DOI: 10.15514/ISPRAS-2019-31(5)-3

Автоматическое доказательство корректности программ с динамической памятью

[1] *Ю.О.Костюков, ORCID: 0000-0003-4607-039X <kostyukov.yurii@gmail.com>*
[1] *К.А.Батоев, ORCID: 0000-0003-1124-7909 <konstantin.batoev@gmail.com>*
[1,2] *Д.А.Мордвинов, ORCID: 0000-0002-6437-3020 <dmitry.mordvinov@jetbrains.com>*
[1] *М.П.Костицын, ORCID: 0000-0001-9982-6571 <mishakosticyn@yandex.ru>*
[1] *А.В.Мисонижник, ORCID: 0000-0002-5907-0324 <misonijnik@gmail.com>*

[1] *Санкт-Петербургский государственный университет,*
199034, Россия, Санкт-Петербург, Университетская набережная, д. 7–9
[2] *JetBrains Research*
197342, Россия, Санкт-Петербург, Кантемировская ул., 2

Аннотация. В данной работе изучаются теоретические основы автоматической модульной верификации императивных программ с динамической памятью. Вводится формализм *композициональной символьной памяти*, который используется для построения композиционального алгоритма, порождающего *обобщённые кучи*. Они являются термами *исчисления символьных куч*, которые описывают состояния *произвольных* циклических фрагментов программы. Выводимые в этом исчислении *кучи* соответствуют достижимым состояниям исходной программы. В работе также устанавливается соответствие между выводом в этом исчислении и исполнением функциональных программ второго порядка без эффектов.

Ключевые слова: формальная верификация; автоматическая верификация; символьное исполнение; анализ программ; динамическая память; композициональность; чистые функции

Для цитирования: Костюков Ю.О., Батоев К.А., Мордвинов Д.А., Костицын М.П., Мисонижник А.В. Автоматическое доказательство корректности программ с динамической памятью. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 37-62. DOI: 10.15514/ISPRAS-2019-31(5)-3

Automatic verification of heap-manipulating programs

[1] *Yu.O. Kostyukov, ORCID: 0000-0003-4607-039X <kostyukov.yurii@gmail.com>*
[1] *K.A. Batoev, ORCID: 0000-0003-1124-7909 <konstantin.batoev@gmail.com>*
[1,2] *D.A. Mordvinov, ORCID: 0000-0002-6437-3020 <dmitry.mordvinov@jetbrains.com>*
[1] *M.P. Kostitsyn, ORCID: 0000-0001-9982-6571 <mishakosticyn@yandex.ru>*
[1] *A.V. Misonizhnik, ORCID: 0000-0002-5907-0324 <misonijnik@gmail.com>*

[1] *Saint Petersburg State University,*
3A bld. 1, Kantemirovskaya st., St Petersburg, Russia, 194100
[2] *JetBrains Research*
2, Kantemirovskaya st., St Petersburg, Russia, 197342

Abstract. Theoretical foundations of compositional reasoning about heaps in imperative programming languages are investigated. We introduce a novel concept of compositional symbolic memory and its relevant properties. We utilize these formal foundations to build up a compositional algorithm that generates

generalized heaps, terms of symbolic heap calculus, which characterize arbitrary cyclic code segments. All states inferred by this calculus precisely correspond to reachable states of the original program. We establish the correspondence between inference in this calculus and execution of pure second-order functional programs. The contribution of this work is as follows: (1) a formal model of compositional symbolic memory is proposed; (2) the properties of its correctness are formulated; (3) the calculus of symbolic heaps has been introduced: the conclusions in this calculus give all attainable states of the program; (4) the concept of generalized heaps is introduced, an algorithm for automatic modular construction of generalized heaps according to an imperative program is proposed; (5) an approach is proposed to reduce the problem of finding an output in calculus of symbolic heaps to the problem of proving the safety of functional programs.

Keywords: formal verification; automatic verification; symbolic execution; static analysis; dynamic memory; heap analysis; compositionality; pure functions

For citation: Kostyukov Yu.O., Batoev K.A., Mordvinov D.A., Kostitsyn M.P., Misonizhnik A.V. Automatic verification of heap-manipulating programs. Trudy ISP RAN/Proc. ISP RAS, vol.31, issue 5, 2019, pp. 37-62 (in Russian). DOI: 10.15514/ISPRAS-2019-31(5)-3

1. Введение

Большая часть современного программного обеспечения написана на языках с динамически выделяемой памятью, таких как C++, Java, C#. Автоматическая верификация и анализ таких программ является очень трудоёмкой задачей [1]. При этом даже корректные с теоретической точки зрения методы могут оказаться неэффективными из-за больших размеров анализируемых программ [2]. Для решения этой проблемы были предложены *композициональные* техники, которые хорошо зарекомендовали себя на практике [3, 4, 5, 6]. Такие техники выполняют анализ функций в *изоляции*, т. е. вне контекста конкретного вызова, и в дальнейшем 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 пространство состояний программы снизу или сверху. Аппроксимирующие снизу подходы рассматривают не все сценарии поведения программы, например, при помощи раскрутки циклов на конечное число шагов [7]. Это позволяет находить ошибки, но не доказывать корректность произвольных программ. Аппроксимирующие сверху подходы анализируют упрощённую версию программы, что на практике приводит к большому числу ложноположительных срабатываний [8]. Таким образом, остаётся актуальной задача точного анализа программ с помощью композициональных техник.

Эту задачу можно решить при помощи введения особой *модели памяти*, представляющей состояния программы. Такая модель 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.

В данной статье предложен подход к модульной верификации программ с динамической памятью. Подход основывается на новой модели *композициональной символьной памяти*. Эта модель основывается на идее *ленивого инстанцирования* [9] и описывает состояние динамической памяти программы символьными выражениями над значениями входных *ячеек памяти*. Адреса этих ячеек могут быть символьно описаны с использованием содержимого других ячеек; это позволяет выражать эффект любой функции на произвольной рекурсивной структуре данных.

Модель композициональной символьной памяти используется для построения *исчисления символьных куч*, которые, в свою очередь, позволяют описывать поведение произвольной императивной программы с динамической памятью при помощи символьных состояний

программы. Термы этого исчисления (т.н. *обобщённые кучи*), построенные по программе, *в точности* описывают эффект этой программы на произвольном состоянии.

В статье также предложена автоматическая процедура композиционального построения обобщённых куч, описывающих произвольные фрагменты императивной программы.

Поскольку для вывода в исчислении символьных куч не требуется хранения контекста, по любой обобщённой символьной куче можно построить эквивалентную ей функциональную программу, сведя задачу проверки корректности императивных программ с динамической памятью к задаче проверки корректности чистых функций. В статье предлагается сведение задачи анализа императивных программ к задаче вывода *уточнённых типов* (refinement types) [10] для функциональных программ, получаемых из обобщённых куч.

Доказательства сформулированных в работе свойств и теорем в основном опущены в виду ограничений на размер статьи. Читатель может ознакомиться с ними в [27].

Вклад данной работы заключается в следующем: (1) предложена формальная модель *композициональной символьной памяти*; (2) сформулированы свойства её корректности; (3) введено *исчисление символьных куч*: выводы в этом исчислении дают все достижимые состояния программы; (4) введено понятие *обобщённых куч*, предложен алгоритм автоматического модульного построения обобщённых куч по императивной программе; (5) предложен подход к сведению задачи поиска вывода в *исчислении символьных куч* к задаче доказательства безопасности функциональных программ.

2. Обзор

Существует большое количество работ, посвящённых автоматическому анализу и доказательству корректности программ с динамической памятью [1, 2, 5, 6]. Подходы делятся на *аппроксимирующие снизу*, *аппроксимирующие сверху* и *точные*. Аппроксимирующие снизу подходы исследуют только часть пространства состояний; они пригодны для поиска ошибок в программах, но не годятся для доказательства корректности программ. Подходы, аппроксимирующие пространство состояний сверху, пригодны для доказательства корректности программ, но на практике порождают большое количество ложноположительных срабатываний. Точные подходы исследуют все пространство состояний, но, насколько известно авторам, в данной статье представлен первый *полностью автоматический* подход к точному анализу императивных программ с динамической памятью.

К подходам, аппроксимирующим пространство состояний снизу, можно отнести динамическое символьное исполнение [11] и ограничиваемую проверку моделей [12]. Одна из основных идей в символьном исполнении программ с динамической памятью – ленивое инстанцирование [9]. Оно позволяет производить анализ программ с рекурсивными структурами данных без ручной спецификации размеров этих структур. Существует множество работ, развивающих эту идею [13, 14], но все они следуют идее классического символьного исполнения с раскруткой отношения перехода. Алгоритм, представленный в данной работе, не раскручивает отношение перехода программы, а строит систему ограничений в формализме композициональной символьной памяти, точно описывающих поведения программы.

Как правило, подходы, аппроксимирующие пространство состояний сверху, основаны на *абстрактной интерпретации* программ [2, 6, 8]. Существует множество подходов к абстрактной интерпретации программ с динамической памятью. Подавляющее большинство из них основаны на *логике с разделением* (separation logic) [15]; абстрактный домен таких анализаторов хорошо подходит для автоматического анализа *формы куч* [16], т.е. анализом того, как объекты в куче *связаны* друг с другом, а не их содержимого.

Анализаторы, которые пользуются логикой с разделением, являются, как правило, композициональными [2, 5, 6]. Логика с разделением адаптирована для символьного исполнения программ [17]. Основной проблемой логики с разделением является её невыразительность — она хорошо подходит для рассуждения о форме куч, но не подходит для анализа *данных* в динамической памяти, потому что введённая в этой логике разделяющая конъюнкция не позволяет выражать сложных свойств. Существуют подходы на основе логики с разделением, которые позволяют производить более точный анализ динамической памяти вплоть до побайтовых манипуляций с памятью [18]. Однако практическая применимость таких подходов сопряжена с большим количеством ложноположительных срабатываний.

Существуют также работы, основанные на идее сведения задачи верификации программ с динамической памятью к решению системы рекурсивно-логических ограничений [3, 20]. Такие подходы, как и наш, используют SMT-решатели [19] для решения ограничений и вывода индуктивных инвариантов системы. Логические решатели позволяют верифицировать программы, не порождая ложных срабатываний.

