Санкт-Петербургский Государственный
Университет
Математико-механический факультет
Кафедра системного программирования

Применение предикативной абстракции программы для решения потоковых задач

Дипломная работа студента 544-а группы
Буре Дмитрия Владимировича

Научный руководитель	..... (Друнин А.В.)
Рецензент к.ф.-м.н.	..... (Булычев Д.Ю.)
“Допустить к защите”, заведующий кафедрой, профессор, д.ф-м.н.	..... (Терехов А.Н.)

Санкт-Петербург
2007

Содержание

1 Введение

В современной теории компиляции важную роль играют задачи анализа потока данных. А именно, анализа изменения значения переменных в процессе выполнения программы. Помимо классических задач на практике, например в реинжиниринге [3], часто возникают смежные проблемы, имеющие не столь формализованные подходы к решению и требующие разработки новых алгоритмов. Часть алгоритмов применяется однократно, во время первичного анализа программы, другая позволяет пользователю анализировать структуру исходной программы динамически, что накладывает дополнительные требования к производительности.

Несмотря на различия в решении некоторых потоковых задач многие из них используют общие или близкие подходы. Так, например, существуют известные алгоритмы статического определения значения переменных в точке, хотя далеко не всегда необходимо собирать информацию обо всех переменных программы. В тоже время возникают задачи проверки валидности указателей и определения ограничений на входные параметры необходимые для достижимости выбранной точки. Несмотря на схожие постановки задач, каждая из них на практике требует разработки отдельного, независимого алгоритма.

Целью данной работы является обобщение некоторых задач анализа потока данных и предложения механизма их решения. Кроме того, в процессе работы необходимо было проанализировать возможность применения некоторых методов, распространенных в узких областях статического анализа. Так идеи проверок на модели [21] и абстрактной интерпретации программ [11] используются, в основном, как теоретическое обоснование алгоритмов анализа потока данных, а диаграммы двоичных решений [10] применяются в задачах достигающих определений и распределения регистров [2], но совершенно не упоминаются в контексте протягивания констант.

В главе 2 приведен обзор существующих подходов к решению потоковых задач. Особое внимание уделяется способам промежуточного представления анализируемых программ. Глава 3 содержит теоретическое описание основных идей предложенного подхода и используемой абстракции. В главе 4 приведено описание разработанного алгоритма на примере решения задачи протягивания констант. Глава 5 подводит итог дипломной работы, выявляя особенности предложенного механизма.

2 Обзор существующих подходов

В общем случае алгоритм статического анализа программ принимает на вход дерево разбора программы. При этом, перед проведением непосредственно вычислений необходимо собрать информацию о логике программы. Под логикой программы, в зависимости от типа поставленной задачи, может пониматься как достаточно очевидная информация, например, о последовательности выполнения операторов или использовании переменных, которая явно следует из дерева разбора, так и более сложная, требующая предварительного анализа, например, набор определений, достигающих некоторую точку.

Таким образом, алгоритмы анализа программ оперируют не с самой программой, а с некоторой ее абстракцией или представлением.

2.1 Представление на основе графов

Наиболее распространенным представлением программы является *граф потока управления* [1, 20, 4], отражающий порядок выполнения операторов. Узел графа представляет собой оператор программы (или её линейный участок), направленное ребро - передачу управления. Таким образом задаются все возможные пути исполнения программы. Граф потока управления используется для нахождения недостижимого кода программы и для итеративного решения потоковых задач.

На основе графа потока управления и информации об использованных переменных строится *множество достигающих определений* [1, 20] для каждого оператора программы. Определение d некоторой переменной x считается достигающим точки p, если имеется путь от точки d до точки p, такой, что определение d на нем не переприсваивается, т.е. d является последним присваиванием переменой x на этом пути.

Множество достигающих определений можно представить в виде набора *цепочек присваивание-использование* (def-use chains). Анализ достигающих определений и построение таких цепочек требует решения соответствующей потоковой задачи, но полученные результаты могут быть использованы для построения более подробных абстракций программы и решения более сложных задач.

Например, граф, вершинами которого являются экземпляры переменных программы, а ребра расставлены на основе def-use цепочек, называется *графом зависимости по данным (data dependence graph)* [21]. Граф зависимости по данным является транзитивным замыканием def-use цепочек. Типичное применение такой структуры является итеративное решение задачи протягивания констант [26]. Итеративно протягивая

значения переменных от инициализирующих присваиваний вдоль ребер графа зависимости по данным можно получить возможные значения переменных во всех интересующих точках программы.

Несколько иным подходом является объединение информации о потоке данных и потоке управления. Например, в работе [15] предлагается использовать *граф программных зависимостей (program dependence graph)* для эффективной реализации оптимизирующих алгоритмов. Узлами такого графа являются операторы, связанные двумя типами ребер: управления и данных.

Отношение управления возникает между операторами X и Y в том случае, если существует путь P из X в Y, в котором встречается узел Z являющийся постдоминатором Y, при этом X не является постдоминатором Y.

Определение 1 *Вершина w называется постдоминатором для v, если любой путь в графе с началом в v содержит вершину w. [17]*

Отношение по данным возникает между операторами X и Y в том случае, если изменение порядка выполнения этих операторов ведет к изменению значения хотя бы одной переменной из Y. Такая структура оказывается эффективной для задач, в которых одновременно используется информация о порядке выполнения операторов и потоке данных, например раскрутка циклов. Кроме того, в отличие от других итеративных потоковых алгоритмов, решение, основанное на графе программных зависимостей, позволяет проводить инкрементальные оптимизации по мере изменения потока данных программы.

Другим примером аналогичного подхода может служить работа [22], в которой рассматривается исполнимая графовая структура *граф потоковых зависимостей (dependence flow graph)*, расширяющая понятие графа зависимости по данным. Под исполнимостью авторы понимают возможность описания семантики программы с помощью предложенной структуры. Для этой цели вводится несколько типов ребер: функциональные, потоковые, императивные и т.п. Кроме того, исходный язык расширяется двумя дополнительными операторами store и load, соответственно сохраняющие и считывающие значения из заданной области памяти. Результатом работы является реализация алгоритма протягивания констант но основе графа потоковых зависимостей.

2.2 Символические вычисления

Применение графовых структур представления программы позволяет использовать некоторые дополнительные методы для анализа пере-

менных, например символические вычисления.

Определение 2 *Символическими вычислениями называется манипуляция уравнениями и выражениями в символической форме без вычисления конкретных значений представляемыми этими символами*

Классическим применением символических вычислений является автоматическое доказательство теорем и верификация программ. Однако, проведение синтаксических манипуляций выражений позволяет определять эквивалентность некоторых конструкций программы, что может быть использовано для удаления избыточных вычислений. Кроме того, мы можем ассоциировать переменные программы с выражениями определяющими их значений и проводить подстановку или упрощение выражений без непосредственного вычисления значений [19].

