{\sigma_{sc}(x) \leq t_r \qquad \ldots}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:W(x,sc,v)} \langle h', \psi[i \mapsto T'], \sigma_{sc}\rangle}\ \text{Read-SC}$$

$$\frac{\ldots}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:W(x,sc,v)} \langle h', \psi[i \mapsto T'], \sigma_{sc}[x \mapsto t_w]\rangle}\ \text{Write-SC}$$

$$\frac{t_r = max_{ts}(h, x) \qquad \ldots}{\langle h, \psi, \sigma_{sc}\rangle \xrightarrow{i:RMW(x,mo,v_r,v_w)} \langle h', \psi[i \mapsto T'], \sigma'_{sc}\rangle}\ \text{Read-Modify-Write}$$

Рис. 20: Отношение перехода для подсистемы памяти SRA

Мы не будем подробно описывать все входящие в него правила. Вместо этого продемонстрируем пример исполнения программы в модели памяти SRA и проследим как в ходе этого исполнения изменяется состояние подсистемы памяти.

Рассотрим вариант программы MP (листинг 21). В данной программе не используются последовательно консистентные операции, поэтому

писи определяются последовательно, от более ослабленных к более строгим, при этом более строгое правило использует посылки более слабого (данный факт обозначается с помощью многоточия), но накладывает некоторые новые ограничения. Правило для чтения-модификации-записи включает все посылки для правил чтения/записи с соответствующими модификаторами доступа, а также дополнительное ограничение на метку времени прочитанного сообщения.

$$\begin{array}{l} [\mathtt{x}]_{rlx} := 42; \\ [\mathtt{f}]_{rel} := 1; \end{array} \Bigg\| \begin{array}{l} \mathtt{r}_1 := [\mathtt{f}]_{acq}; \\ \mathtt{r}_2 := [\mathtt{x}]_{rlx}; \end{array}$$

Рис. 21: Пример программы Message Passing (MP)

нам не понадобитя рассматривать фронт последовательной консистентности. Изначально история содержит инициализирующие сообщения $H = \{\langle x@0 = (0, \bot)\rangle, \langle y@0 = (0, \bot)\rangle\}$, и обоим потокам видны только эти сообщения $\sigma_{init} = [x@0, y@0]$, $T_1 = T_2 = (\sigma_{init}, \sigma_{init}, \sigma_{init})$. Предположим, что исполнение программы будет происходить по следующему сценарию: сначала исполняются команды первого потока, а затем команды второго потока. После выполнения записи $[\mathtt{x}]_{rlx}$ `:= 42` в историю будет добавлено сообщение $\langle x@1 = (42, \sigma_{init})\rangle$. Также обновятся базовый фронт и фронт захвата первого потока: $T_1.cur = T_1.acq = [x@1, y@0]$. После выполнения высвобождающей записи $[\mathtt{f}]_{rel}$ `:= 1` обновится фронт высвобождения потока $T_1.rel = T_1.cur = [x@1, y@1]$, данный фронт будет также записан в новое сообщение $\langle f@1 = (1, T_1.rel)\rangle$. Затем начнут исполняться команды второго потока. Чтение $\mathtt{r}_1$ `:=` $[\mathtt{f}]_{acq}$ может прочитать либо первое, либо второе сообщение для локации f (так как $T_2.cur(f) = 0$). Предположим, что будет прочитано второе сообщение $\langle f@1 = (1, \sigma)\rangle$. В этом случае будет выполнено слияние фронта захвата потока с фронтом из прочитанного сообщения $T_2.acq = [x@1, y@1]$. Так как чтение является захватывающим, также обновится базовый фронт $T_2.cur = T_2.acq = [x@1, y@1]$. Следующая операция чтения $\mathtt{r}_2$ `:=` $[\mathtt{x}]_{rlx}$ будет обязана прочитать второе сообщение для локации x (т.к. $T_2.cur(x) = 1$). Данное сообщение содержит значение 42.