3. Демонстрационный язык

В данном разделе мы определяем демонстрационный язык программирования с операциями над динамической памятью, для которого позже будет представлена процедура автоматической модульной верификации. Синтаксис языка представлен на рис. 1.

$$
\begin{array}{l}
Program ::= Statement^{*} \\
Statement ::= label : Statement \\
\qquad\qquad \mid Location := Expression \\
\qquad\qquad \mid \boldsymbol{goto}\ \{Expression \rightarrow label\}^{+} \\
\qquad\qquad \mid \boldsymbol{fail} \mid \boldsymbol{halt} \\
Expression ::= \boldsymbol{null} \mid \boldsymbol{true} \mid \boldsymbol{false} \mid \mathbb{N} \\
\qquad\qquad \mid Expression\ BinOp\ Expression \\
\qquad\qquad \mid UnOp\ Expression \\
\qquad\qquad \mid Location \\
\qquad\qquad \mid \boldsymbol{new}\ \{ident\ =\ Expression\}^{+} \\
Location ::= ident \mid Location.ident
\end{array}
$$

Рис. 1. Синтаксис демо-языка
Fig. 1. Demo language syntax

Состояния программы на этом языке описываются состояниями динамической памяти. Язык не содержит функций, операций ветвления и циклов, однако включает оператор условного перехода по метке $\boldsymbol{goto}\ \{Expression \rightarrow label\}^{+}$. Слева от стрелки находится условие перехода, справа — метка, на которую нужно перейти. Оператор последовательно вычисляет условия слева от стрелок и переходит по первой метке справа от стрелки, чьё условие истинно.

Идентификаторами $ident$ в языке называются идентификаторы в стиле языка C; $BinOp$ *и* $UnOp$ это простые арифметические и булевы операторы.

Оператор $\boldsymbol{new}\ \{ident\ =\ Expression\}^{+}$ выделяет в памяти новый объект с именами полей $ident$, инициализируя каждый из них соответствующим $Expression$. Используя точку, как в определении $Location$, можно получить доступ к полям объектов. Доступ может быть вложенный, например $list.RootNode.Next.Key$.

В каждой переменной находятся либо примитивные значения (**null**, **true**, 42), либо ссылки на объекты в динамической памяти. Оператор $:=$ перезаписывает ссылку. Например, в листинге 1 в строке 3 в переменную p помещается ссылка на l, сам объект не копируется.

Затем, в строке 7 переменная p начинает ссылаться на новый объект, а переменная l по-прежнему указывает на старый. В строке 10 меняется само содержимое памяти, независимо от переменных p *и* l.

```
1.  RemoveAll:
2.  l := new {Key = x, Next = l}
3.  p := l
4.  RemoveAllIterate:
5.  goto {p.Next = null  -> RemoveAllFinalize,
6.        p.Next.Key = x -> RemoveAllRemoveElement}
7.  p := p.Next
8.  goto {true -> RemoveAllIterate}
9.  RemoveAllRemoveElement:
10. p.Next := p.Next.Next
11. goto {true -> RemoveAllIterate}
12. RemoveAllFinalize:
13. l := l.Next
14.
15. p := l
16. Contains:
17. goto {p = null  -> Exit,
18.       p.Key = x -> Error}
19. p := p.Next
20. goto {true -> Contains}
21.
22. Error: fail
23. Exit: halt
```

Листинг 1. Программа, модифицирующая динамическую память
Listing 1. Example of heap-manipulating program.

Далее мы будем рассматривать только корректно типизированные программы: арифметические операции применяются согласно их стандартной семантике; поле и то, что в него записывается, согласованы по типам; выражения в **goto** имеют булевский тип; поля всегда читаются успешно (кроме чтения из **null**); выражения и имена полей, используемые в операторе **new**, согласованы по типам. Также мы предполагаем, что программы не читают из неинициализированных локаций, всякая программа содержит инструкцию **halt**, все метки, на которые есть переходы, определены и имена идентификаторов, меток и ключевые слова языка не пересекаются. Предложенный язык не даёт возможности управлять памятью на низком уровне, в частности, освобождать выделенную память. Таким образом, программы на этом языке допускают всего два вида ошибок, которые необходимо находить: доступ к полю по ссылке, содержащей **null**, и достижимость инструкции **fail**.

4. Композициональная символьная память

В центре нашего подхода стоит формализм композициональных символьных куч. Концепция *композициональной символьной памяти* (КСП), определяемая в этом разделе, основана на идее *ленивого инстанцирования* [9, 13]. Ленивое инстанцирование — это техника, позволяющая строить *конечные* символьные выражения для рекурсивных структур данных, таких как связные списки и деревья. При этом предлагается инициализировать поля рекурсивных структур данных *по требованию* вместо инициализации всей структуры одномоментно, что потребовало бы заранее заданных

ограничений на её размер. Это, например, позволяет анализировать списки и деревья, не зная заранее их размер.

1. **goto** {**true** -> F}	c
2. G:	
3. x := x + 1	G (стр. 2-4): $\{x \mapsto LI(x) + 1\}$
4. **goto** {**true** -> Exit}	F (стр. 5-7): $\{x \mapsto 42\} \circ \{x \mapsto LI(x) + 1\} =$
5. F:	$= \{x \mapsto 42 + 1\}$
6. x := 42	
5. **goto** {**true** -> G}	
8. Exit: **halt**	

Рис. 2. Пример композиции состояний
Fig. 2. Heap composition example.

Основная идея КСП состоит в том, чтобы трактовать лениво инициализированные локации $LI(x)$[1] как *ячейки*, в которые будут подставлены значения из контекстного состояния. Например, состояние, описывающее код после метки G на рис. 2 (стр. 2-4), содержит незаполненную ячейку x. Это означает, что оно описывает *эффект* этого фрагмента кода на *произвольном* контекстном состоянии, т.е. состоянии с произвольным значением x. Таким контекстным состоянием может быть, например, состояние после метки F до перехода на G (стр. 5-6). Чтобы получить полное состояние после метки F (стр. 5-7), необходимо заранее подсчитанный эффект G применить к текущему состоянию F. Для этого нужно заполнить ячейки состояния G значениями из текущего контекста F. Кучу, полученную в результате этого процесса, мы называем *композицией* куч. Заметим, что адаптация этой идеи к программам произвольной сложности требует некоторых дополнительных усилий.

Далее будет описан формализм композициональной символьной памяти. Доказательства теорем приведены в [27].

4.1 Символьные выражения

Чтобы представлять произвольные состояния программ на нашем демо-языке, необходимо ввести понятие символьных выражений. Определение символьного выражения представлено на рис. 3.

Символьный терм ($term$) — это либо арифметическое выражение ($arith$), либо символьный адрес локации в памяти (loc).

$$
\begin{aligned}
term ::= \;& arith \mid loc \\
arith ::= \;& \mathbb{N} \mid arith \pm arith \mid -arith \mid LI^{arith}(loc) \\
& \mid union^{arith}(\langle guard, arith \rangle^*) \\
loc ::= \;& null \mid 0x[0-9]^+ \mid ident \\
& \mid loc.FieldName \mid LI^{loc}(loc) \\
& \mid union^{loc}(\langle guard, loc \rangle^*) \\
guard ::= \;& \top \mid \bot \mid \neg guard \mid guard \wedge guard \mid guard \vee guard \\
& \mid arith = arith \mid arith < arith \mid loc = loc
\end{aligned}
$$

[1] LI — lazy instantiation

Рис. 3. Грамматика символьных выражений
Fig. 3. Symbolic terms.

Символьные значения записываются как $LI^*(x)$, что означает «ленивое инстанцирование x». Далее всюду тип символьного значения либо не важен, либо очевиден из контекста, потому мы будем его опускать и писать просто $LI(x)$. Заметим, что локация-источник символьного значения LI может быть также символьной, так что, например, термы вида $LI(LI(list).Key) + 1$ также допустимы.

```
1. a := new {Key = 15, Next = null}
2. a.Next := a
3. Next := a
4. c := d.Next
```

$$\begin{cases} a & \mapsto 0x1 \\ 0x1.Key & \mapsto 15 \\ 0x1.Next & \mapsto 0x1 \\ LI(b).Next & \mapsto 0x1 \\ c & \mapsto LI(LI(d).Next) \end{cases}$$

Рис. 4. Пример различных типов локаций
Fig. 4. Different location types example.

Локации могут быть *конкретными*, т.е. известными в текущем контексте ($0x[0-9]^+$, порождаются оператором **new**); именованными ($ident$, глобальные переменные, a, b, c); ссылками на конкретное поле; ленивыми инстанцированиями других локаций. Например, на рис. 4, где представлен фрагмент кода и соответствующее ему символьное состояние памяти, локация b неизвестна, поэтому запись в её поле $Next$ порождает запись по символьной локации $LI(b).Next$. Аналогично неизвестна локация d, но также неизвестен и элемент, на который ссылается её поле $Next$, из-за чего символьная локация c указывает на символьный терм $LI(LI(d).Next)$.

Вернёмся к определению символьных выражений (рис. 3). На этом рисунке $guard$ обозначает *ограничение,* представленное в виде логической формулы (условие пути или условие перехода по метке). Такие формулы «защищают» элементы *символьных объединений*. Символьное объединение[2] ($union$) — это обобщение символа $ite(cond, x, y)$. Как и в случае символьных значений, мы опускаем тип символьных объединений и пишем просто $union(*)$. За символьным объединением мы закрепляем следующую семантику: $x = union(\langle g_1, v_1\rangle, \dots, \langle g_n, v_n\rangle)$ тогда и только тогда, когда $(g_1 \wedge x = v_1) \vee \dots \vee (g_n \wedge x = v_n)$. Символьные объединения позволяют при помощи одного символьного состояния описать несколько веток исполнения программы.

Замечание. Потребуем, чтобы ограничения в объединениях *не пересекались*, т. е. только одно ограничение должно выполняться при подстановке конкретных значений вместо символьных. Однако несколько ограничений могут выполняться одновременно в том случае, если они «защищают» одно и то же значение. Например, допустимы объединения вида

$$union(\langle LI(x) = LI(y), LI(LI(y).Key) + 7\rangle,$$
$$\langle LI(x) = LI(z), LI(LI(z).Key) + 7\rangle).$$

Мы рассматриваем содержимое объединений как множество пар, потому пишем, например, $union(\{\langle g, v\rangle\} \cup X)$. Чтобы избежать перегрузки синтаксиса лишними скобками, мы опускаем их далее при записи одноэлементных множеств. Так, пример выше можно записать следующим образом: $union(\langle g, v\rangle \cup X)$. Также мы опускаем круглые скобки там, где не возникает двусмысленности, например, пишем $union\langle x > 5{,}42\rangle$ или $union\{\langle g_i, v_i\rangle | 1 \le i \le n\}$

Выражение на рис. 3 — это либо терм, либо ограничение. *Примитивные* выражения — это натуральные числа, именованные локации, конкретные адреса в памяти, $\boldsymbol{null}$, $\top$ и $\bot$.

[2] Понятие символьных объединений заимствовано из [21]

Операциями являются сложение, вычитание, унарный минус, сравнения, логические связки и чтение поля. В тех случаях, когда вид операции не важен, мы пользуемся следующей нотацией: $op(e_1, \dots, e_n)$.

Замечание. Равенство символьных термов является *семантическим*, например, $2 * (x + 1) = x + x + 4 - 2$ и $union(\langle x + 5 = y + 4{,}7\rangle, \langle \bot, 42\rangle) = union\langle x + 1 = y, 7\rangle$.

Далее мы перечисляем некоторые очевидные свойства символьного объединения.