Существует большое количество работ связанных с использованием возможностей символических вычислений на структуре данных, представляющей собой объединение потока управления и потока данных. Например, для *графа системных зависимостей (system dependence graph)* [8] был предложен алгоритм сведения, основанный на символических вычислениях, позволяющий решать задачу протягивания констант за счет редукции графа.

Граф системных зависимостей использует три типа ребер. Ребра потока управления могут исходить из первого оператора программы, из операторов ветвления или из оператора вызова процедуры и отражают последовательность выполнения операторов. Ребра потока данных отражают зависимость присваивание - использование или задают порядок присваиваний. Третий тип ребер отвечает за передачу управления и контекста при вызове процедур (для этой же цели вводятся дополнительные вершины входящего и выходящего контекста процедуры). Алгоритм сведения заключается в последовательном удалении вершин без входных ребер и модификациях соседних операторов в зависимости от типа удаленного узла. Например, в случае присваивания, использование соответствующей переменной заменяется на правую часть этого присваивания во всех операторах, доступных из данного по ребру потока данных. Для операторов ветвления, в зависимости от истинности (ложности) условий, удаляются вершины, доступные из данного по ложному (истинному) потоковому ребру. На рисунке 1, приведен пример программы, его графа системных зависимостей и итераций алгоритма сведения.

Еще большее применение символические вычисления получили в работах [25, 9]. В качестве представления программы авторы используют *граф потока значений (value flow graph)*, узлами которого являются

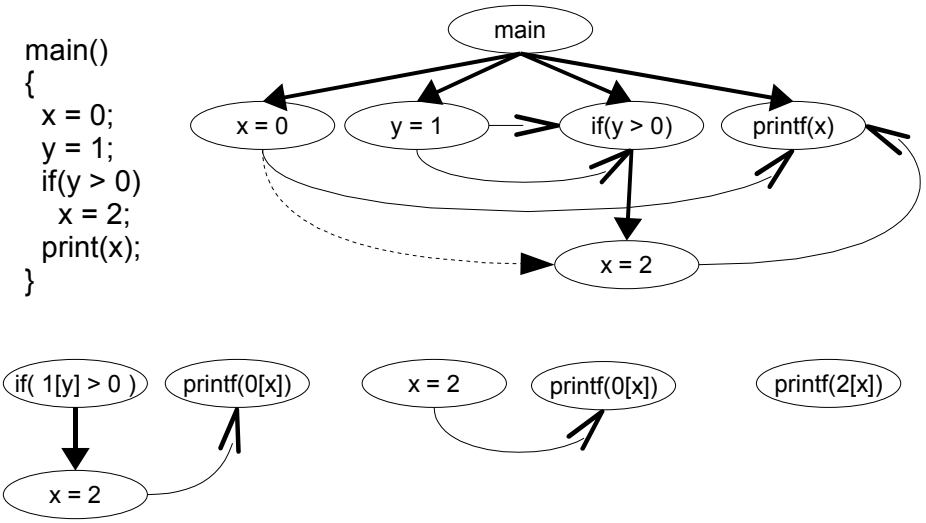


Рис. 1: Исходная программа, граф системных зависимостей, итерации алгоритма сведения

все операторы вычислений программы. В работах предлагается алгоритм оперирующий синтаксическими конструкциями языка для определения семантической эквивалентности вычислений производимых в разных точках программы. Семантическая эквивалентность позволяет разбить выражения программы на классы эквивалентности и выявить семантические зависимости между операторами присваивания. Достоинством такого подхода является его бо́льшая точность по сравнению со стандартными алгоритмами, например при анализе переменных цикла.

2.3 Static Single Assignment Form

Большинство приведенных выше графовых представлений используется для сбора конечной информации о программе (например, значений переменных). Существуют также промежуточные представления программы, используемые для сбора некоторой предварительной информации или улучшения работы последующих анализов. Примером является *Static Single Assignment (SSA)* [12] форма программы. После преобразования программы в SSA форму она приобретает два важных свойства:

1. Каждое использование переменной достижимо ровно из одного присваивания.

2. Программа содержит φ-функции, которые различают значения переменных, приходящие по разным ребрам графа потока управле-

ния.

Первое свойство SSA формы используется для решения задачи протягивания констант с удалением неисполнимых веток [26]. Без использования SSA формы информация о потоке данных должна быть пересчитана после удаления каждого блока, что приводит к высокой сложности алгоритма. Второе свойство формы позволяет проводить глобальную нумерацию значений, которая используется для удаления избыточных вычислений и для определения эквивалентности программ.

i:=0	i_1:=0
l:=1	l_1:=1
repeat	repeat
{	{
	$l_2 := \varphi(l_5, l_1)$
if(Q)	$if(Q)$
l:=i	$l_3 := i_1$
else	else
l:=3	$l_4 := 3$
	$l_5 := \varphi(l_3, l_4)$
}	}
until (T)	until (T)

Таблица 1: Исходная программа, программа в Static Single Assignment форме

φ-функция имеет вид $U = \varphi(V, W, ...)$, где U,V,W,... переменные, а операторы изменяющие значения переменных V,W,... - предшественники узла графа потока управления с φ-функцией. Переменные записываются в определенном порядке, а именно j-й операнд φ-функции соответствует j-ому предшественнику. Если управление переходит к φ-функции по j-ому предшественнику, то φ-функция динамически интерпретируется как присваивание j-ого операнда в переменную U. Каждое исполнение φ-функции использует только один операнд, но какой именно определяется потоком управления непосредственно перед φ. В таблице 1 приведен пример программы в SSA форме.

Говорят, что программа находится в *минимальной SSA форме*, если количество φ-функций в ней минимально. Теоретически, достаточно использовать любую SSA форму программы для дальнейшей работы, но большое число φ-функций приводит к большему количеству дополни-

тельных вычислений. Исходя из этого необходимо добавлять φ-функции только если их наличие действительно необходимо.

Классический алгоритм вычислений φ-функций [12] основан на вычислении границы доминаторов (dominance frontier). Сложность такого алгоритма зависит от размера границы доминаторов, которая в большинстве случаев линейна, но в худшем случае может быть квадратична по сравнению с числом узлов графа потока управления.

Позже в работе [13] был предложен алгоритм, который вместо вычисления границы доминаторов использует алгоритм сокращения сбалансированных путей (balanced path compression algorithm) Тарьяна и алгоритм нумерации узлов обходом в глубину.

