УДК 004.415.53

Построение срезов для программ на динамических языках

© А.О. Крючков, В.А. Крищенко

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Предложен метод построения срезов, учитывающий особенности динамических языков программирования и основанный на построении траектории в ходе интерпретации программы и последующем формировании на ее основе графа программных зависимостей. Метод реализован для построения срезов программ на языке Lua путем модификации его интерпретатора.

Ключевые слова: *срезы программ, анализ программ, динамические языки.*

Срез программы — это подмножество ее операторов, влияющее на результат, вычисляемый в заданной точке программы [1]. Построение срезов применяется в отладке, сопровождении и тестировании программ, а также при оптимизации и распараллеливании программ [2].

Существующие методы построения срезов опираются на статический анализ зависимостей по данным и управлению, т. е. предполагают, что все возможные зависимости заранее известны. Методы построения динамических срезов [3], учитывающих зависимости, возникающие в программе на конкретных исходных данных, также предполагают, что множество всех зависимостей в программе известно заранее, и выбирают подмножество задействованных зависимостей на основании траектории выполнения программы.

Таким образом, существующие методы построения срезов применимы только для программ, не меняющих структуру межоператорных зависимостей во время выполнения. В последние годы возросла популярность так называемых динамических языков, характеризующихся, среди прочего, динамической типизацией, возможностью модификации типов данных и объектов и встроенными высокоуровневыми средствами кодогенерации «на лету». Для программ на этих языках статический анализ зависимостей затруднен или невозможен. Работы [4, 5] показывают, что многие реальные программы на таких динамических языках опираются на их динамические возможности, и, следовательно, существующие методы построения срезов к ним не применимы. Таким образом, актуальна проблема построения срезов для программ на динамических языках.

Межоператорные зависимости и срезы. Выделяют два типа зависимостей между операторами: по данным и по управлению. Зависимость по данным оператора O_2 от O_1 возникает, когда при выполнении O_2 используются значения, полученные в O_1. Многие работы по срезам (в частности [1]) определяют эту зависимость через наличие пересечения двух множеств: переменных, значение которых устанавливается одним оператором ($DEF(O_1)$), и переменных, на которые ссылается другой оператор ($REF(O_2)$). В динамических языках понятие «переменной» зачастую отсутствует, вместо этого используются понятия «значение» и «имя значения», поэтому далее зависимость по данным будет определяться в терминах значений.

Оператор O_2 зависит от оператора O_1 по управлению, если результат выполнения оператора O_1 определяет, будет ли выполнен оператор O_2 или нет. Зависимости по управлению в структурированных программах возникают в пределах одной процедуры. Это также верно для неструктурированных программ, не использующих нелокальную передачу управления в форме операторов halt или механизма исключений и ограничивающихся конструкциями break, continue и локальным оператором goto. Оператор goto, позволяющий перейти к произвольной метке в любой части программы, отсутствует в современных динамических языках.

Зависимости в программе можно представить несколькими способами, наиболее распространенный среди которых — граф программных зависимостей (ГПЗ) [6]. Это ориентированный граф, вершинами которого являются множества операторов программы, а дуги соответствуют зависимостям по данным и управлению (рис. 1).

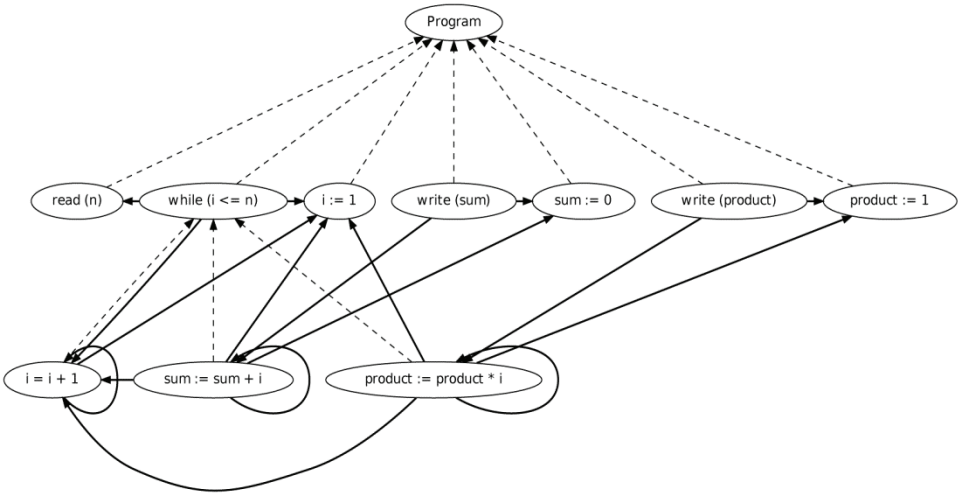

Рис. 1. Граф программных зависимостей

В качестве примера представлен граф программных зависимостей для следующей программы (см. рис. 1). Зависимости по данным изображены на рисунке сплошными стрелками, по управлению — прерывистыми.

```
var i, sum, product: integer;
begin
    read(n);
    i := 1;
    sum := 0;
    product := 1;
    while i <= n do
       sum := sum + i;
       product := product * i;
       i := i + 1;
    end;
    write(sum);
    write(product);
end.
```

Когда ГПЗ построен, вычисление среза по заданному оператору в программе, которому соответствует одна из вершин ГПЗ, сводится к определению множества вершин графа, достижимых по дугам зависимостей из вершины-критерия, которое и будет являться срезом. При определении достижимости дуги зависимостей по данным и управлению равнозначны. Задачу построения среза, таким образом, можно считать решенной при наличии ГПЗ для анализируемой программы.