Утверждение 1.

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

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

$$union(\langle g, union(\langle g_1, v_1\rangle, \dots, \langle g_n, v_n\rangle)\rangle \cup X) = union(\{\langle g \wedge g_1, v_1\rangle, \dots, \langle g \wedge g_n, v_n\rangle\} \cup X)$$

В частности,

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

(c) $union(\{\langle g_1, v\rangle, \dots, \langle g_n, v\rangle\} \cup X) = union(\langle g_1 \vee \dots \vee g_n, v\rangle \cup X)$

В частности,

$$union(\{\langle g, v\rangle, \langle g \wedge g_1, v\rangle, \dots, \langle g \wedge g_n, v\rangle\} \cup X) = union(\langle g, v\rangle \cup X)$$

(d) $op(union(\langle g_1^1, e_1^1\rangle, \dots, \langle g_{n_1}^1, e_{n_1}^1\rangle), \dots, union(\langle g_1^m, e_1^m\rangle, \dots, \langle g_{n_m}^m, e_{n_m}^m\rangle)) = union(\{\langle g_{i_1}^1 \wedge \dots \wedge g_{i_m}^m, op(e_{i_1}^1, \dots, e_{i_m}^m)\rangle | 1 \le i_j \le n_j\})$

В частности, для непересекающихся ограничений $g_1, \dots, g_n$

$$op(union(\langle g_1, e_1^1\rangle, \dots, \langle g_n, e_n^1\rangle), \dots, union(\langle g_1, e_1^m\rangle, \dots, \langle g_n, e_n^m\rangle)) = union(\langle g_1, op(e_1^1, \dots, e_1^m)\rangle, \dots, \langle g_n, op(e_n^1, \dots, e_n^m)\rangle)$$

4.2 Символьные кучи

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

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

Заметим, что это более сильное ограничение, чем накладывает само понятие функции ($x \equiv y \Rightarrow \sigma(x) \equiv \sigma(y)$). Это связано с тем, что равенство термов и локаций в нашем подходе является не синтаксическим, а семантическим. Так, например, функция $\{LI(x).Key \mapsto 10;\ LI(y).Key \mapsto 15\}$ не является символьной кучей, т.к. при подстановке вместо x и y, например, $0x1$, получится, что этот адрес указывает на два разных значения. Напротив, следующая функция является символьной кучей: $\{LI(x).Key \mapsto union(\langle LI(x) = LI(y), 15\rangle, \langle LI(x) \neq LI(y), 10\rangle); LI(y).Key \mapsto 15\}$.

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

Определение 3. Пусть $x \in dom(\sigma)$ или x — символьная локация. Тогда определим чтение локации x в символьной куче σ следующим образом:

$$read(\sigma, x) \stackrel{\text{def}}{=} union(\{\langle x = l, \sigma(l)\rangle | l \in dom(\sigma)\} \cup \langle \bigwedge_{l \in dom(\sigma)} x \neq l, LI(x)\rangle). \quad (2)$$

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

Очевидно, что при $x \in dom(\sigma)$ выполнено $read(\sigma, x) = \sigma(x)$. Одно из ограничений $x = l$ будет выполняться, тогда как один из элементов конъюнкции $\wedge_{l \in dom(\sigma)} x \neq l$ будет, наоборот, невыполним, следовательно $read(\sigma, x)$ не сможет вернуть значение $LI(x)$.

Сделаем следующее наблюдение: фактически, из опр. 3 следует, что множество ограничений в формуле (2) может содержать пересечения, т.е. в куче могут содержаться две (или более) символьные локации, которые совпадают при некоторых конкретных подстановках. Инвариант символьной кучи (1) позволяет обойти возможную проблему с совпадающими адресами: благодаря ему при совпадении ограничений не будет конфликтов между «защищаемыми» значениями.

Пример 1. *Пусть* $\sigma = \{0x1.A \mapsto 42; LI(x).B \mapsto union(\langle LI(x).B = 0x1.A, 42\rangle, \langle LI(x).B \neq 0x1.A, 7\rangle)\}$. *Тогда*

$$read(\sigma, 0x1.A) = union(\langle 0x1.A = 0x1.A, 42\rangle, \langle 0x1.A = LI(x).B, union(\dots)\rangle,$$
$$\langle 0x1.A \neq 0x1.A \wedge LI(x).B \neq 0x1.A, LI(0x1.A)\rangle)) =$$
$$= union(\langle \top, 42\rangle, \langle 0x1.A = LI(x).B, union(\dots)\rangle, \langle \bot, LI(0x1.A)\rangle) = 42$$
$$read(\sigma, LI(y).B) =$$
$$= union(\langle LI(y).B = 0x1.A, 42\rangle, \langle LI(y).B = LI(x).B,$$
$$union(\langle LI(x).B = 0x1.A, 42\rangle, \langle LI(x).B \neq 0x1.A, 7\rangle)\rangle,$$
$$\langle LI(y).B \neq 0x1.A \wedge LI(y).B \neq LI(x).B, LI(LI(y).B)\rangle) \overset{\text{утв. 1}}{=}$$
$$= union(\langle LI(y).B = 0x1.A, 42\rangle, \langle LI(y).B = LI(x).B \wedge LI(x).B \neq 0x1.A, 7\rangle,$$
$$\langle LI(y).B \neq 0x1.A \wedge LI(y).B \neq LI(x).B, LI(LI(y).B)\rangle)$$

4.3 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. $\sigma \bullet op(e_1, \dots, e_n) \overset{\text{def}}{=} op(\sigma \bullet e_1, \dots, \sigma \bullet e_n)$.
3. $\sigma \bullet union\{\langle g_1, t_1\rangle, \dots, \langle g_n, t_n\rangle\} \overset{\text{def}}{=} union\{\langle \sigma \bullet g_1, \sigma \bullet t_1\rangle, \dots, \langle \sigma \bullet g_n, \sigma \bullet t_n\rangle\}$.
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 п.4 заполняет ячейку значением из σ.

Определение 5. *Композиция символьных куч* σ *и* σ' — это частичная функция $\sigma \circ \sigma' : loc \to term$, определяемая так: $(\sigma \circ \sigma')(x) \overset{\text{def}}{=}$

$$\overset{\text{def}}{=} union(\{\langle x = \sigma \bullet l, \sigma \bullet (\sigma'(l))\rangle | l \in dom(\sigma')\} \cup \langle \underset{l \in dom(\sigma')}{\wedge} x \neq \sigma \bullet l, \sigma(x)\rangle).$$

Композиция $\sigma \circ \sigma'$ определена на всех локациях, удовлетворяющих ограничению $x = \sigma \bullet l$, где $l \in dom(\sigma')$, и на всех локациях $dom(\sigma)$ (здесь и далее запись $\{\sigma \bullet a | a \in A\}$ сокращается как $\sigma \bullet A$). Из этого следует, что:

$$dom(\sigma \circ \sigma') = dom(\sigma) \cup \sigma \bullet dom(\sigma'). \quad (3)$$

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 заполнением символьных ячеек из σ_2 значениями из контекста σ_1 с последующей их записью в контекст σ_1.

Пример 2. Пусть $\sigma = \{x \mapsto 42; y \mapsto 7\}$ и $\sigma' = \{y \mapsto LI(x) - LI(y)\}$. Тогда $\sigma \circ \sigma' = \{x \mapsto 42; y \mapsto 42 - 7\}$.

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

Теорема 2. Для произвольной символьной кучи σ, локации x и выражения e справедливо следующее:

(a) $read(\epsilon, x) = LI(x)$
(b) $\epsilon \bullet e = e$
(c) $\epsilon \circ \sigma = \sigma$
(d) $\sigma \circ \epsilon = \sigma$

Стоит отметить, что имеется некоторое сходство между чтением (опр. 3) и композицией (опр. 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 определения оператора $find$, который далее используется для трансляции обобщённых куч в чистые функции.

Определение 6. $find(\sigma, x, \tau, d) \overset{\text{def}}{=}$

$$\overset{\text{def}}{=} union(\{\langle x = \tau \bullet l, \tau \bullet (\sigma(l))\rangle | l \in dom(\sigma)\} \cup \langle \underset{l \in dom(\sigma)}{\wedge} x \neq \tau \bullet l, d\rangle)$$

Теор. 2 позволяет компактно выразить $read$ и композицию куч через $find$:

$$read(\sigma, x) = find(\sigma, x, \epsilon, LI(x)) \quad (4)$$
$$(\sigma \circ \sigma')(x) = find(\sigma', x, \sigma, \sigma(x)) \quad (5)$$

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

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

1.	`goto {true -> F}`	
2.	`G:`	$\sigma_G = \{x \mapsto (LI(LI(a).Key) + 5)\}$
3.	`x := a.Key + 5`	$read(\sigma_G, x) = \sigma_G(x) = LI(LI(a).Key) + 5$
4.	`goto {true -> Exit}`	$\sigma_F \bullet read(\sigma_G, x) = 10 + 5$
5.	`F:`	
6.	`a := new {Key = 10}`	$\sigma_F \circ \sigma_G = \{a \mapsto 0x1;\ 0x1.Key \mapsto 10;\ x \mapsto (10 + 5)\}$
7.	`goto {true -> G}`	$read(\sigma_F \circ \sigma_G, \sigma_F \bullet x) = read(\sigma_F \circ \sigma_G, x) = 10 + 5$
8.	`Exit:`	
9.	`r := x`	
10.	`halt`	

Рис. 5. Чтение из композиции состояний
Fig. 5. Read from composition example

Теорема 3 говорит о корректности чтения из композиции состояний. Например, имея состояния $\boldsymbol{\sigma_F}$ и $\boldsymbol{\sigma_G}$ исполняемых друг за другом фрагментов кода (как на рис. 5) и читая после фрагмента $\boldsymbol{G}$ переменную $\boldsymbol{x}$, можно прочитать переменную из позднего состояния $\boldsymbol{G}$, а затем заполнить результат из контекста $\boldsymbol{F}$. Однако возможно также прежде воспроизвести эффект $\boldsymbol{G}$ поверх эффекта $\boldsymbol{F}$ ($\boldsymbol{\sigma_F \circ \sigma_G}$), и прочитать $\boldsymbol{x}$ из уточнённого состояния. Теорема 3 утверждает, что результаты двух таких операций совпадут.

Теорема 4. Для всех символьных куч σ, σ' и символьного выражения e справедливо следующее: $(\sigma \circ \sigma') \bullet e = \sigma \bullet (\sigma' \bullet e)$.

Допустим, имеется три последовательных фрагмента кода с метками F, G и H, причём каждый фрагмент заканчивается переходом на следующую метку в этом списке. Интуитивно, итоговое символьное состояние всего кода не должно зависеть от порядка применения эффектов этих фрагментов. Следующая теорема показывает, что КСП обладает указанным свойством.