Наиболее удачный алгоритм вычисляющий положение φ-функции за линейное время [24] основан на построении и обходе специального представления программы - *Dominator-Join графа*, которое является дополнением дерева доминаторов ребрами потенциально объединяющими информацию о потоке данных.

3 Предлагаемый подход

В главе 2 было рассмотрено несколько распространенных подходов к решению потоковых задач. Заметим, что любой используемый алгоритм основан на некотором представлении программы, будь то дерево разбора, графы зависимостей или SSA форма, а вычисления проводятся над некоторой абстрактной интерпретацией программы [11]. Обобщим это соображение.

3.1 Абстрактная интерпретация

Определение 3 *Абстрактной интерпретацией программы называется устойчивая аппроксимация семантики программы, основанная на монотонных функциях над упорядоченными множествами.*

Иными словами, это частичное исполнение программы для сбора необходимой информации без проведения всех вычислений. Принято считать, что абстрактная интерпретация программ суть теория статического анализа, в то время как анализ потока данных является её практическим применением. Далее мы попытаемся описать наш вариант использования абстрактной интерпретации для потоковых задач.

Рассмотрим простой пример:

```
read x;
while(even x)
{
  x := x div 2;
  x := succ x;
}
```

Построим *конкретную интерпретацию программы*, т.е. множество всех возможных путей исполнения программы для всех возможных входных параметров из области определения (в данном случае множество натуральных чисел). Состояние будем записывать как $n \vdash p$. Представление интерпретации в виде дерева будем называть *деревом вычисления программы.*

$Val = N$
$2n \vdash even(x) \rightarrow 2n \vdash x := x \ \ div \ \ 2$
$2n + 1 \vdash even(x) \rightarrow 2n + 1 \vdash exit$

$2n \vdash x := x \ \ div \ \ 2 \rightarrow n \vdash x := succ \ \ x$
$2n + 1 \vdash x := x \ \ div \ \ 2 \rightarrow n \vdash x := succ \ \ x$
$n \vdash x := succ \ \ x \rightarrow n + 1 \vdash even(x)$

Преобразуем полученные выражения таким образом, чтобы переход между состояниями основывался на операциях с символическим представлением, а не со значениями переменных времени исполнения. Область определения Val будет представлять из себя множество $\{e, o\}$.

$Val = \{e, o\}$
$e \vdash even(x) \rightarrow e \vdash x := x \ \ div \ \ 2$
$o \vdash even(x) \rightarrow o \vdash exit$
$v \vdash x := x \ \ div \ \ 2 \rightarrow e \vdash x := succ \ \ x$
$v \vdash x := x \ \ div \ \ 2 \rightarrow o \vdash x := succ \ \ x$
$v \in \{e, o\}$
$e \vdash x := succ \ \ x \rightarrow o \vdash even(x)$
$o \vdash x := succ \ \ x \rightarrow e \vdash even(x)$

В результате мы получили пример абстрактной интерпретации программы. Введем смежное понятие коллекционной семантики.

Определение 4 *Коллекционной семантикой в общем случае назовем информацию, собранную из узлов и путей дерева вычислений.*

На практике часто необходима информация об истинности некоторого свойства программы на входе или в определенной точке. Имея дерево t и условие φ мы пишем $t \models \varphi$ если φ верно. В таком случае коллекционная семантика будет выглядеть как $coll_t = \{\varphi |\ t \models \varphi\}$

Определение 5 *Верификация моделей (model checker) – это набор идей и методов для построения моделей работающих программ, математической формулировки требований к ним, отражающих правильность их работы или некоторые свойства, а также создания алгоритмов для формальной проверки выполнения этих требований. Верификация моделей может быть применена к любой схеме абстрактной интерпретации*

Коллекционная семантика может быть использована в качестве модели программы. Как было замечено, верификация модели применима к любой абстрактной интерпретации программы, в том числе к дереву вычисления программы. Тогда проверка свойств коллекционной семантики эквивалентна анализу потока данных. Для доказательства этого факта в

работе [23] предлагается формулировать задачи анализа потока данных в терминах μ -вычислений.

Например рассмотрим *задачу определения популярных выражений.* Выражение A op B называется популярным в точке p, если оно используется на всех путях выходящих из p перед тем, как одна из переменных изменит значение. Для итеративного решения основанного на битвекторах используется формула:

$$VBE(p) = Used(p) \cup (notModified(p) \cap (\bigcap_{p' \in succ(p)} VBE(p')))$$

В терминах модальных μ -вычислений задачу можно переформулировать следующим образом:

$$isVBE(e) = \nu\ Z.isUsed(e) \vee (\neg isModified(e) \wedge \triangle Z)$$

Где для состояния s, утверждения ψ и идентификатора Z,

$s \models \nu Z.\psi$, если $\forall i \geq 0, s \models \phi_i$, где $\psi_0 = true; \psi_{i+1} = [\psi_i/Z]\psi$
$s \models \triangle\psi$, если $\forall s_i$ таких, что $s \rightarrow s_i, s_i \models \psi$

Используя приведенную формализацию задачи авторы доказывают ее эквивалентность верификации модели.

3.2 Предикативная абстракция

Далее в этой главе мы будем говорить об одном из типов абстрактных интерпретации, так называемой предикативной абстракции программы [5]. Рассмотрим программу P и множество $E = \{\varphi_1, ..., \varphi_n\}$ простых логических выражений над переменными из P и константами языка.

Определение 6 *Предикативная абстракция программы (булева программа [6]), обозначаемая BP(P,E), строится на основе исходной, последовательной трансляцией всех операторов, т.е. имеет поток управления идентичный исходной P, но содержит только логические переменные. Точнее BP(P,E) содержит n логических переменных* $V = \{b_1, ..., b_n\}$*, где каждая переменная* b_i *представляет собой предикат* $\varphi_i \in E$ $(1 \leq i \leq n)$*. Множество путей исполнения BP(P,E) является надмножеством множества путей исполнения P.*

Переменными булевой программы являются логические выражения полученные на основе операторов присваиваний и контролирующих

условий операторов перехода. То есть, интерес представляют лишь операторы исходной программы влияющие на поток управления и поток данных. Для задания семантики этих операторов мы будем использовать понятие слабейшего предусловия.