При построении срезов для программы и конкретного набора ее исходных данных построенный ГПЗ модифицируется некоторым образом с учетом траектории выполнения, собранной при фактическом запуске программы. Простейший вариант, описанный в [7], предполагает ограничение ГПЗ по множеству вершин, входящих в траекторию (т. е. операторов, которые были реально выполнены). После получения ограниченного графа построение среза идет как обычно.

Особенности динамических языков. В качестве определяющих особенностей динамических языков, влияющих на создание срезов, можно выделить динамическую типизацию, возможность модификации типов данных и кодогенерации «на лету», а также интерпретацию как основной способ исполнения программ. Наиболее распространенные на данный момент динамические языки программирования, такие как Python, Ruby, Lua, Perl, PHP и многие диалекты Lisp, обладают всеми перечисленными особенностями в совокупности.

Динамическая типизация означает, что понятие типа данных в языке связывается только с конкретным значением во время выполнения программы, а не с переменной или именем на стадии компиля-

ции. Динамическая типизация реализуется путем добавления к каждому значению некоторого набора метаданных. Не зная, какому классу принадлежит объект, нельзя сказать, какой именно код выполнится при вызове метода этого объекта или при обращении к его свойству. Следовательно, статическое построение графа зависимостей по управлению невозможно.

Модификация типов данных и объектов во время выполнения программы в общем случае означает, что нельзя делать каких-либо утверждений о типе объекта в произвольный момент времени, если мы располагаем только информацией о типе в момент его создания. Эта особенность также означает, что одного идентификатора типа значения недостаточно, чтобы определить его поведение в качестве объекта. При использовании программистом модификации типов данных прослеживание каких-либо зависимостей может осуществляться только в момент выполнения кода.

Генерация программного кода «на лету» является наиболее очевидным препятствием для статического анализа программ на динамических языках, поскольку интересующий нас программный код буквально отсутствует до начала выполнения программы. Типичной реализацией генерации кода «на лету» является функция **eval**, принимающая на вход исходный код. После формирования промежуточного представления сгенерированный код ничем не отличается от остального кода программы.

Интерпретация является основным способом выполнения программ на большинстве динамических языков. Различные виды динамического поведения, описанные выше, делают статическую компиляцию программ на динамических языках в машинный код затруднительной или невозможной. Вместо этого компиляция обычно осуществляется в некоторое промежуточное представление — байт-код виртуальной машины языка или абстрактное дерево синтаксиса (AST) — которое затем подвергается интерпретации.

При наличии интерпретатора для языка появляется возможность перенести обязанность ведения траектории на интерпретатор, дополнив его цикл выполнения инструкций. При наличии исходных текстов интерпретатора можно расширить набор используемых интерпретатором метаданных, включив в него некоторую информацию, используемую при построении срезов.

Метод построения срезов. Для программ на динамических языках собирать информацию о зависимостях до выполнения программы не имеет смысла, поэтому приходится получать ее непосредственно во время выполнения программы. Поскольку для работы метода необходим запуск анализируемой программы, то метод позволяет получать только динамические срезы.

Метод (рис. 2) предполагает формирование траектории выполнения программы — последовательности записей о каждом выполненном операторе вместе с информацией о том, от каких других операторов зависел (по данным или управлению) результат выполнения в каждом конкретном случае.

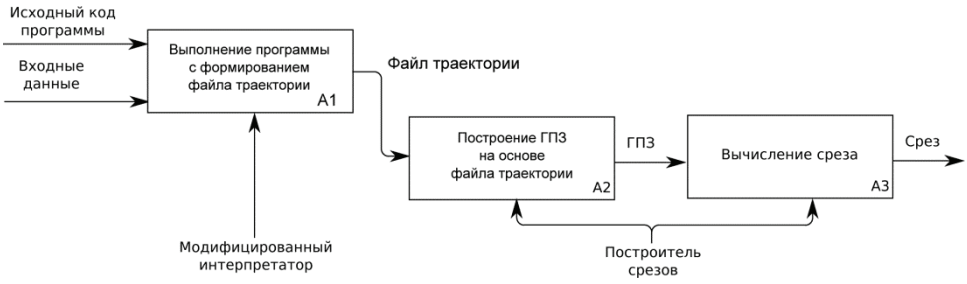

Рис. 2. Функциональная схема метода построения срезов

Зависимости по данным прослеживаются на уровне значений языка. Предполагается, что каждый выполняемый оператор использует ноль или более значений, полученных при выполнении других операторов, и сам вычисляет ноль или более новых значений либо модифицирует старые значения. Множество операторов, от которого данный оператор зависит по данным, определяется, таким образом, через множество использованных им значений.