Теорема 5. Для всех символьных куч σ_1, σ_2 и σ_3 справедливо следующее:

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

Теорема 6. Пусть Σ — множество всех символьных куч. Тогда $(\Sigma, \circ)$ — моноид.

Доказательство. Из теоремы 1 следует замкнутость множества Σ относительно операции $\circ$, по теореме 2 ϵ — нейтральный элемент, и по теореме 5 операция $\circ$ ассоциативна.

4.4 Объединение символьных куч

До сих пор мы рассматривали символьные состояния только для линейных фрагментов кода, в которых все переходы по меткам были безусловными ($\boldsymbol{goto}\ \{true \to \cdots\}$). Для более сложных состояний необходим новый оператор, позволяющий объединить несколько состояний в одно.

Определение 7. *Объединением* $\sigma = merge(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle)$ 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: $(merge\langle g_i, \sigma_i\rangle)(x) \stackrel{\text{def}}{=} union\langle g_i, read(\sigma_i, x)\rangle$.

Теорема 7. Для любой символьной кучи σ и произвольных символьных локаций x, y справедливо следующее:

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

Далее покажем, что оператор $merge$ обладает интуитивными свойствами. **Теорема 8.** Для любых символьных куч $\sigma_1, \ldots, \sigma_n$ и произвольных непересекающихся ограничений $g_1, \ldots, g_n$, справедливо утверждение о том, что $merge\langle g_i, \sigma_i\rangle$ — символьная куча.

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

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

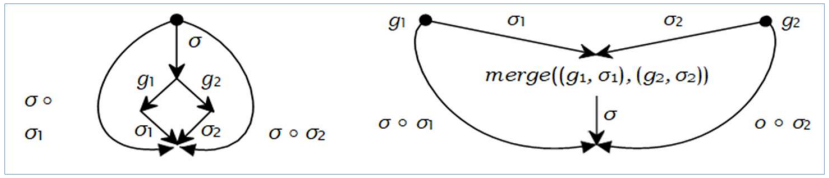

Рис. 6. Композиция объединения куч
Fig. 6. Composition of heap merge

Как уже было показано, результат вычисления символьного состояния не зависит от порядка применения эффектов. Необходимо показать, что это свойство КСП выполняется и для нового оператора объединения состояний. Для этого нужно рассмотреть два случая расстановки объединения и композиции, представленные на рис. 6.

Теорема 10. Для любых символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$ и произвольных непересекающихся ограничений $g_1, \ldots, g_n$ выполняется следующее утверждение:

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

Интересно, что симметричный случай гораздо сложнее. Например, рассмотрим $\sigma_1 = \{x \mapsto LI(a)\}$, $\sigma_2 = \{x \mapsto LI(b)\}$, $\sigma = \{LI(x).Key \mapsto 42\}$. Тогда:

$$dom(merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle) \circ \sigma) =$$
$$= dom(merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle)) \cup merge(\langle g, \sigma_1\rangle, \langle \neg g, \sigma_2\rangle) \bullet dom(\sigma) =$$
$$= \{x, union(\langle g, LI(a).Key\rangle, \langle \neg g, LI(b).Key\rangle)\},$$

что не то же самое, что

$$dom(merge(\langle g, \sigma_1 \circ \sigma\rangle, \langle \neg g, \sigma_2 \circ \sigma\rangle)) = dom(\sigma_1 \circ \sigma) \cup dom(\sigma_2 \circ \sigma) =$$

$$= \{LI(a).Key, LI(b).Key, x\}.$$

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:

$$LI(union(\langle g_1, x_1\rangle, \ldots, \langle g_n, x_n\rangle)) = union(\langle g_1, LI(x_1)\rangle, \ldots, \langle g_n, LI(x_n)\rangle). \quad (6)$$

Теорема 11. Для любой символьной кучи σ и произвольных непересекающихся ограничений $g_1, \ldots, g_n$, а также любых локаций $x_1, \ldots, x_n$ справедливо следующее утверждение:

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

Теперь необходимо сформулировать вспомогательное утверждение, которое позволит доказать симметричную теорему о композиции с объединением: $merge\langle g_i, \sigma_i\rangle \bullet e = union\langle g_i, \sigma_i \bullet e\rangle$. Однако оно не всегда верно: может случиться так, что $g_1 \vee \ldots \vee g_n \neq \top$, что, в свою очередь, 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 $union(\langle g_1, 42\rangle, \ldots, \langle g_1, 42\rangle) = union\langle g_1 \vee \ldots \vee g_n, 42\rangle \neq 42$. Чтобы указанное выше утверждение выполнялось, необходимо ограничить результат условием существования вычисления $g_1 \vee \ldots \vee g_n$.

Теорема 12. Для любых символьных куч $\sigma_1, \ldots, \sigma_n$, произвольных непересекающихся ограничений $g_1, \ldots, g_n$ и некоторого выражения e справедливо следующее утверждение:

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

Таким образом, нельзя приравнять $merge(\langle g_1, \sigma_1\rangle, \ldots, \langle g_n, \sigma_n\rangle) \circ \sigma$ и $merge(\langle g_1, \sigma_1 \circ \sigma\rangle, \ldots, \langle g_n, \sigma_n \circ \sigma\rangle)$ как теоретико-множественные объекты. Однако следующая теорема показывает, что определение операции чтения позволяет избежать этой проблемы: в теоретико-множественном смысле кучи могут быть не равны как отображения различных множеств ключей, однако с точки зрения операции чтения они будут совпадать.

Теорема 13. Для любых символьных куч $\sigma, \sigma_1, \ldots, \sigma_n$, произвольных непересекающихся ограничений $g_1, \ldots, g_n$ и произвольной локации x справедливо следующее:

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

```
1. Abs:
2. goto {x >= 0 -> Exit}
3. x := -x
4. Exit: halt
```

Листинг 2. Объединение состояний
Listing 2. Merge states

При помощи операции объединения возможно описать состояние программы на лист. 2 следующим образом:

$$\sigma_- = \{x \mapsto -LI(x)\}$$
$$\sigma = merge(\langle LI(x) \geq 0, \epsilon\rangle, \langle \neg(LI(x) \geq 0), \sigma_-\rangle)$$
$$read(\sigma, x) \overset{\text{Teop.11}}{=} union(\langle LI(x) \geq 0, read(\epsilon, x)\rangle, \langle \neg(LI(x) \geq 0), read(\sigma_-, x)\rangle) =$$
$$= union(\langle LI(x) \geq 0, LI(x)\rangle, \langle \neg(LI(x) \geq 0), -LI(x)\rangle).$$

4.5 Запись в символьную кучу

До сих пор были рассмотрены операции с кучами как с готовыми объектами. Для того, чтобы строить кучу из пустого состояния ϵ, необходима операция *записи* в символьную память. Для её определения воспользуемся следующим сокращением: $ite(c, a, b) \stackrel{\text{def}}{=} union(\langle c, a\rangle, \langle \neg c, b\rangle)$.

Определение 8. *Запись символьного значения v в символьную локацию y символьной кучи σ — это символьная куча $write(\sigma, y, v)$, такая что для всех $x \in dom(write(\sigma, y, \cdot)) = dom(\sigma) \cup \{y\}$, $(write(\sigma, y, v))(x) \stackrel{\text{def}}{=} ite(x = y, v, \sigma(x))$.*

Заметим, что инвариант кучи (1) для записей выполняется тривиально. Следующие теоремы показывают, что операция записи сохраняет свойство композициональности относительно других операций.

Теорема 14. Для любой символьной кучи σ, произвольных символьных локаций x, y и любого символьного выражения v справедливо следующее:

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

Теорема 15. Для любых символьных куч σ, σ', произвольной символьной локации y и произвольного символьного выражения v справедливо следующее:

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

Теорема 16. Для любых символьных куч $\sigma_1, \dots, \sigma_n$, любых непересекающихся ограничений $g_1, \dots, g_n$, и произвольной символьной локации y и символьного выражения v справедливо следующее:

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

5. Исчисление символьных куч

Представленные операторы КСП уже позволяют описывать символьные состояния произвольных фрагментов кода без циклов в графе потока управления. Однако идея подстановки из контекстной кучи позволяет нам пойти дальше и определить исчисление, описывающее исполнение произвольных императивных программ с динамической памятью.

5.1 Обобщённые символьные кучи

Для начала определим формальный язык $Heap$ нашего исчисления. Термы языка $Heap$ будем называть *обобщёнными кучами*. Напомним, что Σ — это множество всех символьных куч, которые были определены в предыдущем разделе.

Обобщённая куча может быть либо обычной символьной кучей (из Σ), которую мы будем называть *определённой*, либо объединением обобщённых куч по непересекающимся ограничениям, либо композицией обобщённых куч, либо записью в обобщённую кучу, либо неподвижной точкой цикла в графе потока управления, которую мы будем называть *рекурсивным состоянием* (рис. 7).

$$\begin{aligned} Heap ::= \ & \Sigma \\ \mid \ & Heap \circ Heap \\ \mid \ & merge(\langle guard, Heap\rangle^*) \\ \mid \ & write(Heap, loc, term) \\ \mid \ & Rec(id) \end{aligned}$$

Рис. 7. Обобщённые кучи
Fig. 7. Generalized heaps

Интуитивно, рекурсивные состояния программы — это состояния, зависящие от самих себя, т.е. чтение из которых требует предыдущую версию того же состояния. В общем случае такое состояние нельзя выразить в виде конечной композиции других состояний, поэтому для них вводится новый символ. Идентификатор id в $Rec(id)$ уникальным

образом описывает цикл в графе потока управления: он состоит из метки из исходного кода и некоторого.

Чтобы адаптировать операции КСП, необходимо расширить синтаксис символьных выражений, как показано на рис. 8.

$$\begin{aligned} term ::= \ & arith \mid loc \\ arith ::= \ & \mathbb{N} \mid arith \pm arith \mid -arith \mid LI^{arith}(\boldsymbol{Heap}, loc) \\ & \mid union^{arith}(\langle guard, arith\rangle^*) \\ loc ::= \ & null \mid 0x[0-9]^+ \mid ident \\ & \mid loc.FieldName \mid LI^{loc}(\boldsymbol{Heap}, loc) \\ & \mid union^{loc}(\langle guard, loc\rangle^*) \\ guard ::= \ & \top \mid \bot \mid \neg guard \mid guard \wedge guard \mid guard \vee guard \\ & \mid arith = arith \mid arith < arith \mid loc = loc \end{aligned}$$

Рис. 8. Грамматика обобщённых символьных выражений
Fig. 8. Symbolic generalized terms

На чтение из *определённой* кучи $read(\sigma, x) = LI(x)$ можно смотреть как на следующее сообщение: «в σ недостаточно информации, чтобы узнать x, — требуется дополнительный контекст». Сама операция чтения построена таким образом, что мы можем однозначно сказать, было ли успешно чтение x из σ, — и если было, то предъявить результат. При чтении из *обобщённой* кучи, например, $read(Rec(F), x)$, могут возникнуть сложности. Если циклический фрагмент кода по метке F меняет содержимое x, то мы не можем его корректно прочитать — для этого необходимо заранее знать, сколько раз исполнится F, что в общем случае невозможно. Очевидно также, что мы не можем отбросить $Rec(F)$ и вернуть просто $LI(x)$.

