Межпроцедурный анализ указателей

Дроздов А.Ю, Владиславлев В.Е.
Институт микропроцессорных вычислительных систем РАН
sasha@mcst.ru, vladisla@mcst.ru

Введение

Межпроцедурный анализ указателей [3, 4] необходим для того, чтобы уже на стадии компиляции программы определить возможные значения указателей, которые в ней используются. Наличие такой информации помогает компилятору в определении того, какие группы операций могут выполняться параллельно. Это, в свою очередь, дает возможность для более эффективной работы многих оптимизирующих преобразований. Отсутствие же информации о значениях указателей заставляет делать самое грубое предположение: любые два указателя пересекаются. Если в редких частных случаях информацию о значении указателей еще удается получить и без межпроцедурного анализа, то выявить независимость пары операций, одна из которых является операцией вызова, без межпроцедурного анализа нельзя.

На более формальном уровне задачей межпроцедурного анализа указателей является вычисление так называемой *points-to функции*, которая в каждой точке программы для каждого указателя выдает множество фрагментов памяти, адреса которых в этой точке он может содержать. Задача межпроцедурного анализа укладывается в рамки идеи абстрактной интерпретации [5]. Суть её в построении такого приближения семантики программы, которое отображало бы интересующее нас свойство поведения этой программы на стадии исполнения.

Предлагаемый алгоритм анализа обладает такими характеристиками, как потоковая зависимость (flow-sensitivity) и контекстная зависимость (context-sensitivity) [3]. Первое означает, что алгоритм учитывает управление внутри процедуры, что, в принципе, приводит к повышению его точности. Второе – что он пытается различать информацию, приходящую в процедуру по различным путям во время исполнения. Но так как число таких путей может быть потенциально бесконечным, алгоритму необходимо объединять те из них, которые он считает наиболее близкими (и объединение которых внесет минимально возможный консерватизм в его результаты). Основным механизмом, который используется для обеспечения свойства контекстной зависимости, является механизм частичной трансферной функции (ЧТФ) [2]. Он позволяет весьма эффективно выбирать соотношение между скоростью проведения анализа и его точностью, ибо, в общем случае, межпроцедурный анализ указателей является достаточно дорогим процессом, как с точки зрения требуемой памяти, так и с точки зрения времени его проведения.

Большое внимание в работе уделено проблеме обработки рекурсивных циклов. Для решения проблем, которые с этим связаны, было предложено обобщение понятия ЧТФ, которое теперь может содержать информацию о points-to функции не только одной процедуры.

Промежуточное представление

Семантика программы выражает множество всех поведений программы, исполняемой при всевозможных допустимых входах. Сначала программист выражает эту семантику на языке программирования. Затем компилятор переводит её в форму, более удобную для проведения анализов и преобразований. Такая форма называется *промежуточным представлением*. Для проведения глобального анализа указателей было выбрано смешанное контекстно-потоковое промежуточное представление. Это значит, что передать значение от одной операции к другой можно двумя способами. Первый (контекстный) способ заключается в том, что результат первой операции записывается в некоторую переменную или по некоторому указателю, а аргументом второй операции является операция чтение этого объекта или по этому указателю. Второй (потоковый) способ заключается в явном указании того, что результат одной операции является аргументом другой. Такой способ передачи возможен только между операциями, для которых верно, что одна из них выполняется тогда и только тогда, когда выполняется и другая. Кроме того, представление обладает тем свойством, что граф, узлами которого являются операции, а дуги отображают передачу значений между ними потоковым способом, является древовидным.

Потоковый способ передачи значений позволяет существенно сэкономить на числе переменных, необходимых для передачи значений между операциями, а свойство древовидности выражения позволяет легко обходить деревья аргументов, что не требует дополнительной маркировки его узлов-операций.

Выбранное представление обладает тем свойством, что любая модификация памяти может быть осуществлена только посредством операции записи, а получение значения из памяти требует операции чтения. Ниже показано (рис.1), чему в промежуточном представлении соответствуют основные (с точки зрения проводимого анализа) языковые выражения.

Языковое выражение	Промежуточное представление
p = x	op1 READ(x) op2 WRITE(p,op1)
p = *x	op1 READ(x) op2 READ(op1) op3 WRITE(y,op2)
p = &x	op1 OBJ_PTR(x) op2 WRITE(p,op1)
*p = *x	op1 READ(p) op2 READ(x) op3 READ(op2) op4 WRITE(op1,op3)
p = *(x+8)	op1 READ(x) op2 CONST(8) op3 ADD_P(op1,op2) op4 READ(op3) op5 WRITE(p,op4)

Рис. 1. Примеры соответствия промежуточного представления языковым выражениям.

Аналитические структуры

Теперь кратко перечислим некоторые аналитические структуры, которые необходимы для проведения анализа. Первая из них – CFG (control flow graph или *граф управления потоком*) процедуры [1], который строится на основе промежуточного представления и отображает поведение процедуры с точки зрения потока управления. Узлы графа пронумерованы в соответствии с так называемой RPO-нумерацией (Reverse Post Order) [1], причем поддержан механизм быстрого получения узла по его номеру. Обход узлов CFG в соответствии с указанной нумерацией обладает тем свойством, что обработка любого узла происходит после того, как обработаны все его *предшественники*, за исключением предшественников по обратным дугам.

На основе CFG строится *дерево доминаторов* (Dominator-Tree) [1], которое задает частичный порядок доминирования на узлах CFG. Также на основе CFG для каждого узла строится так называемый IDF (Iterated Dominants Frontier или *итерационный фронт доминирования*) [1]. Эта аналитическая структура позволяет для любого узла CFG быстро находить все узлы схождения управления, которые достижимы из этого узла. Это, в свою очередь, дает возможность эффективно и экономно распространять по CFG интересующее нас свойство посредством построения ϕ-функций в узлах схождения управления.

Ещё одна структура данных, необходимая для проведения анализу, это *стек активаций* [4]. С его помощью моделируется стек, который возникнет при исполнении анализируемой программы. Каждая запись этого стека содержит два поля: частичную трансферную функцию с информацией о points-to функции вызванной процедуры и узел CFG вызывающей процедуры, из которого произошел вызов. В начале анализа в верхушку стека помещается ЧТФ стартовой процедуры (в случае С-программы это main), и нуль в качестве точки вызова. С помощью стека активаций легко определить, что функции принадлежат рекурсивному циклу: как только в процессе анализа встречается вызов процедуры, ЧТФ которой уже находится в стеке, все процедуры в стеке от его верхушки до вызванной процедуры, которую назовем *головой рекурсии*, принадлежат рекурсивному циклу. Множество этих процедур назовем *простым рекурсивным циклом* (ПРЦ). Однако рекурсивный цикл, возникающий при исполнении программы, может состоять не только из процедур одного ПРЦ. Если какие-то ПРЦ пересекаются, то необходимо объединить их. Таким образом, любой ПРЦ принадлежит ровно одному такому объединению, которое будем называть максимальным рекурсивным циклом, или просто *рекурсивным циклом*.

Введенное понятие рекурсивного цикла можно пояснить следующим образом. Пусть для анализируемой программы построен граф вызовов: узлы – это функции программы, а направленные дуги между ними отображают тот факт, что одна функция вызывает другую. Тогда простым рекурсивным циклом будет множество процедур, составляющих любой простой цикл в графе вызовов, а рекурсивным циклом будет множество процедур, составляющих его сильносвязную компоненту (strong connected component, SCC).

Пространство имен

Каждой переменной программы соответствует свой собственный фрагмент памяти. Для того, чтобы, с одной стороны, обобщить понятие переменной, а с другой – уточнить его, введем необходимые определения.

Определение 1. *Блок имен* это переменная или объединение имен. *Имя* это блок имен, уточненный множеством локализаций. Разные блоки имен должны представлять непересекающиеся фрагменты памяти.
Определение 2. *Множеством локализаций* блока имен будем называть подмножество множества целых чисел, которое задается парой $\langle f, s\rangle \in \mathrm{Z}\times\mathrm{N}$ следующим образом: $\langle f, s\rangle = \{f + i \cdot s \mid i \in Z\}$. Первый элемент этой пары называется *смещением* множества локализаций, а второй – его *шагом*. Множество локализаций, у которого шаг равен нулю, назовем точечным. Очевидно, что неточечное множество локализаций может быть задано бесконечным множеством различных пар. Поэтому для взаимной однозначности между множествами локализаций и парами, которые их задают, потребуем дополнительно, чтобы для пар с ненулевым шагом выполнялось условие $0 \le f < s$.