В динамических языках каждое значение в памяти сопровождается набором метаданных; наш метод предполагает включение в этот набор нового поля *DEF_AT* — идентификатора оператора, вычислившего или последним модифицировавшего это значение. Траектория также будет содержать информацию о зависимостях по управлению.

Далее подробно рассмотрены основные этапы метода.

Построение траектории. Для построения траектории выполнения программы нужен модифицированный интерпретатор языка. Схема измененного процесса выполнения представлена на рис. 3, а детализация изменений, сделанных в основном цикле интерпретатора, — на рис. 4. Блоки, выделенные серым, не были подвержены изменениям.

Формируемая траектория содержит записи нескольких типов, несущих следующую информацию об отношениях между операторами: зависимость по данным, зависимость по управлению, вызов процедур. Записи о вызовах процедур необходимы для построения межпроцедурных срезов, где недостаточно одних зависимостей по данным и управлению [8].

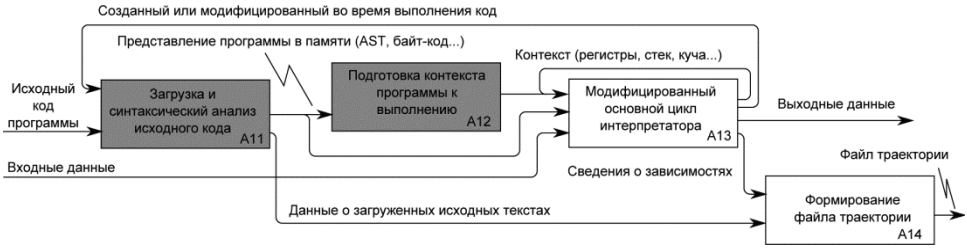

Рис. 3. Процесс выполнения программы модифицированным интерпретатором

Рис. 4. Модифицированный основной цикл интерпретатора

Зависимости по данным формализуются записями *REFS*, имеющими следующий вид: O_1 *REFS* O_2, где O_1 и O_2 — идентификаторы операторов. Идентификатор определяет положение оператора в коде программы и может представлять собой, например, пару «имя файла + номер строки» либо более точные координаты (конкретный вид идентификатора не существенен для описываемого метода).

Для формирования таких записей код выполнения каждой инструкции языка в интерпретаторе изменяется по следующему образцу (`referenced_values` — используемые значения или операнды, `calculated_values` — вычисляемые значения).

```
current_pos := определить_положение_текущей_инструкции;
for v in referenced_values:
    ref_pos := DEF_AT(v);
    добавить_в_траекторию('$current_pos REFS $ref_pos');
calculated_values := выполнить_текущую_инструкцию;
for v in calculated_values:
    DEF_AT(v) := current_pos;
```

Зависимости по управлению проще всего определяются для интерпретаторов, использующих AST-дерево: достаточно найти ближайшего родителя выполняемого сейчас узла, содержащего условие выбора той или иной ветви управления. В общем же случае потребуется составлять граф потока управления (ГПУ) — ориентированный граф, вершинами которого являются простые (линейные) блоки программы, а дуги обозначают отношение предшествия (или следования) — и определять зависимости по управлению на основе ГПУ. Само построение ГПУ тривиально выполняется на основе любого промежуточного представления программы, будь то AST-дерево или байт-код.

Если поток управления структурирован, зависимости по управлению никогда не пересекают границы процедур (сам вызов процедуры не является зависимостью по управлению, так как не содержит условный переход). Такие виды неструктурированной передачи управления, как break, continue и локальный goto, также не пересекают границы процедур.

Межпроцедурные зависимости по управлению возникают при использовании механизма обработки исключений, использовании оператора halt (который можно рассматривать как частный случай необрабатываемого исключения) и оператора goto с переходом на произвольное место в программе (последняя возможность отсутствует в современных динамических языках). Заметим, что с точки зрения ГПУ любые исключения могут возникнуть в середине линейного блока программы и граф потока управления придется перестраивать, даже если исключение будет обработано локально. Такую задачу необходимо рассматривать отдельно; в данной работе мы ограничимся внутрипроцедурными зависимостями по управлению и будем строить ГПУ для отдельно взятых процедур.

С учетом возможностей кодогенерации в динамических языках необходимо отложить составление ГПУ до того момента, когда выполняемый код гарантированно будет сформирован, а именно, до момента захода интерпретатора в процедуру. Будем предполагать, что пока процедура выполняется, структура потока управления в ней остается неизменной.

Процедура извлечения зависимостей по управлению из ГПУ является ресурсоемкой и разрабатываемый метод предполагает ее вынесение на этап обработки траектории, чтобы снизить нагрузку на модифицированный интерпретатор. Сам же интерпретатор будет заниматься сериализацией ГПУ для каждой процедуры, в которую он заходит, и записью сериализованного представления в траекторию. Простейшим сериализованным представлением ГПУ является множество дуг — пар (o_1, o_2), означающих «из оператора o_1 возможен

переход в оператор o_2». Группировка операторов в простые блоки будет выполнена при считывании траектории.