Таким образом, основная идея расширения термов состоит в «запоминании» источника каждого символьного значения (выделено жирным на рис. 8). Такое расширение является корректным, поскольку старые символьные значения $LI(x)$ теперь станут $LI(\epsilon, x)$. Уточнение также может быть расширено: $\tau \bullet LI(\sigma, x) = read(\tau \circ \sigma, \tau \bullet x)$. Это не нарушит свойства КСП, так как по теор. 3, мы имеем $\tau \bullet read(\sigma, x) = read(\tau \circ \sigma, \tau \bullet x)$.

5.2 Правила редукции

Правила редукции символьных куч представлены на рис. 9. Буквами H обозначены элементы языка $Heap$, буквами t — символьные термы. $H[A/X]$ означает одновременную подстановку A вместо X в обобщённую кучу H; здесь A и X могут быть как обобщёнными кучами, так и символьными термами.

Корректность всех правил, кроме (8), обоснована в разд. 4. Правила (7) и (8) представляют собой аналог α-конверсии и β-редукции в λ-исчислении. $Body(x)$ представляет собой описание поведения участка кода, которому соответствует обобщённая куча $Rec(x)$. Если посмотреть на исчисление символьных куч как на некоторый язык программирования, то $Rec(x)$ соответствовало бы имени функции, а $Body(x)$ — её телу. Обобщённую кучу назовём *нередуцируемой*, если к ней нельзя применить ни одного правила за исключением (7). $H \to^n H'$ означает, что куча H редуцируется в H' за n шагов. $H \to^* H'$ означает, что существует $n \geq 0$, что $H \to^n H'$.

Теперь продемонстрируем, как можно представлять символьные состояния циклических фрагментов кода на примере с лист. 3. Так как выход из цикла зависит от получаемого списка p, и его длина заранее неизвестна, то состояние этого фрагмента нельзя представить в виде какой-либо конечной композиции определённых символьных

состояний. Однако можно построить обобщённую кучу $Body(Inc)$, описывающую поведение метки Inc.

$$\frac{t_1 = t_2}{H \to H[t_2/t_1]}\ (7)$$

$$H \to H[Body(id)/Rec(id)]\ (8)$$

$$\epsilon \circ H \to H$$

$$H \circ \epsilon \to H$$

$$H_1 \circ (H_2 \circ H_3) \to (H_1 \circ H_2) \circ H_3$$

$$H \circ merge\langle g_i, H_i\rangle \to merge\langle H \bullet g_i, H \circ H_i\rangle$$

$$merge\langle g_i, H_i\rangle \circ H \to merge\langle g_i, H_i \circ H\rangle$$

$$H \circ write(H', x, v) \to write(H \circ H', H \bullet x, H \bullet v)$$

$$write(merge\langle g_i, H_i\rangle, x, v) \to merge\langle g_i, write(H_i, x, v)\rangle$$

$$\frac{g \text{ невыполнимо}}{merge(\langle g, H\rangle \cup X) \to merge(X)}$$

$$\frac{g \text{ общезначимо}}{merge\langle g, H\rangle \to H}$$

Рис. 9. Правила редукции
Fig. 9. Reduction rules

```
1. Inc:
2. goto {p = null -> Exit}
3. p.Key := p.Key + 1
4. p := p.Next
5. goto {true -> Inc}
6. Exit: halt
```

Листинг 3. Фрагмент кода с циклом в графе потока управления
Listing 3. Code snippet with a cycle in a control flow graph

Пусть σ_0 — некоторая символьная куча, представляющая начальное состояние исполнения, а обобщённая куча $Rec(Inc)$ соответствует метке Inc. Тогда поведение всего кода в лист. 3 на состоянии σ_0 будет описываться обобщённой кучей $\sigma_0 \circ Rec(Inc)$. Применение правил редукции к $\sigma_0 \circ Rec(Inc)$ будет соответствовать вычислению кода с метки Inc на состоянии σ_0. Покажем это на примере.

Пусть $\sigma_0 = \{p \mapsto 0x1, 0x1.Key \mapsto 10, 0x1.Next \mapsto 0x2, 0x2.Key \mapsto 20,\ 0x2.Next \mapsto null\}$.

Опишем теперь поведение циклического региона, помеченного Inc. В начале исполнения происходит ветвление по условию $p = null$. Если $p \neq null$, то стр. 3-5 увеличивают значение ключа в узле связного списка и переходят к следующему элементу. Поведение этого участка кода можно описать символьным объединением двух эффектов: пустого эффекта ϵ (т.к. переход на стр. 2 не меняет состояния) и эффекта σ нерекурсивного кода на стр. 3-4, где $\sigma = \{LI(p).Key \mapsto LI(LI(p).Key) + 1, p \mapsto LI(LI(p).Next)\}$.

Таким образом, поведение региона Inc описывается обобщённой кучей

$$Body(Inc) = merge(\langle LI(p) = null, \epsilon\rangle, \langle LI(p) \neq null, \sigma \circ Rec(Inc)\rangle).$$

Теперь опишем процесс редукции кучи $\sigma_0 \circ Rec(Inc)$.

$\sigma_0 \circ Rec(Inc) \to \sigma_0 \circ Body(Inc) \to$
$merge(\langle \sigma_0 \bullet (LI(p) = null), \sigma_0 \circ \epsilon\rangle, \langle \sigma_0 \bullet (LI(p) \neq null), \sigma_0 \circ (\sigma \circ Rec(Inc))\rangle) \to^4$
$merge(\langle 0x1 = null, \sigma_0\rangle, \langle 0x1 \neq null, (\sigma_0 \circ \sigma) \circ Rec(Inc)\rangle) \to^2 \sigma_1 \circ Rec(Inc) \to^2$

$merge(\langle \sigma_1 \bullet (LI(p) = null), \sigma_1 \circ \epsilon\rangle, \langle \sigma_1 \bullet (LI(p) \neq null), \sigma_1 \circ (\sigma \circ Rec(Inc))\rangle) \to^4$
$merge(\langle 0x2 = null, \sigma_1\rangle, \langle 0x2 \neq null, (\sigma_1 \circ \sigma) \circ Rec(Inc)\rangle) \to^2 \sigma_2 \circ Rec(Inc) \to^2$
$merge(\langle \sigma_2 \bullet (LI(p) = null), \sigma_2 \circ \epsilon\rangle, \langle \sigma_2 \bullet (LI(p) \neq null), \sigma_2 \circ (\sigma \circ Rec(Inc))\rangle) \to^4$
$merge(\langle null = null, \sigma_2\rangle, \langle null \neq null, (\sigma_2 \circ \sigma) \circ Rec(Inc)\rangle) \to^2 \sigma_2$

Здесь

$\sigma_1 \stackrel{\text{def}}{=} \sigma_0 \circ \sigma = \{p \mapsto 0x2, 0x1.Key \mapsto 11, 0x1.Next \mapsto 0x2,$
$0x2.Key \mapsto 20, 0x2.Next \mapsto null\},$
$\sigma_2 \stackrel{\text{def}}{=} \sigma_1 \circ \sigma = \{p \mapsto null, 0x1.Key \mapsto 11, 0x1.Next \mapsto 0x2,$
$0x2.Key \mapsto 21, 0x2.Next \mapsto null\}.$

Обобщённая куча σ_2 не редуцируема (заметим, что нередуцируемым будет любой элемент Σ). σ_0 и σ_1 представляют собой состояния изначальной императивной программы в процессе её исполнения, а σ_2 — её конечное состояние (при запуске на σ_0).

Исчисление символьных куч позволяет описывать произвольные поведения программ с динамической памятью без потери информации.

6. Композициональное символьное исполнение

В данном разделе предложен алгоритм композиционального символьного исполнения, который позволяет автоматически проверять достижимость ошибок при произвольном графе потока управления. Он основан на подходе символьного исполнения программ [22].

6.1 Метод описания путей в графе потока управления

Этот метод является основным в предлагаемом алгоритме, т.к. выполняет описание *всех* путей в графе потока управления. Данный метод, в частности, позволяет *автоматически* строить обобщённые символьные кучи $Body(id)$, описывающие поведения циклических фрагментов программы.

Прямолинейным способом построения обобщённых куч по императивной программе является введение куч $Rec(l)$ для каждой инструкции l и взятие композиции со всеми $Rec(l')$, в которые есть переход из инструкции l. Однако такой подход порождает слишком большую систему взаимно-рекурсивных определений: фактически, количество символов-абстракций $Rec(l)$ было бы равно количеству инструкций в программе. В данном разделе описывается метод описания всех возможных путей исполнения программы через введение меньшего числа символов-абстракций.

Определение 9. *Вершинами* V_G графа потока управления G будут номера l инструкций, а *рёбрами* E_G — пары номеров (l_x, l_y), указывающие на возможность передачи управления от инструкции l_x к инструкции l_y. В графе потока управления существует *начальная* вершина, которая соответствует первой инструкции программы и в которую не ведёт ни одно ребро. Согласно грамматике демо-языка (см. рис. 1), большинство рёбер будут иметь вид $(l, succ(l))$. Однако благодаря оператору **goto** возможны переходы к произвольным инструкциям, кроме начальной.

Определение 10. Проведём обход графа G в глубину и для каждой вершины v вычислим время «выхода» $time(v)$ из обхода. Вершина l называется *рекурсивной*, если существует ребро (l', l) такое, что $time(l') \geq time(l)$. Множество рекурсивных вершин будем обозначать RV.

Для описания путей необходимо ввести операцию *конкатенации* двух путей в графе. Неформально, конкатенация путей p_1 и p_2 — это путь $p_1 \circ p_2$, который содержит рёбра пути p_1, за которыми следуют ребра пути p_2. Конкатенация двух множеств путей P_1 и P_2

определяется через конкатенацию двух путей: $P_1 \circ P_2 = \{p_1 \circ p_2 | p_1 \in P_1, p_2 \in P_2\}$. Символ ε означает *пустой путь*, а символ $\cup$ — объединение множеств путей.

