DOI: 10.18721/JCSTCS.11203
УДК 004.434

РЕЛЯЦИОННОЕ ПРОГРАММИРОВАНИЕ С МЕМОИЗАЦИЕЙ И ОТРИЦАНИЕМ

Е.А. Моисеенко, А.В. Подкопаев

Санкт-Петербургский государственный университет,
Санкт-Петербург, Российская Федерация

Реляционная парадигма позволяет описывать программы как набор отношений. Отношения не делают различий между входными и выходными параметрами. Благодаря этому одна и та же реляционная программа может использоваться для решения нескольких связанных проблем. В статье представлена реализация двух расширений реляционного программирования, полезных при разработке интерпретаторов: табличной мемоизации и конструктивного отрицания. Мемоизация позволяет эффективно обходить пространство состояний интерпретатора, а отрицание – проверять, что заданное состояние интерпретатора недостижимо. Полезность данных расширений продемонстрирована на примере интерпретатора для многопоточного императивного языка программирования.

Ключевые слова: реляционное программирование, декларативное программирование, логическое программирование в ограничениях, табличная мемоизация, конструктивное отрицание.

Ссылка при цитировании: Моисеенко Е.А., Подкопаев А.В. Реляционное программирование с мемоизацией и отрицанием // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 2018. Т. 11. № 2. С. 35–46. DOI: 10.18721/JCSTCS.11203

RELATIONAL PROGRAMMING WITH MEMOIZATION AND NEGATION

E.A. Moiseenko, A.V. Podkopaev

St. Petersburg State University, St. Petersburg, Russian Federation

The relational paradigm allows to express programs as relations. Unlike functions, the relations do not distinguish input and output parameters, thus a single relational program can be used to solve several related problems. Relational interpreters are of particular interest. These interpreters can execute a program, check that the program satisfies a set of constraints or generate a program that has specified properties. In order to take advantages of the relational interpreter, the developer needs to define the semantics of the programming language as a relation. In this work, we present an implementation of two useful extensions of relational programming: tabling and constructive negation. Tabling helps to traverse the state space of the interpreter

efficiently. Constructive negation allows to check that some state of the interpreter is unreachable. We show how this extensioncan be used on an example of a relational interpreter for a concurrent imperative programming language.

Keywords: relational programming, declarative programming, constraint logic programming, tabling, constructive negation.

Citation: Moiseenko E.A., Podkopaev A.V. Relational programming with memoization and negation. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems, 2018, Vol. 11, No. 2, Pp. 35–46. DOI: 10.18721/JCSTCS.11203

Введение

Реляционное программирование – это вид декларативного программирования, в рамках которого программы представляются как набор отношений. Среди реляционных программ особый интерес представляют реляционные интерпретаторы, способные не только исполнять программу, но и проверять удовлетворяет ли программа набору ограничений или генерировать программу, обладающую заданными свойствами [1]. Таким образом, на базе одного реляционного интерпретатора могут быть разработаны различные языковые инструменты, такие как символьный интерпретатор, верификатор, генератор кода и др. Для того чтобы воспользоваться всеми преимуществами реляционного интерпретатора, разработчику необходимо описать семантику языка в терминах отношений.

Для задания семантики императивных программ часто используются системы помеченных переходов (СПП) [2]. В наиболее общем виде СПП – это множество состояний и отношение перехода между парой состояний по заданной метке. При реализации семантики императивного языка как СПП особенно полезными оказываются такие расширения реляционного программирования, как табличная мемоизация [9, 11] и конструктивное отрицание [3, 7, 8, 10]. Мемоизация позволяет запоминать и неоднократно использовать результаты запросов к отношениям, тем самым избегая повторных вычислений. Представив пространство состояний СПП как отношение и выполнив его мемоизацию, можно затем эффективно выполнять его обход. Конструктивное отрицание позволяет построить логическое отрицание запроса. При помощи конструктивного отрицания становится возможно, например, проверить, что заданное состояние СПП недостижимо.

В рамках данной работы мы реализовали табличную мемоизацию и конструктивное отрицание для языка реляционного программирования OCanren [17]. Используя эти расширения, мы разработали интерпретатор минималистичного императивного многопоточного языка программирования, а затем применили данный интерпретатор для верификации нескольких многопоточных алгоритмов, таких как алгоритм взаимного исключения Деккера [16], алгоритм взаимного исключения Петерсона [18], алгоритм барьера и др.

1. Реляционное программирование

Парадигма реляционного программирования использует многие идеи из *логического программирования в ограничениях* (constraint logic programming) [4] и, в частности, от языков семейства Prolog [9, 11]. Одним из представителей реляционной парадигмы является язык программирования MiniKanren [1]. MiniKanren состоит из небольшого числа примитивов. Благодаря этому он может быть легко реализован как встраиваемый предметно-ориентированный язык (embedded domain specific language). Одной из таких реализаций является OCanren [18], встроенный в функциональный язык OCaml.