Процедуру сериализации ГПУ можно описать следующим псевдокодом, в котором *operators* — массив всех операторов текущей процедуры.

```
добавить_в_траекторию('CFG_START');
for i := 1 to length(operators):
    o1 := operators[i];
    if is_conditional_jump(o1):
       for o2 in jump_targets(o1):
          добавить_в_траекторию('$o1 JUMPSTO $o2');
    else if is_return_from_procedure(o1):
       добавить_в_траекторию('$o1 JUMPSTO End');
    else:
       o2 := next_operator(o1);
       добавить_в_траекторию('$o1 JUMPSTO $o2');
добавить_в_траекторию('CFG_END');
```

Вызовы процедур не относятся к зависимостям по управлению, так как условный переход при самом вызове отсутствует. Информация о вызовах процедур используется для построения межпроцедурных срезов [8]. В траекторию добавляются следующие типы записей: вызов процедуры — O_1 *FUNCALL* O_2 и возврат из процедуры — O_1 *RETURN*.

Записи *FUNCALL* предшествуют записи *REFS* для каждого фактического параметра, передаваемого в процедуру. Аналогично после записи *RETURN* должны следовать записи *REFS*, соответствующие возвращенным из процедуры значениям. Обработка инструкции вызова процедуры в модифицированном операторе принимает следующий вид:

```
current_pos := определить_положение_текущей_инструкции;
func_def_pos := DEF_AT(вызываемая_процедура);
for v in actual_arguments:
    ref_pos := DEF_AT(v);
    добавить_в_траекторию('$current_pos REFS $ref_pos');
добавить_в_траекторию('$current_pos FUNCALL $func_def_pos');
перейти_к_выполнению_вызываемой_процедуры;
```

При возврате из процедуры записи траектории формируются по следующему алгоритму:

```
current_pos :=
      определить_положение_места_возврата_из_процедуры;
call_pos := определить_положение_места_вызова_процедуры;
for v in returned_values:
    ref_pos := DEF_AT(v);
    добавить_в_траекторию('$current_pos REFS $ref_pos');
```

```
    DEF_AT(v) := call_pos;
добавить_в_траекторию('$current_pos RETURN');
добавить_в_траекторию('$call_pos REFS $current_pos');
вернуться_к_выполнению_вызывающей_процедуры;
```

Каждой записи *FUNCALL* в траектории должна соответствовать парная запись *RETURN*. Все записи, находящиеся между такой парой записей, относятся к выполнению вызванной процедуры (и других процедур, вызванных из нее).

Обработка траектории и построение ГПЗ. Метод предполагает последовательное считывание записей траектории выполнения программы и добавление дуг в составляемый граф программных зависимостей (рис. 5).

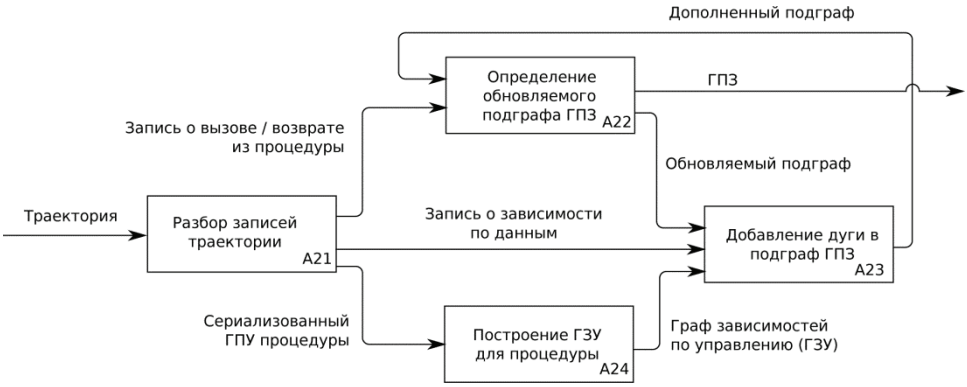

Рис. 5. Обработка траектории и построение ГПЗ

Обработка траектории начинается с инициализации ГПЗ с пустым множеством дуг и множеством вершин, состоящим из одной вершины *Entry*. Множество вершин (операторов) пополняется каждый раз, когда в траектории встречается идентификатор оператора, отсутствующего в графе.

Зависимости по данным обрабатываются проще всего: дуги таких зависимостей $O_1 \rightarrow O_2$ добавляются в ГПЗ при обработке записей траектории вида O_1 *REFS* O_2.

Зависимости по управлению определяются на основе сериализованных ГПУ для каждой процедуры. Восстановление ГПУ по записям *CFG_START*, *CFG_END*, O_1 *JUMPSTO* O_2 траектории описывается следующим алгоритмом:

```
N := emptyset;
E := emptyset;
while true:
    record := прочитать_запись_траектории
    if record = 'CFG_END':
       break
```

```
        o1, o2 := извлечь_поля_JUMPSTO(record);
        N := union( N, {o1, o2} );
        E := union( E, {(o1, o2)} );
    Start := найти_вершину_графа_без_предшественников(N, E);
    /* вершина End уже явно присутствует во множестве N */
    ГПУ := (N, E, Start, End);
```

По восстановленному ГПУ составляется граф зависимостей по управлению (ГЗУ), вершинами которого являются операторы, а дуги $v_1 \to v_2$ означают «оператор v_1 зависит от v_2 по управлению». При построении статического среза ГЗУ стал бы подграфом ГПЗ; так как мы строим динамический срез, только часть дуг ГЗУ будет перенесена в ГПЗ.