4. Апробация

При апробации реляционного интерпретатора была поставлена задача убедиться в том, что интерпретатор корректно работает в различных моделях памяти, а также измерить время его работы при исполнении запросов верификации и синтеза.

В первую очередь, чтобы проверить, что интерпретатор корректно работает со всеми поддержанными моделями памяти, интерпретатор был протестирован на наборе “лакмусовых” тестов. Лакмусовыми тестами называются небольшие многопоточные программы (обычно состоящие из 2-4 потоков с несколькими инструкциями в каждом). Как правило, каждый лакмусовый тест проверят наличие или отсутствие в модели памяти определенного слабого поведения.

Для каждого лакмусового теста и каждой модели памяти была дана спецификация на возможные конечные состояния интерпретатора. Например, для теста Store Buffering в случае модели памяти SC проверялось, что не существует конечного состояния интерпретатора в котором `r1 = 0` $\wedge$ `r2 = 0`, а в случае моделей TSO и SRA наоборот, что такое конечное состояние может быть достигнуто. Задачей реляционного интерпретатора была проверка данной спецификации.

Листинги лакмусовых тестов вместе с проверяемой спецификацией приведены в таблице 1.

Реляционный интерпретатор также был применен для верификации и синтеза кода синхронизции нескольких классических многопоточных алгоритмов. В задаче верификации на вход интерпретатору подавался инвариант и многопоточная программа. Задачей интерпретатора было проверить корректность программы. В задаче синтеза на вход интерпретатору подавался инвариант и шаблон программы, в котором были опущены модификаторы доступа, на их место были подставлены свободные переменные. Задачей интерпретатора было вывести подстановку для переменных, стоящих на месте модификаторов доступа, гарантирующую корректность программы. Задача синтеза модификаторов доступа в модели памяти SC не актуальна, так как в данной модели

Название теста	Спецификация поведения		
	SC	TSO	SRA
Store Buffering			
$[\mathtt{x}] := 1; \mid\mid [\mathtt{y}] := 1;$ $\mathtt{r}_1 := [\mathtt{y}]; \mid\mid \mathtt{r}_2 := [\mathtt{x}];$	$\forall. r_1 \neq 0 \vee r_2 \neq 0$	$\exists. r_1 = 0 \wedge r_2 = 0$	
$[\mathtt{x}]_{sc} := 1; \mid\mid [\mathtt{y}]_{sc} := 1;$ $\mathtt{r}_1 := [\mathtt{y}]_{sc}; \mid\mid \mathtt{r}_2 := [\mathtt{x}]_{sc};$	$\forall. r_1 \neq 0 \vee r_2 \neq 0$		
Load Buffering			
$\mathtt{r}_1 := [\mathtt{x}]; \mid\mid \mathtt{r}_2 := [\mathtt{y}];$ $[\mathtt{y}] := 1; \mid\mid [\mathtt{x}] := 1;$	$\forall. r_1 \neq 1 \vee r_2 \neq 1$		
Message Passing			
$[\mathtt{x}] := 1; \mid\mid \mathtt{r}_1 := [\mathtt{f}];$ $[\mathtt{f}] := 1; \mid\mid \mathtt{r}_2 := [\mathtt{x}];$	$\forall. \neg(r_1 = 1 \wedge r_2 = 0)$		$\exists. r_1 = 1 \wedge r_2 = 0$
$[\mathtt{x}] := 1; \mid\mid \mathtt{r}_1 := [\mathtt{f}]_{acq};$ $[\mathtt{f}]_{rel} := 1; \mid\mid \mathtt{r}_2 := [\mathtt{x}];$	$\forall. \neg(r_1 = 1 \wedge r_2 = 0)$		
Coherence of Read-Read			
$[\mathtt{x}] := 1; \mid\mid [\mathtt{x}] := 2;$ $\mathtt{r}_1 := [\mathtt{x}]; \mid\mid \mathtt{r}_3 := [\mathtt{x}];$ $\mathtt{r}_2 := [\mathtt{x}]; \mid\mid \mathtt{r}_4 := [\mathtt{x}];$	$\forall. \neg((r_1 = 1 \wedge r_2 = 2 \wedge r_3 = 2 \wedge r_4 = 1) \vee (r_1 = 2 \wedge r_2 = 1 \wedge r_3 = 1 \wedge r_4 = 2))$		
Independent Reads of Independent Writes			
$[\mathtt{x}] := 1; \mid\mid [\mathtt{y}] := 1;$ $\mathtt{r}_1 := [\mathtt{x}]; \mid\mid \mathtt{r}_3 := [\mathtt{y}];$ $\mathtt{r}_2 := [\mathtt{y}]; \mid\mid \mathtt{r}_4 := [\mathtt{x}];$	$\forall. \neg(r_1 = 1 \wedge r_2 = 0 \wedge r_3 = 1 \wedge r_4 = 0)$		$\exists. r_1 = 1 \wedge r_2 = 0 \wedge r_3 = 1 \wedge r_4 = 0$
Write-to-Read Causality			
$[\mathtt{x}] := 1; \mid\mid \mathtt{r}_1 := [\mathtt{x}]; \mid\mid \mathtt{r}_2 := [\mathtt{y}];$ $[\mathtt{y}] := \mathtt{r}_1; \mid\mid \mathtt{r}_3 := [\mathtt{x}];$	$\forall. \neg(r_2 = 1 \wedge r_3 = 0)$		$\exists. r_2 = 1 \wedge r_3 = 0$