Основы. Процесс программирования на реляционном языке можно условно разделить на два этапа: определение типов данных и отношений и формирование запросов к отношениям. Проведя аналогию с языком SQL, можно сказать, что отношения соответствуют таблицам, а запросы – SELECT-запросам к базе данных. При

формировании запроса в реляционном языке разработчик передает некоторые аргументы отношения, а другие аргументы заменяет свободными переменными. Если отношение может быть выполнено в контексте переданных аргументов, то запрос завершается успехом (`success`) и возвращает потенциально бесконечный список подстановок для свободных переменных, на которых выполняется отношение. Если отношение не может быть выполнено, то запрос завершается неудачей (`fail`).

```
let append xs ys =
  match xs with
  | []      → ys
  | x::xs′  → x::(append xs′ ys)

let appendº xs ys zs =
  ((xs ≡ []) ∧ (ys ≡ zs))
  ∨
  fresh (x xs′ zs′)
    (xs ≡ x::xs′) ∧
    (zs ≡ x::zs′) ∧
    (appendº xs′ ys zs′)
```

Листинг 1. Функция `append` и отношение `append°`

Рассмотрим функциональную программу конкатенации списков `append` и ее реляционный аналог appendє (листинг 1). При помощи отношения `append°` мы можем выполнить конкатенацию списков, сгенерировать все разбиения списка на пару списков или проверить, что конкатенация пустого списка с любым другим не изменяет последний (листинг 2). В ответе на последний запрос можно видеть *свободную логическую переменную*, обозначенную как $\__0$. Свободная логическая переменная в ответе на запрос означает, что на ее место может быть подставлен любой терм соответствующего типа.

Рассмотрим более подробно определение отношения `append°`. Можно видеть, что определение состоит из бинарного отношения ≡, вызова других отношений (в данном случае рекурсивного вызова `append°`), логических связок ∧ (конъюнкции) и ∨ (дизъюнкции), а также конструкции `fresh`.

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

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

run * (appendº [] q q) ⇝ success {
  q = _0
}
```

Листинг 2. Примеры запросов к отношению `append°`

Бинарное отношение эквивалентности `t ≡ u` определено на множестве логических термов. Два терма `t` и `u` находятся в отношении `t ≡ u`, если они унифицируются [15], т. е. существует подстановка, заменяющая вхождения свободных переменных в `t` и `u` на другие логические термы таким образом, что `t` и `u` становятся синтаксически эквивалентными. Например, термы `Cons(_0, Nil)` и `Cons(1, _1)` унифицируются, т. к. существует подстановка, которая ставит в соответствие первой переменной константу 1, а второй переменной — терм `Nil`.

Конструкция `fresh` служит для введения новых «свежих» переменных. Свежая переменная может быть унифицирована с любым другим термом. Далее в программе такая переменная может быть заменена на данный терм. В коде отношения `append°` с помощью `fresh` вводятся три новых переменных `x, xs′, zs′`. Эти переменные затем используются для того, чтобы декомпозировать исходные списки `xs` и `zs` на голову и хвост.

Отрицание. Помимо перечисленных выше конструкций в языке OCanren также имеется *ограничение неэквивалентности* ≢ (disequality constraint). Ограничение `t ≢ u` выполняется тогда и только тогда, когда термы `t` и `u` не связаны отношением `t ≡ u`. С помощью ограничения ≢ может быть определено отношение `remove°` (листинг 3), которое связывает список `ys` со списком `zs`, в котором удалено первое вхождение `x`.

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

Листинг 3. Отношение `remove°`

Отношение ≢ представляет собой очень ограниченную форму отрицания в реляционном программировании. К сожалению, этой формы отрицания нередко недостаточно для выражения многих полезных программ в реляционной форме. В частности, невозможно обобщить отношение `remove°` таким образом, чтобы оно выполняло удаление первого элемента списка, удовлетворяющего произвольному предикату `p°` (т. к. в третьем дизъюнкте отношения `remove°` требуется проверка, что элемент списка `y` не удовлетворяет предикату `p°`).

Ниже, в разделе 4, описан метод конструктивного отрицания [3, 7, 8, 10]. Данный метод впервые был предложен для языков семейства Prolog, однако в этой работе, насколько нам известно, впервые реализовано конструктивное отрицание для встроенного реляционного языка программирования.

Мемоизация. Это метод оптимизации программ, заключающийся в сохранении результата работы функции на конкретных аргументах и неоднократного использования этого результата при повторных вызовах функции [14].

Табличный метод мемоизации (*tabling*) позволяет выполнять мемоизацию отношений, которая способна существенно ускорить исполнение запросов к рекурсивным отношениям. Кроме того, некоторые запросы к мемоизированной версии отношения способны завершаться за конечное время, тогда как аналогичные запросы к немемоизированной версии не завершаются.

Табличный метод мемоизации известен в контексте языка программирования Prolog [9, 11]. Ниже, в разделе 3, мы представим улучшенный алгоритм табличной мемоизации для OCanren, поддерживающий отношения с ограничениями неэквивалентности.

2. Реляционный интерпретатор

В данном разделе описана общая структура реляционного интерпретатора, т. е. интерпретатора, заданного как отношение, для простого многопоточного императивного языка программирования. Также продемонстрируется как данный интерпретатор может использоваться для проверки инвариантов программ.

Реляционные СПП. Для того чтобы описать семантику императивного языка программирования в реляционной форме, используются системы помеченных переходов (СПП). В рамках данной статьи определим СПП как тройку (S, L, R), где S – это множество состояний, L – множество меток, а $R \subseteq L \times S \times S$ – отношение перехода. СПП тривиальным образом представима как реляционная программа: множествам состояний S и меток L соответствуют типы данных `State` и `Label`, а отношение перехода R может быть определено как отношение `step° :: Label × State × State`.

В качестве множества меток будем рассматривать множество пар $Tid \times Action$, где первый элемент является идентификатором потока, который выполняет действие, а второй элемент определяет само действие. В случае императивной программы, работающей с разделяемой памятью, действием может быть, например, чтение значения из разделяемой переменной или запись значения в разделяемую переменную.

Обход пространства состояний. При помощи отношения `step°` возможно описать отношение достижимости `reachable°` (листинг 4). Данное отношение параметризовано предикатом `p°` и связывает начальное состояние `t` с некоторым достижимым состоянием `t''`, на котором выполняется `p°`. Можно видеть, что отношение достижимости определено индуктивно. Любое состояние, на котором выполняется `p°`, достижимо из самого себя. Кроме того, состояние

t'' достижимо из состояния t, если существует переход из t в t' и t'' достижимо из t'.

