Санкт-Петербургский государственный университет

Программная инженерия
Кафедра системного программирования

Костюков Юрий Олегович

Композициональная верификация программ с динамической памятью на основе дизъюнктов Хорна

Выпускная квалификационная работа бакалавра

Научный руководитель:
ст. преп. Д. А. Мордвинов

Рецензент:
программист ООО «Интеллиджей Лабс» Д. С. Косарев

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Yurii Kostyukov

Compositional precise Horn-based verification of heap-manipulating programs

Bachelor's thesis

Scientific supervisor:
Senior lecturer Dmitry Mordvinov

Reviewer:
Software Developer at IntelliJ Labs Co. Ltd. Dmitry Kosarev

Saint-Petersburg
2019

Оглавление

Введение

Бóльшая часть современного программного обеспечения написана на языках с динамически выделяемой памятью, таких как `C++`, `Java`, `C#`. Автоматическая верификация и анализ программ на таких языках является трудоёмкой задачей [10]. При этом даже корректные с теоретической точки зрения методы анализа могут оказаться неэффективными из-за больших размеров анализируемых программ [8]. Для решения проблемы применения верификации к большим проектам были предложены *композициональные* техники анализа программ, которые хорошо зарекомендовали себя на практике [2, 6, 11, 24]. Такие техники выполняют анализ функций в *изоляции*, то есть вне контекста конкретного вызова, и переиспользуют промежуточные результаты анализа. Таким образом, можно свести верификацию больших систем к задаче верификации набора небольших фрагментов кода.

Большинство существующих композициональных техник являются *неточными* в том смысле, что они аппроксимируют снизу или сверху пространство состояний программы. Аппроксимирующие снизу подходы рассматривают не все поведения программы, например, при помощи раскрутки циклов на конечное число шагов [26]. Аппроксимирующие сверху подходы анализируют упрощённую версию программы, что на практике приводит к большому числу ложных срабатываний. Примером такого подхода может служить абстрактная интерпретация [9], которая работает в абстрактном домене. Таким образом, стоит задача точного анализа программ, которую можно решить при помощи особой *модели памяти*, т.е. представления состояний программы. Модель памяти должна быть достаточно гибкой и выразительной, чтобы с её помощью можно было описать *произвольные* свойства программы, тем самым охватив все сценарии её поведения. Свойства программ в модели могут быть выражены в виде логических формул.

Специальным видом логических формул являются дизъюнкты Хорна, которые с каждым годом всё активнее используются в верификации [5, 16, 22, 24]. Существует значительное количество решателей

(solvers), работающих с дизъюнктами Хорна — Spacer [3], Eldarica [15], HoIce [17], QARMC [25]. Во многих случаях они показывают лучшие результаты, чем классические SMT-решатели. Однако сведение программ с динамической памятью к дизъюнктам Хорна до сих пор оставалось открытой проблемой.

На кафедре системного программирования СПбГУ ведутся исследования по применимости дизъюнктов Хорна к анализу программ с динамической памятью. Для этого разрабатывается формализм *композициональной символьной памяти* и инструмент V# для верификации .NET-программ. Используемый в инструменте V# подход позволяет сводить программы к дизъюнктам Хорна второго и более порядков. Далее дизъюнкты Хорна высших порядков необходимо сводить к дизъюнктам Хорна первого порядка, для которых существуют эффективные решатели. Таким образом, имея V# и решатель дизъюнктов высших порядков, можно получить инструмент автоматической композициональной верификации программ с динамической памятью.

1. Постановка задачи

Целью данной работы является разработка подхода к автоматической композициональной верификации программ с динамической памятью в рамках проекта V#. Для её достижения были поставлены следующие задачи:

- предложить подход к композициональной верификации программ с динамической памятью;
- доказать корректность предложенного подхода;
- выполнить его реализацию;
- провести апробацию данного подхода.

2. Основные понятия

2.1. Композициональность

Под композициональностью будем понимать возможность анализа фрагментов кода *в изоляции*, т.е. как отдельной программы, независимо от того, как исполнение попадает в текущую точку кода. Результатом анализа фрагмента кода является некоторый набор артефактов, которые могут быть *переиспользованы* позже при обращении к данному фрагменту кода.

```
class Node {
    int Key;
    Node Tail;
    Node(int k, Node t) {
        Key = k;
        Tail = t;
    }
}

void inc(Node node) {
    if (node != null) {
        node.Key = node.Key + 1;
    }
}

void Main() {
    Node x = new Node(42, null);
    inc(x);
}
```

Листинг 1: Пример: композициональный анализ кода

Результатом анализа функции `inc` на лист. 1 может быть следующая тройка Хоара:

$$\left\{ \; node.Key \geq 0 \; \right\} \; inc(node) \; \left\{ \; node.Key > 0 \; \right\}.$$

Эта тройка может быть далее переиспользована при анализе функции

`Main`, например, чтобы доказать, что после вызова `inc` $x.Key$ будет положителен.

2.2. Аппроксимации состояний программ

Будем говорить, что техника анализа *аппроксимирующая снизу* [1], если она рассматривает часть достижимых состояний программы. Любая найденная при помощи такой техники ошибка будет являться валидной ошибкой в исходной программе. Однако любые доказательства корректности при помощи такой техники нельзя рассматривать как валидные, потому что ошибка может содержаться в одном из исключённых состояний.

Аналогично, будем говорить, что техника анализа *аппроксимирующая сверху* [1], если она рассматривает помимо достижимых состояний также те, в которые программа попасть не может. Соответственно, если удалось доказать корректность аппроксимации, исходная программа также будет корректна. С другой стороны, найденная в аппроксимации ошибка может не быть ошибкой в исходной программе, потому что она может содержаться в одном из «лишних» состояний.

Таким образом, *точной* будем называть технику, которая рассматривает все достижимые состояния программы и только их.

3. Обзор предметной области

3.1. Хорн/SMT-решатели

Последние десять лет хорошие результаты показывают инструменты анализа, опирающиеся на инструменты проверки выполнимости логических формул. Ведущими исследователями в области верификации был выдвинут тезис о том, что именно решатели логики должны стать твёрдым основанием в сфере анализа программ [5, 16].

SMT-решатели — решатели логики первого порядка с ограничениями. Они принимают на вход формулу логики первого порядка с символами из заранее заданных теорий. Ниже представлен пример формулы, описывающей отсортированный массив:

$$\forall i \forall j (i \leq j \leq n \Rightarrow a[i] \leq a[j]).$$

Если формула выполнима, решатель вернёт SAT[1] и модель с оценкой свободных переменных. Иначе решатель возвращает UNSAT[2]. Так как некоторые теории или их комбинации неразрешимы, решатель может иногда возвращать UNKNOWN или зависать.

В анализе программ при помощи логических решателей обычно проверяется достижимость той или иной ветки исполнения. Например, если ветка ограничивается условием $p(x)$, решателю будет послан запрос: «выполнимо ли $p(x)$?». Если решатель возвращает SAT и модель, это означает, что ветка достижима, а в модели закодирована конкретная трасса (значение x), которая приведёт к попаданию в данную ветку (например, контрпример, если ветка ведёт к падению). Если же решатель вернул UNSAT, данная ветка не достижима ни при каких условиях.

Хорн-решатели базируются на SMT-решателях и ориентированы на решения систем дизъюнктов Хорна с ограничениями [4]. Трансляция в дизъюнкты Хорна применяется, например, для анализа функциональных программ [7]. Например, на лист. 2 описана функция Маккарти 91,

[1] англ. *satisfiable*, «выполнимо»

[2] англ. *unsatisfiable*, «невыполнимо»

которая возвращает 91 для всех $x \leq 101$. Можно проверить это свойство, записав следующий набор дизъюнктов и передав их решателю (запятыми обозначаются конъюнкции, а левыми стрелками — импликации):

$$\begin{aligned} mc(x, r) &\leftarrow x > 100, r = x - 10 \\ mc(x, r) &\leftarrow x \leq 100, y = x + 11, mc(y, z), mc(z, r) \\ r = 91 &\leftarrow mc(x, r), x \leq 101 \end{aligned}$$

Различают дизъюнкты Хорна *высших* порядков и *первого* порядка (по аналогии с логиками высших порядков и первого порядка). В дизъюнктах Хорна высшего порядка предикаты могут принимать в качестве аргументов другие предикатные символы (аналогично функциям высших порядков в функциональном программировании). Наиболее известные решатели дизъюнктов Хорна первого порядка — Spacer [3], Eldarica [15], HoIce [17], QARMC [25].

Так как задачи решения дизъюнктов Хорна высших порядков и вывода ограниченных (refinement) типов эквивалентны [14], среди Хорн-решателей высших порядков можно перечислить следующие наиболее известные — MoCHi [18, 23], RType [17], RCaml [28], DOrder [29].

3.2. Существующие модели динамической памяти

При повышении точности моделей памяти резко возрастает их сложность, что приводит к падению эффективности анализа (см. *взрыв путей исполнения* и т.п.). Для решения этой проблемы применяются различные *композициональные* техники анализа программ, построенные на соответствующих композициональных моделях памяти.

Ключевыми подходами к анализу программ с динамической памятью являются: (а) кодирование поведений программ в разрешимые фраг-

```
let rec mc (x : int) : int =
  if x > 100 then (x - 10) else mc (mc (x + 11))
```

Листинг 2: Функция Маккарти 91

менты логики первого порядка с ограничениям [2,24]; (b) анализ *«формы кучи»* (shape analysis), например при помощи *логики разделения* (separation logic) [6,20]; (c) трансляция во фрагменты функциональных языков [13].

3.2.1. Кодирование программы в логические формулы: композициональное символьное исполнение

Типичным примером инструмента, основанного на композициональном символьном исполнении, может служить Pex [26], предназначенный для генерации тестовых покрытий для .NET-приложений. Для него была предложена композициональная модель памяти [2], которая позволила увеличить размер покрытий и скорость работы инструмента, а также уменьшить число исследуемых путей. Суть подхода состоит в построении по каждой функции SMT-формул первого порядка, описывающих часть поведений функции (пример на рис. 1).

Pex основан на динамическом символьном исполнении — аппроксимирующей снизу технике, которая позволяет рассуждать только о *части* возможных состояний программ (что нормально для инструмента генерации тестовых покрытий, но неприемлемо для верификатора).

3.2.2. Кодирование программы в логические формулы: абстрактная интерпретация

В качестве примера рассмотрим SeaHorn [24] — инструментарий для верификации LLVM. Композициональность и эффективность до-

```
int abs(int x) {
    if (x > 0) return x;
    else if (x == 0)
        return 100;
    else return -x;
}
```

$$\begin{aligned}\forall x.D_{abs}(x) \Leftrightarrow\ & (x \leq 0 \wedge x = 0 \wedge abs(x) = 100) \\ & \vee (x \leq 0 \wedge x \neq 0 \wedge abs(x) = -x) \\ & \vee (x > 0 \wedge abs(x) = x)\end{aligned}$$

(a) Пример функции (b) Пример построенной по функции формулы

Рис. 1: Анализ функции при помощи Pex
(пример взят из статьи [26])

стигаются путём использования решателя дизъюнктов Хорна первого порядка с ограничениями. Динамическая память и указатели представляются соответственно при помощи теорий массивов и линейной арифметики в SMT. Пример порождаемых дизъюнктов представлен на рис. 2. SeaHorn в отличие от Pex полагается на более выразительный фрагмент логики, который в частности позволяет описывать циклы (как на рис. 2). В Pex же анализ циклов и рекурсии производится классическими неточными методами, например раскруткой на фиксированное число шагов.

Несмотря на то, что SeaHorn использует выразительный фрагмент логики и эффективные решатели, этот инструмент основан на абстрактной интерпретации — аппроксимирующей сверху технике, которая работает в абстрактном домене, и потому допускает *ложные срабатывания* (т.е. инструмент корректно выделяет безопасные участки программы, но может неверно указать ошибку в коде).