3.2.1 Слабейшее предусловие

Определение 7 *$WP(s, \varphi)$ называется слабейшим предусловием предиката φ учитывающим оператор s, если $WP(s, \varphi)$ является слабейшим предикатом, чья истинность перед s ведет к истинности φ после выполнения s.*

Например, для вычисления $WP(x = e, \varphi)$ необходимо заменить все использования x в φ на e:

$$WP(x = x + 1, x < 5) = (x + 1) < 5 = (x < 4)$$

Предположим, оператор s находится между двумя точками программы p и $\acute{p}$. Если предикат $\varphi \in E$ соответствует переменной b, то мы можем положить, что b истинно в BP(P,E) между точками p и $\acute{p}$ если булева переменная $\acute{b}$ соответствующая $WP(s, \varphi)$ истинна в точке p. Однако переменная $\acute{b}$ может не существовать, если $WP(s, \varphi)$ не принадлежит E.

Например, для $E = \{(x < 5), (x = 2)\}$, $WP(x = x + 1, x < 5) = (x < 4) \notin E$. В таком случае необходимо усилить WP до выражения над предикатами из E. $(x = 2) \Rightarrow (x < 4)$, поэтому если $(x = 2)$ верно перед оператором $x = x + 1$, то $(x < 5)$ верно после.

Формализуем усиление предикатов. Для переменной $b_i \in V$, положим $\epsilon(b_i)$ указывающий на соответствующий предикат φ_i, а $\epsilon(\neg b_i)$ указывающий на предикат $\neg\varphi_i$. Естественным образом расширим ϵ до дизъюнктивной нормальной формы. Для каждого предиката φ и набора булевых переменных V, положим $F_v(\varphi)$ эквивалентным наибольшей дизъюнктивной нормальной формы $(c_{j_1} \vee ... \vee c_{j_l})$ над V, такой что $\epsilon(c_{j_i})$ имеет следствием φ. Тогда предикат $\epsilon(F_v(\varphi))$ является слабейшим предикатом на $\epsilon(V)$ из которого следует φ.

Например, $\epsilon(F_v(x < 4)) = (x = 2)$.

Определим также $G_v(\varphi)$ как $\neg F_v(\neg\varphi)$. Предикат $\epsilon(G_v(\varphi))$ определяет сильнейший предикат над $\epsilon(V)$ которые следует из φ.

3.2.2 Абстракция присваиваний

Рассмотрим оператор присваивания $x := e$ в точке l программы P. BP(P,E) будет содержать в точке l одновременные присваивания в буле-

вы переменные в области видимости l. Булева переменная b_i в BP(P,E) может принимать значение true после l, если $F_v(WP(x = e, \varphi))$ истинно после l. Аналогично b_i может принимать значение false после l, если $F_v(WP(x = e, \neg\varphi))$ истинно перед l. Причем оба предиката не могут одновременно быть истинны. Если же ни один из предикатов не является истинным перед l, то b_i должно принимать недетерминированное значение. Это происходит в случае если предикаты из E не достаточно строги чтобы предоставить необходимую информацию.

Таким образом BP(P,E) определяет следующий вид параллельного присваивания:

```
b_1, ..., b_n =
  choose(F_v(WP(x=e,φ_1)), F_v(WP(x=e,¬φ_1))),
  ...,
  choose(F_v(WP(x=e,φ_n)), F_v(WP(x=e,¬φ_n)));

bool choose(bool pos, bool neg) {
  if(pos) { return true; }
  if(neg) { return false; }
  return unknown();
}

bool unknown() {
  if(*) { return true; }
  else { return false; }
}
```

Абстрактное условие $*$ означает, что обе ветки исполнения возможны. Заметим, абстрактная интерпретация оператора присваивания основана на вычислении слабейшего предусловия, которое является локальным для каждого присваивания и может быть вычислено исключительно синтаксической манипуляцией предикатов.

3.2.3 Абстракция ветвлений

Любой оператор ветвления программы можно представить в виде комбинации операторов условного и безусловного перехода. Тогда каждый оператор безусловного перехода эквивалентно копируется в булеву программу. Трансляция условных переходов более сложна.

Рассмотрим условие $if(\varphi)\{...\}else\{...\}$ в программе P. В начале ветки then в P предикат φ истинен. Поэтому в начале соответствующей

ветки then программы BP(P,E), условие $G_v(\varphi)$ должно быть истинно. Соответственно $\neg\phi$ истинно в else ветке P, а $G_v(\neg\varphi)$ должно быть истинно в BP(P,E).

```
if(*)
{
  assume(G_v(φ))
  ...
}
else
{
  assume(G_v(¬φ))
  ...
}
```

$assume(\psi)$ никогда не может быть ложью. Поток исполнения, для которого ψ ложно в точке assume, игнорируется.

3.2.4 Построение

Построение предложенной предикативной абстракции заключается в последовательном преобразовании операторов исходной программы в булеву форму. На первом шаге формируется множество предикатов E на основе присваиваний и контролирующих условий исходной программы. Далее происходит непосредственная трансляция. Поиск слабейших предусловий основан на синтаксических манипуляциях выражений, т.е. на символических вычислениях.

Такой подход позволяет проводить особенно точный анализ изменения значения переменных и применим на практике в случае достаточно простых анализируемых выражений. Например в случае анализа указателей в основном нас интересует сравнение переменных и достаточно простая арифметика.

В случае анализа значений всех переменных программы, особенно в языках со сложной системой типов, использование символических вычислений на этапе построения булевой программы существенно ухудшает производительность алгоритма. В таком случае имеет смысл рассматривать слабейшее предусловие равным исходному предикату, а также перенести анализ контролирующих операторов assume до завершающего вывода результата анализа.

3.2.5 Устойчивость

Существенным свойством предикативной абстракции является ее устойчивость. То есть, каждый осуществимый путь в P остается осуществимым в BP. Если X состояние программы P после исполнения пути p, существует исполнение пути p в булевой программе BP заканчивающееся состояние Y таким, что для любого $1 \leq i \leq n$, φ_i верно в X если b_i истинно в Y. В результате, при преобразовании программы в булеву форму все возможные пути ее исполнения сохраняются, что позволяется проводить аналогичный анализ на предложенной абстракции.

В работе [5] приводится доказательство устойчивости предложенного преобразования и подробное описание синтаксиса булевой программы.

3.3 Верификация модели

Предикативная абстракция является одним из типов абстрактной интерпретации программы. Из сказанного выше следует, что правильно задавая интересующие свойства (модель) и проводя их проверку (верификацию) на булевой программе можно получить решение некоторых потоковых задач. Так, например, для анализа валидности указателей в работе [7] к булевой программе применяется алгоритм достигающих определений. Аналогичный метод может быть применен для решения других задач, например, для вычисления значений переменных.