Таблица 1: Лакмусовые тесты

все операции над разделяемыми переменными полностью упорядочены.

В качестве алгоритмов для проверки были выбраны: программа передачи сообщений между двумя потоками, алгоритм взаимного исключения Деккера, алгоритм взаимного исключения Петерсона, алгоритм

Название теста	SC	TSO	SRA
Message Passing	0.01±0.01	0.02±0.00	0.02±0.01
Barrier	0.08±0.01	0.14±0.01	0.12±0.00
Cohen Lock	0.11±0.00	0.35±0.00	0.21±0.01
Dekker Lock	0.11±0.01	1.08±0.01	0.21±0.01
Peterson Lock	0.26±0.01	2.91±0.01	0.48±0.01

Таблица 2: Результаты применения интерпретатора для верификации

взаимного исключения Коэна, алгоритм барьера. Для каждого алгоритма был также предоставлен проверяемый инвариант. Листинги алгоритмов вместе с проверяемым инвариантом приведены в приложении А. Результаты замеров приведены для задач верификации приведены в таблице 2, а для задач синтеза в таблице 3. В таблицах указано среднее время работы в секундах [2] с 95% доверительным интервалом (по результатам 10 запусков). В таблице с результатами запуска интерпретатора для синтеза в некоторых ячейках присутствует значение `OOM` (out of memory). Это означает, что данный тест не завершился по причине исчерпания виртуальной памяти.

По результатам замеров можно видеть, что время работы интерпретатора в слабых моделях выше, чем в модели SC. Это связано с тем, что в моделях TSO и SRA пространство состояний программ, как правило, существенно больше, чем в модели SC. В задачах генерации синхронизации время работы интерпретатор стремительно увеличивается вместе с увеличением сложности программы. Это связано с тем, что каждый раз, когда реляционный интерпретатор встречает свободную переменную в исходной программе, он делает недетерминированный

[2]Конфигурация машины, на которой выполнялись измерения: Intel® Core™ i5-6300HQ CPU 2.30GHz × 4; 8GB RAM; Ubuntu 17.10 x64.