```
let reachable° p° t t'' =
  ((p° t) ∧ (t ≡ t'')) ∨
  fresh (t')
    (step° t t') ∧
    (reachable° t' t'')
```

Листинг 4. Отношение достижимости

У приведенного выше определения reachable° имеется существенный недостаток: оно неэффективно обходит пространство состояний, поскольку одно и то же состояние может быть посещено несколько раз. Рассмотрим ромбовидную систему переходов, изображенную на рис. 1. Предположим, что требуется обойти все состояния, достижимые из состояния S_1. Начав обход, запрос тут же обнаружит, что из начального состояния существует два перехода в состояния S_2 и S_3. Далее запрос запустит два независимых подзапроса: один для обхода состояний, достижимых из состояния S_2, и другой для обхода состояний, достижимых из S_3. Оба подзапроса продолжат обход, обнаружив состояние S_4 и все состояния, достижимые из него, однако эти состояния будут посещены дважды.

Для того чтобы исправить этот недостаток, необходимо сохранять множество посещенных состояния. Данного эффекта можно добиться, применив к отношению

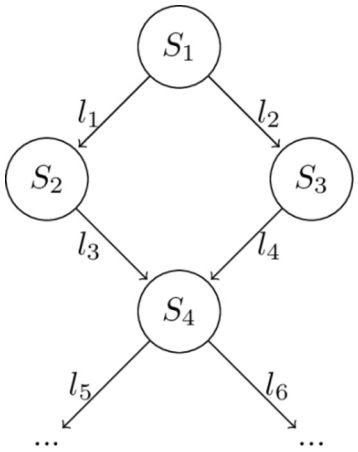

Рис. 1. Ромбовидная система переходов

reachable° табличный метод мемоизации реляционных программ. В рамках данной работы поддержка табличной мемоизации была добавлена в OCanren.

Проверка инвариантов. В предыдущем разделе было показано, что отношение reachable° может использоваться для обхода пространства состояний системы помеченных переходов, соответствующей императивной программе. С помощью данного отношения может быть определено отношение invariant° (листинг 5), проверяющее, что для данной программы выполняется инвариант, заданный предикатом safe°. Для проверки недостижимости ошибочного сосстояния необходим оператор отрицания. В языке OCanren отсутствовала поддержка отрицания, поэтому в рамках данного исследования мы разработали соответствующее расширение, основанное на методе конструктивного отрицания [3, 7, 8, 10].