Если множество локализаций имеет ненулевой шаг, это означает, что анализ не в состоянии определить, с каким именно конечным фрагментом памяти будет работать операция. Множество локализаций вида $\langle 0,1\rangle$ будем называть *максимальным*, ибо оно описывает всю возможную память некоторого блока имен.

Теперь описание того, как операции некоторой процедуры работают с памятью, можно осуществлять с помощью имен или их объединения (блоков имен). Множество всех тех имен, с которыми работают операции процедуры, назовем *пространством имен* этой процедуры. Из определения следует, что каждой переменной соответствует некоторый блок имен, и что разным переменным соответствуют разные блоки. Каждому блоку имен соответствует определенный тип. Если блок имен это переменная, то тип наследуется от этой переменной следующим образом:

- формальный тип – приписывается блоку, который есть переменная – формальный параметр процедуры;
- локальный тип – приписывается блоку, соответствующему локальной переменной процедуры;
- глобальный тип – приписывается блоку, соответствующему глобальной переменной.

Однако не вся память, с которой работает программа, соответствует какой-то поименнованой переменной. Для единообразия описания работы с памятью введем еще один тип:

- динамический тип – приписывается блоку имен, через который происходит работа с памятью, выделенной одной из процедур захвата памяти malloc(), calloc(), realloc()…

Все вышеперечисленные типы блоков имен будем называть *пользовательскими* типами. Для наглядности, в соответствии с [3], приведем следующую таблицу (рис.2), в которой указано, чему во введенных терминах соответствуют некоторые языковые конструкции.

Тип	Выражение	Соответствующее имя
type *p	P	[p,<0,0>]
struct {type *F;}r	r.F	[r,<f,0>]
type *a[i]	a[i]	[a,<0,s>]
type *p;	*(p op X)	[p,<0,1>]
struct {type *F[];} r	r.F[i]	[r,<f(mod s),s>]
struct {type *F;} a[]	a[i].F	[a,<f,s>]

Рис. 2. Пример того, в какие имена переходят языковые выражения.
f – смещение поля F от начала структуры.
s – размер элемента структуры.

Два имени пересекаются [одно имя содержит другое], если это имена одного блока и множества их локализаций пересекаются [множество локализаций первого содержит множество локализаций второго]. Объединением двух имен одного блока будем называть минимальное имя, которое содержит оба исходных имени. В большинстве случаев объединение двух имен в смысле данного выше определения, будет содержать имена, которые не принадлежали ни первому, ни второму имени. Поэтому представление используемой памяти в виде имен часто является неточным, однако оно позволяет совершать теоретико-множественные операции (объединение, пересечение, разность …) с потенциально бесконечными множествами за время, не зависящее от размеров конкретных множеств (другими словами, за константное время).

Частичная трансферная функция

Трансферная функция процедуры отражает поведение этой процедуры в зависимости от той точки, в которой она была вызвана. Далее точку вызова процедуры будем называть её *вызывающим контекстом*. *Частичная трансферная функция* процедуры (ЧТФ) (Partial Transfer Function, PTF [2]) отражает её поведение только на некотором подмножестве вызывающих контекстов. Совокупность всех таких контекстов называется *областью определения* ЧТФ. Если некоторый вызывающий контекст принадлежит области определения ЧТФ, то нет необходимости заново анализировать процедуры. Достаточно применить ЧТФ к данному контексту и получить вносимое процедурой изменение в points-to функцию вызывающего контекста.

В рассматриваемом случае анализа указателей ЧТФ будет содержать информацию о том, как вызов процедуры, для которой она построена, меняет значения указателей в вызывающем контексте. В каждом конкретном вызывающем контексте, принадлежащем области определения ЧТФ, значения аргументов ЧТФ, очевидно, могут быть разными. Соответствие между аргументами ЧТФ и их конкретными значениями в текущем вызывающем контексте называется *функцией связывания* этого контекста с ЧТФ.

Работа с памятью в некоторой процедуре может осуществляться не только через переменные (которым уже были сопоставлены блоки имен), но и через указатели, содержащие, например, адреса переменных вызывающего контекста. Фактически, значения этих указателей являются, наравне со значениями формальных параметров, аргументами процедуры. Для того, чтобы сделать эти аргументы более явными, введем понятие *расширенного параметра* (РП), который является обобщением понятия (поименнованого) параметра процедуры. Расширенные параметры представляют значения указателей, используемых процедурой, в вызывающем контексте на входе в процедуру.

Теперь, для того, чтобы в терминах имен можно было описывать работу с той памятью, которую представляют РП, введем новый тип блоков имен, называемый *нелокальным*. Нелокальный блок имен соответствует некоторому множеству имен вызывающего контекста, которые представляются расширенными параметрами. Нелокальные блоки имен, как и любые другие, должны удовлетворять тому требованию, что два различных блока имен представляют непересекающиеся фрагменты памяти. Каждому РП соответствует ровно один нелокальный блок. Если значения, представляемые какими-то расширенными параметрами, пересекаются, то им должен соответствовать один и тот же нелокальный блок. Для демонстрации введенный понятий, рассмотрим пример (рис. 3).

Пример	**Связывающая функция для процедуры foo( )**		
	объекты ЧТФ	значение ф-ции связывания	соответствие РП → НЛБ

int x, y, z, *p, *q;
 	
int foo(int **par1,
int **par2)
{
int *swap = *par1;
*par1 = *par2;
*par2 = swap;
}
 	
main()
{
if (cond)
q = &y;
p = &x;
else
q = &x;
p = &z;
 	
foo(&p, &q);
}
	РП(par1)	{p}	N_0

	РП(par2)	{q}	N_1
	РП($name_1$)	{x, z}	N_2
	РП($name_2$)	{x, y}	N_2
	N_0	{p}	РП(par1)
	N_1	{q}	РП(par2)
	N_2	{x, y, z}	РП($name_1$), РП($name_2$)

Рис. 3. Пример, демонстрирующий понятия РП, нелокального блока (НЛБ) и функции связывания на них.
$name_i$ – имя блока N_i, i=1,2,

Кроме уже описанного свойства ЧТФ (отображать вносимые процедурой изменения в points-to функцию вызывающей процедуры в зависимости от вызывающего контекста), она обладает также и еще одним свойством: содержать информацию о points-to функции той процедуры, для которой она построена. Выражена эта информация в так называемой структуре присваиваний, описанию которой посвящен следующий раздел.

Анализ одной процедуры

Для нужд анализа произведем дополнительное разбиение CFG процедуры так, чтобы каждый узел CFG содержал не более одного присваивания, не более одной операции CALL, а узлы схождения управления не содержали бы никаких операций. Функция EvalProc, отвечающая за обработку одной процедуры, получает на вход ptf – ЧТФ процедуры. Суть обработки заключается в итерации по узлам CFG процедуры в соответствии с RPO-нумерацией до тех пор, пока points-to функция этой процедуры в

данном вызывающем контексте не будет окончательно вычислена. Ниже приведен псевдокод этой процедуры (рис. 4.), после которого даны необходимые пояснения.

```
func EvalProc( PTF ptf) {
do {
        changed = FALSE;
        foreach node n ∈ ptf.proc.cfg
            if ( n is assignment )
                EvalAssign( n, ptf);
            else if ( n is join )
                EvalJoin( n, ptf);
            else if ( n is call)
                EvalCall( n, ptf);
        endfor;
} while (changed);
}
```

Рис. 4. Псевдокод процедуры EvalProc.