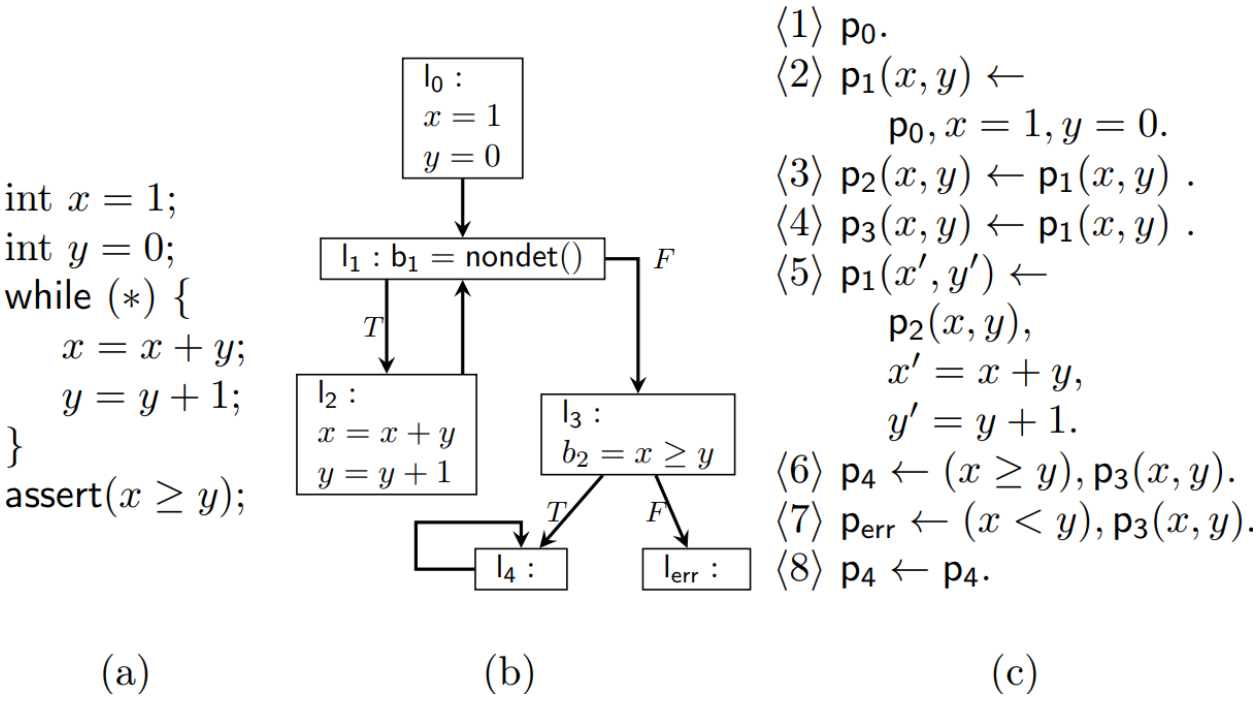

Рис. 2: Анализ функции при помощи SeaHorn:
(a) программа, (b) граф потока управления, (c) построенные по программе дизъюнкты
(пример взят из статьи [24])

3.2.3. Анализ формы кучи: графовые модели памяти

Predator [20] — инструмент композиционального анализа C кода, многократный победитель секций Memory Safety и Heap (низкоуровневый анализ памяти) на соревнованиях SV-COMP, посвящённых анализу программ. Его характерными чертами являются совмещение техник символьного исполнения и абстрактной интерпретации в домене *символьных графов памяти* (symbolic memory graphs, SMG) [12], вдохновлённом логикой с разделением. SMG *ориентирован* на доказательства свойств вложенных дву- и односвязных списков, которые часто встречаются, например, в коде ядра Linux. Это позволило инструменту Predator хорошо себя показать не только на соревнованиях, но и в анализе реальных проектов, таких как драйвера Linux, lvm2, менеджер памяти Firefox и других.

В то же время основные достоинства инструмента Predator являются его же основными недостатками. В случае возникновения необходимости расширения области анализа на новые структуры, например на деревья, необходимо будет также значительно расширить инструмент, исправив базовые алгоритмы. Более того, некоторые алгоритмы могут оказаться нерасширяемыми. Тем самым, такую кропотливую работу необходимо проводить для каждой возможной структуры, что означает отсутствие точного анализа для произвольных структур, определённых пользователем.

Анализ памяти в Predator можно, скорее, назвать *анализом формы кучи* (shape analysis) [8], т.е. анализом того, как объекты в куче *связаны* друг с другом. Такой вид анализа даёт возможность доказать, например, что функция копирования списка создаёт новые объекты в памяти, не пересекающиеся с исходным списком. Однако этот подход слишком «грубый», чтобы, например, доказать качественные утверждения о содержимом списка. В инструменте Predator в сложных случаях информация из графа памяти удаляется, что позволяет модели быть простой и корректной, абстрактной (аппроксимацией сверху), но неточной.

3.2.4. Анализ формы кучи: логика разделения

Логика разделения (separation logic) [19, 21] позволяет судить об отсутствии пересечений объектов в динамической памяти. Такие ограничения на память задаются при помощи специальных логических связок: разделяющей конъюнкции и импликации. Композициональность достигается при помощи соответствующих правил вывода (например т.н. «Frame rule») и т.н. *биабдукции* [8]. Однако логика разделения (и построенные на ней инструменты, например Infer [6]) ориентирована на *анализ формы кучи*, т.е. на то, как объекты связаны друг с другом в памяти, а не на их содержимое. Такой неточный анализ на практике приводит к ложным срабатываниям.

3.2.5. Трансляция в функциональные языки

Как было замечено в работе [27], циклы и рекурсивные функции в императивных языках зачастую реализуют функциональные паттерны обхода коллекций (`map, filter,` ...), что позволяет транслировать код на императивных языках в функциональные языки. Эта идея была подхвачена работой [13], в которой предложен метод анализа императивных программ с динамической памятью при помощи трансляции циклов и непосредственно динамической памяти в чистые функции. Например, императивная программа с доступом к динамической памяти на лист. 3 будет транслирована в функциональную программу на лист. 4.

```
auto a = new int[n];
int l = 0;
while (l < n) {
    a[l] = l + 3;
    l++;
}
```

Листинг 3: Пример программы, меняющей динамическую память (пример взят из статьи [13])

```
let l = λ(i). if i ≥ 0 then i + 1 else 0
let a = λ(i). if i ≥ 0 then a(i - 1)[i ↦ i + 3] else []
```

Листинг 4: Пример трансляции программы в функциональный язык (пример взят из статьи [13])

Здесь i — это параметр, отражающий *время*, т.е. номер итерации цикла. В конструкции $a(i-1)[i \mapsto i+3]$ запись $a(i-1)$ означает получение массива «в прошлый момент времени», в который, конструкцией $[i \mapsto i+3]$, по адресу i кладётся значение $i+3$. Анализ построенной функциональной программы далее производится путём упрощения по набору правил переписывания. Заметим, что ни во время трансляции, ни во время переписывания приближенные операции не выполняются.

Таким образом, современные методы анализа динамической памяти либо некомпозициональны и, соответственно, неэффективны, либо неточны: являются аппроксимациями снизу или сверху. Наиболее перспективным подходом к точному композициональному анализу динамической памяти может стать трансляция программ в формулы логики высших порядков, для которых существуют различные решатели.

4. Подход к композициональной верификации

Основная идея предлагаемого подхода состоит в том, чтобы при помощи алгоритма композиционального символьного исполнения транслировать код на исходном языке в символьные состояния и символьные выражения (формализм *композициональной символьной памяти*), из которых далее строить *уравнения на состояния*. Кодируя эти уравнения в дизъюнкты Хорна и передавая их в сторонний решатель, можно получить результаты верификации исходной программы. Схема подхода представлена на рис. 3.

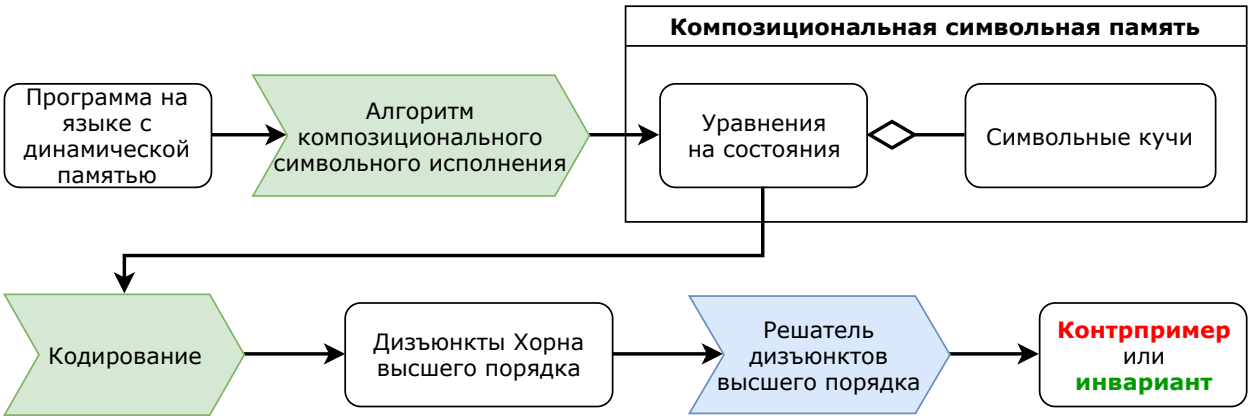

Рис. 3: Автоматическая композициональная верификация программ с динамической памятью. Синим отмечены сторонние инструменты.

4.1. Композициональная символьная память

В рамках нашей группы формальной верификации был разработан формализм *композициональной символьной памяти* (КСП). Далее будет проведён его краткий обзор. Доказательства всех теорем приведены в приложении А.

4.1.1. Символьные выражения

Перед тем, как ввести понятие символьной памяти, следует определить символьные термы, локации и ограничения.

Символьный терм ($term$) — это либо некоторое выражение над термами (в общем виде $op(term, \ldots, term)$, может быть построением слож-

$$
\begin{aligned}
term &::= leaf \mid op(term, \ldots, term) \mid LI(loc) \mid loc \mid union\big(\langle guard, term\rangle^*\big) \\
loc &::= \text{именованная локация} \mid (\text{конкретный адрес}).field \mid null \\
&\quad \mid LI(loc).field \mid union\big(\langle guard, loc\rangle^*\big) \\
guard &::= \top \mid \bot \mid \neg guard \mid guard \wedge guard \mid guard \vee guard \\
&\quad \mid term = term \mid loc = loc
\end{aligned}
$$

Рис. 4: Символьные термы

ных структур, например кортежей), либо символьный адрес локации в памяти (loc), либо объединение термов ($union$).

В куче есть именованные и неименованные локации. Неименованная локация может быть конкретной или символьной (обозначается $LI(x)$). $s.field$ — доступ к полю структуры.

Заметим, что локация-источник символьного значения LI может быть также символьной, так что, например, термы вида $LI\big(LI(node).field\big) + 1$ также допустимы.

$guard$ — это *ограничение*, логическая формула (условие пути или условие ветвления). Такие формулы «защищают» элементы *символьных объединений* ($union$). Символьное объединение — это обобщение широко используемого символа $ite(condition, thenExpr, elseExpr)$. За ними будет закреплена следующая семантика: $x = union\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)$ т. и т.т., когда $(g_1 \wedge x = v_1) \vee \ldots \vee (g_n \wedge x = v_n)$.

Выражениями назовём термы и ограничения. *Примитивные* выражения — это натуральные числа, именованные локации, конкретные адреса в куче, $null$, $\top$ и $\bot$. *Операциями* являются сложение, вычитание, унарный минус, сравнения, логические связки, взятие элемента кортежа и т.п.

Равенство символьных термов *семантическое*, то есть, например, $2*(x+1) = x+x+4-2$ и $union\big(\langle x+5 = y+4, 7\rangle, \langle \bot, 42\rangle\big) = union\langle x+1 = y, 7\rangle$.

Далее следует первый набор свойств, который позволит далее пользоваться формализмом как системой переписывания.

Свойство 1. (a) $union\langle \top, v\rangle = v$

(b) $union\big(\langle \bot, v\rangle \cup X\big) = union\big(X\big)$

(c) $union\Big(\Big\langle g, union\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)\Big\rangle \cup X\Big) =$
$= union\Big(\big\{\langle g \wedge g_1, v_1\rangle, \ldots, \langle g \wedge g_n, v_n\rangle\big\} \cup X\Big)$
В частности,

- $union\big(\langle g, union\big(\varnothing\big)\rangle \cup X\big) = union\big(X\big)$
- $union\Big\langle g_1 \vee \ldots \vee g_n, union\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)\Big\rangle =$
$union\Big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\Big)$

(d) $union\big(\big\{\langle g_1, v\rangle, \ldots, \langle g_n, v\rangle\big\} \cup X\big) = union\big(\langle g_1 \vee \ldots \vee g_n, v\rangle \cup X\big)$
В частности, $union\big(\big\{\langle g, v\rangle, \langle g \wedge g_1, v\rangle, \ldots, \langle g \wedge g_n, v\rangle\big\} \cup X\big) =$
$union\big(\langle g, v\rangle \cup X\big)$

(e) $op\Big(union\big(\langle g_1^1, e_1^1\rangle, \ldots, \langle g_{n_1}^1, e_{n_1}^1\rangle\big), \ldots, union\big(\langle g_1^m, e_1^m\rangle, \ldots, \langle g_{n_m}^m, e_{n_m}^m\rangle\big)\Big) =$
$= union\Big(\Big\{\Big\langle g_{i_1}^1 \wedge \ldots \wedge g_{i_m}^m, op(e_{i_1}^1, \ldots, e_{i_m}^m)\Big\rangle \mid 1 \le i_j \le n_j\Big\}\Big)$
В частности, для непересекающихся ограничений $g_1, \ldots, g_n$,
$op\Big(union\big(\langle g_1, e_1^1\rangle, \ldots, \langle g_n, e_n^1\rangle\big), \ldots, union\big(\langle g_1, e_1^m\rangle, \ldots, \langle g_n, e_n^m\rangle\big)\Big) =$
$= union\Big(\Big\langle g_1, op(e_1^1, \ldots, e_1^m)\Big\rangle, \ldots, \Big\langle g_n, op(e_n^1, \ldots, e_n^m)\Big\rangle\Big)$

4.1.2. Символьные кучи

Определение 1. *Символьная куча* — это частичная функция $\sigma : loc \to term$, удовлетворяющая следующему требованию (*инвариант кучи*):

$$\forall x, y \in dom(\sigma), union\langle x = y, \sigma(x)\rangle = union\langle x = y, \sigma(y)\rangle. \qquad (1)$$

Определение 2. Пустая куча ϵ — это частичная функция с областью действия $dom(\epsilon) = \varnothing$ (она, очевидно, удовлетворяет (1)).

Определение 3. *Чтение* локации x в символьной куче σ возвращает

терм и определяется следующим образом:

$$read(\sigma, x) = union\Big(\big\{\langle x = l, \sigma(l)\rangle \mid l \in dom(\sigma)\big\} \cup \langle \bigwedge_{l \in dom(\sigma)} x \neq l, LI(x)\rangle\Big). \quad (2)$$

Интуитивно, чтение пытается сопоставить (возможно символьную) ссылку x с каждым (возможно символьным) адресом локации в σ. Если ссылка и некоторый адрес совпали, результатом чтения будет значение, лежащее по этому адресу. Если не было найдено ни одного совпадения, возвращается символьное значение $LI(x)$.