Булева переменная может принимать истинное, ложное или неопределенное значение (обозначаемое $*$). Состояние программы в операторе s определяется набором значений логических переменных в области видимости оператора s (т.е. вектором, где один элемент соответствует одной переменной). А множеством достижимых состояний булевой программы в s является множество векторов. Это важное отличие от традиционного алгоритма достигающих определений позволяет сохранить более подробную информацию о потоках исполнения.

4 Описание алгоритма

В данной главе мы рассмотрим детали реализованного алгоритма в применении к задачи протягивания констант. Разработка происходила в рамках системы автоматизированного реинжиниринга приложений Relativity Modernization Workbench[27], позволяющего анализировать несколько входных языков. Для сохранения языконезависимости предлагаемого подхода была реализована оболочка, инкапсулирующая входные конструкции. Единственной частью, требующей подробной информации об используемом языке в случае протягивания констант, является среда времени исполнения, которая проводит вычисления над конкретными значениями переменных, используя информацию о системе типов. В отличие от традиционного подхода, использование среды времени исполнения происходит не постоянно, а лишь на завершающей стадии алгоритма для выведения значений исходной программы из набора логических уравнений.

В текущей реализации на синтаксис входной программы накладываются некоторые ограничения:

- Любой оператор программы изменяет значение лишь одной переменной
- Все условные ветвления представляются оператором if с двумя ветками возможного исполнения
- Для каждой переменной существует присваивание, отражающее инициализацию переменной

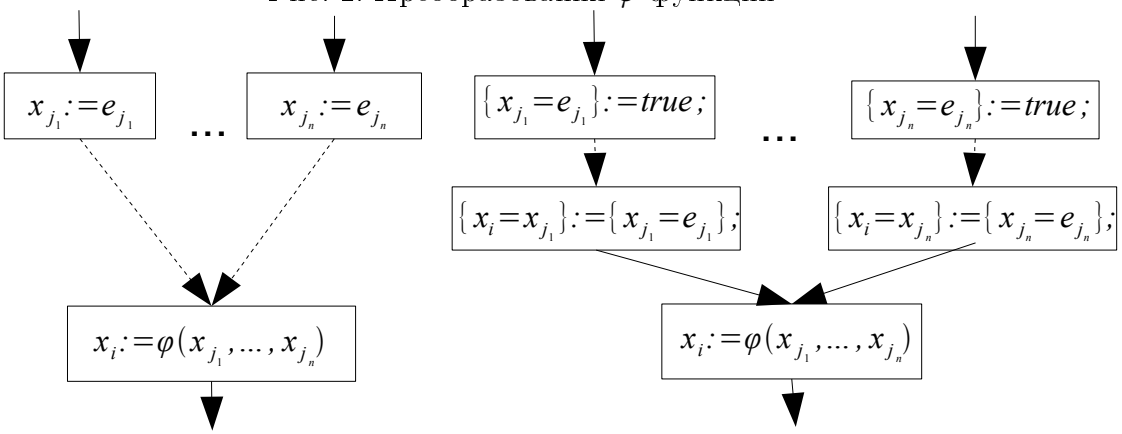
Вводимые ограничения могут быть преодолены путем введением препроцессора, преобразующего программу без изменения семантики.

Как было отмечено выше, для определения значения переменной в точке необходим не только набор операторов, приводящих к этой точке, но и их порядок. Для этой цели в реализации использована нумерация присваиваний SSA формы. Кроме того, реализованный алгоритм состоит из модуля построения булевой программы, модуля анализа достигающих определений и модуля вывода результирующих значений.

4.1 Построение SSA формы

Построение SSA формы основано на алгоритме предложенном авторами работы [24]. Основной информацией, используемой при расстановке φ-узлов, традиционно является граница доминаторов. Оптимизация ее

Рис. 2: Преобразования φ-функции



вычисления позволяет добиться линейности алгоритма. В данном случае оптимизация основана на введении порядка вычисления. Предположим вершина y является предком вершины x в дереве доминаторов. Если доминаторная граница x была вычислена, нет необходимости ее пересчета при вычислении границы y.

Для определения порядка обхода узлов используется структура *DJ-графа (Dominator-Join graph)*, где узлы представляют собой линейные участки программы. Используются два типа ребер: доминаторные (дублирующие ребра дерева доминаторов) и объединяющие. Объединяющее ребро соединяет вершины x и y если между x и y существует ребро в графе потока управления и x не является непосредственным доминатором y. Алгоритм обходит все вершины в порядке убывания длины пути в дереве доминаторов от корня до вершины. Доминаторные ребра используются для задания порядка обхода, в то время как объединяющие нужны для определения кандидатов на расположение φ-узлов. Для хранения кандидатов в правильном порядке используется специальная списковая структура. Сложность такого алгоритма составляет $O(|E|)$, где $|E|$ - количество ребер в DJ-графе.

После размещения в программе φ-узлов, необходимо перенумеровать все переменные программы. Эта задача решается обходом в глубину дерева доминаторов. При этом для каждой переменной необходимо хранить на стеке последний использованный номер.

4.2 Построение булевой программы

Основная задача модуля построения булевой программы заключается в преобразовании операторов изменяющих значения переменных. Для простоты будем рассматривать лишь операторы присваивания, в том числе присваивания φ-функций.

Присваивание вида x:=e задает новую булеву переменную $\{x = e\}$ и булевую форму оператора присваивания $\{x = e\} := true$. Остальным переменным, вида $\{x = a\}$, присваивается false.

Каждая φ-функция вида $x_i := \varphi(x_{j_1}, ..., x_{j_n})$ задает n булевых переменных $\{x_i = x_{j_1}\}$, ..., $\{x_i = x_{j_n}\}$ (рисунок 2). Истинность k-ой булевой переменной в точке p, непосредственно перед исполнением φ-функции, определяется динамически в зависимости от потока, приводящего к точке p. При передачи управления φ-функции по k-ому предку в графе потока управления, переменной $\{x_i = x_{j_l}\}$ присваивается true при $l = k$ и false при $l \neq k$. Например:

```
if(...)
  move '1' to x4.
else
  move '2' to x3.

move φ(x3, x4) to x2
```

преобразуется в:

```
if(...)
{
  {x4 = '1'} := true;
  {x2 = x4} := {x4 == '1'};
}
else
{
  {x3 = '2'} := true;
  {x2 = x3} := {x3 == '2'};
}
```

Операторы, задающие неопределенное значение переменных (например, чтение из базы данных) порождают переменные вида $\{x = undef\}$.