Обработка отдельной записи, осуществляемая процедурой EvalAssign, состоит из определения тех имен, в которые может производиться запись (имена левой части), и множества имен, указатели на которые могут быть записаны в имена левой части (это имена правой части); для определения этих множеств служит процедура EvalExpr, описанная ниже. Когда эти множества определены, создаются отношения, называемое *присваиваниями*, которые ставят в соответствие каждому имени из множества имен левой части всё множество имен правой части. Эти присваивания привязываются к узлу n CFG процедуры. Присваивания в каждое имя организованы в виде отдельного дерева, узлами которого являются все те узлы CFG, в которых была запись в это имя. Расположение узлов в этом дереве соответствует отношению доминирования, что позволяет эффективно искать ближайший доминирующий узел с присваиванием в данное имя. Если было создано новое присваивание с непустой правой частью, или же произошло пополнение правой части уже имеющегося присваивания, то поднимается флаг changed, который сообщает о необходимости новой итерации по процедуре.

С помощью IDF находится множество узлов схождения управления в CFG, которые достижимы из данного узла n и в эти узлы распространяется информация о том, что в данное имя было присваивание. Когда в процессе очередной итерации алгоритм доходит до обработки узла схождения управления, в нем уже сосредоточена информация о тех именах, для которых необходимо вычислить f-функции. Вычислить f-функцию для некоторого имени name означает создать присваивание в это имя с правой частью, равной объединению множеств правых частей присваиваний в name в каждой из веток управления, сходящихся в данном узле.

Вообще говоря, число имен одного блока может быть бесконечным. Если в цикле происходит обход массива путем прибавления к указателю константы на каждой итерации, то на каждой же итерации будет появляться новое имя, отличающееся от предыдущего смещением в его множестве локализаций. Для сходимости этого итерационного алгоритма необходимо произвести *расширение значений присваивания* [3]. Суть его состоит в том, что в голове каждого цикла, куда в виде f-функций стекается информация обо всех присваиваниях цикла, создаются присваивания, у которых в правой части содержится не более одного имени каждого блока имен. Если имен больше, то они заменяются именем, которое содержит все имеющееся; такое имя

всегда существует, так как имя с максимальным множеством локализаций содержит все имена данного блока.

Обработка выражения (EvalExpr) позволяет определить те множества имен, с которыми имеет дело EvalAssign. Как было сказано при описании промежуточного представления, каждое выражение имеет структуру дерева. Обходя рекурсивно дерево операций, алгоритм каждой его вершине сопоставляем множество имен. Листовыми операциями дерева выражения могут быть только следующие операции:

- CONST; предполагается, что константа не может быть адресом никакого имени. Поэтому этой операции сопоставляется пустое множество имен.
- OBJ_PTR(name); такой операции ставится в соответствие множество из единственного имени name.
- READ(name); множество, которое следует поставить в соответствие этой операции, зависит не только от name, но и от узла CFG, в котором находится операция. Для вычисления этого множества используем процедуры GetPointsTo(name,n,ptf).

GetPointsTo(name,n,ptf) служит для получения в некоторой точке программы, а именно в узле n CFG процедуры ptf.proc, множества значений некоторого имени name. Принцип её работы таков: рассматривается структура присваиваний в данное имя name в обрабатываемой процедуре. Если в этой структуре существует присваивание в каком-то узле m, доминирующем узел n, то результатом GetPointsTo будет то множество имен, которое найденное присваивание ставит в соответствие имени name. В примере, приведенном на рис. 5, значение GetPointsTo(loc_p,n,ptf)={loc}, где n – узел, содержащий последнее присваивание процедуры bar, а ptf – ЧТФ этой процедуры. Если же доминирующего узла с присваиванием в name не нашлось, то возможны следующие ситуации:

- name имеет локальный тип. Тогда результатом будет пустое множество. В примере на рис 5. такая ситуация возникает для локального имени, созданного на основе переменной loc.
- name имеет формальный, глобальный или нелокальный тип и в данном вызывающем контексте его возможными значениями не могут быть указатели ни на какие имена вызывающего контекста. Тогда результатом GetPointsTo снова будет пустое множество. Однако само имя будет занесено во множество имен, которые читались, но в данном вызывающем контексте не содержали никаких указателей в качестве значений. В дальнейшем будем обозначать это множество как NPI (Non-Pointer Inputs). Это множество необходимо для описания механизма переиспользования ЧТФ. В приведенном примере (рис. 5) в такое множество попадает глобальное имя, соответствующее переменной glob и формальное имя, соответствующее par1.
- наконец, name имеет один из типов, перечисленных в предыдущем пункте, но среди его возможных значений могут быть указатели на имена вызывающего контекста. В этом случае по имени name создается расширенный параметр, который и призван выражать значения указателя name, которые используются в процедуре. В рассматриваемом примере (рис. 5) расширенный параметр создается по par2. Функция связывания ставит ему в соответствие локальное имя процедуры func, созданное по переменной a. Если в ЧТФ уже существует такой РП, что множества значений, представляемые РП(name) и этим РП, пересекаются, то необходимо только что созданному РП(name) поставить в соответствие тот же нелокальный блок, что и РП; при этом множество имен, представляемых нелокальным блоком, пополняется

именами, представляемыми РП(name). Если же такого РП не существует, необходимо создать новый нелокальный блок имен для представления значений РП в вызывающем контексте, которому функция связывания поставит в соответствие все те имена, которые представляет РП(name). В любом случае, результатом GetPointsTo будет имя нелокального блока с множеством локализаций вида $\langle 0,0 \rangle$. А для того, чтобы при последующих разыменованиях name, получение нелокального имени вписывалось в общую схему, в стартовом узле CFG создадим присваивание, которое имени name ставит в соответствие созданное нелокальное имя.

```
int *glob;

func(  void ) {
   int a;
   bar( NULL, &a)
}

bar( int* par1, int* par2) {
   int loc, *loc_p = &loc;

   if ( cond )
      *par1 = *glob; /* READ(par1)=∅, READ(glob)=∅,
                        и  glob, par1 → NPI */
   else
      *par2 = loc;   /* RAED(par2) = РП(par2),
                        READ(loc) = ∅ */

   glob = loc_p;      /* READ(loc_p) = loc */
}
```

Рис. 5. Пример, поясняющий работу функции GetPointsTo.

Пусть теперь в процессе обработки выражения встретилась операция, каждому из аргументов которой поставлено в соответствие множество имен (возможно пустое). Если рассматриваемая операция принадлежит классу операций над адресами и вторым её аргументом является константа, то к смещению множества локализаций каждого имени из множества имен её первого аргумента, прибавляется эта константа. Получившееся множество имен ставим в соответствие обрабатываемой операции. Если второй аргумент не является константой, то все имена из множества имен первого аргумента делаем максимальными. Это соответствует тому, что анализ не в состоянии определить точечный фрагмент памяти, с которым работает операция. Если же рассматриваемая операция не принадлежит классу операций работы с адресами, то множество имен, которое ставится ей в соответствие, есть объединение множеств имен её аргументов.

В той части работы, в которой шла речь о создании присваивания, говорилось, что имени ставится в соответствие множество имен, которое является результатом обработки выражения в правой части операции записи. Это действие требует определённого уточнения. Если имя, в которое происходит присваивание, представляет единственный фрагмент памяти, то все происходит равно так, как это было описано. Однако если имя представляет не единственный фрагмент, то множество имен правой части необходимо объединить с множеством имен из доминирующего присваивания, которое может быть получено процедурой GetPointsTo. Это необходимо сделать

потому, что статически не ясно в какой именно кусок памяти происходит запись и, следовательно, нет гарантии, что происходит затирание старых значений. Если имя имеет тип, отличный от нелокального, то достаточно посмотреть, является ли его множество локализаций точечным. Если же имя имеет нелокальный тип, то этого недостаточно. Необходимо посмотреть, какие имена вызывающего контекста представляет соответствующий нелокальный блок имен. Если в вызывающем контексте нелокальный блок представляет единственное точечное имя, тип которого отличен от нелокального, то станем называть его *сингулярным*. Если же нелокальный блок представляет единственное нелокальное точечное имя, то он является сингулярным, в том и только том случае, когда нелокальный блок представляемого имени, в свою очередь, является сингулярный. Только для точечных имен сингулярного блока можно не объединять множество правой части с множеством, даваемым процедурой GetPointsTo. Описанное свойство алгоритма называется свойством *сильного обновления*.