Название теста	TSO	SRA
Message Passing	0.05±0.00	0.36±0.01
Barrier	2.39±0.04	238.67±1.79
Cohen Lock	1.00±0.05	2.48±0.07
Dekker Lock	76.49±0.51	OOM
Peterson Lock	5300.51±10.09	OOM

Таблица 3: Результаты применения интерпретатора для синтеза кода синхронизации

выбор. Таким образом, пространство поиска может расти экспоненциально с ростом количества свободных переменных в программе. Так как каждая операция с разделяемой переменной проаннатирована модификатором доступа, количество свободных переменных в программе равно количеству операций с разделяемыми переменными.

Таким образом можно видеть, что интерпретатор работает за приемлимое время в задачах верификации, но в задачах синтеза время работы быстро растет вместе с увелечением программы. Это означает, что интерпретатор применим для проверки и синтеза небольших модельных программ и алгоритмов, но для применения его к более сложным программам требуется усовершенствование подхода.

Заключение

В ходе данной работы были достигнуты следующие результаты.

- Были реализованы расширения для языка OCanren, предназначенные построении реляционных интерпретаторов, а именно мемоизация и конструктивное отрицание.
- Разработан реляционный интерпретатор для многопоточных программ на языке OCanren.
- Была произведена апробация интерпретатора на ряде классических многопоточных алгоритмов.

Исходный код интерпретатора находится в свободном доступе и может быть найден по адресу `https://github.com/eucpp/relcppmem` (дата обращения 10.05.2018). Версия языка OCanren, дополненная разработанными в рамках данной работы расширениями, доступна по адресу `https://github.com/eucpp/OCanren` (дата обращения 10.05.2018).

Целью данной работы являлось применение идей реляционного программирования к задаче верификации и синтеза многопоточных программ. Наша гипотеза состояла в том, что полезные свойства реляционного программирования, такие как декларативность и гибкость помогут в разработке инструментов для решения данной задачи. В частности, ожидалось, что возможность "запуска" реляционного интерпретатора в "обратную сторону" (т.е. передав на вход отношению частичную программу с логическими переменными и ожидаемый результат) позволит получить инструмент для синтеза программ. На практике такой подход приводит к необходимости поиска в большом пространства состояний, который не может быть выполнен с помощью полного обхода.

Таким образом, для масштабирования реляционного интерпретатора необходимо применять более сложные подходы и методы. Одним из возможных улучшений может быть встраивание в реляционный язык техник символьного исполнения [14, 32] и абстрактной интерпретации [7, 8] для более компактного представления пространства

состояний. Другое направлением — интеграция языка OCanren с современными решателями формул в теориях (SMT solvers) для улучшения производительности. В частности, может представлять интерес интеграция с решателем Z3, поддерживающим решение систем хорновских дизъюнктов [3,12]. Наконец, ещё одно перспективное направление — это применение техник суперкомпиляции и частичного исполнения [13, 23].

A. Многопоточные Алгоритмы

A.1. Передача сообщения

Инвариант (для терминальных состояний): $r_2 = 1$

```
                ||  repeat
[x] := 1;       ||    r1 := [f]
[f] := 1        ||  until (r1);
                ||  r2 := [x]
```

A.2. Алгоритм Коэна

Инвариант (для терминальных состояний): $d = 1$

```
r1 := choice(1, 2);                ||  r1 := choice(1, 2);
[x] := r1;                         ||  [y] := r1;
repeat                             ||  repeat
  r2 := y_acq                      ||    r2 := [x]
until (r2);                        ||  until (r2);
if (r1 = r2) then                  ||  if (r1 != r2) then
  // start of critical section     ||    // start of critical section
  r3 := [v];                       ||    r3 := [v];
  [v] := r3 + 1;                   ||    [v] := r3 + 1;
  // end of critical section       ||    // end of critical section
fi                                 ||  fi
```