Ниже приводится алгоритм построения графа зависимостей по управлению (N_c, E_c) на основе графа потока управления (N, E, $Start \in N$, $End \in N$). Алгоритм основан на работе [6], в которой рассматривается построение более общего вида ГЗУ.

1. Дополнить ГПУ мнимой вершиной *Entry* и дугами $Entry \to Start$, $Entry \to End$.

2. Построить дерево пост-доминаторов (N, E_d) (ДПД) по дополненному ГПУ.

3. Положить $S \subseteq E$: $(a, b) \in S \Leftrightarrow b$ не является пост-доминатором a.

4. Инициализировать ГЗУ (N_c, E_c): $N_c = N \backslash \{End\}$, $E_c = \varnothing$.

5. Повторять следующие шаги для каждой дуги $(a, b) \in S$:

а) найти $L = L(a, b)$ — ближайшего общего предка a и b в ДПД;

б) если $L(a, b) = a$, примем $v_0 = a$; иначе $v_0 = L$. Найдем в ДПД путь p из v_0 в b.

в) все вершины в этом пути, исключая v_0, зависят по управлению от a, что следует отразить добавлением дуг в граф зависимостей по управлению (N_c, E_c): $(v, a) \in E_c \forall v \in p \backslash \{v_0\}$.

Результаты выполнения шагов этого алгоритма для конкретного ГПУ показаны на рис. 6 (вершина Start в ГПУ совпадает с вершиной 1).

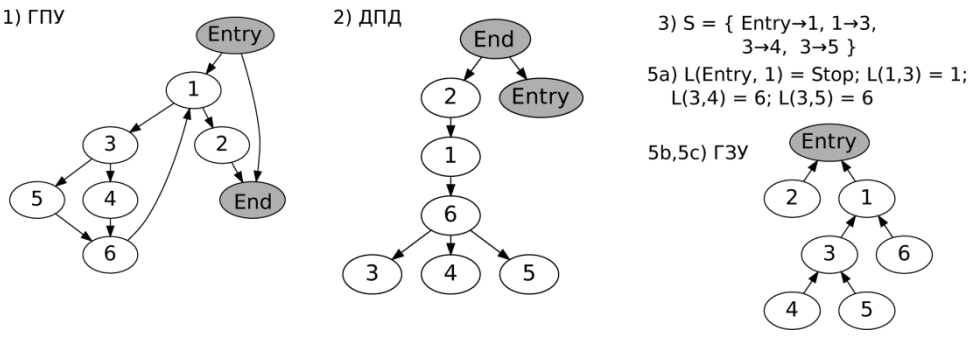

Рис. 6. Результаты выполнения шагов алгоритма получения графа зависимостей по управлению

Построение дерева пост-доминаторов для ГПУ сводится к построению дерева доминаторов для обратного ГПУ, в котором дуга $a \rightarrow b$ означает, что блок a непосредственно следует за блоком b. В свою очередь, алгоритмы построения дерева доминаторов широко известны и описаны, в частности в работе [9].

Граф зависимостей по управлению (N_c, E_c), составленный из линейных блоков (непрерывных участков программы, начинающихся меткой перехода, заканчивающихся оператором условного перехода либо безусловным переходом к вершине *End* и не содержащих иных меток перехода или операторов перехода), можно «развернуть» в граф (N_{co}, E_{co}) составленный из операторов. Пусть $Ops(v)$, $v \in N_c$ — множество операторов, из которых состоит линейный блок v, а $CondJump(v) \in Ops(v)$ — оператор условного перехода, которым заканчивается v. Тогда $N_{co} = \cap_{v \in Nc} Ops(v)$; $(o_1, o_2) \in E_{co} \Leftrightarrow \exists\ (v_1, v_2) \in E_c$: $(o_1 \in Ops(v_1) \wedge o_2 = CondJump(v_2))$.

Если поток управления является структурированным, каждая вершина ГЗУ v_1 будет зависеть по управлению ровно от одной другой вершины v_2; то же самое верно для ГЗУ, составленного из операторов.

В случае наличия операторов локального перехода, таких как break или continue, потребуется формировать ГПУ особым образом, описанным в работе [10]. Каждый из этих операторов представляется как оператор условного перехода, условие в котором всегда истинно, и две исходящие из этого оператора дуги — одна в место перехода (начало или конец цикла), вторая — к следующему оператору, как будто вместо break или continue присутствует no-op.

Располагая операторным ГЗУ (N_{co}, E_{co}), можно добавлять его дуги в ГПЗ в виде дуг зависимостей по управлению, но только тогда, когда обе вершины-оператора, соединяемые дугой, уже присутствуют в ГПЗ (т. е. были выполнены), поскольку мы отслеживаем динамические зависимости по управлению.