Как уже говорилось, блоки имен должны представлять непересекающиеся фрагменты памяти. Возможно, однако, что в процессе анализа процедуры появятся два различных нелокальных блока, которым функция связывания ставит в соответствие пересекающиеся множества имен вызывающего контекста. Например, существовало два РП, которым были сопоставлены различным нелокальным блоки. Затем был создан еще один (третий) РП, значения которого пересекаются со значениями обоих существующих РП. Необходимо взять нелокальных блок, соответствующий одному из этих РП, сопоставить его вновь созданному РП и пополняя функцию связывания этого нелокального блока значениями, представляемыми новым РП. Таким образом, в структуре ЧТФ появляется два нелокальных блока, которые представляют пересекающиеся множества имен. Нет необходимости устранять подобный дефект сразу же при его появлении. Достаточно и конце каждой итерации по процедуре запустить процесс так называемой *канонизации нелокальных блоков*, который объединит пересекающиеся нелокальные блоки и во всех структурах ЧТФ заменит устаревшие блоки на новые.

Если в процедуре ведется явная работа с глобальной переменной и эта же переменная доступна через некоторый указатель, то нелокальный блок, сопоставленный РП по этому указателю, будет представлять глобальное имя, построенное по данной глобальной переменной. В этом случае два имени разных блоков имен (глобального и нелокального) будут пересекаться. Но, по определению, имена различных блоков не должны пересекаться. Для устранения подобной ситуации необходимо обобщить глобальное имя до имени нелокального блока. Обобщить – потому что нелокальный блок, кроме данного глобального имени, может представлять и другие имена. Так же, как и канонизацию нелокальных блоков, данное обобщение достаточно делать в конце каждой итерации.

Эффект, вносимый вызовом некоторой процедуры в points-to функцию обрабатываемой процедуры вычисляется функцией EvalCall. Её описанию посвящен следующий раздел работы. Здесь же будут описаны некоторые частные случаи, которые не требуют межпроцедурной техники.

При условии, что компилятору подан флаг “доверия к именам стандартных библиотечных процедур”, можно утверждать, что вызовы некоторых библиотечных процедур (например, sin(x), cos(x), …) не меняет points-to функцию вызывающей

процедуры. Про вызовы других известно без непосредственной обработки, как именно они меняют points-to функцию вызывающей процедуры (например, про вызов log(x) известно, что он может только изменить значение флага errno). Подобным образом (без непосредственной обработки) следует поступать и с вызовами процедур захвата динамической памяти: необходимо создать присваивание, левая часть которого получается также как при обработки обычного присваивания – процедурой EvalExpr, а правой частью является имя динамического типа. Вопрос лишь в том, когда создавать новое динамическое имя, а когда следует взять уже существующее.

Предлагается две стратегии работы с динамическими именами. Первая (грубая) заключается в том, что существует единственное динамическое имя, область видимости которого – вся программа. При таком подходе работа с динамическим именем ничем не отличается от работы с любым глобальным именем. Вторая стратегия заключается в том, что динамическое имя, составляющее правую часть присваивания, характеризуется путем в графе вызовов, который привел к рассматриваемому вызову процедуры захвата памяти. Этот подход правомерен, ибо в таком случае различным динамическим именам соответствуют непересекающиеся фрагменты памяти. Число же таких имен в программах, активно работающих с динамической памятью, может быть очень велико. Это, безусловно, увеличивает точность, однако скорость проведения анализа падает, а расходы на память возрастают. Компромиссом является рассмотрение не всего пути в графе вызовов, а лишь финального его отрезка фиксированной длины; он-то и будет характеристикой динамического имени. При таком подходе областью видимости динамических имен будет процедура. Различие между локальными именами и динамическими лишь в том, что после выхода из процедуры, в которой последние были созданы, память, которую они представляют, может быть доступна и в вызывающем контексте. Для этого в вызывающем контексте необходимо создать имена (тоже динамического типа), которые бы представляли эту память. Подробнее это будет описано в следующей части работы.

Межпроцедурный анализ

Эта часть работы посвящена описанию того, что происходит, когда в процессе итерации по процедуре алгоритм встречает операцию вызова. Как было сказано выше, для оценки изменения, вносимого в points-to функцию обрабатываемой процедуры вызовом другой процедуры, служит функция EvalCall. На рис. 6 приведен псевдокод этой функции.

```
func EvalCall( n, ptf){
procedures = FindCallTarget(n, ptf);
foreach proc ∈ procedures  {
bind = RecordActuals( n, proc);

if ( proc ∉ CallStack ) {
        /* нерекурсивный вызов */

tgt_ptf = GetPTF(bind, proc, n, ptf);

if ( needVisit ) {
push(tgt_ptf, bind, CallStack);
EvalProc(  tgt_ptf);
pop( CallStack);
}
ApplySummury( tgt_ptf, bind, n, ptf);
```

```
} else {
        /* рекурсивный вызов */

tgt_ptf = GetPTFFromCallStack( proc);
EvalRerursion(tgt_ptf, bind, n, ptf);
}

endfor;
}
```

Рис. 6. Псевдокод процедуры EvalCall , обрабатывающей операции вызова.

Переменная bind служит для хранения информации о функции связывания вызывающего контексте (узла n CFG процедуры ptf.proc) с ЧТФ обрабатываемой процедуры proc. Функция RecordActuals служит для инициализации этой структуры и подсчета функции связывания для формальных параметров proc. Для нахождения множества процедур, которые будут вызваны в данной точке, используется процедура FindCallTargets. При вызове по косвенности необходимо воспользоваться уже имеющимися результатами анализа для определения возможных значений указателя, по которому происходит вызов. По этому указателю создается РП, в котором запоминается множество процедур (в частности, пустое множество тоже), которое в данном контексте может быть через него вызвано.

Для каждой из найденных процедур определяется, лежит ли ЧТФ этой процедуры в стеке активаций или нет. Если лежит, то вызов является рекурсивным, а вызванная процедура объявляется головой рекурсии. За обработку рекурсивного вызова отвечает процедура EvalRecur, которая будет описана ниже. В случае нерекурсивного вызова необходимо получить ЧТФ для этой процедуры. Возможны три случая:

- В процессе анализа процедура встретилась впервые. Тогда для этой процедуры создается новая ЧТФ и поднимается флаг необходимости обработки (needVisit=TRUE).
- Процедура уже обрабатывалась и для неё существует ЧТФ (или даже несколько ЧТФ), но данный вызывающий контекст не лежит ни в одной из областей определения этих ЧТФ, то есть ни одна из них не может быть применена в данном вызывающем контексте без дополнительной обработки. Тогда, либо создается новая ЧТФ, либо берется одна из имеющихся и её область определения дополняется текущим вызывающим контекстом. Ниже этот процесс будет описан более детально. В любом случае, поднимается флаг необходимости дополнительной обработки (needVisit=TRUE).
- Наконец, если существует ЧТФ, область определения которой содержит данный вызывающий контекст, то ЧТФ может быть применена к данному контексту без дополнительной обработки (needVisit=FALSE).

Последним этапом обработки процедуры является собственно пополнение points-to функции вызывающей процедуры – ApplySummary. Так как points-to функция процедуры выражена в структуре присваиваний ЧТФ, необходимо в точке вызова процедуры создать присваивания, соответствующие изменениям, вносимым вызванной процедурой. Для каждого имени name ЧТФ вызванной процедуры (tgt_ptf), в которое было присваивание, собирается множество его значений в каждом из узлов выхода из процедуры (посредством функции GetPointsTo(name, exit_node, tgt_ptf)). Затем эти множества имен объединяются и с помощью функции связывания выражаются в

терминах имен ЧТФ вызываемой процедуры (ptf). Получившееся множество используется в качестве правых частей присваиваний, которые создаются в вызывающем контексте во все имена ЧТФ ptf, которые соответствуют имени name ЧТФ tgt_ptf. Процесс выражения имен вызванной процедуры в терминах имен вызывающей с помощью функции связывания называется *трансляцией*.