```
let invariant° safe° t = ¬ (
  fresh (t')
    (reachable° (¬safe°) t t')
)
```

Листинг 5. Проверка инварианта

3. Табличный метод мемоизации

Идея метода табличной мемоизации заключается в неоднократном использовании ответов на запрос. С мемоизированным отношением связывается таблица, в которой ключом является запрос (т. е. набор переданных аргументов отношения), а значением – список ответов на данный запрос. В дальнейшем, если в ходе исполнения программы вновь будет сделан запрос, эквивалентный некоторому более раннему, ответы на данный запрос будут извлечены из таблицы. Схема алгоритма табличной мемоизации представлена на рис. 2.

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

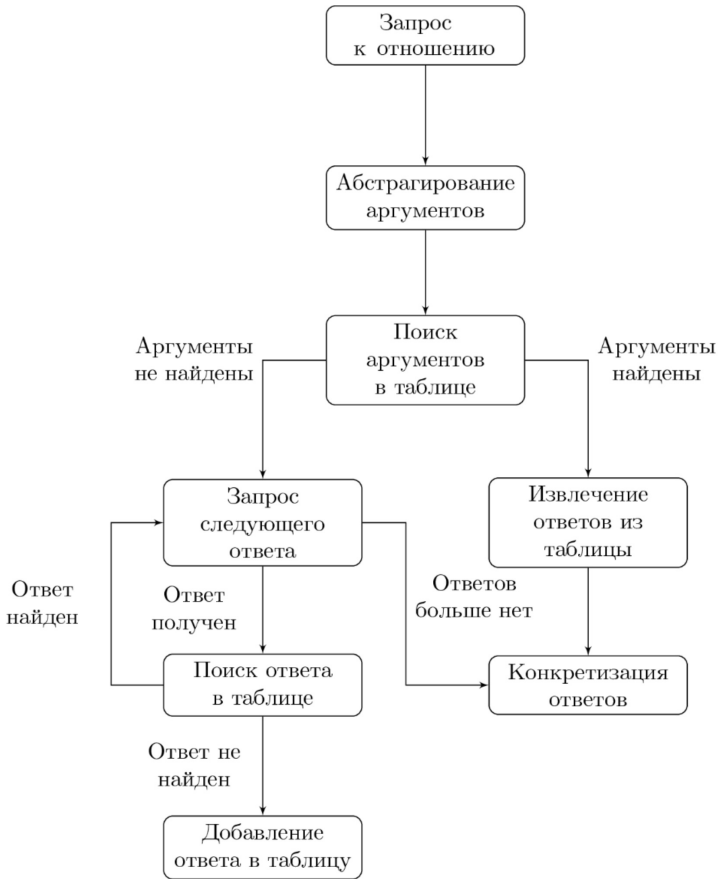

Рис. 2. Блок-схема алгоритма табличной мемоизации

до одного запроса. Следует отметить, что сильная абстракция может привести к тому, что запросы к рекурсивным отношениям перестанут завершаться, т. к. после абстракции в рекурсивный вызов попадут слишком общие аргументы. В нашей реализации при абстракции аргументов отбрасывается вся информация об ограничениях неэквивалентности. Так, например, запрос `reachable° s q{≠ s′}` (где `q` — свободная переменная) будет абстрагирован до запроса `reachable° s q`.

Полученный после абстракции список аргументов используется как ключ для поиска записи об эквивалентном запросе в таблице. Запросы считаются эквивалентными, если их аргументы *альфа-эквивалентны*, т. е. синтаксически равны с точностью до переименования переменных. В качестве структуры данных для хранения множества запросов алгоритм использует хеш-таблицу. Перед вычислением хеш-функции от логического терма сначала выполняется переименование всех переменных в порядке обхода терма в глубину (это необходимо для того, чтобы альфа-эквивалентные термы имели одинаковое значение хеш-функции).

Если запись не была найдена в таблице, необходимо выполнить запрос к оригинальной, немемоизированной версии отношения, чтобы получить ответы. После получения очередного ответа выполняется проверка: не был ли данный ответ добавлен в таблицу ранее. Если такой ответ встречается впервые, он добавляется в таблицу.

Если запись была найдена, т. е. запрос отношения с соответствующими аргументами уже выполнялся когда-то, 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 (в частности, в нашей реализации удаляется информация об ограничениях неэквивалентности). Абстрагированные аргументы затем используются для вызова оригинальной, немемоизированной версии отношения. 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 отброшены, как не удовлетворяющие исходному запросу. Таким образом, хотя информация об ограничениях неэквивалентности отбрасывается на стадии абстракции, на этапе конкретизации происходит проверка совместимости ограничений с полученными ответами. Возвращаясь к примеру с запросом `reachable° s q{≠ s′}`, получив список ответов на его абстрагированную версию `reachable° s q`, например, два ответа `reachable° s s′` и `reachable° s s′′`, алгоритм на стадии конкретизации отбросит первый ответ как неудовлетворяющий исходному запросу.

4. Конструктивное отрицание

Конструктивное отрицание [3, 7, 8, 10] — это форма отрицания в реляционном программировании, которая, получив список ответов на запрос, строит на основе этого списка набор ограничений. Немного упростив, можно сказать, что конструктивное отрицание меняет местами отношения эквивалентности и ограничения неэквивалентности, а также конъюнкции и дизъюнкции.

В рамках данной работы реализовано конструктивное отрицание для языка программирования OCanren. Эта реализация основана на теоретическом обосновании данной формы отрицания, приведенном в [8].

Стратифицированные программы. Метод конструктивного отрицания имеет ограничения и применим не ко всем реляционным программам. В частности, оператор отрицания может быть применен только к запросу, имеющему конечное количество ответов. Более того, семантика реляционной программы с отрицанием определена только для *стратифицированных* программ [8].

Для того чтобы дать определение стратифицированной реляционной программы, свяжем с каждой программой ориентированный граф, где вершинами выступают отношения. Между двумя вершинами A и B есть ребро, если в определении отношения A есть вызов отношения B. Будем помечать ребра меткой «+» если B встречается в определении A не под оператором отрицания, и меткой «−» иначе. Реляционная программа является стратифицированной, если в соответствующем графе нет циклов, содержащих хотя бы одно ребро с меткой «−».