Очевидно, что при $x \in dom(\sigma)$ $read(\sigma, x) = \sigma(x)$. Одно из ограничений $x = l$ будет выполняться, когда как один из конъюнктов $\bigwedge_{l \in dom(\sigma)} x \neq l$ будет, наоборот, невыполним, а значит значение $LI(x)$ получиться не может.

Можно сделать одно важное наблюдение об определении 3. Технически множество ограничений в формуле (2) может содержать пересечения: в куче могут быть две (или более) символьные локации, которые могут совпадать при некоторых конкретных подстановках. Инвариант кучи (1) позволяет обойти возможную проблему с совпадающими адресами: благодаря нему при совпадении ограничений не будет конфликтов между «защищаемыми» значениями.

Пример 1. (Адреса записаны в шестнадцатеричной форме, чтобы отличать их от обыкновенных чисел).

Пусть $\sigma = \{\texttt{0x1} \mapsto 42; LI(x) \mapsto union(\langle LI(x) = \texttt{0x1}, 42\rangle, \langle LI(x) \neq$

$\texttt{0x1}, 7\rangle)\}$. Тогда

$$
\begin{aligned}
&read(\sigma, \texttt{0x1}) = \\
&= union\Big(\big\langle \texttt{0x1} = \texttt{0x1}, 42\big\rangle, \\
&\qquad \big\langle \texttt{0x1} = LI(x), union(\langle LI(x) = \texttt{0x1}, 42\rangle, \langle LI(x) \neq \texttt{0x1}, 7\rangle)\big\rangle, \\
&\qquad \big\langle \texttt{0x1} \neq \texttt{0x1} \wedge LI(x) \neq \texttt{0x1}, LI(\texttt{0x1})\big\rangle\Big) = \\
&= union\Big(\big\langle \top, 42\big\rangle, \big\langle LI(x) = \texttt{0x1}, union(\langle LI(x) = \texttt{0x1}, 42\rangle, \langle LI(x) \neq \texttt{0x1}, 7\rangle)\big\rangle, \\
&\qquad \big\langle \bot, LI(\texttt{0x1})\big\rangle\Big) = 42
\end{aligned}
$$

$$
\begin{aligned}
&read(\sigma, LI(y)) = \\
&= union\Big(\big\langle LI(y) = \texttt{0x1}, 42\big\rangle, \\
&\qquad \big\langle LI(y) = LI(x), union(\langle LI(x) = \texttt{0x1}, 42\rangle, \langle LI(x) \neq \texttt{0x1}, 7\rangle)\big\rangle, \\
&\qquad \big\langle LI(y) \neq \texttt{0x1} \wedge LI(y) \neq LI(x), LI\big(LI(y)\big)\big\rangle\Big) \overset{\text{Св.1}}{=} \\
&= union\Big(\big\langle LI(y) = \texttt{0x1}, 42\big\rangle, \big\langle LI(y) = LI(x) \wedge LI(x) \neq \texttt{0x1}, 7\big\rangle, \\
&\qquad \big\langle LI(y) \neq \texttt{0x1} \wedge LI(y) \neq LI(x), LI\big(LI(y)\big)\big\rangle\Big)
\end{aligned}
$$

4.1.3. Композиция символьных куч

Определение 4. *Уточнение* выражения e в контексте символьной кучи σ обозначим $\sigma \bullet e$ и определим следующим образом.

1. Если e — это примитивное значение, то $\sigma \bullet e \overset{\text{def}}{=} e$.
2. $\sigma \bullet op(e_1, \ldots, e_n) \overset{\text{def}}{=} op(\sigma \bullet e_1, \ldots, \sigma \bullet e_n)$.
3. $\sigma \bullet union\{\langle g_1, t_1\rangle, \ldots, \langle g_n, t_n\rangle\} \overset{\text{def}}{=} union\{\langle \sigma \bullet g_1, \sigma \bullet t_1\rangle, \ldots, \langle \sigma \bullet g_n, \sigma \bullet t_n\rangle\}$.
4. $\sigma \bullet LI(l) \overset{\text{def}}{=} read(\sigma, \sigma \bullet l)$.

Интуитивно, $\sigma \bullet e$ — это выражение, получаемое подстановками значений из σ в символьные ячейки e: первые три пункта определения сохраняют структуру e, а 4 заполняет ячейку значением из σ.

Определение 5. *Композиция* символьных куч σ и σ' — это частичная функция $\sigma \circ \sigma' : loc \to term$, определяемая следующим образом:

$$(\sigma \circ \sigma')(x) \stackrel{\text{def}}{=} union\Big(\big\{\big\langle x = \sigma \bullet l, \sigma \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \cup \\ \cup \big\langle \bigwedge_{l \in dom(\sigma')} x \neq \sigma \bullet l, \sigma(x)\big\rangle\Big) \tag{3}$$

$\sigma \circ \sigma'$ определена на всех локациях, удовлетворяющих $\sigma \bullet l$, где $l \in dom(\sigma')$, и на всех локациях $dom(\sigma)$ (здесь запись $\{\sigma \bullet a \mid a \in A\}$ сокращается как $\sigma \bullet A$):

$$dom(\sigma \circ \sigma') = dom(\sigma) \cup \sigma \bullet dom(\sigma') \tag{4}$$

Композиция символьных куч отражает последовательную композицию в программировании: если σ_1 — это эффект фрагмента кода `A` и σ_2 — эффект фрагмента кода `B`, тогда $\sigma_1 \circ \sigma_2$ — это эффект `A;B`. Интуитивно, $\sigma_1 \circ \sigma_2$ — это символьная куча, полученная заполнениями символьных ячеек из σ_2 значениями из контекста σ_1 с последующей их записью в контекст σ_1.

Пример 2. Пусть $\sigma = \{\texttt{0x1} \mapsto 42; \texttt{0x2} \mapsto 7\}$ и $\sigma' = \{\texttt{0x2} \mapsto LI(\texttt{0x1}) - LI(\texttt{0x2})\}$. Тогда $\sigma \circ \sigma' = \{\texttt{0x1} \mapsto 42; \texttt{0x2} \mapsto 42 - 7\}$

Теорема 1. Если σ и σ' — символьные кучи, то $\sigma \circ \sigma'$ также символьная куча.

Теорема 2. Для произвольных символьных кучи σ, локации x и выражения e справедливо следующее:

(a) $read(\epsilon, x) = LI(x)$

(b) $\epsilon \bullet e = LI(e)$

(c) $\epsilon \circ \sigma = \sigma$

(d) $\sigma \circ \epsilon = \sigma$

Теорема 3. Для всех символьных куч σ, σ' и символьных локаций x справедливо следующее:

$$\sigma \bullet read(\sigma', x) = read(\sigma \circ \sigma', \sigma \bullet x).$$

Теорема 3 — это ключевое свойство КСП. Допустим, функция f, находясь в состоянии σ, вызывает функцию g с эффектом σ'. Из теоремы следует, что исследование g в изоляции с последующим чтением значения локации x и заполнением символьных ячеек σ' из контекстного состояния (то есть $\sigma \bullet read(\sigma', x)$) даст тот же результат, что и обычный вызов g с последующим чтением локации (то есть $read(\sigma \circ \sigma', \sigma \bullet x)$).

Лемма 1. Для всех символьных куч σ, σ' и символьных выражений e справедливо следующее:

$$(\sigma \circ \sigma') \bullet e = \sigma \bullet (\sigma' \bullet e).$$

Рассмотрим ситуацию, когда функция f вызывает g, а функция g вызывает h. Можно вычислить состояние вплоть до вызова h и затем воспроизвести на нём эффект h. Также можно вычислить эффект g (включающий эффект h) и воспроизвести его в контексте состояния f. Корректный формализм композициональной памяти не должен зависеть от порядка вычисления и воспроизведения результатов в таких случаях. Следующая теорема показывает, что КСП обладает указанным свойством.

Теорема 4. Для всех символьных куч σ_1, σ_2 и σ_3 справедливо следующее:

$$(\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3).$$

Обозначим с помощью Σ множество всех символьных куч.

Теорема 5. $(\Sigma, \circ)$ — моноид.

Доказательство. Теорема 1 показывает замкнутость множества Σ относительно операции $\circ$, по Теореме 2, ϵ — нейтральный элемент, и по Теореме 4, $\circ$ ассоциативна. □

Рассмотрим пример того, как символьные кучи и их композиции уже могут описывать части реальных программ. Третьей строке функции

```
1  class Node {
2      int Key;
3      Node Tail;
4      Node(int k, Node t) {
5          Key = k;
6          Tail = t;
7      }
8  }
9
10 void inc(Node node) {
11     if (node != null) {
12         node.Key = node.Key + 1;
13     }
14 }
15
16 void Main() {
17     Node x = new Node(42, null);
18     inc(x);
19 }
```

Листинг 5: Вызов из контекста

`inc` (строка 12 на лист. 5) будет соответствовать следующая[3] куча:

$$\{LI(node).Key \mapsto LI(LI(node).Key) + 1\}$$

Если произвести *подстановку* символьных значений ($LI(\dots)$) из более точного контекста, например из функции `Main`, которая вызывает `inc`, получится куча:

$$\texttt{Main} : \{x.Key \mapsto 42\} \circ inc = \{x.Key \mapsto 42 + 1\}.$$

(Здесь вместо $LI(node)$ был подставлен x, а вместо $LI(node).Key$ подставлен 42).

Таким образом, КСП обладает ожидаемыми свойствами композициональной памяти. В следующих двух секциях формализм КСП будет расширен двумя новыми операциями над символьными кучами: объединением и записью.

[3]Куча, описывающая всё поведение `inc` устроена сложнее

4.1.4. Объединение символьных куч

Так как в коде возможны разветвления, например в условных конструкциях и циклах, необходим оператор объединения символьных куч: он будет применяться в точках сочленения программы.

Определение 6. *Объединением* $\sigma = merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)$ символьных куч $\sigma_1, \ldots, \sigma_n$ по непересекающимся ограничениям $g_1, \ldots, g_n$ будем называть частичную функцию с $dom(\sigma) = \bigcup_{i=1}^{n} dom(\sigma_i)$, для которой выполняется следующее:

$$\begin{aligned}\big(merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)\big)(x) &\stackrel{\text{def}}{=} \\ &\stackrel{\text{def}}{=} union\Big(\langle g_1, read(\sigma_1, x)\rangle, \ldots, \langle g_n, read(\sigma_n, x)\rangle\Big).\end{aligned}$$

Лемма 2. Для любой символьной кучи σ и символьных локаций x, y справедливо следующее:

$$union\big(\big\langle x = y, read(\sigma, x)\big\rangle\big) = union\big(\big\langle x = y, read(\sigma, y)\big\rangle\big).$$

Теорема 6. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$ и непересекающихся ограничений $g_1, \ldots, g_n$, $merge\langle g_i, \sigma_i\rangle$ — символьная куча.

Теорема 7. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$ и локаций x справедливо следующее:

$$read\Big(merge\big\langle g_i, \sigma_i\big\rangle, x\Big) = union\Big\langle g_i, read(\sigma_i, x)\Big\rangle.$$

Теперь покажем, что объединение символьных куч инвариантно относительно стратегий поиска и объединения, выбранных в движке сим-

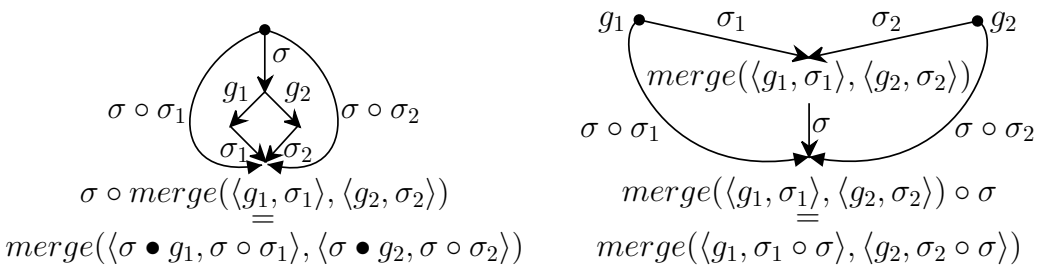

Рис. 5: Композиция объединений куч

вольного исполнения. То есть вне зависимости от того, в каких точках движок символьного исполнения производит объединение состояний, конечное состояние должно быть тем же самым. Чтобы гарантировать это, необходимо рассмотреть два случая объединений, представленные на рис. 5.