Осталось описать, как происходит трансляция присваиваний, содержащих динамические имена. Подход, при котором существует единственное динамическое имя (которое с точки зрения области его видимости является глобальным), не требует доработки описанной схемы пополнения points-to функции вызывающей процедуры. Напротив, подход, при котором динамическое имя лежит в пространстве имен только той ЧТФ, в которой оно было создано, требует уточнения процесса трансляции, ибо память, описываемая этими именами, может использоваться и в вызывающем (относительно обрабатываемой процедуры) контексте. А, следовательно, для представления работы с этой памятью необходимо создать в вызывающем контексте соответствующие динамические имена. Но делать это следует лишь для имен, работа с которыми действительно возможна вне процедуры их создания. Множество таких имен формируется следующим образом. Все имена динамического типа, которые попали в имена, возвращаемые оператором return, заносятся в это множество. Если существует присваивание, которое будет оттранслировано в вызывающий контекст, в правой части которого находится динамическое имя, оно также добавляется в формируемое множество. Если есть присваивание в динамическое имя, которое уже находится в формируемом множестве, то все динамические имена из правой части этого присваивания также добавляются в множество. Очевидно, в силу конечности числа присваиваний в ЧТФ, этот итерационный процесс сходится.

После того, как множество сформировано, в ЧТФ вызывающей процедуры по каждому из его имен создается динамическое имя, которые характеризуются тем же отрезком пути в графе вызовов, что и имя-оригинал. Создание происходит лишь в том случае, когда имени с такой характеристикой в ЧТФ еще нет (иначе берется существующее). Таким образом, функция связывания пополняется соответствием между динамическими блоками вызывающего контекста и текущей ЧТФ. И лишь после этого осуществляет пополнение points-to функции вызывающего контекста, как это было описано выше.

Определенную сложность для предложенной схемы межпроцедурного анализа представляют нелокальные переходы, которые в языке C, например, осуществляются посредством вызова системных функций setjmp() и longjmp(). Подход к их обработке такой: в процессе анализа процедуры запоминаются узлы CFG, в которых встречались вызовы этих функций. При трансляции результатов в вызывающий контекст происходит обход всех узлов, в которых встретилась функция longjmp(). В каждом из них, как и в узлах выхода из процедуры, собираются множества значений для всех имен, в которые были присваивания. Если в самой процедуре есть вызовы функции setjmp(), то в этих узлах создаются копии набранных присваиваний. Затем это множество присваиваний транслируется в вызывающий контекст и пополняется множеством присваиваний вызывающей процедуры, со значениями в точке вызова. Если в вызывающей процедуре есть узлы с функцией setjmp(), то копии всех набранных присваиваний создаются в этих узлах. Далее продолжается трансляция множества присваиваний вверх по стеку с пополнением его присваиваниями из

процедур в стеке со значениями в точках вызова. При наличии узлов с функцией setjmp(), в них создаются копии набранных присваиваний.

Ниже описывается процесс выбора ЧТФ для обработки процедуры. Пусть в вызванной процедуре существует набор ЧТФ (в частности, одна). Задача состоит в том, чтобы сравнить текущий (новый) вызывающий контекст с областями определений имеющихся ЧТФ. Для этого необходимо перечислить те характеристики, которые формируют область определения ЧТФ.

- Рассмотрим множество имен, которое ранее было обозначено, как NPI – это те имена, по которым алгоритм при обработке процедуры безуспешно пытался создать РП. Если в новом контексте попытка создания РП увенчается успехом, то область определения существующей ЧТФ не содержит новый вызывающий контекст.
- Если в ЧТФ каким-то двум РП были поставлены в соответствие различные нелокальным блоки, а в новом вызывающем контексте значения, которые они представляют, пересекаются и, следовательно, им следует сопоставить один и тот же нелокальный блок, то ЧТФ также не может быть переиспользована в новом контексте. Если же в новом вызывающем контексте некоторое глобальное имя обобщается до нелокального имени, чего не было раньше, то опять же необходим пересчет ЧТФ.
- Пусть в ЧТФ есть РП, для которых было сохранено множество процедур для вызова по косвенности (которое может быть пустым). Тогда для переиспользования ЧТФ необходимо, чтобы множество процедур, которые РП представляет в новом вызывающем контексте, было подмножеством множества процедур, сохраненных в РП. Если это не так, то ЧТФ снова неприменима.

Возможен случай, когда у процедуры нет ЧТФ, подходящей для переиспользования в данном вызывающем контексте, и в соответствии с какими-то ограничениями на их число нет возможности создать новую ЧТФ. Необходимо выбрать одну из множества существующих, область определения которой придется расширить, добавив к ней рассматриваемый вызывающий контекст. Это изменение сделает ЧТФ более универсальной с точки зрения множества контекстов, в которых она применима, но, возможно, менее точной, с точки зрения тех контекстов, в которых она была применена без обобщения. Решение о выборе носит эвристический характер и для построения эвристики предлагается найти числовое выражение для сравнения контекстов, в которых ЧТФ были построены с текущим контекстом, в котором одну из них необходимо будет пересчитать.

Число вновь созданных РП по именам из множества NPI и число новых функций, которые добавляются к множеству процедур, приписанных РП – уже числовые характеристики. Чем они меньше, тем меньше будет загрублена ЧТФ и тем быстрее она будет пересчитана. Необходимо также учесть то, как число новых функций распределено по множеству РП – чем меньше РП, в которых произошло изменение, тем лучше. Теперь перейдем к описанию того, как сравнивать контексты по степени пересечения РП.

Пусть для некоторой процедуры существует ЧТФ ptf. В новом вызывающем контексте по структуре этой ptf создается её копию – ptf′. Процесс этот итерационный; ниже приведены его этапы.

База. На множестве блоков имен задается отношение r следующим образом. Каждый блок имен глобального или формального типа находятся в отношении с самим собой. Далее, это отношение итерационно пополняется парами нелокальных блоков, причем первый элемент каждой такой пары из r будет нелокальным блоком ЧТФ в новом контексте ptf′, а второй – нелокальным блоком существующей ЧТФ ptf. Кроме того, создается множество имен, называемое *рабочим множеством*, в которое помещаются все те имена формального и глобального типа, по которым в ptf были созданы РП.

Шаг. По именам из рабочего множества алгоритм пытается создать РП в ptf′. Возможно, что по некоторым из этих имен создать РП в новом вызывающем контексте нельзя, ибо они ничего в нем не представляют. Если же РП удалось создать, ему сопоставляется нелокальный блок, исходя из значения функции связывания на этом РП. А множество значений функции связывания на нелокальном блоке пополняется значениями созданного РП. Если в ptf′ и ptf существуют РП по именам, блоки которых находятся в отношении r, а множества локализаций равны, то пара нелокальных блоков, поставленных им в соответствие, добавляется в отношение r. После чего происходит очистка рабочего множество имен, для того, чтобы сформировать его заново. Теперь в него войдут имена ЧТФ нового контекста ptf′.

Для формирования рабочего множества рассматриваются все имена ptf, созданные по нелокальным блокам, которые являются вторыми элементами только что добавленных в отношение r пар. Для каждого такого имени name рассматривается множество всех имен ptf′, которые созданы на основе нелокальных блоков, находящихся в отношении r с блоком имени name, и имеющие то же множество локализаций, что и name. Объединение таких множеств для всех указанных имен из ptf и составляет новое рабочее множество.

Если в процессе вычисления функции связывания её значение изменилось хотя бы для одного нелокального блока, то шаг повторяется для всех тех пар блоков, которые были добавлены в r на текущем шаге.

Решение. Когда процесс создания копии ЧТФ для нового вызывающего контекста окончен, необходимо, как и в случае обработки процедуры, провести процесс канонизации нелокальных блоков ptf′. Объединение нескольких нелокальных блоков в один должно найти свое отражение в структуре отношения r. После этого рассматривается двудольный граф, узлами которого являются нелокальные блоки, а дуги отображают наличие пары блоков в построенном отношении. Если существует узел, соответствующий нелокальному блоку ptf′, степень которого больше единицы, это означает, что некоторые РП ptf, которые раньше представляли непересекающиеся множества имен, в новом контексте представляют пересекающиеся. А значит ptf не может быть применима без пересчета в новом контексте.

Следует заметить, что если некоторая ЧТФ была выбрана для переиспользования или пересчета в контексте, в котором не все из её РП, представляют имена вызывающего контекста, то такие РП не удаляются и имена, по которым они были созданы, не переводятся во множество NPI. Вместо этого таким РП функция связывания ставит в соответствие пустое множество. Таким образом, число РП любой ЧТФ никогда не уменьшается.