Предлагаемый метод позволяет описывать в точности все пути в произвольном графе потоке управления при помощи рекурсивных символов и их рекурсивных описаний, на базе которых далее будут строиться обобщённые символьные кучи.

$$\Pi(u,v,D) = \bigcup_{(u,v)\in E_G} \{(u,v)\} \cup \bigcup_{\substack{t\notin RV\cup\{v\} \\ (u,t)\in E_G}} (u,t) \circ \Pi(t,v,D) \cup \bigcup_{\substack{t\in RV\setminus(D\cup\{v\}) \\ (u,t)\in E_G}} (u,t) \circ Rec(t, D\cup\{t\}) \circ \Pi(t,v,D\cup\{t\})$$

$$Rec(u,D) = \{\varepsilon\} \cup \Pi(u,u,D) \circ Rec(u,D)$$

Символом $\Pi(u,v,D)$ обозначим множество путей из вершины u в вершину v, параметризованное множеством *пройденных* рекурсивных вершин D. Множество D ограничивает переходы по рёбрам: ребро (l_x, l_y) является «допустимым», если оно ведёт в конечную вершину (т.е. $l_y = v$) или $l_y \neq v$ и вершина l_y не была ещё посещена (т.е. $l_y \notin D$). *Рекурсивным символом* $Rec(u,D)$ обозначим множество путей-циклов из вершины u в вершину u с множеством D, имеющим тот же смысл, как и для $\Pi(u,v,D)$.

Интуитивно, $\Pi(u,v,D)$ соответствует символьным кучам, полученным в результате символьного исполнения программы от инструкции с номером u до инструкции с номером v, когда исполнение не посещало инструкции с номерами из множества $D\backslash\{v\}$. В свою очередь, *рекурсивный символ* $Rec(u,D)$ соответствует *обобщённой символьной куче* $Rec(id)$, у которой уникальным идентификатором id является пара (u,D), а его описание — это обобщённая символьная куча $Body(u,D)$. Кроме того, оператор $\circ$ соответствует операции *композиции* состояний, символ $\cup$ — операции *объединения* состояний ($merge$), а ε — пустой куче ϵ.

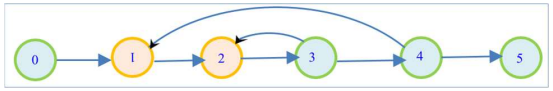

Рис. 10. Пример графа потока управления с вложенными циклами
Fig. 10. An example of a control flow graph with nested loops

Покажем на примере, как можно описать все пути в графе потока управления при помощи итеративного построения Π и Rec. На рис. 10 представлен пример графа потока управления программы с вложенными циклами. Для простоты изложения номер вершины v равен времени $time(v)$. На рис. 10 вершины 1 и 2 являются рекурсивными. Ниже представлено описание всех путей в графе из 0 в 5, использующее рекурсивные символы.

$\Pi(0,5,\varnothing) = (0,1) \circ Rec(1,\{1\}) \circ (1,2) \circ Rec(2,\{1,2\}) \circ (2,3) \circ (3,4) \circ (4,5)$

$Rec(1,\{1\}) = (1,2) \circ Rec(2,\{1,2\}) \circ (2,3) \circ (3,4) \circ (4,1) \circ Rec(1,\{1\}) \cup \{\varepsilon\}$

$Rec(2,\{1,2\}) = (2,3) \circ (3,2) \circ Rec(2,\{1,2\}) \cup \{\varepsilon\}$

$\Pi(0,5,\varnothing)$ обозначает все пути из вершины 0 в вершину 5. При переходе по ребру $(0,1)$ метод попадает в вершину 1. Так как она *рекурсивная* (поскольку лежит в RV), метод вводит для неё символ $Rec(1,\{1\})$ и начинает создавать *рекурсивное* описание этого символа.

Это описание начинается с перехода в вершину 2 и введения рекурсивного символа для неё. Поскольку были пройдены обе *рекурсивные* вершины, то при создании символа $Rec(2,\{1,2\})$ множество $D = \{1,2\}$. Рекурсивное определение для $Rec(2,\{1,2\})$ выглядит следующим образом: все пути из 2 в 2, не проходящие через $D = \{1,2\}$ в середине пути,

— это повторения пути $(2,3) \circ (3,2)$. Кроме того, любое множество $Rec(\cdot)$ путей из себя в себя содержит пустой путь ε.

Далее продолжается описание $Rec(1,\{1\})$: после перехода из 1 в 2 происходит конкатенация пути $(1,2)$ и путей $Rec(2,\{1,2\})$ из 2 в себя, а к множеству D добавляется вершина 2. После этого путь, возвращающийся в вершину 1, очевиден: $(2,3) \circ (3,4) \circ (4,1)$.

Затем процесс возвращается к построению $\Pi(0,5,\varnothing)$. После перехода по ребру $(0,1)$ и создания символа $Rec(1,\{1\})$ к множеству D добавляется вершина 1. Затем происходит переход по ребру $(1,2)$ и добавление соответствующего символа $Rec(2,\{1,2\})$, а также к множеству D добавляется 2. Поскольку описание для символа $Rec(2,\{1,2\})$ уже было построено при построении описания $Rec(1,\{1\})$, то метод не будет строить его заново. Далее следует тривиальный путь до вершины 5: $(2,3) \circ (3,4) \circ (4,5)$.

Таким образом, метод позволяет описывать все пути в графе потока управления и только их. Формальное доказательство этого факта приведено в [27].

6.2 Алгоритм композиционального символьного исполнения

Благодаря соответствию между рекурсивными символами и их описаниями, с одной стороны, и обобщёнными кучами $Rec(id)$ и $Body(id)$, с другой стороны, можно получить *алгоритм автоматической проверки достижимости ошибок в программах с динамической памятью и произвольными графами потока управления*, не раскручивающий отношение перехода.

Предлагаемый алгоритм использует символьное исполнение и операции над символьными кучами (см. лист. 4). Также он использует оракул SAT, проверяющий достижимость веток исполнения. Для данного алгоритма важно понятие *состояния исполнения*.

Определение 10. *Состояние исполнения* — это кортеж (l, pc, σ, D), где l — номер инструкции, pc – *условие пути* (path condition — символьная формула, описывающая ограничения на достижимость инструкции), σ — символьная куча, D — множество посещённых рекурсивных вершин.

Важнейшей функцией алгоритма является $Exec$, которая может быть вызвана либо из *начальной* (стр. 2), либо из *рекурсивной* вершины (стр. 51). В первом случае её результатом является символьная куча, соответствующая путям исполнения, которые привели к инструкциям ***halt*** (стр. 10), а также выводится множество путей, приводящих к ошибкам (стр. 53). Во втором случае результатом является символьная куча $Body(l_0, D_0)$, описывающая поведения циклического участка графа потока управления. В стр. 37-40 происходит добавление *обобщённого состояния*, представляющего собой композицию *построенного состояния* σ' *с рекурсивным состоянием* $Rec(l_0, D_0)$. В конце функции (стр. 52) добавляется завершающее состояние для случая, когда исполнение не вернулось в *рекурсивную* вершину l_0, соответствующее *пустому пути* ε из определения рекурсивных символов в методе описания путей в графе.

Алгоритм выбирает следующее состояние исполнения из рабочего множества ($pickNext$), затем исполняет соответствующую инструкцию, порождая новые состояния исполнения (стр. 9-35), и добавляет их в рабочее множество, объединяя те состояния исполнения, у которых равны номера инструкций l и совпадают множества *посещённых* рекурсивных вершин D (стр. 47-50). Стоит отметить, что предлагаемый алгоритм не раскручивает циклы, а вводит рекурсивные состояния $Rec(l_0, D_0)$ (стр. 12) и обобщённые кучи $Body(l_0, D_0)$ (стр. 46) для их описания. Все обобщённые кучи $Body(\cdot,\cdot)$ используются для определения выполнимости ограничений пути (стр. 21, 23, 31, 34).

```
1  ∀l ∈ RV, ∀D  Body(l, D) ← ϵ;
2  return Exec(start, ∅);
3  Function Exec(l_O : Vertex, D_O : Vertex set)
4      pc_r, σ_r ← (⊥, ϵ);
5      W ← {(l_O, ⊤, ϵ, D_O)}; Errors ← ∅;
6      while W ≠ ∅ do
7          (l, pc, σ, D), W ← pickNext(W);
8          S ← ∅;
9          switch instr(l)
10             case halt:
11                 if l_O ∉ RV then
12                     pc_r ← pc_r ∨ pc;
13                     σ_r ← merge((pc_r , σ_r ), (pc,σ));
14             case fail:  Errors ← Errors ∪ {pc};
15             case ident := expression
16                 σ, value ← Eval(σ,expression);
17                 S ← {(succ(l), pc, write(σ, ident, value))};
18             case Location.field := expression
19                 σ, value ← Eval(σ,expression);
20                 σ, loc ← Eval(σ,Location);
21                 if SAT(Body, σ, pc ∧ loc ≠ null) then
22                     S ← {(succ(l), pc ∧ loc ≠ null, write(σ, loc.field, value))};
23                 if SAT(Body, σ, pc ∧ loc = null) then
24                     Errors ← Errors ∪ {pc ∧ loc = null};
25             case label : statement
26                 S ← {(succ(l), pc, σ)};
27             case goto labels
28                 guard_succ ← ⊤;
29                 forall (expression → l') ∈ labels do
30                     σ, guard ← Eval(σ, expression);
31                     if SAT(Body, σ, pc ∧ guard ∧ guard_succ) then
32                         S ← S ∪ {(l', pc ∧ guard ∧ guard_succ, σ)};
33                     guard_succ ← guard_succ ∧ ¬guard;
34                 if SAT(Body, σ, pc ∧ guard_succ) then
35                     S ← S ∪ {(succ(l), pc ∧ guard_succ, σ)};
36         forall (l', pc', σ') ∈ S do
37             if l' = l_O then
38                 σ' ← σ' ∘ Rec(l_O, D_O);
39                 pc_r, σ_r ← (pc_r ∨ pc', merge ((pc_r, σ_r), (pc' , σ')));
40                 continue;
41             elif l' ∈ D then continue;
42             elif l' ∈ RV then
43                 D ← D ∪ {l'};
44                 σ' ← σ' ∘ Rec(l', D);
45                 if Body(l', D) = (⊥, ϵ) then
46                     Body(l', D) ← Exec(l', D);
47             if ∃(l'', pc'', σ'', D'') ∈ W : l' = l'' ∧ D = D'' then
48                 W ← W \ {(l'', pc'', σ'', D'')};
49                 W ← W ∪ {(l', pc' ∨ pc'', merge( (pc', σ'), (pc'', σ'')) , D)};
50             else W ← W ∪ {(l', pc', σ', D)};
51     if l_O ∈ RV then
52         return merge (pc_r, σ_r), (¬pc_r, ϵ) ;
53     print Errors;
54     return σ_r
```

Листинг 4. Алгоритм композиционального символьного исполнения
Listing 4. Compositional symbolic execution algorithm

Функция $Eval(\sigma, expression)$ вычисляет выражения, трактуя арифметические и булевы операции стандартным образом и читая переменные из состояния σ.