Теорема 8. Для любых символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$ и непересекающихся ограничений $g_1, \ldots, g_n$ выполняется следующее утверждение:

$$\sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big) = merge\big(\langle \sigma \bullet g_1, \sigma \circ \sigma_1\rangle, \ldots, \langle \sigma \bullet g_n, \sigma \circ \sigma_n\rangle\big).$$

Интересно, что симметричный случай гораздо сложнее. Например, возьмём $\sigma_1 = \{\texttt{0x1} \mapsto 0, \texttt{0x2} \mapsto 0, x \mapsto \texttt{0x1}\}$, $\sigma_2 = \{\texttt{0x1} \mapsto 0, \texttt{0x2} \mapsto 0, x \mapsto \texttt{0x2}\}$, $\sigma = \{LI(x) \mapsto 42\}$. Тогда

$$\begin{aligned} &dom\big(merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle) \circ \sigma\big) = \\ &= dom\big(merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle)\big) \cup \\ &\qquad \cup\, merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle) \bullet dom(\sigma) = \\ &= \{\texttt{0x1}, \texttt{0x2}, x, union(\langle g, \texttt{0x1}\rangle, \langle \neg g, \texttt{0x2}\rangle)\}, \end{aligned}$$

что не то же самое, что

$$\begin{aligned} &dom\big(merge(\langle g, \sigma_1 \circ \sigma\rangle, \langle \neg g, \sigma_2 \circ \sigma\rangle)\big) = \\ &= dom(\sigma_1 \circ \sigma) \cup dom(\sigma_2 \circ \sigma) = \{\texttt{0x1}, \texttt{0x2}, x\}. \end{aligned}$$

Чтобы избежать проблем такого вида, далее мы будем требовать, чтобы символьные ячейки $LI(*)$ удовлетворяли следующему дополнительному свойству: для любых непересекающихся ограничений $g_1, \ldots, g_n$ и символьных локаций $x_1, \ldots, x_n$ должно выполняться условие 5.

$$\begin{aligned} &LI\big(union(\langle g_1, x_1\rangle, \ldots, \langle g_n, x_n\rangle)\big) = \\ &= union\big(\langle g_1, LI(x_1)\rangle, \ldots, \langle g_n, LI(x_n)\rangle\big) \end{aligned} \tag{5}$$

Лемма 3. Для всех символьных куч σ, непересекающихся ограничений

$g_1, \ldots, g_n$ и локаций $x_1, \ldots, x_n$ справедливо следующее:

$$read\big(\sigma, union\langle g_i, x_i\rangle\big) = union\big\langle g_i, read(\sigma, x_i)\big\rangle.$$

Теперь мы хотим сформулировать вспомогательное утверждение: $merge\langle g_i, \sigma_i\rangle \bullet e = union\langle g_i, \sigma_i \bullet e\rangle$. Однако оно не всегда верно: может случиться так, что $g_1 \vee \ldots \vee g_n \neq \top$. Это может произойти, когда некоторые пути исполнения были отброшены, что может привести к попытке уточнить терм в несуществующей куче. Например, можно ожидать, что уточнение терма 42 в любой куче даст 42, однако в рамках наших определений мы получим $union\big(\langle g_1, 42\rangle, \ldots, \langle g_1, 42\rangle\big) = union\langle g_1 \vee \ldots \vee g_n, 42\rangle \neq 42$. Чтобы указанное выше утверждение выполнялось, необходимо ограничить результат условием того, что «вычисление существует».

Лемма 4. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$ и выражений e справедливо следующее:

$$union\Big\langle g_1 \vee \ldots \vee g_n, merge\langle g_i, \sigma_i\rangle \bullet e\Big\rangle = union\Big\langle g_i, \sigma_i \bullet e\Big\rangle.$$

Несмотря на то, что технически нельзя приравнять $merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big) \circ \sigma$ и $merge\big(\langle g_1, \sigma_1 \circ \sigma\rangle, \ldots, \langle g_n, \sigma_n \circ \sigma\rangle\big)$ как теоретико-множественный объекты, следующая теорема показывает, что это не будет большой проблемой благодаря определению чтения: «физически» кучи могут быть отображениями различных множеств ключей, однако с точки зрения движка символьного исполнения они будут совпадать.

Теорема 9. Для всех символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$ и локаций x справедливо следующее:

$$union\Big\langle g_1 \vee \ldots \vee g_n, read\big(merge\langle g_i, \sigma_i\rangle \circ \sigma, x\big)\Big\rangle = read\big(merge\langle g_i, \sigma_i \circ \sigma\rangle, x\big).$$

Теперь можно вернуться к нашему примеру (фрагмент из него показан на лист. 6). Полностью куча, описывающая `inc`, будет выглядеть

следующим образом:

$$merge(\langle (LI(node) = 0), \epsilon \rangle,$$
$$\langle \neg(LI(node) = 0), \{LI(node).Key \mapsto LI(LI(node).Key) + 1\} \rangle)$$

4.1.5. Запись в символьную кучу

В данной секции будем пользоваться следующим сокращением:

$$ite(c, a, b) \stackrel{\text{def}}{=} union\big(\langle c, a \rangle, \langle \neg c, b \rangle\big).$$

Определение 7. *Запись* символьного значения v в символьную локацию y символьной кучи σ — это символьная куча $write(\sigma, y, v)$, такая что для всех $x \in dom\big(write(\sigma, y, \cdot)\big) = dom(\sigma) \cup \{y\}$,

$$\big(write(\sigma, y, v)\big)\big(x\big) \stackrel{\text{def}}{=} ite\big(x = y, v, \sigma(x)\big)$$

Заметим, что инвариант кучи (1) для записей выполняется тривиально.

Теорема 10. Для всех символьных куч σ, символьных локаций x, y и символьных выражений v справедливо следующее:

$$read\big(write(\sigma, y, v), x\big) = ite\big(x = y, v, read(\sigma, x)\big).$$

Теорема 11. Для всех символьных куч σ, σ', символьных локаций y и символьных выражений v справедливо следующее:

$$\sigma \circ write(\sigma', y, v) = write(\sigma \circ \sigma', \sigma \bullet y, \sigma \bullet v).$$

```
void inc(Node node) {
    if (node != null) {
        node.Key = node.Key + 1;
    }
}
```

Листинг 6: Функция с ветвлением

Теорема 12. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$, символьных локаций y и символьных выражений v справедливо следующее:

$$write\big(merge\langle g_i, \sigma_i\rangle, y, v\big) = merge\big\langle g_i, write(\sigma_i, y, v)\big\rangle.$$

4.2. Уравнения на состояния

Представленные операторы КСП уже позволяют описать произвольные линейные участки кода на языке с динамической памятью. Формализм может быть далее расширен для описания всех поведений программ с циклами, рекурсией, функциями высшего порядка и т.д. Для этого строятся *уравнения на состояния*, которые и будут описывать произвольные поведения функций. Далее будет показан пример построения уравнения на состояния.

Уравнения на состояния функций с лист. 7 выглядят следующим

```
1  class Node {
2      int Key;
3      Node Tail;
4      Node(int k, Node t) {
5          Key = k;
6          Tail = t;
7      }
8  }
9
10 void inc(Node node) {
11     if (node != null) {
12         node.Key += 1;
13         inc(node.Tail);
14     }
15 }
16
17 void Main() {
18     Node c = new Node(30, null);
19     Node b = new Node(20, c);
20     Node a = new Node(10, b);
21     inc(a);
22     Console.WriteLine(a.Key);
23 }
```

Листинг 7: Рекурсивный вызов из контекста

образом:

$$Rec\big(inc\big) = Merge\big(\langle LI(node) = 0, \epsilon\rangle, \langle LI(node) \neq 0, h_1 \circ Rec\big(inc\big)\rangle\big)$$
$$App\big(Main\big) = h_2 \circ Rec\big(inc\big)$$

h_1 — куча, соответствующая строке 12 (инструкции `node.Key += 1;`); h_2 — куча, соответствующая строкам 18—20 (инициализации списка (`a, b, c`)). *Rec* и *App* — вспомогательные символы, позволяющие описывать поведения рекурсивных и обычных функций.

Уравнения на состояния следует читать следующим образом. В каком состоянии будет программа после вызова функции `inc`? Это будет состояние $Rec\big(inc\big)$. Оно является объединением двух состояний. Если $LI(node) = 0$ (`node == null` в исходном коде), его состояние будет пустым. Иначе оно является композицией состояния h_1 (в котором меняется `node.Key`) и состояния $Rec\big(inc\big)$, то есть его самого. Таким образом рекурсивная природа функции `inc` проявляется в рекурсивности уравнения на её состояние.

Легко заметить, что по уравнениям на состояния можно тривиально восстановить исходный код программы. Интуитивно это означает, что уравнения на состояния эквиваленты коду, то есть описывают все поведения программы и только их.

Таким образом, проанализировать программу есть то же самое, что решить систему уравнений на состояния. Чтобы это сделать, уравнения на состояния специализируются и сводятся к дизъюнктам Хорна с ограничениями, которые далее могут быть решены эффективными автоматическими решателями.

Доказательства корректности КСП были частично механизированы в интерактивной системе доказательств Coq[4].

[4] `https://github.com/Columpio/verified-symbolic-memory/tree/disj-def`

5. Реализация подхода

Композициональная модель символьной памяти используется в разрабатываемом проекте V#[5] как промежуточный шаг для трансляции .NET-программ в дизъюнкты Хорна высшего порядка с ограничениями (схема работы V# представлена на рис. 6). Язык разработки — F#.

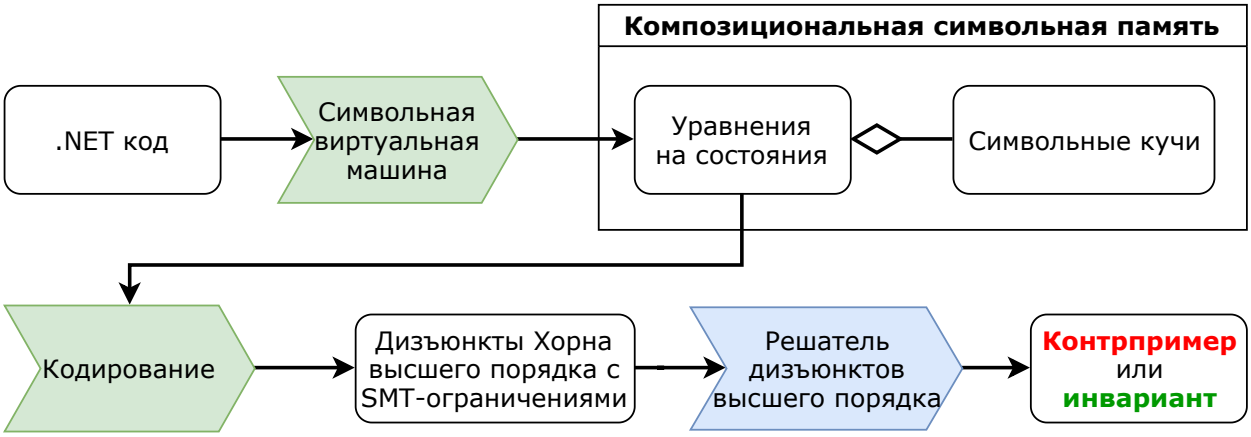

Рис. 6: Автоматическая композициональная верификация .NET программ. Зелёным отмечены компоненты проекта V#; синим отмечены сторонние инструменты.

При помощи техники композиционального символьного исполнения [2] V# транслирует байт-код .NET в символьные термы и уравнения на состояния. Далее уравнения на состояния кодируются в дизъюнкты Хорна высшего порядка с SMT-ограничениями, которые передаются стороннему решателю. Итоговая композициональность всей цепочки достигается за счёт композициональности модели памяти V# и композициональности стороннего решателя.

V# является инструментарием для построения анализаторов, верификаторов, генераторов тестовых покрытий и т.п. Так как трансляция точна в том смысле, что полученные дизъюнкты описывают те и только те поведения, которые были у исходной программы, получаемые анализаторы и верификаторы могут регулировать точность анализа и повышать её до бесконечности.

На рис. 7 представлена структура проекта V#.

[5] https://github.com/dvvrd/VSharp

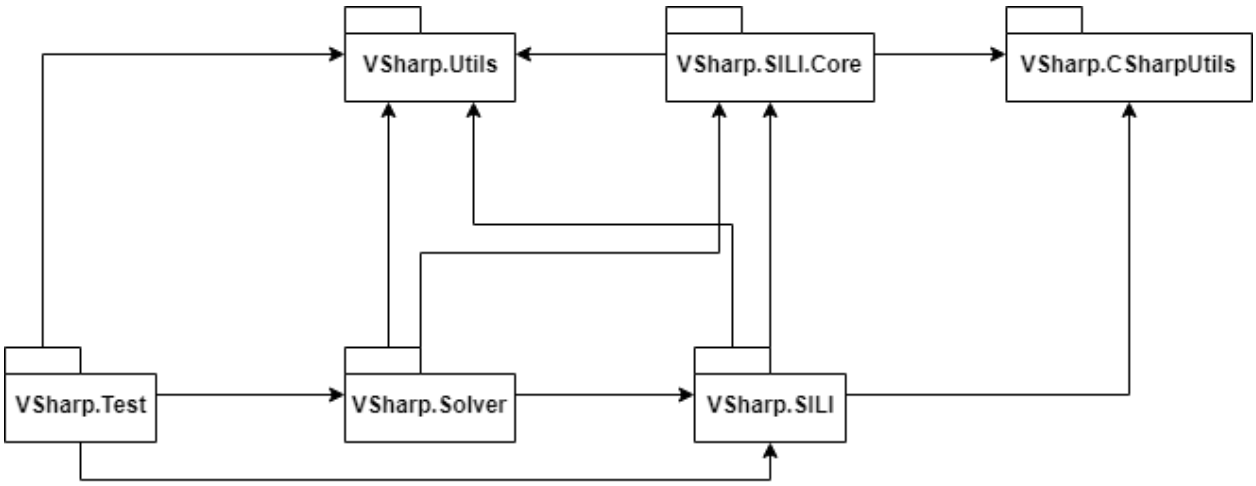

Рис. 7: Структура проекта V#

Модуль `VSharp.SILI`[6]`.Core` отвечает за проведение операций над символьными термами (упрощение арифметики и пропозициональных формул, приведения типов и т.п.) и над символьными состояниями (чтение, запись, объединение, композиция состояний — в соответствии с вышеизложенной теорией).

Модуль `VSharp.SILI` реализует высокоуровневый алгоритм композиционального символьного исполнения: исполняет абстрактное синтаксическое дерево кода, пользуясь операциями из ядра.

Модуль `VSharp.Solver` отвечает за решение ограничений на условия пути. В частности, он реализует[7] кодирование уравнений на состояния и термов в дизъюнкты Хорна высших порядков.

Модуль `VSharp.Test` отвечает за тестирование инструмента. Тестирование проводится на наборе C# программ.