С помощью построенного графа можно вывести числовую эвристику для оценки степени близости нового контекста к области определения имеющейся ЧТФ. Например, это может быть средняя степень узлов графа, соответствующих РП воссозданной ЧТФ ptf′. Если, наоборот, существует несколько ЧТФ, подходящих для переиспользования в новом контексте, то схожим образом можно оценить, применение какой из них в новом контексте будет давать наименее грубый результат. Для этого можно рассмотреть ту же эвристическую функцию, но на узлах, соответствующих РП существующей ЧТФ ptf.

Далее приведено описание того, как обрабатывается рекурсивный вызов – процедура EvalRecur(). Первая проблема, которая возникает при рассмотрении рекурсии, связана с тем, что в некоторых случаях правило, по которому определяется, что значения двух указателей пересекаются, нуждается в уточнении. В нерекурсивном случае для определения этого факта достаточно было выяснить: пересекаются ли в вызывающем контексте множества имен, которые они представляют. В случае рекурсии это не так. Рассмотрим следующий пример (рис. 7).

```
void recurse (struct ListN *tail) {

        struct ListN head, *tmp;
        head.next = tail;

        if ( condition ) {
                recurse( &head);
        }

        tmp = &head;
        while( tmp != NULL ) {
                tmp->data=0;
                tmp = tmp->next;
        }
}
```

Рис. 7. Пример рекурсии, для которой определение пересечения требует глобализации имени.

Если не вносить никаких изменений в существующую схему, то обработка рекурсивного вызова каждый раз будет создаваться новое нелокальный блок для представления локальной переменной head из очередной активации процедуры recurse. Строго говоря, несмотря на то, что все эти нелокальные блоки, в конечном счете, представляют фрагмент памяти, соответствующий одной и той же переменной, они действительно не пересекаются. Но так как статически алгоритм не может делать никаких утверждений о глубине рекурсии, число нелокальных имен будет неограниченно расти и на каждой итерации цикла значение указателя tmp будет представлено уникальным нелокальным именем. Для того, чтобы избежать этой ситуации и обеспечить сходимость алгоритма, предлагается следующий подход: разным локальным переменным, соответствующим разным активациям процедуры сопоставляется один и тот же блок имен. Тогда в приведенном примере нелокальное имя пересекается с локальным. Однако для того, чтобы определить, что эти блоки имен пересекаются, необходимо выразить все нелокальные имена в терминах имен программы (тех, которые не являются нелокальными). Для этого производится так называемая *глобализация*: все локальные и формальные имена процедур рекурсивного

цикла делаются глобальными в рамках этого рекурсивного цикла. Это достигается посредством приписывания им нового *циклового типа*. Цель этого – сделать все присваивания в имена нового типа видимыми во всех процедурах цикла.

В результате работы анализа ЧТФ любой процедуры из рекурсивного цикла должна содержать информацию о том, как цикл в целом меняет points-to функцию вызывающего (относительно всего цикла) контекста. Поэтому алгоритм анализа вынужден совершать итерации по процедурам цикла до тех пор, пока ЧТФ каждой из них не перестанет изменяться. Однако подобный итерационный процесс требует больших временных затрат. Решение, которое предлагается в этой работе, состоит в том, чтобы рассматривать весь рекурсивный цикл как единую процедуру и вычислять для неё единственную ЧТФ, содержащую информацию о points-to функции всех процедур цикла. Такую ЧТФ будем называть *ЧТФ рекурсивного цикла*.

Этот подход не требует разделения вызывающих контекстов на рекурсивные и нерекурсивные, не требует держать несколько функций связывания, не требует трансляции результатов анализа из рекурсивного контекста внутрь цикла, а также ускоряет сходимость. Кроме того, с помощью рекурсивного цикла можно создать такую же ситуацию, что возникает при обработке внутрипроцедурного цикла: неограниченный рост имен одного блока. Предложенный подход позволяет не вводить нового межпроцедурного оператора расширения значений в головах рекурсивного цикла, а уложить решение в существующую схему.

Еще одна сложность в обработке рекурсивных циклов состоит в том, что заранее неизвестно, какие именно процедуры будут составлять рекурсивный цикл. Это определяется «на лету» во время работы алгоритма. Начиная обрабатывать процедуру, алгоритм не знает, окажется она рекурсивной или нет.

Как только в процессе анализа обнаруживается рекурсивный вызов, все ЧТФ, находящиеся в стеке активаций от его верхушки до головы рекурсии изымаются, и создается ЧТФ рекурсивного цикла, в которой фиксируется множество процедур, входящих в этот цикл. Пусть, для начала, все изъятые ЧТФ были построены для процедур. Созданная ЧТФ помещается в стек активаций и начинается обработка процедуры, являющейся текущей головой рекурсивного цикла. Если встречается вызов процедуры, принадлежащей рекурсивному циклу, то вместо того, чтобы считать функцию связывания, создается присваивание в рамках этой же ЧТФ в цикловые имена, соответствующие параметрам процедуры тех значений, с которыми была вызвана процедура и начинается обработка вызванной процедуру. При этом создаются присваивания в той же ЧТФ. Если же встречается вызов процедуры не из цикла, он обрабатывается обычным образом (как было описано выше). Если же при обнаружении рекурсивного цикла в стеке активаций от верхушки до головы рекурсии содержатся ЧТФ, построенных не только для процедур, но и для рекурсивных циклов, необходимо объединить их в одну ЧТФ. Для этого выбирается некоторая ЧТФ рекурсивного цикла, которая пополняется множеством процедур, входящих в текущий элементарный рекурсивный цикл и во все рекурсивные циклы, ЧТФ которых находятся в стеке. А к пространству цикловых имен добавляются имена, созданные по локальным и формальным переменным всех этих процедур.

Настройки анализа

Приведенная схема анализа во многих случаях даёт достаточно точный результат. Однако при обработке больших программ такая схема требует затрат за время выполнения анализа. Ниже описываются те подходы, которые позволяют путем снижения точности проводимого анализа существенно снизить временные затраты.

- Отказ от различных имен одного блока. Это предполагает, что множество локализаций любого имени является максимальным. Такое решение, кроме того, что экономит память в структурах приваиваний, позволяет существенно ускорить операции с множествами имен.
- Отказ от чувствительности анализа к управлению процедуры (или flow-insensitive версия анализа). Это означает, что все присваивания происходят в стартовом узле CFG процедуры. Нет необходимости вычислять f-функции и любое присваивание сразу же становится видимым в каждой точке процедуры. Такой подход существенно уменьшает число итераций по процедуре, необходимых для вычисления points-to функции в текущем вызывающем контексте. В ЧТФ для рекурсивного цикла также все присваивания производятся в одной точке.
- Моноверсия анализа – для каждой процедуры создается только одна ЧТФ, которая при необходимости обобщается для новых вызывающих контекстов. Это дает значительную экономию памяти и не требует сравнения текущего вызывающего контекста с областью определения существующих ЧТФ, что позволяет ускорить проведение анализа.

Так как ЧТФ рекурсивного цикла содержит информацию обо всем цикле, её пересчет весьма критичен по скорости анализа. Ниже приводятся ситуации, которые заставляют пересчитывать ЧТФ рекурсивных циклов и указываются способы избежать этого.

Число процедур, которые могут являться входами в рекурсивный цикл, бывает весьма велико. Каждый раз, когда алгоритм попадает на обработку рекурсивного цикла через новый вход, необходимо, вообще говоря, заново пересчитать его ЧТФ, так как еще не возникала ситуация, когда формальные параметры этой процедуры представляли бы что-либо в вызывающем контексте (вызывающем – относительно всего цикла). Однако можем считать, что любая процедура рекурсивного цикла может быть входом в этот цикл и уже при первой обработке создать присваивания в текущем (для всего рекурсивного цикла) вызывающем контексте, которые каждому формальному имени ЧТФ поставят в соответствие фиктивные имена. Тогда чтение этих формальных имен приведет к созданию по ним множества РП и соответствующих нелокальных блоков, которые, при необходимости, попадут в структуру присваиваний ЧТФ. И в случае, когда вход в рекурсивный цикл будет осуществлен через новую процедуру, пересчет ЧТФ не будет осуществляться лишь на том основании, что через данные параметры процедуры не приходила информация извне рекурсивного цикла. Заметим, что видом фиктивных присваиваний можно также регулировать соотношение между скоростью и точностью анализа: если все присваивания имеют одну и ту же правую часть, то всем создаваемым РП будут сопоставлен один и тот же нелокальный блок; это снижает точность, но повышается вероятность перепримененения ЧТФ в новых контекстах.