Стоит заметить, что в качестве побочного эффекта $Eval(\sigma, expression)$ может добавить к множеству $Errors$ новое условие пути, защищающее обращение к нулевому адресу. Для вычисления $Eval(\sigma, \textbf{new}\{Field_i \to Expr_i\})$ будет создан новый уникальный адрес

$0xNNN$[3], все $Expr_i$ вычислятся в v_i и состояние будет итеративно обновлено: $\sigma' = write(\sigma, 0xNNN.Field_i, v_i)$. Функция $Eval$ вернёт новое состояние σ' и адрес созданного объекта $0xNNN$. Например, для фрагмента на лист. 5 может быть получено следующее новое состояние:

$$0x40.K \mapsto 30 \quad 0x41.K \mapsto 10 \quad 0x42.K \mapsto 50$$
$$0x40.L \mapsto 0x41 \quad 0x41.L \mapsto null \quad 0x42.L \mapsto null$$
$$0x40.R \mapsto 0x42 \quad 0x41.R \mapsto null \quad 0x42.R \mapsto null$$

```
1.  x = new {K = 30;
2.              L = new {K = 10; L = null; R = null};
3.              R = new {K = 50; L = null; R = null}}
```

Листинг 5. Программа, выделяющая память
Listing 5. Heap-allocating program

6.3 Корректность алгоритма композиционального символьного исполнения

Определение 12. *Замкнутым* назовём терм, не содержащий $LI(\cdot)$.

Конкретная куча — это (тотальное) отображение из замкнутых локаций в замкнутые термы. Множество конкретных куч обозначим за Σ_G.

Конкретная куча представляет собой состояние динамической памяти при конкретном исполнении программы. Заметим, что $(\Sigma_G, \circ)$ — правый идеал в моноиде $(\Sigma, \circ)$: если $\sigma \in \Sigma_G, \tau \in \Sigma$, то $\sigma \circ \tau \in \Sigma_G$. Этот факт позволяет легко доказать следующее утверждение.

Утверждение 2. Если для $\sigma \in \Sigma_G$ и обобщённой кучи H, $\sigma \circ H \to^* H'$ для некоторой нередуцируемой $H' \in Heap$, то $H' \in \Sigma_G$.

Пусть $T: \mathbb{N} \times \Sigma_G \to \mathbb{N} \times \Sigma_G$ — отношение перехода некоторой программы на демо-языке (т.е. отображение, которое номеру инструкции и состоянию программы сопоставляет следующую инструкцию и состояние, полученное исполнением входной инструкции на входном состоянии). Обозначим $T^n(l, \sigma) \stackrel{\text{def}}{=} \underbrace{T(\ldots T}_{n \text{ раз}}(l, \sigma))$.

Следующая теорема (которую мы оставляем без доказательства) говорит о корректности алгоритма на лист. 4.

Теорема 17. Пусть T — отношение перехода программы P на демо-языке, l_0 — номер начальной инструкции, F — множество номеров инструкций ***halt*** и ***fail*** в программе, $\sigma_0 \in \Sigma_G$, $H \stackrel{\text{def}}{=} Exec(l_0, \varnothing)$. Также допустим, что оракул SAT всегда отвечает правильно. Тогда $T^n(l_0, \sigma_0) = (f, \tau)$ для некоторых $n \in \mathbb{N}, f \in F, \tau \in \Sigma_G$ т. и т. т., к. $\sigma_0 \circ H \to^* \tau$.

7. Трансляция символьных куч в чистые функции

Для проверки достижимости некоторого пути исполнения программы, алгоритм из разд. 6 обращается к функции-оракулу SAT. В данном разделе мы определяем $SAT(Body, \sigma, g)$ и тем самым завершаем построение композициональной процедуры верификации. Мы сведём задачу выполнимости ограничения g к задаче доказательства безопасности функциональной программы без эффектов, состоящей из чистых функций второго порядка[4].

[3] В текущем изложении некорректно обрабатывается случай с выделением объекта в циклическом регионе; для корректной работы определение уточнения должно быть изменено: $\sigma \bullet 0xNNN$ должно порождать *новый* адрес $0xMMM$

[4] Функцией *первого порядка* называется функция, которая не принимает в аргументы другие функции. Функцией *второго порядка* называется функция, которая принимает в аргументы функции только первого порядка — и не выше.

7.1 Оператор Find

Этот оператор, определённый в разд. 4.3, играет важную роль в трансляции обобщённых куч в чистые функции, обобщая чтение и композицию символьных куч. Сформулируем его основное свойство.

Утверждение 3. Для всех определённых символьных куч σ, σ', τ таких, что для каждого символьного выражения e, выполняется $(\tau \circ \sigma) \bullet e = \tau \bullet (\sigma \bullet e)$, и всех символьных выражений $x \in loc$, $d \in term$, верно следующее:

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

Далее, можно заметить, что:

$$read(\sigma, x) = find(\sigma, x, \epsilon, LI(x)) = find(\sigma, x, \epsilon, read(\epsilon, x)),$$
$$(\sigma \circ \sigma')(x) = find(\sigma', x, \sigma, \sigma(x)) = find(\sigma', x, \sigma, read(\sigma, x)).$$

Это даёт возможность определить оператор $find$ для обобщённых куч следующим образом (обозначив прежний $find$ как $find^{\Sigma}$):

$$find(\sigma, x, \tau) \stackrel{\text{def}}{=} \begin{cases} find^{\Sigma}(\sigma, x, \tau, find(\tau, x, \epsilon)), & \text{если } \sigma \in \Sigma \\ LI(\tau \circ \sigma, x), & \text{иначе} \end{cases} \quad (10)$$

7.2 Трансляция обобщённых куч в функции второго порядка

Будем говорить, что символьный терм t находится в *нормальной форме*, если он содержит объединения только на верхнем уровне, т.е. $t = union(\langle g_1, t_1\rangle, \ldots, \langle g_n, t_n\rangle)$ и ни одно из ограничений g_i и ни один из термов t_i не содержат внутри $union$. Будем также говорить, что *ограничение* находится в нормальной форме, если оно не содержит объединений. Каждое символьное выражение может быть нормализовано: по утв. 1 и (6), вложенные объединения могут быть линеаризованы. Если t не содержит объединений, тогда его нормальной формой будем называть $union\langle \top, t\rangle$. По определению, ограничение $g \equiv union(\langle g_1, c_1\rangle, \ldots, \langle g_n, c_n\rangle)$ может быть переписано в $(g_1 \wedge c_1) \vee \ldots \vee (g_n \wedge c_n)$.

Рассмотрим символьную ячейку $LI(\sigma, x)$. Заметим, что такие ячейки с $\sigma \neq \epsilon$ появляются только в последней ветке определения (10), т.е. можно рассматривать $LI(\sigma, x)$ как $find(\sigma, x, \epsilon)$. Уточнение такого выражения в контексте τ даст $\tau \bullet find(\sigma, x, \epsilon) \stackrel{\text{утв.3}}{=} find(\sigma, \tau \bullet x, \tau)$. Это даёт возможность транслировать символьные выражения в функции второго порядка.

Далее, с помощью τ обозначим функцию первого порядка «чтение из контекстной кучи». Преобразование символьного выражения e в выражение функционального языка при контекстной куче τ обозначим как $[\![e]\!]_{\tau}$. Это преобразование состоит из трёх следующих шагов.

1. Нормализация e и преобразование верхнеуровневого объединения в конструкцию ветвления.
2. Замена всех ячеек $LI(\sigma, x)$ на $find(\sigma, [\![x]\!]_{\tau}, \tau)$.
3. Специализация оператора $find$ согласно правилам (10). На этом шаге все термы вида $find(\sigma, x, \tau)$ транслируются в применения функций второго порядка find_{σ}. Телом функции find_{σ} будет результат применения этих трёх шагов к соответствующему правилу (10). При появлении композиции $\sigma \circ \sigma'$ контекстное состояние становится частичным применением find_{σ} к текущему контекстному состоянию τ.

Вместо формального описания целевого функционального языка программирования и алгоритма трансляции мы продемонстрируем процесс трансляции на примере. Допустим, необходимо ответить на запрос $SAT(Body, Rec(f), LI(\epsilon, a) * 3 < 17)$. Пусть $Body(f) = merge(\langle c, \epsilon\rangle, \langle \neg c, \sigma \circ Rec(f)\rangle)$, где σ — это некоторая обобщённая куча. Тогда

необходимо проверить выполнимость ограничения $g = (Rec(f) \bullet (LI(\epsilon, a) * 3) < 17) = (LI(Rec(f), a) * 3 < 17)$.

Сначала определим контекстную функцию первого порядка τ_0, которая будет принимать адрес и выполнять ленивое инстанцирование символьных локаций, т.е. возвращать недетерминированные значения. Далее, вычислим $[\![g]\!]_{\tau_0}$.

Первый шаг не порождает условных конструкций. После второго шага выражение g становится следующим: $find(Rec(f), a, \tau_0) * 3 < 17$. Третий шаг порождает новую функцию второго порядка $\text{find}_{Rec(f)}$. Таким образом, закодированное значение g будет: $[\![g]\!]_{\tau_0} = (find_{Rec(f)}\ \tau_0\ a) * 3 < 17$.

Проверить выполнимость g — это то же самое, что проверить безопасность программы “assert(¬ g)”.

Теперь мы должны задать тела полученных функций $find$. Пусть find_f принимает контекстную функцию первого порядка τ и локацию x. Тело функции $\text{find}_{Rec(f)}$ мы получим, применяя шаги 1-3 к $Body(f)$:

$$find(merge(\langle c, \epsilon\rangle, \langle \neg c, \sigma \circ Rec(f)\rangle), x, \tau) \stackrel{(10)}{=} ite(\tau \bullet c, find(\epsilon, x, \tau), find(\sigma \circ Rec(f), x, \tau))$$

Это объединение будет нормализовано и транслируется в ветвление в теле $\text{find}_{Rec(f)}$; ленивые ячейки в c заменятся применениями $find$, которые будут также специализированы. Итеративное применение этих шагов даст код для g, представленный ниже.

Идея такой трансляции заключается в том, что операции композиции могут быть заменены частичными применениями функций. Это позволяет сохранять справедливость того факта, что контекстные функции не поднимаются выше первого порядка. Таким образом получается трансляция в чистые функции второго порядка.

```
1. assert(not((find_Rec(f) τ a) * 3 < 17))
2. find_Rec(f) τ x =
3.   if [[g]]_τ then find_ε τ x
4.   else find_σ∘Rec(f) τ x
5. find_ε τ x = τ x
6. find_σ∘Rec(f) τ x = find_Rec(f) (find_σ τ) x
7. find_σ τ x = …
```

Замечание. Есть несколько способов улучшить эту трансляцию. Во-первых, можно специализировать не только по куче, но и по типу локации, что даст более специализированные функции. Это также необходимо, чтобы получить из алгоритма трансляции типобезопасные функции. Во-вторых, полученная программа может быть частично исполнена, чтобы удалить тривиальные чтения, как, например, find_{ϵ}. В-третьих, именованные локации могут передаваться как обычные аргументы, так как их адреса никогда не меняются. Эти три улучшения позволяют получить автоматическую трансляцию, результаты которой будут схожи с приведённой в [27] (прил. А).