Модули `VSharp.Utils` и `VSharp.CSharpUtils` являются прикладными, и реализуют различные операции над неизменяемыми коллекциями.

[6] «SILI» расшифровывается как «symbolic intermediate language interpreter» (символьный интерпретатор промежуточного языка — имеется в виду MSIL).

[7] `https://github.com/Columpio/VSharp/tree/Encoding`

6. Апробация

Предложенный подход был реализован в проекте V# — символьной виртуальной машине .NET. В качестве решателя дизъюнктов высших порядков был выбран RType [17] (инструмент вывода ограниченных типов). На обработку каждого запроса был установлен лимит времени в 30 секунд.

В рамках апробации были сформулированы следующие вопросы.

Q1 Как ведёт себя предложенный подход на программах с нерекурсивными структурами данных?

Q2 Как ведёт себя предложенный подход на программах с рекурсивными структурами данных?

В табл. 1 приведены результаты экспериментов на наборе простых рекурсивных программ, меняющих нерекурсивные структуры данных и проверяющих математические свойства функций. В первом столбце указаны исходные имена верифицируемых функций, написанных на языке C#. Во втором столбце указаны номера логических запросов к решателю дизъюнктов высшего порядка. Исходный код содержит большое число ветвлений, поэтому многие тесты требуют проверки нескольких запросов. Таким образом, по результатам безопасности запросов можно судить о безопасности исходной программы на C#. В третьем столбце указан результат, порождённый сторонним решателем RType. В четвёртом столбце указаны верные ответы на запросы. В пятом столбце указано время решения запроса, в миллисекундах. *Q1*: подход оказывается эффективным на простых рекурсивных программах с нерекурсивными структурами данных.

В табл. 2 приведены результаты экспериментов на наборе рекурсивных программ с рекурсивными структурами данных. В наборе содержатся различные тесты на обход рекурсивных структур данных, в том числе с их изменением (например, в тесте «TestRemoveAllContains» из списка удаляются все вхождения числа и затем рекурсивно проверяется его отсутствие). Красными отмечены запросы, на которые был

Название теста	№	Результат V#+ RType	Верный результат	Время, мс
JustCallTestSafe	1	Безопасно	Безопасно	1014
TestMutateRecursiveSafe	1	Безопасно	Безопасно	4385
TestGcdSameSafe	1	Безопасно	Безопасно	11965
TestFactorialIsGreaterSafe	1	Безопасно	Безопасно	1078
TestFieldsEqualRecursiveSafe	1	Безопасно	Безопасно	1009
TestFieldIsGreaterSafe	1	Безопасно	Безопасно	1013
TestEqualFieldsSafe	1	Безопасно	Безопасно	990
TestGcdIsLessSafe	1	Небезопасно	Небезопасно	1116
TestGcdIsLessSafe	2	Небезопасно	Небезопасно	931
TestGcdIsLessSafe	3	Безопасно	Безопасно	935
TestGcdEqualSafe	1	Безопасно	Безопасно	1006
CheckMc91Safe	1	Безопасно	Безопасно	982
JustCallTestUnsafe	1	Небезопасно	Небезопасно	1140
JustCallTestUnsafe	2	Безопасно	Безопасно	1053
CheckMc91Unsafe	1	Небезопасно	Небезопасно	2128
CheckMc91Unsafe	2	Небезопасно	Небезопасно	1099
TestGcdSameUnsafe	1	Небезопасно	Небезопасно	6432
TestGcdSameUnsafe	2	Безопасно	Безопасно	1226
TestFactorialIsGreaterUnsafe	1	Небезопасно	Небезопасно	1200
TestFactorialIsGreaterUnsafe	2	Небезопасно	Небезопасно	1026
TestGcdIsLessUnsafe	1	Небезопасно	Небезопасно	939
TestGcdIsLessUnsafe	2	Небезопасно	Небезопасно	1316
TestGcdIsLessUnsafe	3	Небезопасно	Небезопасно	2307
TestGcdIsLessUnsafe	4	Безопасно	Безопасно	945

Таблица 1: Результаты экспериментов: программы с нерекурсивными структурами данных

получен неверный ответ. Их наличие связано с тем, что подход, лежащий в основе инструмента RType, корректен, но неполон, что иногда приводит к появлению ложных контрпримеров (ответов «Небезопасно» для безопасных запросов). *Q2*: тем не менее, подход показывает успешные результаты на многих нетривиальных рекурсивных программах с рекурсивными структурами данных.

Название теста	№	Результат V#+ RType	Верный результат	Время, мс
TestDecreasingLast	1	Безопасно	Безопасно	904
TestDecreasingLast	2	Безопасно	Безопасно	897
TestMax	1	Безопасно	Безопасно	888
TestMax	2	Безопасно	Безопасно	896
TestContainsLast	1	Безопасно	Безопасно	1173
LengthZero	1	Небезопасно	Небезопасно	904
LengthZero	2	Небезопасно	Небезопасно	899
LengthZero	3	Безопасно	Безопасно	918
TestLastReverse	1	Безопасно	Безопасно	1038
LastNodeOfReverse	1	Безопасно	Безопасно	1007
LastTest	1	Небезопасно	Небезопасно	926
LastTest	2	Небезопасно	Небезопасно	897
LastTest	3	Безопасно	Безопасно	921
TestItemContains	1	Небезопасно	Небезопасно	915
TestItemContains	2	Небезопасно	Небезопасно	991
TestItemContains	3	Безопасно	Безопасно	1113
LastNodeAndReverse	1	Безопасно	Безопасно	911
LastNodeAndReverse	2	Безопасно	Безопасно	891
LastNodeAndReverse	3	Небезопасно	Безопасно	1024
TestReverseNull	1	Безопасно	Безопасно	908
ContainsDecreasingOne	1	Безопасно	Безопасно	978
TestRemoveOneLength	1	Небезопасно	Небезопасно	1151
TestRemoveOneLength	2	Небезопасно	Небезопасно	1252
TestRemoveOneLength	3	Небезопасно	Безопасно	1216
TestLengthOfReverse	1	Безопасно	Безопасно	1406
TestDoubleCrop	1	Небезопасно	Безопасно	4496
TestCreate	1	Безопасно	Безопасно	889
TestCreate	2	Безопасно	Безопасно	886
TestCreate	3	Безопасно	Безопасно	916
TestRemoveAllContains	1	Небезопасно	Небезопасно	887
TestRemoveAllContains	2	Небезопасно	Небезопасно	884
TestRemoveAllContains	3	Безопасно	Безопасно	1089
LengthPositive	1	Безопасно	Безопасно	899
TestDecreasingIsDecreasing	1	Безопасно	Безопасно	965
ContainsDecreasing	1	Безопасно	Безопасно	989
TestReverseOfReverse	1	Безопасно	Безопасно	1030
TestDecreasingIncIsDecreasing	1	Безопасно	Безопасно	886
TestDecreasingIncIsDecreasing	2	Безопасно	Безопасно	4258
TestContainsReverse	1	Безопасно	Безопасно	899
TestContainsReverse	2	Безопасно	Безопасно	888
TestDecreasingConsIsDecreasing	1	Безопасно	Безопасно	894
TestDecreasingConsIsDecreasing	2	Безопасно	Безопасно	964

Таблица 2: Результаты экспериментов: программы с рекурсивными структурами данных

Заключение

В рамках данной работы были получены следующие результаты.

- Предложена *композициональная модель символьной памяти*, которая в сочетании с алгоритмом композиционального символьного исполнения формирует подход к верификации программ с динамической памятью.

- Доказана корректность предложенной модели памяти, при этом доказательства частично механизированы в интерактивной системе проверки доказательств Coq.

- Подход реализован в проекте V#, в том числе реализовано кодирование уравнений на состояния в дизъюнкты Хорна высших порядков.

- Проведена апробация подхода на наборе нетривиальных рекурсивных программ с изменяемыми рекурсивными структурами данных.

Список литературы

[1] Albarghouthi Aws, Gurfinkel Arie, Chechik Marsha. From under-approximations to over-approximations and back // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2012. — P. 157–172.

[2] Anand Saswat, Godefroid Patrice, Tillmann Nikolai. Demand-driven compositional symbolic execution // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 367–381.

[3] Automatic abstraction in SMT-based unbounded software model checking / Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, Edmund M Clarke // International Conference on Computer Aided Verification / Springer. — 2013. — P. 846–862.

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

[5] Bjørner Nikolaj, McMillan Kenneth L, Rybalchenko Andrey. Program Verification as Satisfiability Modulo Theories. // SMT@ IJCAR. — 2012. — Vol. 20. — P. 3–11.

[6] Calcagno Cristiano, Distefano Dino. Infer: An automatic program verifier for memory safety of C programs // NASA Formal Methods Symposium / Springer. — 2011. — P. 459–465.

[7] Champion Adrien, Kobayashi Naoki, Sato Ryosuke. HoIce: An ICE-Based Non-linear Horn Clause Solver // Asian Symposium on Programming Languages and Systems / Springer. — 2018. — P. 146–156.

[8] Compositional shape analysis by means of bi-abduction / Cristiano Calcagno, Dino Distefano, Peter W O'hearn, Hongseok Yang // Journal of the ACM (JACM). — 2011. — Vol. 58, no. 6. — P. 26.

[9] 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.

[10] Distefano Dino. Attacking large industrial code with bi-abductive inference // International Workshop on Formal Methods for Industrial Critical Systems / Springer. — 2009. — P. 1–8.

[11] Distefano Dino, Parkinson J Matthew J. jStar: Towards practical verification for Java // ACM Sigplan Notices / ACM. — Vol. 43. — 2008. — P. 213–226.

[12] Dudka Kamil, Peringer Petr, Vojnar Tomáš. Byte-precise verification of low-level list manipulation // International Static Analysis Symposium / Springer. — 2013. — P. 215–237.

[13] Essertel Grégory, Wei Guannan, Rompf Tiark. Precise Reasoning with Structured Heaps and Collective Operations à la Map/Reduce. — 2018.

[14] Hashimoto Kodai, Unno Hiroshi. Refinement type inference via horn constraint optimization // International Static Analysis Symposium / Springer. — 2015. — P. 199–216.

[15] Hojjat Hossein, Rümmer Philipp. The ELDARICA Horn Solver // 2018 Formal Methods in Computer Aided Design (FMCAD) / IEEE. — 2018. — P. 1–7.

[16] Horn clause solvers for program verification / Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, Andrey Rybalchenko // Fields of Logic and Computation II. — Springer, 2015. — P. 24–51.

[17] ICE-based refinement type discovery for higher-order functional programs / Adrien Champion, Tomoya Chiba, Naoki Kobayashi, Ryosuke Sato // International Conference on Tools and Algorithms

for the Construction and Analysis of Systems / Springer. — 2018. — P. 365–384.

[18] Kobayashi Naoki, Sato Ryosuke, Unno Hiroshi. Predicate abstraction and CEGAR for higher-order model checking // ACM SIGPLAN Notices / ACM. — Vol. 46. — 2011. — P. 222–233.

[19] O'Hearn Peter, Reynolds John, Yang Hongseok. Local reasoning about programs that alter data structures // International Workshop on Computer Science Logic / Springer. — 2001. — P. 1–19.

[20] Predator shape analysis tool suite / Lukáš Holík, Michal Kotoun, Petr Peringer et al. // Haifa Verification Conference / Springer. — 2016. — P. 202–209.

[21] Reynolds John C. Separation logic: A logic for shared mutable data structures // Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on / IEEE. — 2002. — P. 55–74.

[22] Rümmer Philipp, Hojjat Hossein, Kuncak Viktor. Classifying and solving horn clauses for verification // Working Conference on Verified Software: Theories, Tools, and Experiments / Springer. — 2013. — P. 1–21.

[23] Sato Ryosuke, Iwayama Naoki, Kobayashi Naoki. Combining higher-order model checking with refinement type inference // Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation / ACM. — 2019. — P. 47–53.

[24] The SeaHorn verification framework / Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, Jorge A Navas // International Conference on Computer Aided Verification / Springer. — 2015. — P. 343–361.

[25] Synthesizing software verifiers from proof rules / Sergey Grebenshchikov, Nuno P Lopes, Corneliu Popeea, Andrey Rybalchenko // ACM SIGPLAN Notices / ACM. — Vol. 47. — 2012. — P. 405–416.

[26] Tillmann Nikolai, De Halleux Jonathan. Pex–white box test generation for. net // International conference on tests and proofs / Springer. — 2008. — P. 134–153.

[27] Translating imperative code to MapReduce / Cosmin Radoi, Stephen J Fink, Rodric Rabbah, Manu Sridharan // ACM SIGPLAN Notices / ACM. — Vol. 49. — 2014. — P. 909–927.

[28] Unno Hiroshi, Torii Sho, Sakamoto Hiroki. Automating induction for solving horn clauses // International Conference on Computer Aided Verification / Springer. — 2017. — P. 571–591.

[29] Zhu He, Petri Gustavo, Jagannathan Suresh. Automatically learning shape specifications // ACM SIGPLAN Notices / ACM. — Vol. 51. — 2016. — P. 491–507.

A. Приложение

Теорема 1. Если σ и σ' — символьные кучи, то $\sigma \circ \sigma'$ также символьная куча.