Ситуацией, схожей с уже описанной, является следующая. Внутри рекурсивного цикла происходит чтение многих глобальных имен, но запись в них некоторых адресов осуществляется после выхода из рекурсивного цикла. Если таким образом инициализируется большое количество глобальных имен, то каждый вызов рекурсивного цикла приводит к необходимости пересчета. Для того, чтобы этого

избежать, в вызывающем контексте создаются фиктивные присваивания, которые позволят анализу создать необходимые РП и учесть соответствующие нелокальные имена в структурах присваиваний. Это в большинстве случаев позволит не пересчитывать ЧТФ цикла. Опять же, видом фиктивных присваиваний можно регулировать соотношение точность-скорость, как это было описано выше.

Возможна следующая редкая, но очень неприятная ситуация. Процедура f(func*) – голова рекурсивного цикла, в качестве параметра получает адрес некоторой процедуры для вызова её по косвенности внутри цикла. Если вызов рекурсивного цикла через процедуру f осуществляется часто, причём с разными значениями параметра, то каждый раз ЧТФ цикла придется пересчитывать. Возможен следующий выход: до проведения анализа, во время предварительного прохода по представлению (который и так осуществляется для служебных нужд) для каждой процедуры собирается информация о значениях её параметров типа «ссылка на процедуру», в том случае, когда в качестве параметра явно фигурирует имя функции. Если в процессе анализа, данная процедура оказалась головой рекурсивного цикла, то при первой же обработке в качестве значения параметра процедуры рассматривается не та единственная функция, которая была подана в качестве аргумент данного вызова, а все набранные во время предобработки процедуры. При таком походе нет необходимости пересчитывать ЧТФ цикла при остальных вызовах f с новыми значениями параметра.

Наследование результатов анализа

После того, как алгоритм завершил последнюю итерацию по главной процедуре программы, анализ закончен. Его результатами являются множества ЧТФ для каждой процедуры, причем число ЧТФ для разных процедур может быть разным. Если у каких-то процедур нет ни одной ЧТФ, это значит, что они не обрабатывались а, следовательно, представляют собой мертвый код и могут быть удалены. Для остальных процедур совокупность ЧТФ представляет собой их обобщенную трансферную функцию в контексте анализируемой программы.

Рассматриваются только операции работы с памятью (READ и WRITE) и операции вызова (CALL). Каждой такой операций одной процедуры ставится в соответствие упорядоченный набор, длина которого равна числу ЧТФ, созданных для этой процедуры в процессе анализа. Для операций работы с памятью элементом такого набора будет множество имен, с которым операция может работать во всех тех контекстах, в которых применялась соответствующая ЧТФ. Для операции вызова элементом набора будет пара множеств: одно – множество имен, в которые внутри вызываемой процедуры (или нескольких процедур) может осуществляться запись, другое – множество имен, которые в этой процедуре (процедурах) могут читаться.

Построение таких наборов осуществляется следующим образом. Для каждой ЧТФ процедуры один раз производится обход представления этой процедуры, почти так, как это делалось в процессе проведения анализе, с той лишь разницей, что теперь нет необходимости обрабатывать операции вызова, ибо весь вклад от них уже учтен в ЧТФ. Также нет необходимости уходить в вызывающий контекст для выяснения, можно ли по некоторому имени name создать РП: если в одном из тех контекстов, в которых использовалась данная ЧТФ, РП можно было создать, значит, он был создан, если нет, то результатом операции чтения name станет пустое множество. Как и в процессе анализа, в процессе обхода представления и вызывается функция EvalExpr, которая

ставит в соответствие операциям READ и WRITE множества имен, с которыми они работают. Для того, чтобы создать пару множеств для операции CALL, обходится множество всех присваиваний в узле вызова. Объединение всех имен, в которые в данном узле существуют присваивания, дает множество имен, в которые может происходить запись. А объединение всех правых частей этих присваиваний дает множество имен, которые в процедуре могут читаться. Однако в случае flow-insensitive версии анализа нет возможности среди всех присваиваний процедуры выделить те, которые отражают изменения, вносимые конкретной операцией вызова. Поэтому в процессе проведения такой версии анализа в каждой ЧТФ для каждой операции вызова хранится пара множеств: первое содержит имена, которые процедуры, вызываемые этой операцией, могут читать, второе – в которые они могут писать. При трансляции результатов в вызывающий контекст эти множества пополняются.

Для проверки того, что две операции рассматриваемых типов могут (при одной о той же активации процедуры) работать с одним и тем же фрагментом памяти (иначе говоря, могут *конфликтовать*), следует найти пересечения всех пар множеств имен, относящихся к одной и той же ЧТФ. Если существует непустое пересечение, то существуют контексты, в которых операции могут конфликтовать.

В процессе анализа в структуре ЧТФ можно сохранять информацию о том, в каких вызывающих контекстах процедуры она использовалась. Тогда, если существует такой вызывающий контекст, в котором какие-то пары её операций не конфликтуют во всех тех ЧТФ, которые использовались для обработки этого контекста, целесообразно создать копию этой процедуры (другими словами, клонировать процедуру). В операциях этой копии наследуются только те множества имен, которые соответствуют указанным ЧТФ, а в рассматриваемом контексте вместо вызова исходной процедуры подставляется вызов её копии.

Опишем теперь, что происходит с результатами анализа при inline-подстановке одной процедуры в другую. Очевидно, что оставить результаты анализа неизменными нельзя, даже если число ЧТФ у обеих процедур одинаковое, ибо результаты анализа каждой из них выражены в терминах пространства имен разных ЧТФ и уже нет функций связывания, которые бы позволили перетранслировать имена вызываемой процедуры в имена вызывающей. Предлагаемое решение состоит в следующем. Во время наследования результатов анализа, в каждой операции создается одноэлементный список, в единственный элемент которого помещается набор множеств. Каждое множество набора содержит имена, с которыми работает операция в тех контекстах, в которых применялась соответствующая ЧТФ. При inline-подстановке для каждой операции-копии (READ, WRITE, CALL), которая создается в вызывающей процедуре по операции-оригиналу подставляемой процедуры, заводится список. Этот список есть конкатенация списка результатов операции вызова, вместо которой происходит подстановка, (этот список идет сначала), и списка результатов операции-оригинала. Таким образом, каждая новая inline-подстановка удлиняет список результатов операций. Тогда, если вопрос о конфликте возникает между операциями, одна из которых существовала в процедуре до подстановки, а другая появилась после неё, то ответ будет таким же, как если бы вопрос о конфликте ставился для первой из этих операций и операции вызова, вместо которой произошла подстановка. Если же вопрос ставится для операций, которые обе появились после подстановки, то первые равные элементы списков результатов пропускаются, а вопрос о конфликте имеет такой же ответ, как для операций-оригиналов в вызываемой процедуре до подстановки.

Ещё одним полезным результатом анализа, который следует сохранить, является множество процедур для вызова по косвенности. При финальном обходе представления для каждой операции вызова по косвенности можно получить соответствующий этому вызову РП, в котором сохранено множество процедур. Объединение таких множеств для всех ЧТФ процедуры дает требуемый результат, который и сохраняется в операции.

Предложенная схема анализа позволяет получать ответ на вопрос о зависимости по памяти между операциями. Однако если множество имен, которое было сопоставлено некоторой операции, содержит нелокальное имя, нельзя определить, с какими именно фрагментами памяти имеет дело эта операция. Наличие же подобной информации может привести к более точной работе некоторых оптимизаций.