Реализация путезависимого решения отличается дополнительным анализом всех путей исполнения, учитывая отношения между операто-

рами ветвления для удаления несуществующих путей. Для сохранения информации об условных ветвлениях программы мы определяем дополнительные булевы переменные, основанные на контролирующих условиях. Так, для оператора $if(e)then\,\{...\}\,else\,\{...\}$, булева переменная $\{e\}$ должна быть использована в начале двух возможных веток исполнения в операторах $assume(\{e\} = true)$ и $assume(\{e\} = false)$. Например:

```
if(z1='4')
  ...
else
  ...
```

преобразуется в:

```
if(*)
{
  assume({z1 == '4'});
  ...
else
{
  assume(!{z1 == '4'});
  ...
}
```

Остальные операторы языка, не влияющие на потоки управления и данных, заменяются на неинтерпретируемый оператор skip.

4.3 Анализ достигающих определений

Алгоритм достигающих определений состоит из этапа анализа используемых переменных и итеративного применения потоковой функции.

На этапе анализа переменных для каждого узла графа потока управления формируются три множества переменных (KILL, GEN, ASSERT). Множество GEN определяет переменные получающие значения в операторе. KILL задает множество переменных, перезаписываемых в операторе. ASSERT используется для проверки истинности(ложности) переменных оператора.

В реализации один элемент битового вектора соответствует одному определению булевой переменной. Предположим, что в программе имеется n определений, тогда начальным состоянием программы является

множество, состоящие из одного битого вектора длины n вида $\{(0, ..., 0)\}$. Алгоритм заключается в интеративном применении потоковой функции к множеству состояний. Для каждого узла графа потока управления определяется входное (IN) и выходное (OUT) множество состояний.

$$IN_i = \begin{cases} \{(0, ..., 0)\}, & \text{для i - корневой вершины} \quad (1) \\ \bigcup IN_{j_{j \in pred(i)}}, & \text{для остальных случаев} \quad (2) \end{cases}$$

$$OUT_i = \Psi(IN_i)$$

Потоковая функция Ψ для оператора i определяется следующим образом:

- $\forall x \in GEN_i$, значение переменной x в IN_i заменяется на true
- $\forall x \in KILL_i$, значение переменной x в IN_i заменяется на false
- $\forall x \in ASSERT_i$, из множества IN_i удаляются элементы со значением true (или false, в зависимости от типа ветки условного перехода) переменной x.

Кроме того, если i является φ-функцией, множество состояний изменяется по правилу описанном в части 4.2. Итерации заканчиваются при достижении фиксированной точки, т.е. когда множество состоянии OUT_i любой вершины не отличается от множества состояний на предыдущей итерации $O\bar{U}T_i$.

$$\forall i : O\bar{U}T_i = \Psi(IN_i)$$

4.4 Диаграммы двоичных решений

Учитывая необходимость хранения множества битовых векторов для каждого узла требовалось выбрать подходящую структуру данных. Кроме возможности компактного хранения элементов, структура должна предоставлять эффективные теоретико-множественные операции, такие как пересечение и объединение, для снижения стоимости одной итерации алгоритма [14].

Подходящей структурой данных является *диаграммы двоичных решений.*

Определение 8 *Диаграммы двоичных решений (Binary Decision Diagram - BDD) [10] - направленный ациклический граф с*

- *выделенным корнем*

- *двумя терминальными вершинами помеченными 0 и 1*
- *множеством узлов с двумя выходящими ребрами low(u), high(u) и меткой узла var(u)*

Диаграммы двоичных решений используются для неявного представления множества бит-векторов фиксированной длины [10], которые отображаются в 1 если принадлежат множеству, и 0 если нет. Причем при увеличении мощности представляемого множества, размер диаграммы увеличивается медленно. На рисунке 3 приведен пример преставления множества из семи бит векторов.

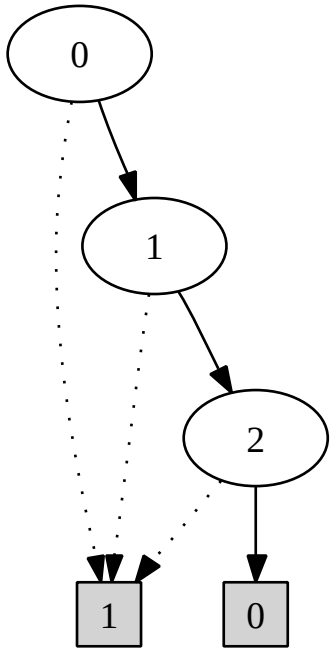


Рис. 3: Представление множества бит векторов $\{(000), (001), (010), (0,1,1), (1,0,0), (1,0,1), (1,1,0)\}$

Рассмотрим логические выражения: $t ::= x|0|1|\neg t|t \wedge t|t \vee t|t \Rightarrow t|t \Leftrightarrow t$. Определим $x \rightarrow y_0, y_1$ как if-then-else оператор следующим образом: $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$ Переменную x, в таком случае, будем называть проверяющим условием. Например $\neg x$ можно записать как $(x \rightarrow 0, 1)$, $x \Leftrightarrow y$ как $x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$, а саму переменную как $x \rightarrow 1, 0$.

Любое логическое выражение может быть записано в таком виде. Для доказательства этого определим $t\,[k/x]$ как подстановку значения k (0 или 1) в выражение t вместо x. Тогда само выражение t эквивалентно $x \rightarrow t\,[1/x]\,, t\,[0, x]$. Если выражение не содержит переменных, оно

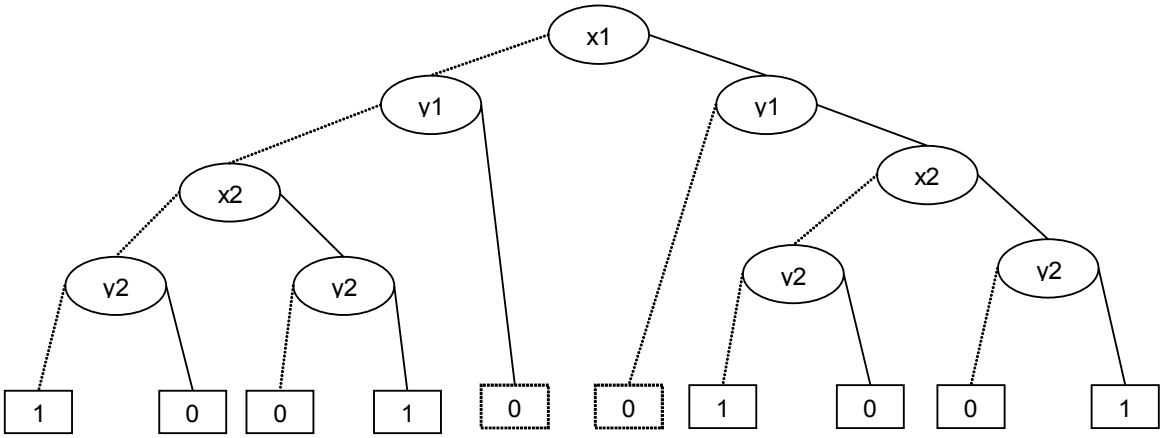


Рис. 4: Представление if-then-else выражений в виде бинарного дерева

эквивалентно константе (0 или 1). Если выражение содержит одну переменную, используя подстановку мы можем свести его к if-the-else форме. Если выражение содержит несколько переменных, будем проводить подстановки рекурсивно, до тех пор, пока не останется ни одной переменной. Процесс завершится так как обе части if-then-else подстановки содержат на одну переменную меньше, чем исходное выражение.