На рис. 3 приведен пример простой стратифицированной программы, а на рис. 4 — пример 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

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

Рис. 3. Пример стратифицированной программы

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

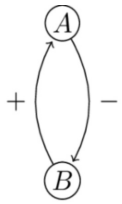

Рис. 4. Пример нестратифицированной программы

с меткой «—». Определение стратифицированной программы гарантирует, что между стратами нет циклических зависимостей.

Интуитивно, нестратифицированная программа содержит противоречащие друг другу отношения. В этом случае невозможно конструктивно построить набор ограничений при отрицании.

Семантика конструктивного отрицания. Определим процедуру конструктивного отрицания более формально. Для этого рассмотрим процедуру ответа на запрос как процесс трансляции реляционной программы в логическую формулу [8]. Напомним, что мы рассматриваем только запросы, имеющие конечное множество ответов. Для начала будем рассматривать простые программы, содержащие только отношения эквивалентности (≡). Для таких программ результатом трансляции будет логическая формула:

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

В данной формуле под кванторами существования стоят переменные, введенные в отношении с помощью конструкции `fresh`. Отношения эквивалентности `t ≡ u` в формуле соответствуют отношениям эквивалентности в коде реляционной программы. Наконец, заметим, что конъюнкции и дизъюнкции в коде реляционной программы можно перегруппировать таким образом, чтобы получить формулу в ДНФ.

Выполнимость конъюнкции отношений эквивалентности `t ≡ u` может быть проверена при помощи алгоритма унификации. Результатом работы данного алгоритма в случае, если набор отношений `t ≡ u` выполним, является подстановка, связывающая свободные переменные. Список данных подстановок является ответом на исходный запрос. Отрицание запроса представим в виде логической формулы:

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

Таким образом, список ответов на отрицание запроса должен соответствовать логическому отрицанию формулы E_+ (2). Подформулы вида `t ≢ u` напоминают ограничения неэквивалентности, однако стоит заметить, что термы `t` и `u` могут содержать универсально квантифицированные переменные из $\overline{x}$. Ограничение $\forall \overline{x}. t \not\equiv u$ выполнимо, если не существует подстановки, связывающей только переменные из $\overline{x}$, которая унифицирует термы `t` и `u`:

$$\forall \overline{x}. t \not\equiv u \Leftrightarrow \neg \exists \overline{x}. t \equiv u. \tag{3}$$

Таким образом, задача выполнимости ограничений обобщения также может быть решена при помощи унификации.

Теперь рассмотрим наиболее общий случай, когда логическая формула, соответствующая ответу на запрос, может содержать отношения эквивалентности и ограничения неэквивалентности с квантором всеобщности. Отрицание формулы, содержащей отношения эквивалентности и ограничения обобщения:

$$\begin{aligned} E &= \exists \overline{x}. \vee_{i=1}^{N} ((\wedge_{j=1}^{M_i} t_{ij} \equiv u_{ij}) \wedge \\ &\wedge (\forall \overline{y}. \wedge_{k=1}^{L_i} \vee_{l=1}^{Q_k} p_{kl} \not\equiv u_{kl})) \\ \neg E &= \forall \overline{x}. \wedge_{i=1}^{N} ((\vee_{j=1}^{M_i} t_{ij} \not\equiv u_{ij}) \vee \\ &\vee (\exists \overline{y}. \vee_{k=1}^{L_i} \wedge_{l=1}^{Q_k} p_{kl} \equiv u_{kl})). \end{aligned} \tag{4}$$

Формула (4) показывает, что при отрицании данной формулы получается дизъюнкция двух подформул. Левая подформула является ограничением неэквивалентности, а правая похожа на формулу положитель-

ной программы, с тем отличием, что термы p и u могут содержать как экзистенциально квантифицированные переменные из $\underline{y}$, так и универсально квантифицированные переменные из $\underline{x}$. Таким образом, ограничения в правом дизъюнкте имеют вид $\forall\underline{x}\exists\underline{y}.p \equiv r$. Выполнимость конъюнкции данных ограничений может быть проверена с помощью модифицированного алгоритма унификации [7]. Алгоритм принимает на вход множество универсально квантифицированных переменных U, множество экзистенциально квантифицированных переменных E, а также множество ограничений (`t ≡ u`), связанных конъюнкцией. Алгоритм возвращает `true` и подстановку, связывающую только свободные переменные, в случае если ограничения выполнимы, и `false` — в противном случае.