A.3. Барьер

Инвариант (для терминальных состояний): $r_3 = 1 \wedge r_6 = 1$

```
                [cnt] := 2;
```

```
[x] := 1;                          | [y] := 1;
// barrier start                   | // barrier start
repeat                             | repeat
  r1 := [cnt];                     |   r4 := [cnt];
  r2 := CAS(cnt, r1, r1 − 1)       |   r5 := CAS(cnt, r1, r1 − 1)
until (r2);                        | until (r5);
if (r1 = 1) then                   | if (r4 = 1) then
  [g] := 1                         |   [g] := 1
else                               | else
  repeat                           |   repeat
    r1 := [g]                      |     r4 := [g]
  until (r1);                      |   until (r4);
fi;                                | fi;
// barrier end                     | // barrier end
r3 := [y]                          | r6 := [x]
```

A.4. Алгоритм Деккера

Инвариант (для терминальных состояний): $v = 2$

```
[x] := 1;
r1 := [y];
while (r1) do
  r2 := [turn];
  if (r2 != 0) then
    [x] := 0;
    repeat
      r2 := [turn]
    until (r2 = 0);
    [x] := 1
  fi;
  r1 := [y]
od;
// start of critical section
r3 := [v];
[v] := r3 + 1;
// end of critical section
[turn] := 1;
[x] := 0
```

```
[y] := 1;
r1 := [x];
while (r1) do
  r2 := [turn];
  if (r2 != 1) then
    [y] := 0;
    repeat
      r2 := [turn]
    until (r2 = 1);
    [y] := 1
  fi;
  r1 := [x]
od;
// start of critical section
r3 := [v];
[v] := r3 + 1;
// end of critical section
[turn] := 0;
[y] := 0
```

A.5. Алгоритм Петерсона

Инвариант (для терминальных состояний): $v = 2$

```
[x] := 1;
[turn] := 1;
repeat
  r1 := [y];
  r2 := [turn];
until (
  (r1 != 1) || (r2 != 1)
);
// start of critical section
r3 := [v];
[v] := r3 + 1;
// end of critical section
[x] := 0
```

```
[y] := 1;
[turn] := 0;
repeat
  r1 := [x];
  r2 := [turn];
until (
  (r1 != 1) || (r2 != 0)
);
// start of critical section
r3 := [v];
[v] := r3 + 1;
// end of critical section
[y] := 0
```

Список литературы

[1] Baader Franz, Snyder Wayne. Unification theory. // Handbook of automated reasoning. — 2001. — Vol. 1. — P. 445–532.

[2] Backtracking, interleaving, and terminating monad transformers:(functional pearl) / Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, Amr Sabry // ACM SIGPLAN Notices. — 2005. — Vol. 40, no. 9. — P. 192–203.

[3] Bjørner Nikolaj, McMillan Ken, Rybalchenko Andrey. On solving universally quantified horn clauses // International Static Analysis Symposium / Springer. — 2013. — P. 105–125.

[4] Byrd William E. Relational programming in minikanren: Techniques, applications, and implementations : Ph. D. thesis / William E Byrd ; Citeseer. — 2009.

[5] Chan David. Constructive negation based on the completed database // Proc. of ICLP-88. — 1988.

[6] Clark Keith L. Negation as failure // Logic and data bases. — Springer, 1978. — P. 293–322.

[7] Counterexample-guided abstraction refinement / Edmund Clarke, Orna Grumberg, Somesh Jha et al. // International Conference on Computer Aided Verification / Springer. — 2000. — P. 154–169.

[8] Cousot Patrick, Cousot Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages / ACM. — 1977. — P. 238–252.

[9] Dijkstra Edsger W. Cooperating sequential processes // The origin of concurrent programming. — Springer, 1968. — P. 65–138.

[10] Friedman Daniel P, Byrd William E, Kiselyov Oleg. The reasoned schemer. — MIT Press, 2005.

[11] Friedman Jason Hemann Daniel P. μKanren: A Minimal Functional Core for Relational Programming.

[12] Hoder Kryštof, Bjørner Nikolaj. Generalized property directed reachability // International Conference on Theory and Applications of Satisfiability Testing / Springer. — 2012. — P. 157–171.