Рассмотрим пример выражения $t = (x \Leftrightarrow y) \wedge (x \Leftrightarrow y)$, проводя рекурсивные подстановки получим:

$$
\begin{aligned}
&t = x \to t_1, t_0 \\
&t_0 = y_1 \to 0, t_{00} \\
&t_1 = y_1 \to t_{11}, 0 \\
&t_{00} = x_2 \to t_{001}, t_{000} \\
&t_{11} = x_2 \to t_{111}, t_{110} \\
&t_{000} = y_2 \to 0, 1 \\
&t_{001} = y_2 \to 1, 0 \\
&t_{110} = y_2 \to 0, 1 \\
&t_{111} = y_2 \to 1, 0
\end{aligned}
$$

На рисунке 4 представлено представление полученных выражений в виде бинарного дерева.

Некоторые выражение (поддеревья), оказавшиеся эквивалентными, можно объединить. Например t_{110} и t_{000}, t_{111} и t_{001}, после t_{00} и t_{11}:

$$
\begin{aligned}
&t = x_1 \to t_1, t_0 \\
&t_0 = y_1 \to 0, t_{00}
\end{aligned}
$$

$t_1 = y_1 \rightarrow t_{00}, 0$
$t_{00} = x_2 \rightarrow t_{001}, t_{000}$
$t_{000} = y_2 \rightarrow 0, 1$
$t_{001} = y_2 \rightarrow 1, 0$

На рисунке 5 представлено сведенное представлений выражений в виде направленного ациклического графа.

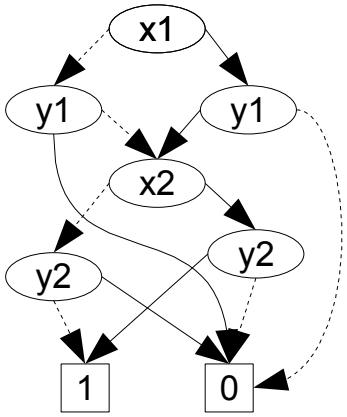


Рис. 5: Представление if-then-else выражений в виде направленного ациклического графа

Порядок переменных на путях от вершины к терминальным узлам соответствует порядку проведенных подстановок: $x_1 < y_1 < x_2 < y_2$. Такой тип BDD называется сортированный, OBBD (ordered). OBBD с удаленными эквивалентными выражениями называется редуцированной, ROBDD.

Определение 9 *BDD называется отсортированной (OBDD), если на всех путях графа переменные находятся в заданном порядке* $x_1 < x_2 < ... < x_n$

Определение 10 *OBDD называется редуцированной, если*

- *в ней не существует узлов u и v, таких что var(u)=var(v), low(u)=low(v), high(u)=high(v)*
- *ни одна вершина u не имеет эквивалентных выходящих ребер, т.е.* $low(u) \neq high(u)$

Для работы с отсортированными редуцированными диаграммами двоичных решений была использована распространенная open source библиотека BUDDY [18], представляющая удобный и достаточно полный интерфейс.

4.5 Вывод результата

Результат работы анализа достигающих определений для булевой программы можно представить в виде набора логических переменных, объединенных отношениями дизъюнкции и конъюнкции, то есть в виде логической формулы. Такая формула неявно содержит информацию о путях исполнения программы приводящих к рассматриваемой точке. Несовместность результирующей формулы в точке p означает недостижимость этой точки ни при каких исходных данных.

В случае совместности, из истинности формулы может быть выведены значения логических переменных в точке [28]. Из истинности булевой переменной b_i следует истинность соответствующего предикатов исходной программы φ_i. В результате мы получаем набор уравнений над интересующими переменными исходной программы. Данные уравнения могут быть использованы для получения конечного результата в зависимости от поставленной задачи.

В нашем случае, для задачи протягивания констант, значениями переменных будут решения уравнений. Переменная может получить одно или несколько значений, неопределенное значение или интервал значений. Для получения конечного результата необходимо провести подстановку уравнений, после чего для каждой интересующей переменной может быть получен набор значений (возможно, неопределенных). В приложении приводится пример работы всех стадий разработанного алгоритма.

В случае анализа указателей решения определяют нулевые или не инициализированные указатели в конкретных точках. При проверке входных параметров программы мы получаем ограничения накладываемые на параметры, необходимые для достижимости выбранных точек и т.д.

4.6 Дальнейшее развитие

В работе был реализован алгоритм протягивания констант для подмножества языка COBOL. При этом, реализованный алгоритм состоит из нескольких независимых модулей. Для поддержки полного набора операторов и типов исходного языка необходимо расширить возможности модуля вывода результатов в сторону большего использования существующей системы времени выполнения. Кроме того, ввиду языконезависимости большинства модулей возможно использование основной части алгоритма для анализа других языков.

Для увеличения абстрактности модулей может быть применен аб-

страктный решатель потоковых задач, позволяющий произвольно задавать потоковые функции без изменения итеративной части алгоритма. Для увеличения точности алгоритма, в случае анализа циклов, может быть применен метод, описанный в работе [8]. А именно, определения зависимостей изменения переменных от порядка итераций цикла.

Развитием тематики предикативной абстракции может быть использование последовательных уточнений множества предикатов [16]. То есть итеративный процесс выбора подходящего набора предикатов, когда на первом шаге выбор происходит на основе неточной информации об интересующих свойствах программы.

5 Заключение

В ходе данной работы был проведен анализ существующих подходов к решению потоковых задач. Были обнаружены сходства некоторых рассматриваемых алгоритмов. Для обобщения задач, смежных с проблемой анализа значения переменных, был предложен способ промежуточного представления программы в виде предикативной абстракции. Построение абстракции основано на выборе множества предикатов исходной программы. Выбор осуществляется в зависимости от интересующих свойств требующих проверку, например валидность указателей или достижимость выбранной точки. Кроме того, выбор в качестве предикатов контролирующих условий операторов перехода позволяет получить путезависимое решение.