Логическая формула, соответствующая всей стратифицированной реляционной программе, строится поэтапно. Выполняется топологическая сортировка страт реляционной программы. Затем транслируется подпрограмма, соответствующая последней в порядке сортировки страте. Затем рассматривается предыдущая в порядке сортировки страта. Выполняется трансляция соответствующей ей подпрограммы, при этом все вхождения оператора отрицания заменяются на отрицание логической формулы, полученной на предыдущем этапе. После этого процесс повторяется для предыдущей страты. Так продолжается, пока не будут обработаны все страты. В итоге будет получена логическая формула, соответствующая исходному запросу. Выполнимость данной формулы может быть сведена к последовательности унификаций.

5. Апробация

Для апробации предложенных в данной работе расширений языка OCanren была рассмотрена задача применения реляционного интерпретатора многопоточных программ (см. раздел 2) к проблемам верификации программ и синтеза кода синхронизации.

Перед выполнением апробации реляционный интерпретатор многопоточных программ был обобщен для возможности исполнения программ в различных *моделях памяти*. Модель памяти определяет семантику взаимодействия потоков с разделяемой памятью. Модель *последовательной согласованности* (sequential consistency) [6] является наиболее простой. В рамках данной модели каждое возможное поведение программы является результатом некоторого чередования инструкции различных потоков. Современные языки программирования и аппаратные архитектуры предоставляют разработчикам более сложные модели, называемые *слабыми моделями памяти*. Помимо модели SC в нашем интерпретаторе реализована поддержка модели Total Store Order (TSO) [13], а также модели Strong Release-Acquire (SRA) [5].

В задачах верификации на вход интерпретатору подавалась многопоточная программа, а также инвариант, выполнимость которого следовало проверить. Были рассмотрены программа передачи сообщения между двумя потоками, алгоритм взаимного исключения Деккера [16], алгоритм взаимного исключения Коэна [12], а также алгоритм барьера для двух потоков. Была произведена попытка верификации данных алгоритмов в трех моделях памяти: SC, TSO и SRA. Известно, что алгоритм Деккера без использования дополнительных операций синхронизации корректен только в модели SC. Реляционный интерпретатор подтвердил этот факт, найдя контрпримеры в случае исполнения алгоритма Деккера в моделях TSO и SRA. Два других алгоритма были успешно верифицированы во всех трех моделях. В табл. 1 приведено среднее время работы интерпретатора для данных тестов с 95 % доверительным интервалом, полученное по десяти запускам.

В задаче синтеза синхронизации были рассмотрены те же алгоритмы, но на вход интерпретатору подавался шаблон программы, в котором были опущены *модификаторы доступа*. Модификаторы доступа определяют уровень синхронизации при чтении или записи разделяемой переменной в слабых моделях. На место модификаторов доступа были подставлены свободные переменные. Задачей реляционного интерпретатора было найти подстановку

Таблица 1

Результаты применения интерпретатора для верификации

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

Таблица 2

Результаты применения интерпретатора для синтеза синхронизации

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

для этих переменных. В модели SC все операции над разделяемыми переменными полностью упорядочены, по этой причине задача генерации модификаторов доступа в данной модели неактуальна. Результаты замеров (среднее и 95 % доверительный интервал времени работы по десяти запускам) приведены в табл. 2.

Из результатов апробации следует, что реляционный интерпретатор хорошо справляется с задачей верификации. В моделях TSO и SRA пространство состояний программ, как правило, существенно больше, чем в модели SC. По этой причине время верификации для этих моделей также увеличивается. С задачей генерации модификаторов доступа интерпретатор справляется намного хуже. Для алгоритма Деккера и Петерсона в модели SRA интерпретатор завершил работу по причине исчерпания виртуальной памяти (our of memory). В модели TSO на данных тестах интерпретатор завершил свою работу, однако это заняло достаточно много времени. Такие результаты связаны с тем, что каждый раз, когда реляционный интерпретатор встречает свободную переменную в исходной программе, он делает недетерминированный выбор. Таким образом, пространство поиска может расти экспоненциально с ростом количества свободных переменных в программе.