[13] Jones Neil D, Gomard Carsten K, Sestoft Peter. Partial evaluation and automatic program generation. — Peter Sestoft, 1993.

[14] King James C. Symbolic execution and program testing // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385–394.

[15] Kosarev Dmitry, Boulytchev Dmitri. Typed Embedding of a Relational Language in OCaml // International Workshop on ML. — 2016.

[16] Lahav Ori, Giannarakis Nick, Vafeiadis Viktor. Taming release-acquire consistency // ACM SIGPLAN Notices / ACM. — Vol. 51. — 2016. — P. 649–662.

[17] Lamport Leslie. How to make a multiprocessor computer that correctly executes multiprocess programm // IEEE transactions on computers. — 1979. — no. 9. — P. 690–691.

[18] Liu Julie Yuchih, Adams Leroy, Chen Weidong. Constructive negation under the well-founded semantics // The Journal of Logic Programming. — 1999. — Vol. 38, no. 3. — P. 295–330.

[19] Manson Jeremy, Pugh William, Adve Sarita V. The Java memory model. — ACM, 2005. — Vol. 40.

[20] Mathematizing C++ concurrency / Mark Batty, Scott Owens, Susmit Sarkar et al. // ACM SIGPLAN Notices / ACM. — Vol. 46. — 2011. — P. 55–66.

[21] Modelling the ARMv8 architecture, operationally: concurrency and ISA / Shaked Flur, Kathryn E Gray, Christopher Pulte et al. // ACM SIGPLAN Notices / ACM. — Vol. 51. — 2016. — P. 608–621.

[22] Nadathur Gopalan, Miller Dale. An overview of Lambda-PROLOG. — 1988.

[23] Pettorossi Alberto, Proietti Maurizio. Synthesis and transformation of logic programs using unfold/fold proofs // The Journal of Logic Programming. — 1999. — Vol. 41, no. 2-3. — P. 197–230.

[24] Podkopaev Anton, Sergey Ilya, Nanevski Aleksandar. Operational aspects of C/C++ concurrency // arXiv preprint arXiv:1606.01400. — 2016.

[25] Przymusinski Teodor C. On constructive negation in logic programming. — MIT Press Cambridge, Massachusetts, 1989.

[26] Schrijvers Tom, Demoen Bart, Warren David S. TCHR: a Framework for Tabled CLP // Theory and Practice of Logic Programming. — 2008. — Vol. 8, no. 4. — P. 491–526.

[27] Stuckey Peter J. Constructive negation for constraint logic programming // Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on / IEEE. — 1991. — P. 328–339.

[28] Swi-prolog / Jan Wielemaker, Tom Schrijvers, Markus Triska, Torbjörn Lager // Theory and Practice of Logic Programming. — 2012. — Vol. 12, no. 1-2. — P. 67–96.

[29] Swift Terrance, Warren David S. XSB: Extending Prolog with tabled logic programming // Theory and Practice of Logic Programming. — 2012. — Vol. 12, no. 1-2. — P. 157–187.

[30] Understanding POWER multiprocessors / Susmit Sarkar, Peter Sewell, Jade Alglave et al. // ACM SIGPLAN Notices. — 2011. — Vol. 46, no. 6. — P. 175–186.

[31] A promising semantics for relaxed-memory concurrency / Jeehoon Kang, Chung-Kil Hur, Ori Lahav et al. // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages / ACM. — 2017. — P. 175–189.

[32] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // arXiv preprint arXiv:1610.00502. — 2016.

[33] A unified approach to solving seven programming problems (functional pearl) / William E Byrd, Michael Ballantyne, Gregory Rosenblatt, Matthew Might // Proceedings of the ACM on Programming Languages. — 2017. — Vol. 1, no. ICFP. — P. 8.

[34] x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors / Peter Sewell, Susmit Sarkar, Scott Owens et al. // Communications of the ACM. — 2010. — Vol. 53, no. 7. — P. 89–97.