Доказательство. Необходимо показать, что $\sigma \circ \sigma'$ удовлетворяет инварианту кучи (1). Рассмотрим $x, y \in dom(\sigma \circ \sigma')$.

$$
\begin{aligned}
&union\Big\langle x = y, (\sigma \circ \sigma')(x)\Big\rangle \overset{\text{Опр.5 и Св.1.(c)}}{=} \\
&= union\Big(\big\{\big\langle x = y \wedge x = \sigma \bullet l, \sigma \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \cup \\
&\qquad \cup \Big\langle x = y \wedge \bigwedge_{l \in dom(\sigma')} x \neq \sigma \bullet l, \sigma(x)\Big\rangle\Big) \overset{\text{Св.1.(c)}}{=} \\
&= union\Big(\big\{\big\langle x = y \wedge x = \sigma \bullet l, \sigma \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \cup \\
&\qquad \cup \Big\langle x = y \wedge \bigwedge_{l \in dom(\sigma')} x \neq \sigma \bullet l, union\big(\langle x = y, \sigma(x)\rangle\big)\Big\rangle\Big) \overset{(1)\text{ для }\sigma}{=} \\
&= union\Big(\big\{\big\langle x = y \wedge y = \sigma \bullet l, \sigma \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \cup \\
&\qquad \cup \Big\langle x = y \wedge \bigwedge_{l \in dom(\sigma')} y \neq \sigma \bullet l, union\big(\langle x = y, \sigma(y)\rangle\big)\Big\rangle\Big) = \\
&= union\Big\langle x = y, (\sigma \circ \sigma')(y)\Big\rangle
\end{aligned}
$$

□

Теорема 2. Для произвольных символьных кучи σ, локации x и выражения e справедливо следующее:

(a) $read(\epsilon, x) = LI(x)$

(b) $\epsilon \bullet e = LI(e)$

(c) $\epsilon \circ \sigma = \sigma$

(d) $\sigma \circ \epsilon = \sigma$

Доказательство. Напомним, что конъюнкция пустого множества формул есть тождественная истина.

(*a*) $read(\epsilon, x) \overset{\text{Опр. 2, 3}}{=} union\big(\varnothing \cup \langle \top, LI(x)\rangle\big) \overset{\text{Св.1.(a)}}{=} LI(x)$

(*b*) Это свойство может быть доказано структурной индукцией по выражению e по Определение 4 и (a).

(*c*) Во-первых, $dom(\epsilon \circ \sigma) \overset{(4)}{=} \varnothing \cup \epsilon \bullet dom(\sigma) \overset{(b)}{=} dom(\sigma)$. Рассмотрим $x \in dom(\sigma)$.

$$
\begin{aligned}
(\epsilon \circ \sigma)(x) &\overset{\text{Опр.5}}{=} union\Big(\big\{\big\langle x = \epsilon \bullet l, \epsilon \bullet (\sigma(l))\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad \cup \Big\langle \bigwedge_{l \in dom(\sigma)} x \neq \epsilon \bullet l, \epsilon(x)\Big\rangle\Big) \overset{(b)}{=} \\
&= union\big(\big\{\langle x = l, \sigma(l)\rangle \mid l \in dom(\sigma)\big\} \cup \langle \bot, \epsilon(x)\rangle\big) \\
&\overset{\text{Св.1.(b)}}{=} \sigma(x)
\end{aligned}
$$

(*d*) $dom(\sigma \circ \epsilon) \overset{(4)}{=} dom(\sigma) \cup \sigma \bullet \varnothing = dom(\sigma)$. Пусть $x \in dom(\sigma)$.

$$(\sigma \circ \epsilon)(x) \overset{\text{Опр.5}}{=} union\big(\varnothing \cup \langle \bigwedge_{l \in \varnothing} x \neq \sigma \bullet l, \sigma(x) \rangle\big) = \sigma(x)$$

□

Стоит отметить, что есть некоторое сходство между чтением (Определение 3) и композицией (Определение 5): объединения осуществляют поиск x среди локаций кучи. Если поиск был успешен, возвращается соответствующее (возможно изменённое) значение, в ином случае — значение по умолчанию. Воспользуемся этим сходством для введения следующего вспомогательного определения.

Определение 8.

$$find(\sigma, x, \tau, d) \overset{\text{def}}{=} union\Big(\big\{\big\langle x = \tau \bullet l, \tau \bullet (\sigma(l)) \big\rangle \mid l \in dom(\sigma)\big\} \cup \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq \tau \bullet l, d \big\rangle\Big) \tag{6}$$

Обобщённая структура $find$ позволит доказать некоторые полезные свойства композиции. Этот оператор станет краеугольным камнем в процессе кодирования уравнений над состояниями в ограниченные дизъюнкты Хорна. Третий аргумент $find$, τ, мы будем называть *контекстной кучей*. Свойства теоремы 2 позволяют компактно выразить $read$ и композицию куч через $find$:

$$read(\sigma, x) = find\big(\sigma, x, \epsilon, LI(x)\big) \qquad \text{и} \tag{7}$$

$$(\sigma \circ \sigma')(x) = find\big(\sigma', x, \sigma, \sigma(x)\big) \tag{8}$$

Докажем фундаментальное свойство $find$, которое будет полезно для доказательства важных свойств композиции и уточнения:

Лемма 5. Для произвольных символьных куч σ, σ', τ таких что для каждого символьного выражения e,

$$(\tau \circ \sigma) \bullet e = \tau \bullet (\sigma \bullet e), \tag{9}$$

и всех символьных выражений $x \in loc$, $d \in term$,

$$find\big(\sigma \circ \sigma', x, \tau, d\big) = find\big(\sigma', x, \tau \circ \sigma, find(\sigma, x, \tau, d)\big)$$

Доказательство.

$$
\begin{aligned}
&find\big(\sigma\circ\sigma',x,\tau,d\big)=\\
&=union\Big(\big\{\langle x=\tau\bullet l,\tau\bullet(\sigma\circ\sigma')(l)\rangle\mid l\in dom(\sigma\circ\sigma')\big\}\cup\\
&\qquad\cup\big\{\langle\bigwedge_{l\in dom(\sigma\circ\sigma')}x\neq\tau\bullet l,d\rangle\big\}\Big)=\\
&=union\Bigg(\Bigg\{\Big\langle x=\tau\bullet l,\tau\bullet union\Big(\big\{\langle l=\sigma\bullet l',\sigma\bullet(\sigma'(l'))\rangle\mid l'\in dom(\sigma')\big\}\cup\\
&\qquad\cup\big\{\langle\bigwedge_{l'\in dom(\sigma')}l\neq\sigma\bullet l',\sigma(l)\rangle\big\}\Big)\Big\rangle\Bigg|l\in dom(\sigma\circ\sigma')\Bigg\}\cup\Big\{\langle\bigwedge_{l\in dom(\sigma\circ\sigma')}x\neq\tau\bullet l,d\rangle\Big\}\Bigg)\overset{\text{Св.1.(c)}}{=}\\
&=union\Big(\big\{\big\langle x=\tau\bullet l\wedge\tau\bullet l=\tau\bullet(\sigma\bullet l'),\tau\bullet\big(\sigma\bullet(\sigma'(l'))\big)\big\rangle\\
&\qquad\mid l'\in dom(\sigma'),l\in dom(\sigma\circ\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle x=\tau\bullet l\wedge\bigwedge_{l'\in dom(\sigma')}\tau\bullet l\neq\tau\bullet(\sigma\bullet l'),\tau\bullet(\sigma(l))\big\rangle\mid l\in dom(\sigma\circ\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle\bigwedge_{l\in dom(\sigma\circ\sigma')}x\neq\tau\bullet l,d\big\rangle\big\}\Big)\overset{(4)}{=}\\
&=union\Big(\big\{\big\langle x=\tau\bullet l\wedge\tau\bullet l=\tau\bullet(\sigma\bullet l'),\tau\bullet\big(\sigma\bullet(\sigma'(l'))\big)\big\rangle\\
&\qquad\mid l'\in dom(\sigma'),l\in dom(\sigma)\cup\sigma\bullet dom(\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle x=\tau\bullet l\wedge\bigwedge_{l'\in dom(\sigma')}\tau\bullet l\neq\tau\bullet(\sigma\bullet l'),\tau\bullet(\sigma(l))\big\rangle\\
&\qquad\mid l\in dom(\sigma)\cup\sigma\bullet dom(\sigma')\big\}\cup\big\{\big\langle\bigwedge_{l\in dom(\sigma)\cup\sigma\bullet dom(\sigma')}x\neq\tau\bullet l,d\big\rangle\big\}\Big)=\\
&=union\Big(\big\{\big\langle x=\tau\bullet l\wedge\tau\bullet l=\tau\bullet(\sigma\bullet l'),\tau\bullet\big(\sigma\bullet(\sigma'(l'))\big)\big\rangle\\
&\qquad\mid l'\in dom(\sigma'),l\in\sigma\bullet dom(\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle x=\tau\bullet l\wedge\bigwedge_{l'\in dom(\sigma')}\tau\bullet l\neq\tau\bullet(\sigma\bullet l'),\tau\bullet(\sigma(l))\big\rangle\\
&\qquad\mid l\in dom(\sigma)\big\}\cup\big\{\big\langle\bigwedge_{l\in dom(\sigma)\cup\sigma\bullet dom(\sigma')}x\neq\tau\bullet l,d\big\rangle\big\}\Big)=\\
&=union\Big(\big\{\big\langle x=\tau\bullet(\sigma\bullet l'),\tau\bullet\big(\sigma\bullet(\sigma'(l'))\big)\big\rangle\mid l'\in dom(\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle x=\tau\bullet l\wedge\bigwedge_{l'\in dom(\sigma')}x\neq\tau\bullet(\sigma\bullet l'),\tau\bullet(\sigma(l))\big\rangle\mid l\in dom(\sigma)\big\}\\
&\qquad\cup\big\langle\bigwedge_{l'\in dom(\sigma')}x\neq\tau\bullet(\sigma\bullet l')\wedge\bigwedge_{l\in dom(\sigma)}x\neq\tau\bullet l,d\big\rangle\Big)\overset{(9)}{=}\\
&=union\Big(\big\{\big\langle x=(\tau\circ\sigma)\bullet l',(\tau\circ\sigma)\bullet(\sigma'(l'))\big\rangle\mid l'\in dom(\sigma')\big\}\cup\\
&\qquad\cup\big\{\big\langle x=\tau\bullet l\wedge\bigwedge_{l'\in dom(\sigma')}x\neq(\tau\circ\sigma)\bullet l',\tau\bullet(\sigma(l))\big\rangle\mid l\in dom(\sigma)\big\}\\
&\qquad\cup\big\langle\bigwedge_{l'\in dom(\sigma')}x\neq(\tau\circ\sigma)\bullet l'\wedge\bigwedge_{l\in dom(\sigma)}x\neq\tau\bullet l,d\big\rangle\Big)\overset{\text{Св.1.(c)}}{=}\\
&=union\Big(\big\{\big\langle x=(\tau\circ\sigma)\bullet l',(\tau\circ\sigma)\bullet(\sigma'(l'))\big\rangle\mid l'\in dom(\sigma')\big\}\cup\\
&\qquad\cup\big\langle\bigwedge_{l'\in dom(\sigma')}x\neq(\tau\circ\sigma)\bullet l',find(\sigma,x,\tau,d)\big\rangle\Big)=\\
&=find\big(\sigma',x,\tau\circ\sigma,find(\sigma,x,\tau,d)\big)
\end{aligned}
$$

□

Лемма 6. Для всех символьных куч σ, σ', τ, удовлетворяющих (9), и выражений $x \in loc$, $d \in term$,

$$\tau \bullet find(\sigma', x, \sigma, d) = find(\sigma', \tau \bullet x, \tau \circ \sigma, \tau \bullet d)$$

Доказательство.

$$\begin{aligned}
&\tau \bullet find(\sigma', x, \sigma, d) = \\
&\tau \bullet union\Big(\big\{\big\langle x = \sigma \bullet l, \sigma \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \cup \\
&\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma')} x \neq \sigma \bullet l, d\big\rangle\Big) \overset{\text{Опр.4}}{=} \\
&= union\Big(\big\{\big\langle \tau \bullet x = \tau \bullet (\sigma \bullet l), \tau \bullet \big(\sigma \bullet (\sigma'(l))\big)\big\rangle \mid l \in dom(\sigma')\big\} \\
&\qquad \cup \big\{\big\langle \bigwedge_{l \in dom(\sigma')} \tau \bullet x \neq \tau \bullet (\sigma \bullet l), \tau \bullet d\big\rangle\big\}\Big) \overset{(9)}{=} \\
&= union\Big(\big\{\big\langle \tau \bullet x = (\tau \circ \sigma) \bullet l, (\tau \circ \sigma) \bullet (\sigma'(l))\big\rangle \mid l \in dom(\sigma')\big\} \\
&\qquad \cup \big\{\big\langle \bigwedge_{l \in dom(\sigma')} \tau \bullet x \neq (\tau \circ \sigma) \bullet l, \tau \bullet d\big\rangle\big\}\Big) = \\
&= find(\sigma', \tau \bullet x, \tau \circ \sigma, \tau \bullet d)
\end{aligned}$$

□

Теорема 3. Для всех символьных куч σ, σ' и символьных локаций x справедливо следующее:

$$\sigma \bullet read(\sigma', x) = read(\sigma \circ \sigma', \sigma \bullet x).$$

Доказательство. Заметим, что по свойствам Теорема 2, (9), выполняющимся для $\sigma = \epsilon$ и $\tau = \epsilon$:

$$\begin{aligned}
&\sigma \bullet read(\sigma', x) \overset{(7)}{=} \sigma \bullet find(\sigma', x, \epsilon, LI(x)) \overset{\text{Лемма 6}}{=} find(\sigma', \sigma \bullet x, \sigma \circ \epsilon, \sigma \bullet LI(x)) \overset{\text{Теор.2, Опр.4}}{=} \\
&= find(\sigma', \sigma \bullet x, \sigma, read(\sigma, \sigma \bullet x)) \overset{(7)}{=} find\big(\sigma', \sigma \bullet x, \sigma, find(\sigma, \sigma \bullet x, \epsilon, LI(\sigma \bullet x))\big) \overset{\text{Лемма 5}}{=} \\
&= find(\sigma \circ \sigma', \sigma \bullet x, \epsilon, LI(\sigma \bullet l)) \overset{(7)}{=} read(\sigma \circ \sigma', \sigma \bullet x)
\end{aligned}$$