6. Обзор связанных работ

Одно из приложений языка MiniKanren – разработка реляционных интерпретаторов. В частности, в [1] представлен реляционный интерпретатор для подмножества функционального языка программирования Racket. Показано, как данный интерпретатор может использоваться для генерации программ по набору тестов или для генерации квайнов (quines) – программ, результатом выполнения которых является исходный текст программы. В отличие от этой работы, мы представляем реляционный интерпретатор для многопоточного императивного языка программирования и демонстрируем, как он может использоваться для проверки инвариантов программ.

Табличный метод мемоизации давно известен в контексте семейства языков программирования Prolog. Например, в языке XSB Prolog [11] табличная мемоизация применяется ко всем отношениям по умолчанию.

Метод конструктивного отрицания,

впервые предложенный в [3], ранее рассматривался в контексте логических языков программирования. В [8] описаны теоретические аспекты конструктивного отрицания, а также приведен алгоритм исполнения логической программы с конструктивным отрицанием для стратифицированных программ. Работа [10] обобщает конструктивное отрицание до логических программ с ограничениями. В [7] описана реализация конструктивного отрицания для более широкого класса программ, чем стратифицированные программы. Несмотря на существование множества работ, описывающих теоретические аспекты конструктивного отрицания, насколько известно нам, все современные реализации языка Prolog используют более простой метод «отрицание как неудача» (negation as a failure). Данный метод некорректен (unsound) в случае, если в запросе под отрицанием встречаются свободные переменные. В нашей статье предпринята попытка реализовать конструктивное отрицание для встроенного реляционного языка программирования.

Заключение

В данной статье представлена реализация двух расширений реляционного языка OCanren: табличной мемоизации и конструктивного отрицания. Табличная мемоизация позволяет разрабатывать эффективные интерпретаторы на реляционном языке программирования. Конструктивное отрицание увеличивает выразительную силу реляционного языка, позволяя выражать новые отношения.

Используя язык OCanren с данными расширениями, мы разработали реляционный интерпретатор для многопоточных императивных программ. Данный интерпретатор позволяет исследовать поведение многопоточных программ, а также проверять их инварианты.

В ходе апробации продемонстрировано, что реляционный подход может использоваться для верификации и генерации синхронизации в небольших многопоточных программах, но для применения данного подхода к более сложным программам требуются некоторые улучшения. Интеграция реляционного языка с современными решателями формул в теориях (SMT solvers) может ускорить проверку выполнимости ограничений. Другим направлением работы является применение техник суперкомпиляции и частичного исполнения.

СПИСОК ЛИТЕРАТУРЫ

1. **Byrd W.E., Ballantyne M., Rosenblatt G., Might M.** A unified approach to solving seven programming problems (functional pearl) // Proc. of the ACM on Programming Languages. 2017. Vol. 1. ICFP. P. 8.

2. **Baier C., Katoen J.P., Larsen K.G.** Principles of model checking. MIT press, 2008.

3. **Chan D.** Constructive negation based on the completed database // Proc. of ICLP-88. 1988.

4. **Jaffar J., Maher M.J.** Constraint logic programming: A survey // The Journal of Logic Programming. 1994. Vol. 19. Pp. 503–581.

5. **Lahav O., Giannarakis N., Vafeiadis V.** Taming release-acquire consistency // ACM SIGPLAN Notices. ACM, 2016. Vol. 51. No. 1. Pp. 649–662.

6. **Lamport L.** How to make a multiprocessor computer that correctly executes multiprocess programm // IEEE Transactions on Computers. 1979. No. 9. Pp. 690–691.

7. **Liu J.Y., Adams L., Chen W.** Constructive negation under the well-founded semantics // The Journal of Logic Programming. 1999. Vol. 38. No. 3. Pp. 295–330.