Основным результатом работы является реализованный алгоритм протягивания констант для программ на языке COBOL (с некоторыми ограничениями синтаксиса), не требующий предварительного анализа достигающих определений. Разработанный механизм позволил разбить алгоритм на несколько последовательно используемых модулей: построение SSA формы, построение предикативной абстракции, итеративное определение достигающих логических присваиваний, вывод результатов. При этом, каждый шаг алгоритма не зависит от окружения, что позволяет использовать различные оптимизации.

Итеративная часть алгоритма свелась к решению задачи достигающих определений на булевой программе, что позволило использовать диаграммы двоичных решений для компактного представления множеств состояний. Кроме того, итеративный модуль не оперирует с конкретными значениями и не использует систему времени выполнения, т.е. является языконезависимым. Отложенные вычисления реальных значений переменных на завершающем этапе позволяют проводить вычисления лишь в интересующих точках программы, применять оптимизации ленивых вычислений и кэширование значений.

Список литературы

[1] Ульман Дж. Ахо А., Сети Р. *Компиляторы: принципы, технологии, инструменты.* М.: Издательский дом Вильямс, 2003.

[2] А.Н. Терехов. Методы синтеза эффективной рабочей программы. 1976.

[3] А.В. Друнин. Автоматическое построение программных компонент на основе устаревших программ. *Автоматизированный Реинжиниринг Программ*, pages 184–206, 2000.

[4] F.E. Allen. Control flow analysis. in proceedings of a symposium on compiler. *SZGPLAN*, 5(9):1–19, 1970.

[5] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[7] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 97–103, New York, NY, USA, 2001. ACM Press.

[8] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In *Computational Complexity*, pages 374–388, 1994.

[9] Rastisalv Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–251, New York, NY, USA, 1998. ACM Press.

[10] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[13] Ron K. Cytron and Jeanne Ferrante. Efficiently computing φ-nodes on-the-fly. *ACM Trans. Program. Lang. Syst.*, 17(3):487–506, 1995.

[14] Devasish Das. Development of a binary decision diagram solver for live variables analysis. Technical report, 2003. Tata reseach design and development centre.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[16] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[17] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[18] Binary Decision Diagram library: BUDDY. http://sourceforge.net/projects/buddy/.

[19] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979.

[20] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] Chandrasekhar Boyapati Paul Darga. Efficient software model checking of data structure properties. *SZGPLAN*, 21:363 – 382, 2006.

[22] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–78, New York, NY, USA, 1991. ACM Press.

[23] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[24] Vugranam C. Sreedhar and Guang R. Gao. Computing φ-nodes in linear time using dj-graphs, 1994.

[25] Bernhard Steffen, Jens Knoop, and Oliver Ruthing. The value flow graph: A program representation for optimal program transformations. In *European Symposium on Programming*, pages 389–405, 1990.

[26] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[27] Relativity Modernization Workbench User Manual. Available with the Modernization Workbench system ©Relativity Technologies (http://www.relativity.com/).

[28] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 145–156, New York, NY, USA, 2006. ACM Press.

A Пример

Рассмотрим пример работы реализованного алгоритма

1. Исходный код:

```
procedure division using y.

  move '' to x.
  move '' to y.
  move '' to z.
p.
  move '4' to z.
  move '0' to x.

  if(z='4')
    move '1' to x
  else
    move '2' to x.

  if(y='exit')
    go to q.

  call x.
q.
  stop run.
```

Первые три оператора присваивания - фиктивные и используются лишь для корректного построения SSA формы. Также будем считать, что переменная y является входным параметром программы.

2. SSA форма:

```
    move '' to x_0.
    move '' to y_0.
    move '' to z_0.
  p.
    move '4' to z_1.
    move '0' to x_1.

    if(z_1='4')
      move '1' to x_4.
    else
      move '2' to x_3.

    move φ(x_3, x_4) to x_2.

    if(y_0='exit')
      go to q.

    call x_2.
  q.
    stop run.
```

3. Предикативная абстракция:

```
p.
  {z1 == '4'} := true;
  {x1 == '0'} := true;

  if(*)
  {
    assume({z1 == '4'});
    {x4 = '1'} := true;
    {x2 = x4} := {x4 == '1'};
  }
  else
  {
    assume(!{z1 == '4'});
    {x3 = '2'} := true;
    {x2 = x3} := {x3 == '2'};
  }

  if(*)
  {
    assume({y0 == 'exit'});
    go to q;
  }

  skip;
q.
  skip;
```

4. Результирующие предикаты для оператора call:

$$(\{z_1 == \text{'4'}\} \vee \{x_1 == \text{'0'}\} \vee \{z_1 == \text{'4'}\} \vee \{x_4 = \text{'1'}\} \vee \{x_2 = x_4\} \vee \neg\{y_0 == \text{'exit'}\})$$
$$\wedge$$
$$(\{z_1 == \text{'4'}\} \vee \{x_1 == \text{'0'}\} \vee \neg\{z_1 == \text{'4'}\} \vee \{x_3 = \text{'2'}\} \vee \{x_2 = x_3\} \vee \neg\{y_0 == \text{'exit'}\})$$

Нас интересуют значения возможные переменной x_2 в операторе call. Упростив полученное выражения подстановками получим:

$$(\{z_1 == \text{'4'}\} \vee \{z_1 == \text{'4'}\} \vee \{x_2 = \text{'1'}\} \vee \neg\{y_0 == \text{'exit'}\}) \wedge (\{z_1 == \text{'4'}\} \vee \neg\{z_1 == \text{'4'}\} \vee \{x_2 = \text{'2'}\} \vee \neg\{y_0 == \text{'exit'}\}) \equiv (\{z_1 == \text{'4'}\} \vee \{x_2 = \text{'1'}\} \vee \neg\{y_0 == \text{'exit'}\})$$

Таким образом, для интересующей точки мы получили, что переменная x принимает единственное значение '1'. Кроме того, для достижимости точки требуется выполнение условия для входной переменной y != 'exit'.

В данном примере мы рассматривали способ нахождения путезависимого решения. Для этого, помимо операторов изменяющие значения переменных, мы принимали во внимание контролирующие выражения операторов условного перехода. Необходимо было проводить проверку совместности полученных выражений и отсеивать лишние результаты. Так, из-за несовместности выражений $\{z_1 == \text{'4'}\}$ и $\neg\{z_1 == \text{'4'}\}$ значение 4 не было использовано в результате.

Для поиска путенезависимого решения достаточно не рассматривать переменные в операторах assume. Для данного примера результатом путенезависимого алгоритма были бы два значения переменной x: '1' и '2'.