Опишем действия при добавлении нового оператора *o_new* в ГПЗ. В следующем псевдокоде *succ* и *pred* — множества непосредственных предшественников и потомков вершины в графе (N_{co}, E_{co}), а функция *add_edge* добавляет в ГПЗ дугу зависимости по управлению, если такой дуги еще нет в ГПЗ.

```
for o2 in pred(o_new):
    add_edge(o2, o_new);
for o2 in succ(o_new):
    add_edge(o_new, o_2);
```

Использование одних и те же вершин и дуг, представляющих зависимости внутри процедуры, для различных ее вызовов приводит к появлению ложных зависимостей между разными местами вызовов.

Эта проблема продемонстрирована на рис. 7 (показаны только зависимости по данным), где видна ложная транзитивная зависимость между операторами *write*(*m*) и *read*(*i*).

```
procedure sum(a, b)
   r := a + b;
   return r;
end;

begin
   read(x);
   read(y);
   read(i);
   read(j);
   m = sum(x, y);
   n = sum(i, j);
   write(m);
   write(n);
end.
```

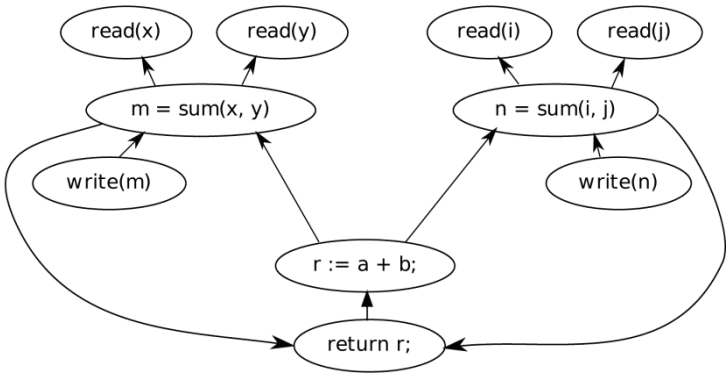

Рис. 7. Ложные зависимости при отсутствии специального подхода к построению ГПЗ процедур

Проблема решается вводом отдельных подграфов для каждой процедуры и соединением их с вершинами вызова особыми дугами — дугами вызова и дугами возврата (рис. 8, показаны линиями с точками). Такой подход описан в работе [8] для статических срезов, но мы введем отдельный подграф для каждого вызова процедуры, поскольку в динамических языках структура зависимостей внутри одной процедуры может меняться от вызова к вызову.

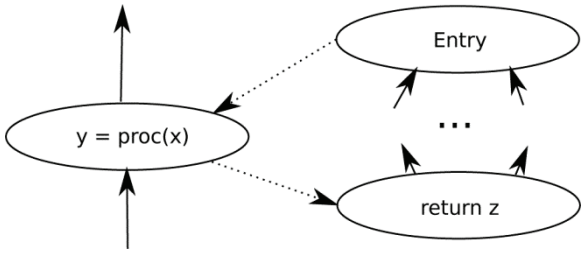

Рис. 8. Дуги вызова и возврата

Для управления подграфами процедур организуем стек подграфов *subgraph_stack*. Изначально в нем содержится пустой подграф основной программы. Все действия по добавлению вершин и дуг проводятся с подграфом на вершине стека. Манипуляции со стеком проводятся при обработке записей *FUNCALL* и *RETURN*.

Дополнительно будем хранить для каждого подграфа G_i в стеке вершину, в которой был сделан вызов процедуры. Далее по тексту эта вершина обозначается как *LAST_CALL_SITE*(G_i). Также в каждом

подграфе процедуры введем вершину — точку входа *Entry*. Эта информация понадобится только на время составления ГПЗ.

Приведем алгоритм обработки записи траектории, имеющей вид O_1 *FUNCALL* O_2.

```
call_pos := O_1;
G_0 := subgraph_stack.peek();
G_1 := создать_новый_пустой_подграф();
Entry := создать_точку_входа();
добавить_вершину_в_подграф(G_0, O_1);
добавить_вершину_в_подграф(G_1, Entry);
пометить_как_вершину_вызова(O_1);
LAST_CALL_SITE(G_0) := O_1;
добавить_дугу_вызова(ENTRY(G_1) -> LAST_CALL_SITE(G_0));
subgraph_stack.push(G1);
```

При обработке записи O_1 *RETURN* в ГПЗ добавляется дуга возврата из процедуры, ведущая из вершины вызова процедуры (*LAST_CALL_SITE*(G_0)) в точку выхода из нее (собственно O_1). После этого подграф вызываемой процедуры извлекается из стека.

В момент извлечения подграфа G_1 из стека можно попытаться найти уже построенный подграф G_e, идентичный извлекаемому G_1, и произвести их объединение с целью сокращения объема памяти, расходуемого на хранение идентичных подграфов. Идентичность здесь можно определить по равенству множеств дуг, исключая при этом сравнении дуги, ведущие из вершины вызова или в нее. Объединение подграфов сводится к замене дуг, ведущих в вершины в G_1, на дуги, ведущие в идентичные вершины в подграфе G_e.