8. **Przymusinski T.C.** On constructive negation in logic programming. Cambridge, Massachusetts, MIT Press, 1989.

9. **Schrijvers T., Demoen B., Warren D.S.** TCHR: a framework for tabled CLP // Theory and Practice of Logic Programming. 2008. Vol. 8. No. 4. Pp. 491–526.

10. **Stuckey P.J.** Constructive negation for constraint logic programming // Proc. of 6th Annual IEEE Symposium on Logic in Computer Science. IEEE, 1991. Pp. 328–339.

11. **Swift T., Warren D.S.** XSB: Extending Prolog with tabled logic programming // Theory and Practice of Logic Programming. 2012. Vol. 12. No. 1-2. Pp. 157–187.

12. **Turon A., Vafeiadis V., Dreyer D.** GPS: Navigating weak memory with ghosts, protocols, and separation // ACM SIGPLAN Notices. ACM, 2014. Vol. 49. No. 10. Pp. 691–707.

13. **Sewell P., Sarkar S., Owens S., Zappa F.N., Myreen M.O.** x86-TSO: a rigorous and

usable programmer's model for x86 multiprocessors // Communications of the ACM. 2010. Vol. 53. No. 7. Pp. 89–97.

14. **Michie D.** Memo functions and machine learning // Nature. 1968. Vol. 218. No. 5136. P. 19.

15. **Baader F., Snyder W.** Unification theory // Handbook of automated reasoning. 2001. Vol. 1. Pp. 445–532.

16. **Dijkstra E.W.** Cooperating sequential processes // The Origin of Concurrent Programming. New York: Springer, 1968. Pp. 65–138.

17. **Kosarev D., Boulytchev D.** Typed embedding of a relational language in OCaml // International Workshop on ML. 2016.

18. **Peterson G.L.** Myths about the mutual exclusion problem // Information Processing Letters. 1981. No. 12(3). Pp. 115–116.

Статья поступила в редакцию 09.05.2018.

REFERENCES

1. **Byrd W.E., Ballantyne M., Rosenblatt G., Might M.** A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages*, 2017, Vol. 1, ICFP, P. 8.

2. **Baier C., Katoen J.P., Larsen K.G.** *Principles of model checking.* MIT press, 2008.

3. **Chan D.** Constructive negation based on the completed database. *Proceedings of ICLP-88*, 1988.

4. **Jaffar J., Maher M.J.** Constraint logic programming: A survey. *The Journal of Logic Programming*, 1994, Vol. 19, Pp. 503–581.

5. **Lahav O., Giannarakis N., Vafeiadis V.** Taming release-acquire consistency. *ACM SIGPLAN Notices.* ACM, 2016, Vol. 51, No. 1, Pp. 649–662.

6. **Lamport L.** How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 1979, No. 9, Pp. 690–691.

7. **Liu J.Y., Adams L., Chen W.** Constructive negation under the well-founded semantics. *The Journal of Logic Programming,* 1999, Vol. 38, No. 3, Pp. 295–330.

8. **Przymusinski T.C.** *On constructive negation in logic programming.* Cambridge, Massachusetts, MIT Press, 1989.

9. **Schrijvers T., Demoen B., Warren D.S.** TCHR: a Framework for Tabled CLP. *Theory and Practice of Logic Programming*, 2008, Vol. 8, No. 4., Pp. 491–526.

10. **Stuckey P.J.** Constructive negation for constraint logic programming. *Proceedings of 6th Annual IEEE Symposium on Logic in Computer Science*, IEEE, 1991. Pp. 328–339.

11. **Swift T., Warren D.S.** XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 2012, Vol. 12. No. 1-2, Pp. 157–187.

12. **Turon A., Vafeiadis V., Dreyer D.** GPS: Navigating weak memory with ghosts, protocols, and separation. *ACM SIGPLAN Notices.* ACM, 2014, Vol. 49, No. 10, Pp. 691–707.

13. **Sewell P., Sarkar S., Owens S., Zappa F.N., Myreen M.O.** x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 2010, Vol. 53, No. 7, Pp. 89–97.

14. **Michie D.** Memo functions and machine learning. *Nature*, 1968, Vol. 218, No. 5136, P. 19.

15. **Baader F., Snyder W.** Unification theory. *Handbook of automated reasoning*, 2001, Vol. 1, Pp. 445–532.

16. **Dijkstra E.W.** Cooperating sequential processes. *The Origin of Concurrent Programming.* Springer, New York, NY, 1968, Pp. 65–138.

17. **Kosarev D., Boulytchev D.** Typed embedding of a relational language in OCaml. *International Workshop on ML*, 2016.

18. **Peterson G.L.** Myths about the mutual exclusion problem. *Information Processing Letters*, 1981, No. 12(3), Pp. 115–116.

Received 09.05.2018.

СВЕДЕНИЯ ОБ АВТОРАХ / THE AUTHORS

МОИСЕЕНКО Евгений Александрович
MOISEENKO Evgenii A.
E-mail: e.moiseenko@2012.spbu.ru

ПОДКОПАЕВ Антон Викторович
PODKOPAEV Anton V.
E-mail: a.podkopaev@2009.spbu.ru

© Санкт-Петербургский политехнический университет Петра Великого, 2018