7.3 Корректность трансляции в чистые функции

Следующая теорема (которую мы оставляем без доказательства) говорит о корректности алгоритма из разд. 7.2.

Теорема 18. Пусть $H \in Heap$, $\sigma_0 \in \Sigma_G$, g — символьное ограничение. Тогда $\tau \bullet g$ выполнимо для некоторого $\tau \in \Sigma_G$, такого что $\sigma_0 \circ H \to^* \tau$, т. и т. т., к. небезопасна

функциональная программа, полученная из $\sigma_0 \circ H$ и g применением алгоритма кодирования обобщённых куч из разд. 7.2.

Таким образом, сводя воедино корректность символьного исполнения и кодирования, получаем следующую теорему.

Теорема 19. Пусть $isSafe(p)$ — оракул, проверяющий безопасность функциональных программ, который всегда возвращает верный результат. Тогда программа на демонстрационном языке безопасна т. и т. т., к. алгоритм с лист. 4 выводит $Errors = \varnothing$.

Доказательство. Следует из теор. 17 и теор. 18.

7.4 Верификация функциональных программ без эффектов

Для доказательства безопасности функциональных программ без эффектов можно применить различные классические техники. Одной из самых успешных является *вывод уточнённых типов* (refinement type inference) [10, 23, 24, 25]. Фреймворки вывода уточнённых типов строят индуктивные инварианты функциональных программ высших порядков из инвариантов первого порядка над значениями с закрытыми типами (ground-types). Более точное описание этого процесса содержится в [25].

Тот факт, что получаемые в результате нашей трансляции функции не выше второго порядка, позволяет специализировать и оптимизировать процедуру вывода уточнённых типов. В контексте нашей работы, наиболее интересны *композициональные* фреймворки вывода уточнённых типов. Примером такого фреймворка является [24].

8. Заключение

В работе представлен подход к композициональному точному анализу программ с динамической памятью. Подход сводит задачу доказательства корректности таких программ к задаче вывода уточняющих типов функциональных программ через построение обобщённых куч, описывающих эффект программы на произвольном состоянии. Имея хорошую модель памяти [26], подход легко адаптировать к промышленным языкам программирования, и тем самым свести задачу формальной верификации программ на таких языках к решению рекурсивно-логических соотношений. Построение практичного верификатора языка C#, основанного на представленном подходе, будет описано в следующих работах. Модель композициональной символьной памяти может также служить хорошим подспорьем для языка спецификации свойств императивных программ с динамической памятью.

Вне контекста формальной верификации работа может рассматриваться как построение интересного соответствия между императивными программами с динамической памятью и чистыми функциями: описанное в статье сведение способно по императивной программе с динамической памятью породить эквивалентную программу на чистом функциональном языке, причём порождённые функциональные программы неочевидным образом кодируют операции над динамической памятью, а композициональное сведение позволяет порождать код функций, не зависящий от её отдельных вызовов.

Предложенный в статье подход выглядит перспективным при обеспечении качества встроенных систем реального времени [28], а также при разработке операционных систем реального времени [29]. Также перспективным является интеграция данного подхода со средствами визуального моделирования ПО, в частности, при верификации исполняемых визуальных спецификаций [30].

Список литературы / References

[1]. Distefano D. Attacking large industrial code with bi-abductive inference. Lecture Notes in Computer Science, vol. 5825, 2009, pp. 1–8.

[2]. Calcagno C. et al. Compositional shape analysis by means of bi-abduction. Journal of the ACM, vol. 58, no. 6, 2011, p. 26:1-26:66.

[3]. Gurfinkel A. et al. The SeaHorn verification framework. Lecture Notes in Computer Science, vol. 9206, 2015, pp. 343–361.

[4]. Anand S., Godefroid P., and Tillmann N. Demand-driven compositional symbolic execution. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 367–381.

[5]. Distefano D. and Parkinson J M. J. jStar: Towards practical verification for Java. ACM Sigplan Notices, vol. 43, issue 10, 2008, pp. 213– 226.

[6]. Calcagno C. and Distefano D. Infer: An automatic program verifier for memory safety of C programs. Lecture Notes in Computer Science, vol. 6617, 2011, pp. 459–465.

[7]. Tillmann N. and De Halleux J. Pex–white box test generation for. net. Lecture Notes in Computer Science, vol. 4966, 2008, pp. 134– 153.

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

[9]. Khurshid S., Păsăreanu C. S., and Visser W. Generalized Symbolic Execution for Model Checking and Testing. Lecture Notes in Computer Science, vol. 2619, 2003, pp. 553–568.

[10]. Vazou N., Bakst A., and Jhala R. Bounded refinement types. ACM SIGPLAN Notices, vol. 50, issue 9, 2015, pp. 48–61.

[11]. Godefroid P. Compositional Dynamic Test Generation. ACM SIGPLAN Notices, vol. 42, issue 1, 2007, pp. 47–54.

[12]. Biere A. et al. Symbolic Model Checking without BDDs. Lecture Notes in Computer Science, vol. 1579, 1999, pp. 193–207.

[13]. Deng X., Lee J. et al. Efficient and Formal Generalized Symbolic Execution. Automated Software Engineering, vol. 19, issue 3, 2012, pp. 233–301.

[14]. Braione P., Denaro G., and Pezz`e M. JBSE: A symbolic executor for java programs with complex heap inputs. In Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 1018–1022.

[15]. Reynolds J. C. Separation logic: A logic for shared mutable data structures. In Proc. 17th Annual IEEE Symposium on Logic in Computer Science, 2002, pp. 55–74.

[16]. Sagiv M., Reps T., and Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems, vol. 24, issue 3. 2002, pp. 217–298.

[17]. Berdine J., Calcagno C., and O'Hearn P.W. Symbolic execution with separation logic. Lecture Notes in Computer Science, vol. 3780, 2005, pp. 52–68.

[18]. Dudka K., Peringer P., and Vojnar T. Byte-precise verification of lowlevel list manipulation. Lecture Notes in Computer Science, vol. 7935, 2013, pp. 215–237.

[19]. Barrett C. and Tinelli C. Satisfiability modulo theories. In Handbook of Model Checking, Springer, 2018, pp. 305–343.

[20]. Kahsai T. et al. Quantified heap invariants for object-oriented programs. In Proc. of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, 2017, pp. 368– 384.

[21]. Torlak E. and Bodik R. A lightweight symbolic virtual machine for solver-aided host languages. ACM SIGPLAN Notices, vol. 49, issue 6, 2014, pp. 530–541.

[22]. Baldoni R. et al. A survey of symbolic execution techniques. ACM Computing Surveys, vol. 51, no. 3, 2018, pp. 50:1-50:39.

[23]. Unno H., Terauchi T., and Kobayashi N. Automating relatively complete verification of higher-order functional programs. ACM SIGPLAN Notices, vol. 48, issue 1, 2013, pp. 75-86.

[24]. Zhu H. and Jagannathan S. Compositional and lightweight dependent type inference for ML. Lecture Notes in Computer Science, vol. 7737, 2013, pp. 295–314.

[25]. Cathcart Burn T., Ong C.-H.L., and Ramsay S. J. Higher-order constrained horn clauses for verification. Proceedings of the ACM on Programming Languages vol. 2, no. POPL, 2017, p. 11:1-11:27.

[26]. Мандрыкин М.У., Мутилин В.С.. Обзор подходов к моделированию памяти в инструментах статической верификации. Труды ИСПРАН, том 29, вып. 1, 2017 г., стр. 195-230 / Mandrykin M.U., Mutilin V.S. Survey of memory modeling methods in static verification tools. Trudy ISP RAN / Proc. ISP RAS, vol. 29, issue 1, 2017, pp. 195-230 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-12.

[27]. Костюков Ю. О., Батоев К. А., Мордвинов Д. А., Костицын М. П., Мисонижник А. В. Автоматическое доказательство корректности программ с динамической памятью. arXiv:1906.10204, 2019 г. / Kostyukov Yu.O., Batoev K.A., Mordvinov D.A., Kostitsyn M.P., Misonizhnik A.V.. Automatic verification of heap-manipulating programs. arXiv:1906.10204, 2019 (in Russian).

[28]. Терехов А.Н. Технология программирование встроенных систем. автореферат дис. доктора физико-математических наук. Новосибирск, 1991 г. / Terekhov A.N. Technology for programming of embedded systems. Abstract of Thesis for obtaining the degree of Doctor of physical and mathematical sciences. Novosibirsk, 1991 (in Russian).

[29]. Новиков Е.М. Развитие ядра операционной системы Linux. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 77-96 / Novikov E.M. Evolution of the Linux kernel. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, 2017, pp. 77-96 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-3.

[1]. Ольхович Л.Б., Кознов Д.В. Метод автоматической валидации UML-спецификаций на языке OCL. Программирование, том 29. № 6, 2003 г., стр. 44-50 / Ol'khovich L., Koznov D.V. OCL-based automated validation method for UML specifications. Programming and Computer Software, vol. 29, № 6, 2003, pp. 323-327.

Информация об авторах / Information about authors

Юрий КОСТЮКОВ получил степень бакалавра в области информационных технологий в Санкт-Петербургском государственном университете в 2019 г. Его исследовательские интересы включают автоматическую верификацию, теорию моделей и конструктивную математику.

Yurii KOSTYUKOV received the bachelor’s degree in information technology from Saint Petersburg State University in 2019. His research interests include automatic verification, model theory and constructive mathematics.

Константин БАТОЕВ получил степень бакалавра в области информационных технологий в Санкт-Петербургском государственном университете в 2019 г. В число научных интересов входят анализ графов потока управления и символьное исполнение.

Konstantin BATOEV received the bachelor’s degree in information technology from Saint Petersburg State University in 2019. His research interests include control flow graph analysis and symbolic execution.

Дмитрий МОРДВИНОВ работает старшим преподавателем в Санкт-Петербургском государственном университете. В настоящее время он готовит диссертацию на соискание степени PhD в области информационных технологий. Область его научных интересов включает формальную верификацию, синтез программ и решение систем дизъюнктов Хорна.

Dmitry MORDVINOV is working as senior lecturer in Saint Petersburg State University. He is currently pursuing the Ph.D. degree in information technology. His area of interest is formal verification, program synthesis and Horn clauses solving.

Михаил КОСТИЦЫН получил степень бакалавра в области информационных технологий в Санкт-Петербургском государственном университете в 2019 г. Его исследовательские интересы включают компьютерную безопасность, анализ строковых выражений и анализ динамической памяти.

Michael KOSTITSYN received the bachelor’s degree in information technology from Saint Petersburg State University in 2019. His research interests include computer security, string and heap analysis.

Александр МИСОНИЖНИК получил степень бакалавра в области информационных технологий в Санкт-Петербургском государственном университете в 2018 г. В число научных интересов входят системы типов, интуиционистская логика и теория категорий.

Aleksandr MISONIZHNIK received the bachelor’s degree in information technology from Saint Petersburg State University in 2018. His research interests include type systems, intuitionistic logic and category theory.