Построение среза. Для построения среза необходимо наличие составленного по описанной выше схеме графа программных зависимостей, а также оператора (критерия среза), для которого будет выполнен поиск влияющих на результат его выполнения операторов. Построение среза начинается с определения вершины или вершин ГПЗ, соответствующих критерию среза. Затем выполняется поиск вершин, достижимых из исходной по дугам зависимостей по данным и управлению. При этом особым образом обрабатываются вершины вызова процедур: при попадании в такую вершину в процессе поиска достижимых рекурсивно запускается новый поиск в подграфе процедуры, в которую ведет дуга «зависимости по возврату из процедуры» из вершины вызова. В процессе рекурсивного поиска запрещается переход по дугам вызова процедур, ведущим в вершины, отличные от той, из которой был запущен поиск: этим гарантируется отсутствие ложных зависимостей между разными местами вызовов одной процедуры.

Множества достижимых вершин, найденных поиском в подграфе основной программы и рекурсивно запущенных поисках в подграфах

процедур, объединяются в одно множество. Это и будет срез программы по заданному критерию.

Ограничения метода. Траектория выполнения программы в описанном методе составляется для одного потока выполнения; соответственно, описанный метод не применим для многопоточных программ. Расширить метод до многопоточного варианта можно добавлением идентификатора потока *thread_id* в каждую запись траектории. Проблемы синхронизации потоков для составления единой траектории либо сопоставления последовательности записей в различных траекториях во времени в данной работе не рассматриваются.

Способ определения зависимостей по управлению, примененный в описанном методе, надежно работает только со структурированным потоком управления. Можно расширить метод для использования с операторами break и continue, используя результаты работы [10], если используемое промежуточное представление программы позволяет отличить break и goto от других типов безусловных переходов. Однако использование нелокальных переходов в форме выброса исключений или остановки программы представляет проблему.

Результаты применения метода. Описанный в работе метод построения срезов программ был реализован для динамического языка Lua. Для реализации метода был модифицирован стандартный интерпретатор Lua 5.1, использующий байт-код в качестве промежуточного представления.

Опыт № 1. В качестве входа используется традиционно приводимая в публикациях по срезам программа, вычисляющая сумму и произведение чисел от 1 до N. Реализация этой программы на языке Lua выглядит следующим образом.

```
local i, n, sum, product:
n = io.read('*number')
sum = 0
product = 1;
for i = 1, n do
    sum = sum + i
    product = product * i
end
print(sum)
print(product)
```

Обработка траектории для $N = 3$ в качестве входных данных дает ГПЗ, показанный на рис. 9. Зависимости по управлению показаны прерывистыми стрелками, по данным — сплошными. Результат построения среза по строке *print*(*sum*) в построенном ГПЗ показан на рис. 10.

Опыт № 2. В этом опыте продемонстрируем программу, использующую вызовы процедур и генерацию кода. Для генерации кода в

Lua применяется стандартная функция *loadstring*, возвращающая новую функцию, при вызове которой и происходит выполнение сгенерированного кода.

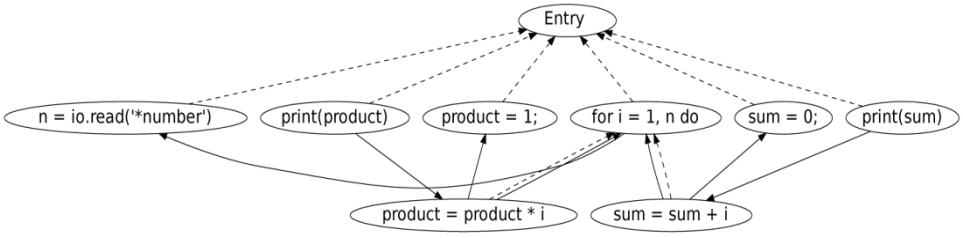

Рис. 9. Граф программных зависимостей из опыта № 1

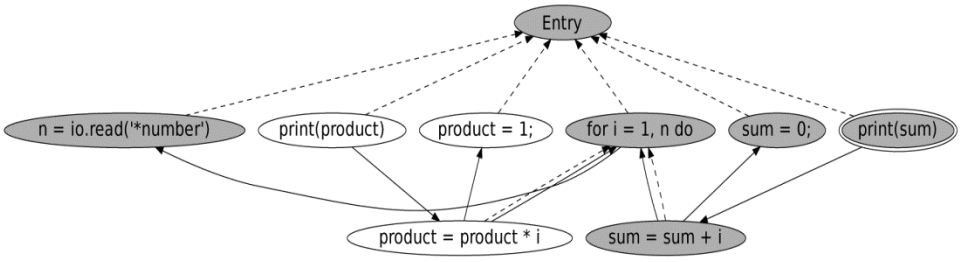

Рис. 10. Срез по оператору `print(sum)` из опыта № 1

Приведенная ниже программа принимает на вход два числа и имя процедуры, в которую затем передаются эти числа. Так, если запустить программу с входными данными «2 3 product», то будет выдан результат «6», а если «2 3 sum» — то «5».

```
local func_name, code, result;
function sum(a, b)
   local r;
   r = a + b;
   return r;
end
function product(a, b)
   local r;
   r = a * b;
   return r;
end
a = oi.read('*number');
b = oi.read('*number');
func_name = oi.read('*line');
code = loadstring("return" .. func_name .. "(a, b)");
print(result);
```

При запуске с входными данными «2 3 product» получается ГПЗ с отмеченными вершинами, входящими в срез по критерию *print*(*result*) (рис. 11). В ГПЗ выделены три подграфа: в порядке сверху вниз это подграф основной программы, подграф сгенерированной с помощью функции *loadstring* процедуры и подграф процедуры *product*, вызванной из сгенерированного кода. Знак вопроса в подграфе сгенерированной процедуры символизирует отсутствие исходного кода для этого оператора. Дуги вызовов и возвратов показаны линиями из точек.