□

Лемма 1. Для всех символьных куч σ, σ' и символьных выражений e справедливо следующее:

$$(\sigma \circ \sigma') \bullet e = \sigma \bullet (\sigma' \bullet e).$$

Доказательство. Доказательство структурной индукцией по Определение 4. Случаи 1—3 тривиальны. Единственное, что необходимо показать, это $(\sigma \circ \sigma') \bullet LI(x) = \sigma \bullet (\sigma' \bullet LI(x))$. Индукционная гипотеза выглядит так:

$$(\sigma \circ \sigma') \bullet x = \sigma \bullet (\sigma' \bullet x)$$

Теперь

$$\sigma \bullet (\sigma' \bullet LI(x)) \overset{\text{Опр.4}}{=} \sigma \bullet read(\sigma', \sigma' \bullet x) \overset{\text{Теор.3}}{=} read\big(\sigma \circ \sigma', \sigma \bullet (\sigma' \bullet x)\big) \overset{\text{I.H.}}{=}$$

$$= read\big(\sigma \circ \sigma', (\sigma \circ \sigma') \bullet x\big) \overset{\text{Опр.4}}{=} (\sigma \circ \sigma') \bullet LI(x)$$

□

Лемма 7. Для всех символьных куч σ, σ' и локаций x,

$$read(\sigma \circ \sigma', x) = find\big(\sigma', x, \sigma, read(\sigma, x)\big)$$

Доказательство.

$$read(\sigma \circ \sigma', x) = find\big(\sigma \circ \sigma', x, \epsilon, LI(x)\big) \overset{\text{Лем.5}}{=} find\big(\sigma', x, \epsilon \circ \sigma, find(\sigma, x, \epsilon, LI(x))\big) =$$
$$= find\big(\sigma', x, \sigma, read(\sigma, x)\big)$$

□

Теорема 4. Для всех символьных куч σ_1, σ_2 и σ_3 справедливо следующее:

$$(\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3).$$

Доказательство. Для начала покажем равенство их областей действия.

$$\begin{aligned}
&dom\big((\sigma_1 \circ \sigma_2) \circ \sigma_3\big) \overset{(4)}{=} dom(\sigma_1 \circ \sigma_2) \cup \big\{(\sigma_1 \circ \sigma_2) \bullet l_3 \mid l_3 \in dom(\sigma_3)\big\} \overset{(4)}{=} \\
&= dom(\sigma_1) \cup \big\{\sigma_1 \bullet l_2 \mid l_2 \in dom(\sigma_2)\big\} \cup \big\{(\sigma_1 \circ \sigma_2) \bullet l_3 \mid l_3 \in dom(\sigma_3)\big\} \overset{\text{Лемма 1}}{=} \\
&= dom(\sigma_1) \cup \big\{\sigma_1 \bullet l_2 \mid l_2 \in dom(\sigma_2)\big\} \cup \big\{\sigma_1 \bullet (\sigma_2 \bullet l_3) \mid l_3 \in dom(\sigma_3)\big\} = \\
&= dom(\sigma_1) \cup \big\{\sigma_1 \bullet l_2 \mid l_2 \in dom(\sigma_2)\big\} \cup \big\{\sigma_1 \bullet l_3' \mid l_3' \in \sigma_2 \bullet dom(\sigma_3)\big\} = \\
&= dom(\sigma_1) \cup \big\{\sigma_1 \bullet l_{23} \mid l_{23} \in dom(\sigma_2) \cup \sigma_2 \bullet dom(\sigma_3)\big\} = \\
&= dom(\sigma_1) \cup \big\{\sigma_1 \bullet l_{23} \mid l_{23} \in dom(\sigma_2 \circ \sigma_3)\big\} \overset{(4)}{=} dom\big(\sigma_1 \circ (\sigma_2 \circ \sigma_3)\big)
\end{aligned}$$

Рассмотрим произвольное $x \in dom\big((\sigma_1 \circ \sigma_2) \circ \sigma_3\big)$,

$$\begin{aligned}
&\big((\sigma_1 \circ \sigma_2) \circ \sigma_3\big)(x) = find\big(\sigma_3, x, \sigma_1 \circ \sigma_2, find\big(\sigma_2, x, \sigma_1, \sigma_1(x)\big)\big) \overset{\text{Лемма 5}}{=} \\
&= find\big(\sigma_2 \circ \sigma_3, x, \sigma_1, \sigma_1(x)\big) = (\sigma_1 \circ \big(\sigma_2 \circ \sigma_3)\big)\big(x\big)
\end{aligned}$$

□

Для упрощения внешнего вида доказательств, мы сокращаем нотацию $merge(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle)$ и пишем просто $merge\langle g_i, \sigma_i\rangle$; аналогично вместо $union\big(\langle g_1, v_1\rangle, \ldots, \langle g_n, v_n\rangle\big)$ далее будем писать $union\langle g_i, v_i\rangle$.

Лемма 2. Для любой символьной кучи σ и символьных локаций x, y справедливо следующее:

$$union\big(\big\langle x = y, read(\sigma, x)\big\rangle\big) = union\big(\big\langle x = y, read(\sigma, y)\big\rangle\big).$$

Доказательство.

$$\begin{aligned}
&union\big\langle x = y, read(\sigma, x)\big\rangle = \\
&= union\Big(\big\langle x = y, union\Big(\big\{\big\langle x = l, \sigma(l)\big\rangle \mid l \in dom(\sigma)\big\} \\
&\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq l, LI(x)\big\rangle\Big)\big\rangle\Big) =
\end{aligned}$$

$$= union\Big(\big\{\big\langle x = y \wedge x = l, \sigma(l)\big\rangle \mid l \in dom(\sigma)\big\} \cup \big\langle x = y \wedge \bigwedge_{l \in dom(\sigma)} x \neq l, LI(x)\big\rangle\Big) =$$
$$= union\Big(\big\{\big\langle x = y \wedge y = l, \sigma(l)\big\rangle \mid l \in dom(\sigma)\big\} \cup \big\langle x = y \wedge \bigwedge_{l \in dom(\sigma)} y \neq l, LI(y)\big\rangle\Big) = union\big\langle x = y, read(\sigma, y)\big\rangle$$

□

Теорема 6. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$ и непересекающихся ограничений $g_1, \ldots, g_n$, $merge\langle g_i, \sigma_i\rangle$ — символьная куча.

Доказательство. Возьмём $x, y \in dom(merge\langle g_i, \sigma_i\rangle) = \bigcup_{i=1}^{n} dom(\sigma_i)$.

$$union\Big\langle x = y, merge\langle g_i, \sigma_i\rangle(x)\Big\rangle = union\Big\langle x = y, union\big\langle g_i, read(\sigma_i, x)\big\rangle\Big\rangle =$$
$$= union\Big\langle x = y \wedge g_i, read(\sigma_i, x)\Big\rangle \overset{\text{Лемма 2}}{=} union\Big\langle x = y \wedge g_i, read(\sigma_i, y)\Big\rangle =$$
$$= union\Big\langle x = y, merge\langle g_i, \sigma_i\rangle(y)\Big\rangle$$

□

Лемма 8. Для всех символьных куч $\sigma_1, \ldots, \sigma_n, \tau$, непересекающихся ограничений $g_1, \ldots, g_n$, символьных локаций x и символьных выражений d,

$$find\Big(merge\big\langle g_i, \sigma_i\big\rangle, x, \tau, read(\tau, x)\Big) = union\Big\langle \tau \bullet g_i, find(\sigma_i, x, \tau, read(\tau, x))\Big\rangle$$

Доказательство. Сначала заметим, что для любого предиката p и непересекающихся множеств A, B верно

$$\bigvee_{u \in A \coprod B} \Big(p(u) \wedge \bigwedge_{v \in B} \neg p(v)\Big) \vee \bigwedge_{u \in A \coprod B} \neg p(u) \iff \bigwedge_{u \in B} \neg p(v) \tag{10}$$

Рассмотрим произвольную символьную кучу σ и произвольное множество символьных локаций A. Можно допустить, что A и $dom(\sigma)$ не пересекаются, иначе возьмём $A := A \setminus dom(\sigma)$. Теперь

$$union\Big(\big\{\big\langle x = \tau \bullet l, \tau \bullet read(\sigma, l)\big\rangle \mid l \in dom(\sigma) \cup A\big\} \cup \big\langle \bigwedge_{l \in dom(\sigma) \cup A} x \neq \tau \bullet l, read(\tau, x)\big\rangle\Big) =$$
$$= union\Bigg(\Bigg\{\Big\langle x = \tau \bullet l, union\Big(\big\{\big\langle \tau \bullet l = \tau \bullet l', \tau \bullet (\sigma(l'))\big\rangle \mid l' \in dom(\sigma)\big\} \cup \big\langle \bigwedge_{l' \in dom(\sigma)} \tau \bullet l \neq \tau \bullet l', \tau \bullet LI(l)\big\rangle\Big)\Big\rangle \Big| l \in dom(\sigma) \cup A\Bigg\} \cup \Big\langle \bigwedge_{l \in dom(\sigma) \cup A} x \neq \tau \bullet l, read(\tau, x)\Big\rangle\Bigg) =$$
$$= union\Big(\big\{\big\langle x = \tau \bullet l \wedge \tau \bullet l = \tau \bullet l', \tau \bullet (\sigma(l'))\big\rangle \mid l \in dom(\sigma) \cup A, l' \in dom(\sigma)\big\} \cup \big\{\big\langle x = \tau \bullet l \wedge \bigwedge_{l' \in dom(\sigma)} \tau \bullet l \neq \tau \bullet l', read(\tau, \tau \bullet l)\big\rangle$$

$$
\begin{aligned}
&\quad \mid l \in dom(\sigma) \cup A\big\} \cup \Big\langle \bigwedge_{l \in dom(\sigma) \cup A} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \Big) \overset{\text{Lm.2}}{=} \\
&= union\Big(\big\{ \big\langle x = \tau \bullet l \wedge x = \tau \bullet l', \tau \bullet (\sigma(l')) \big\rangle \mid l \in dom(\sigma) \cup A, l' \in dom(\sigma) \big\} \cup \\
&\qquad \cup \big\{ \big\langle x = \tau \bullet l \wedge \bigwedge_{l' \in dom(\sigma)} x \neq \tau \bullet l', read(\tau, x) \big\rangle \mid l \in dom(\sigma) \cup A \big\} \cup \\
&\qquad \cup \Big\langle \bigwedge_{l \in dom(\sigma) \cup A} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \Big) \overset{\text{Св.1.(d)}}{=} \\
&= union\Big(\big\{ \big\langle x = \tau \bullet l', \tau \bullet (\sigma(l')) \big\rangle \mid l' \in dom(\sigma) \big\} \cup \\
&\qquad \cup \Big\langle \bigvee_{l \in dom(\sigma) \cup A} \Big(x = \tau \bullet l \wedge \bigwedge_{l' \in dom(\sigma)} x \neq \tau \bullet l' \Big) \vee \bigwedge_{l \in dom(\sigma) \cup A} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \Big) \overset{(10)}{=} \\
&= union\Big(\big\{ \big\langle x = \tau \bullet l, \tau \bullet (\sigma(l)) \big\rangle \mid l \in dom(\sigma) \big\} \cup \Big\langle \bigwedge_{l \in dom(\sigma)} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \Big) = \\
&= find\big(\sigma, x, \tau, read(\tau, x)\big) \qquad (11)
\end{aligned}
$$

Наконец,

$$
\begin{aligned}
&find\Big(merge\big\langle g_i, \sigma_i \big\rangle, x, \tau, read(\tau, x)\Big) = \\
&= union\left(\left\{ \Big\langle x = \tau \bullet l, \tau \bullet union\big\langle g_i, read(\sigma_i, l) \big\rangle \Big\rangle \Big| l \in \bigcup_{j=1}^{n} dom(\sigma_j) \right\} \cup \right. \\
&\qquad \left. \cup \Big\langle \bigwedge_{l \in \bigcup_{j=1}^{n} dom(\sigma_j)} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \right) = \\
&= union\Big\langle x = \tau \bullet g_i, union\Big(\Big\{ \big\langle x = \tau \bullet l, \tau \bullet read(\sigma_i, l) \big\rangle \mid l \in \bigcup_{j=1}^{n} dom(\sigma_j) \Big\} \cup \\
&\qquad \cup \Big\langle \bigwedge_{l \in \bigcup_{j=1}^{n} dom(\sigma_j)} x \neq \tau \bullet l, read(\tau, x) \Big\rangle \Big) \Big\rangle \overset{(11)}{=} \\
&= union\Big\langle \tau \bullet g_i, find(\sigma_i, x, \tau, read(\tau, x)) \Big\rangle
\end{aligned}
$$

□

Теорема 7. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$ и локаций x справедливо следующее:

$$
read\Big(merge\big\langle g_i, \sigma_i \big\rangle, x\Big) = union\Big\langle g_i, read(\sigma_i, x) \Big\rangle.
$$

Доказательство.

$$
\begin{aligned}
&read\Big(merge\big\langle g_i, \sigma_i \big\rangle, x\Big) = find\Big(merge\big\langle g_i, \sigma_i \big\rangle, x, \epsilon, read(\epsilon, x)\Big) = \\
&\overset{\text{Лемма 8}}{=} union\Big\langle \epsilon \bullet g_i, find(\sigma_i, x, \epsilon, read(\epsilon, x)) \Big\rangle = union\Big\langle g_i, read(\sigma_i, x) \Big\rangle
\end{aligned}
$$