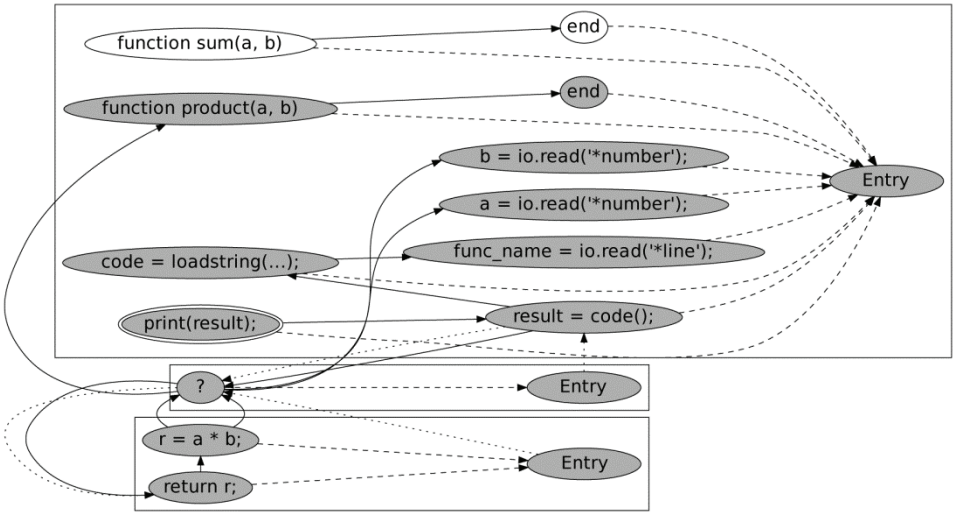

Рис. 11. Срез по оператору `print(result)` из опыта № 2

При указанных входных данных процедура sum, описанная в программе, ни разу не вызывается. В срезе действительно не присутствует ни тело процедуры sum, ни оператор, в котором процедура была сформирована как объект, ни оператор, в котором функция была связана с именем.

Работа выполнена при частичной поддержке Российского фонда фундаментальных исследований (грант № 13-07-00918).

ЛИТЕРАТУРА

[1] Weiser M. Program Slicing. *Proc. of the 5th International Conference on Software Engineering — IEEE Computer Society Press*, 1981, pp. 439–449

[2] В.Н. Касьянов, И.Л. Мирзуитова. *Slicing: срезы программ и их использование*. Сибирское отделение РАН, 2002, 116 с.

[3] Korel B., Laski J.W. Dynamic Program Slicing. *Information Processing Letters*. Elsevier, 1988, vol. 29, no. 3, pp. 155–163.

[4] Richards G., Lebresne S., Burg B., Vitek J. An Analysis of the Dynamic Behavior of JavaScript Programs. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010, pp. 1–12.

[5] Holkner A., Harland J. Evaluating the dynamic behaviour of Python applications. *Proceedings of the Thirty-Second Australasian Conference on Computer Science*, vol. 91, pp. 19–28.

[6] Ferrante J., Ottenstein K. J., Warren J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987, vol. 9, no. 3, pp. 319–349.

[7] Agrawal H., Horgan J.R. Dynamic program slicing. *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pp. 246–256.

[8] Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence graphs. *Proceedings of the SIGPLAN '88 conference on Programming language design and implementation*, 1990, pp. 35–46.

[9] Georgiadis L., Werneck R.F., Tarjan R.E. Finding dominators in practice. *Journal of Graph Algorithms and Applications*, 2006, vol. 10, no. 1, pp. 69–94.

[10] Ball T., Horwitz S. Slicing Programs with Arbitrary Control-flow. *Proceedings of 1st conference on automated algorithmic debugging*, 1993, pp. 206–222.

Статья поступила в редакцию 10.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Крючков А.О., Крищенко В.А.. Построение срезов для программ на динамических языках. *Инженерный журнал: наука и инновации*, 2013, вып. 6. URL: http://engjournal.ru/catalog/it/hidden/777.html

Крючков Алексей Олегович родился в 1990 г., окончил бакалавриат МГТУ им. Н.Э. Баумана в 2011 г. Магистрант кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана. Область научных интересов: анализ программ, сетевые протоколы. e-mail: alexey.kruchkov@gmail.com

Крищенко Всеволод Александрович родился в 1975 г., окончил магистратуру МГТУ им Н.Э. Баумана в 1998 г. Канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии». Автор более 15 научных трудов. Область научных интересов: статический и динамический анализ и верификации программного обеспечения и сетевых протоколов. e-mail: kva@bmstu.ru