□

Теорема 8. Для любых символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$ и непересекающихся ограничений

$g_1, \ldots, g_n$ выполняется следующее утверждение:

$$\sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big) = merge\big(\langle \sigma \bullet g_1, \sigma \circ \sigma_1\rangle, \ldots, \langle \sigma \bullet g_n, \sigma \circ \sigma_n\rangle\big).$$

Доказательство.

$$\begin{aligned}
&dom\big(\sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)\big) = \\
&= dom\big(\sigma\big) \cup \sigma \bullet dom\big(merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)\big) = \\
&= dom\big(\sigma\big) \cup \sigma \bullet \bigcup_{i=0}^{n} dom(\sigma_i) = \bigcup_{i=0}^{n} \Big(dom(\sigma) \cup \sigma \bullet dom(\sigma_i)\Big) = \\
&= \bigcup_{i=0}^{n} dom(\sigma \circ \sigma_i) = dom\big(merge\big(\langle \sigma \bullet g_1, \sigma \circ \sigma_1\rangle, \ldots, \langle \sigma \bullet g_n, \sigma \circ \sigma_n\rangle\big)\big)
\end{aligned}$$

Возьмём $x \in \sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)$. Напомним, что для $x \in dom(\sigma), \sigma(x) = read(\sigma, x)$.

$$\begin{aligned}
&\Big(\sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big)\Big)(x) = \\
&= read\big(\sigma \circ merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big), x\big) \overset{\text{Лемма 7}}{=} \\
&= find\Big(merge\big(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle\big), x, \sigma, read(\sigma, x)\Big) \overset{\text{Лемма 8}}{=} \\
&= union\Big(\big\langle \sigma \bullet g_1, find(\sigma_1, x, \sigma, read(\sigma, x))\big\rangle, \ldots, \big\langle \sigma \bullet g_n, find(\sigma_n, x, \sigma, read(\sigma, x))\big\rangle\Big) \overset{\text{Лемма 7}}{=} \\
&= union\Big(\big\langle \sigma \bullet g_1, read(\sigma \circ \sigma_1, x)\big\rangle, \ldots, \big\langle \sigma \bullet g_n, read(\sigma \circ \sigma_n, x)\big\rangle\Big) \overset{\text{Лемма 7}}{=} \\
&= \Big(merge\big(\langle \sigma \bullet g_1, \sigma \circ \sigma_1\rangle, \ldots, \langle \sigma \bullet g_n, \sigma \circ \sigma_n\rangle\big)\Big)(x)
\end{aligned}$$

□

Лемма 3. Для всех символьных куч σ, непересекающихся ограничений $g_1, \ldots, g_n$ и локаций $x_1, \ldots, x_n$ справедливо следующее:

$$read\big(\sigma, union\langle g_i, x_i\rangle\big) = union\big\langle g_i, read(\sigma, x_i)\big\rangle.$$

Доказательство.

$$\begin{aligned}
&read\big(\sigma, union\langle g_i, x_i\rangle\big) = \\
&= union\Big(\big\{\big\langle union\langle g_i, x_i\rangle = l, \sigma(l)\big\rangle \mid l \in dom(\sigma)\big\} \\
&\qquad\qquad \cup \langle \bigwedge_{l \in dom(\sigma)} union\langle g_i, x_i\rangle \neq l, LI(union\langle g_i, x_i\rangle)\rangle\Big) \overset{\text{Св.1, (5)}}{=} \\
&= union\Big\langle g_i, union\Big(\big\{\big\langle x_i = l, \sigma(l)\big\rangle \mid l \in dom(\sigma)\big\} \\
&\qquad\qquad \cup \langle \bigwedge_{l \in dom(\sigma)} x_i \neq l, LI(x_i)\rangle\Big)\Big\rangle = union\Big\langle g_i, read(\sigma, x_i)\Big\rangle
\end{aligned}$$

□

Лемма 4. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$

и выражений e справедливо следующее:

$$union\Big\langle g_1 \vee \ldots \vee g_n, merge\langle g_i, \sigma_i\rangle \bullet e\Big\rangle = union\Big\langle g_i, \sigma_i \bullet e\Big\rangle.$$

Доказательство. Доказательство структурной индукцией по Определение 4. Возьмём $G \stackrel{\text{def}}{=} g_1 \vee \ldots \vee g_n$ и $M \stackrel{\text{def}}{=} merge\langle g_i, \sigma_i\rangle$.

(a) $union\big\langle G, M \bullet leaf\big\rangle = union\big\langle G, leaf\big\rangle \stackrel{\text{Св.1.(c)}}{=} union\big\langle g_i, leaf\big\rangle = union\big\langle g_i, \sigma_i \bullet leaf\big\rangle$

(b) $union\Big\langle G, M \bullet op(e_1, \ldots, e_m)\Big\rangle = union\Big\langle G, op(M \bullet e_1, \ldots, M \bullet e_m)\Big\rangle \stackrel{\text{I.H.}}{=}$
$= union\Big\langle G, op\Big(union\big\langle g_i, \sigma_i \bullet e_1\big\rangle, \ldots, union\big\langle g_i, \sigma_i \bullet e_m\big\rangle\Big)\Big\rangle \stackrel{\text{Св.1.(c),(e)}}{=}$
$= op\Big(union\big\langle g_i, \sigma_i \bullet e_1\big\rangle, \ldots, union\big\langle g_i, \sigma_i \bullet e_m\big\rangle\Big) \stackrel{\text{Св.1.(e)}}{=} union\big\langle g_i, \sigma_i \bullet op(e_1, \ldots, e_m)\big\rangle$

(c) То же, что в (b)

(d) $union\Big\langle G, M \bullet LI(x)\Big\rangle = union\Big\langle G, read(M, M \bullet x)\Big\rangle \stackrel{\text{Th.7}}{=}$
$= union\Big\langle G, union\big\langle g_i, read(\sigma_i, M \bullet x)\big\rangle\Big\rangle \stackrel{\text{Св.1.(c)}}{=} union\big\langle g_i, read(\sigma_i, M \bullet x)\big\rangle \stackrel{\text{I.H.}}{=}$
$= union\Big\langle g_i, read\big(\sigma_i, union\langle g_j, \sigma_j \bullet x\rangle\big)\Big\rangle \stackrel{\text{Lm.3}}{=} union\Big\langle g_i, union\big\langle g_j, read\big(\sigma_i, \sigma_j \bullet x\big)\big\rangle\Big\rangle \stackrel{g_i \text{ disj.}}{=}$
$= union\big\langle g_i, read(\sigma_i, \sigma_i \bullet x)\big\rangle = union\big\langle g_i, \sigma_i \bullet LI(x)\big\rangle$

□

Теорема 9. Для всех символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$ и локаций x справедливо следующее:

$$union\Big\langle g_1 \vee \ldots \vee g_n, read\big(merge\langle g_i, \sigma_i\rangle \circ \sigma, x\big)\Big\rangle = read\big(merge\langle g_i, \sigma_i \circ \sigma\rangle, x\big).$$

Доказательство. Пусть $G \stackrel{\text{def}}{=} g_1 \vee \ldots \vee g_n$ и $M \stackrel{\text{def}}{=} merge\langle g_i, \sigma_i\rangle$.

$$\begin{aligned}
&union\Big\langle G, read\big(M \circ \sigma, x\big)\Big\rangle \stackrel{\text{Лемма 7}}{=} \\
&= union\Big\langle G, find(\sigma, x, M, read(M, x))\Big\rangle = \\
&= union\Big\langle G, union\Big(\big\{\big\langle x = M \bullet l, M \bullet (\sigma(l))\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq M \bullet l, read(M, x)\big\rangle\Big)\Big\rangle \stackrel{\text{Лем.4, Теор.7, Св.1}}{=} \\
&= union\Big\langle g_i, union\Big(\big\{\big\langle x = \sigma_i \bullet l, \sigma_i \bullet (\sigma(l))\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq \sigma_i \bullet l, read(\sigma_i, x)\big\rangle\Big)\Big\rangle = \\
&= union\big\langle g_i, find\big(\sigma, x, \sigma_i, read(\sigma_i, x)\big)\big\rangle \stackrel{\text{Лем.7}}{=} \\
&= union\big\langle g_i, read(\sigma_i \circ \sigma, x)\big\rangle = read\big(merge\langle g_i, \sigma_i \circ \sigma\rangle, x\big)
\end{aligned}$$

□

Лемма 9. Для всех символьных куч σ, τ, символьных локаций x, y и символьных выражений v, d,

$$find\big(write(\sigma, y, v), x, \tau, d\big) = ite\big(x = \tau \bullet y, \tau \bullet v, find(\sigma, x, \tau, d)\big)$$

Доказательство.

$$\begin{aligned}
&find\big(write(\sigma, y, v), x, \tau, d\big) = \\
&= union\Big(\big\{\big\langle x = \tau \bullet l, \tau \bullet (write(\sigma, y, v)(l))\big\rangle \mid l \in dom(write(\sigma, y, v))\big\} \cup \\
&\qquad \cup \big\langle \bigwedge_{l \in dom(write(\sigma,y,v))} x \neq \tau \bullet l, d\big\rangle\Big) = \\
&= union\Big(\big\{\big\langle x = \tau \bullet l, \tau \bullet ite(y = l, v, \sigma(l))\big\rangle \mid l \in dom(\sigma) \cup \{y\}\big\} \cup \\
&\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma) \cup \{y\}} x \neq \tau \bullet l, d\big\rangle\Big) = \\
&= union\Big(\big\{\big\langle x = \tau \bullet l, ite\big(\tau \bullet y = \tau \bullet l, \tau \bullet v, \tau \bullet (\sigma(l))\big)\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad \cup \big\langle x = \tau \bullet y, \tau \bullet v\big\rangle \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq \tau \bullet l \wedge x \neq \tau \bullet y, d\big\rangle\Big) = \\
&= union\Big(\big\{\big\langle x = \tau \bullet l \wedge \tau \bullet y = \tau \bullet l, \tau \bullet v\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad \cup \big\{\big\langle x = \tau \bullet l \wedge \tau \bullet y \neq \tau \bullet l, \tau \bullet (\sigma(l))\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad \cup \big\langle x = \tau \bullet y, \tau \bullet v\big\rangle \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq \tau \bullet l \wedge x \neq \tau \bullet y, d\big\rangle\Big) = \\
&= ite\Big(x = \tau \bullet y, \tau \bullet v, union\Big(\big\{\big\langle x = \tau \bullet l, \tau \bullet (\sigma(l))\big\rangle \mid l \in dom(\sigma)\big\} \cup \\
&\qquad \cup \big\langle \bigwedge_{l \in dom(\sigma)} x \neq \tau \bullet l, d\big\rangle\Big)\Big) = \\
&= ite\big(x = \tau \bullet y, \tau \bullet v, find(\sigma, x, \tau, d)\big)
\end{aligned}$$

□

Теорема 10. Для всех символьных куч σ, символьных локаций x, y и символьных выражений v справедливо следующее:

$$read\big(write(\sigma, y, v), x\big) = ite\big(x = y, v, read(\sigma, x)\big).$$

Доказательство.

$$\begin{aligned}
&read\big(write(\sigma, y, v), x\big) = find(write(\sigma, y, v), x, \epsilon, LI(x)) \overset{\text{Лем.9}}{=} \\
&= ite\big(x = \epsilon \bullet y, \epsilon \bullet v, find(\sigma, x, \epsilon, LI(x))\big) = ite\big(x = y, v, read(\sigma, x)\big)
\end{aligned}$$

□

Теорема 11. Для всех символьных куч σ, σ', символьных локаций y и символьных выражений v справедливо следующее:

$$\sigma \circ write(\sigma', y, v) = write(\sigma \circ \sigma', \sigma \bullet y, \sigma \bullet v).$$

Доказательство.

$$(\sigma \circ write(\sigma', y, v))(x) = find(write(\sigma', y, v), x, \sigma, \sigma(x)) \overset{\text{Лем.9}}{=}$$

$$= ite\big(x = \sigma \bullet y, \sigma \bullet v, find(\sigma', x, \sigma, \sigma(x))\big) =$$
$$= ite\big(x = \sigma \bullet y, \sigma \bullet v, (\sigma \circ \sigma')(x)\big) = write(\sigma \circ \sigma', \sigma \bullet y, \sigma \bullet v)$$

□

Теорема 12. Для всех символьных куч $\sigma_1, \ldots, \sigma_n$, непересекающихся ограничений $g_1, \ldots, g_n$, символьных локаций y и символьных выражений v справедливо следующее:

$$write\big(merge\langle g_i, \sigma_i\rangle, y, v\big) = merge\big\langle g_i, write(\sigma_i, y, v)\big\rangle.$$

Доказательство.

$$dom\big(write\big(merge\langle g_i, \sigma_i\rangle, y, v\big)\big) = \bigcup_{i=1}^{n} dom(\sigma_i) \cup \{y\} =$$
$$= dom\big(merge\big\langle g_i, write(\sigma_i, y, v)\big\rangle\big)$$

Пусть $x \in dom\big(write\big(merge\langle g_i, \sigma_i\rangle, y, v\big)\big)$.

$$write\big(merge\langle g_i, \sigma_i\rangle, y, v\big)(x) = ite\big(x = y, v, merge\langle g_i, \sigma_i\rangle(x)\big) =$$
$$= ite\big(x = y, v, union\langle g_i, read(\sigma_i, x)\rangle\big) \overset{\text{Св.1}}{=} union\Big\langle g_i, ite\big(x = y, v, read(\sigma_i, x)\big)\Big\rangle =$$
$$= merge\big\langle g_i, write(\sigma_i, y, v)\big\rangle(x)$$

□