Тем не менее, не следует рассматривать информацию о том, с какими фрагментами памяти работает операция, как безусловное уточнение унаследованной информации. Ниже приведен пример (рис. 8), который показывает, что информация о пользовательских именах в операции может давать более консервативный ответ о зависимости между операциями.

```
int a, b;

void
h(void){
        f(&a, &b);
        f(&b, &a);
}

void
f(int*p, int*q){
        g(p, q);
}

void
g(int *par1, int *par2){
        *par1=0;
        *par2=0;
}
```

Рис. 8. Пример показывает, что трансляция в пользовательские имена может давать более грубый ответ.

Если у функций f и g нет никаких других вызывающих контекстов, кроме тех, что указаны в примере, то анализ определит, что обе операции записи процедуры g никогда не конфликтуют (им будут соответствовать разные нелокальные имена). Однако множества пользовательских имен, с которыми эти операции работают, будут совпадать.

Рассмотренный пример показывает также, что поддержание в процессе анализа соответствия между нелокальными блоками и пользовательским именам, не решает проблемы. Второй вызов функции f приведет к тому, что её ЧТФ будет использована без дополнительного пересчета, ибо новый вызывающий контексте полностью совпадает с областью определения ЧТФ. Но в этом случае информация о новых

значениях параметров не достигает процедуры g. Этого можно было бы избежать, если включить в область определения ЧТФ те частные значения, которые принимали её РП. Но такое решение почти полностью лишает предложенный алгоритм возможности переиспользования ЧТФ, что делает его неприменимым на практике. Для решения указанной проблемы предлагается следующий подход.

После того, как отработала моноверсия алгоритма анализ, строится полный граф вызовов программы. Затем выполняется следующее преобразование: каждая сильносвязная компонента этого графа стягивается в один узел. Множество вершин получившегося графа взаимнооднозначно со множеством ЧТФ, которые были построены в процессе проведения анализа. Получившийся граф является ациклическим. В узел этого графа, который соответствует ЧТФ главной процедуре программы не входит ни одна дуга. Начиная с этого узла, на графе проводится RPO-нумерацию. После чего запускается процедура InheritUserNames (её псевдокод приведен на рис. 9), которая реализует алгоритм наследования в операциях пользовательских имен.

```
func InheritUserNames( ptf_graph){

        foreach i = 1 to NodeNumber(ptf_graph) {
                ptf = GetPtfByNumber(ptf_graph, i);

                foreach proc ∈ ptf and each node n ∈ ptf.proc.cfg {

                        if ( n is call ) {

                                /* Наследуем результаты для операции n.call */
                                InheritNamesForNodeOpers(n);

                                /* Для процедур, вызываемых из ЧТФ, строим
                                   функцию связывания для нелокальных блоков. */
                                foreach proc ∈ n {
                                        tgt_ptf = GetPtfByProc(proc);
                                        ConstructBindingFunction( n, tgt_ptf);
                                endfor;

                        } else if ( n is assign ) {

                /* Для операций работы с памятью, принадлежащих
                   узлу  n, наследуем результаты в терминах
                   пользовательских имен */
                                InheritNamesForNodeOpers(n);
                                }
                 endfor;
        endfor;
}
```

Рис. 9. Псевдокод функции наследования результатов анализа в терминах пользовательских имен.

В силу ацикличности графа и упомянутого ранее свойства RPO-нумерации, любой узел будет обрабатываться лишь после того, как будут обработаны все его предшественники, а, следовательно, станут известны объединенные значения всех функций связывания во всех возможных вызывающих контекстах. За вычисление такой функции связывания отвечает процедура ConstructBindingFunction. Кроме того, все

значения функции связывания могут быть выражены в терминах пользовательских имен. В самом деле, в ЧТФ, которая содержит главную процедуру программы, нет никаких имен, кроме пользовательских. Следовательно, для ЧТФ тех процедур, которые вызываются из неё, функция связывания будет содержать только пользовательские имена. Пусть теперь есть процедура, объединенная функция связывания которой содержит только пользовательские имена. Тогда для любой ЧТФ процедуры, вызываемой из рассматриваемой процедуры, функцию связывания строится так, как это делалось во время анализа. После этого все нелокальные имена, которые вошли во множество её значений, заменяются на пользовательские имена из функции связывания рассматриваемой ЧТФ. Когда для каждого нелокального имени известно множество пользовательских имен, которое оно выражает, в операциях процедуры (READ, WRITE, CALL) производятся соответствующие замены.

Результаты

В этой части работы приводятся результаты работы алгоритма анализа на некоторых задачах из тестовых пакетов SpecInt92 и SpecInt95. Замеры проводились на машине со следующими характеристиками: Pentium Xeon CPU 3.06Hz, cashe size 512KB. Та версия алгоритма анализа, которая использовалась для получения нижеприведенных результатов, обладал следующими характеристиками:

- моноверсия (единственная ЧТФ для процедуры);
- flow-insensitive (нечувствительность к потоку управления процедуры);
- не различаются имена одного блока имен;
- включены эвристики, ускоряющие сходимость анализа на рекурсивных циклах.

Сложность каждой из приведенных в табл. 1 задачи характеризуется такими параметрами, как:

- число строк исходного кода задачи (строки).
- число пользовательских процедур в задаче (проц.).
- число процедур в самом большом рекурсивном цикле задачи (ССК – сильно-связная компонента).

Работа алгоритма анализа на указанных задачах характеризуется следующими параметрами:

- временем работы алгоритма на указанной машине, выраженное в секундах (время);
- степень переиспользования ЧТФ без дополнительного анализа (переисп.). Этот параметр показывает среднее значение по всем обработанным процедурам от следующей величины: процентное отношение числа вызывающих контекстов, в которых ЧТФ процедуры была переиспользована без дополнительного анализа, к числу всех вызывающих контекстов этой процедуры, которые встретились в процессе анализа.
- степень порывов зависимостей (разрывы). Этот параметр показывает среднее значение по всем процедурам от следующей величины: процентное отношение числа порванных зависимостей к числу всех возможных пар операций READ, WRITE и CALL данной процедуры.

Табл.1

Результаты анализа на тестах из пакетов SpecInt92 и SpecInt95.

Название теста	строки	проц.	ССК	время	переисп.	разрывы
SpecInt92						
008.espresso	13500	382	2	9	79.5%	64.8%
022.li	7413	380	313	162	88.5%	51.4%
023.eqntott	3376	62	1	1	70.9%	72.9%
026.compress	1503	16	1	1	61.5%	52.5%
072.sc	8086	166	7	24		
085.gcc	84524	1689	350	1508	84.4%	62.3%
SpecInt95						
099.go	28547	394	1	9	75.2%	71.2%
124.m88ksim	17939	286	11	20	83.7%	64.0%
129.compress	1420	30	0	1	68.5%	47.8%
130.li	6916	373	311	189	87.9%	53.0%
134.perl	23678	298	26	54	89.5%	68.1%
147.vortex	52633	1072	37	290	79.3%	50.9%

Анализируя приведенные результаты, не сложно заметить, что время работы алгоритма зависит, во многом, не столько от величины анализируемой задачи (число её строк и процедур), сколько от размеров рекурсивных циклов этой задачи. Именно это является обоснованием того особого внимания, которое в описанной схеме их обработки. Кроме того, следует отметить, что степень переиспользования ЧТФ весьма высока. Это свидетельствует о том, что предложенные эвристики позволяют довольно быстро обобщить область определения ЧТФ процедур задачи для применения её во всех вызывающих контекстах данной процедуры.

Заключение

На сегодняшний день существует много хороших алгоритмов, которые решают задачу межпроцедурного анализа указателей с достаточной степенью детализации на уровне одной процедуры [6]. Однако все они крайне консервативны на межпроцедурном уровне. Попытки создать алгоритм, обладающий хорошей межпроцедурной детализацией анализа, свелись к двум основным подходам. Первый из них основан на использовании графа активаций процедур; при таком подходе результаты анализа привязываются к его узлам [4]. Но в силу того, что размер этого графа растет экспоненциально с увеличением некоторых параметров анализируемой программы, на практике данный подход оказывается применимым для анализа только очень небольших программ.

Второй подход основан на использовании описанного механизма ЧТФ и моделировании стека вызовов в процессе работы алгоритма. Этот подход не обладает указанным недостатком предыдущего, однако в своем изначальном виде также непригоден для анализа больших программ. Его основной недостаток состоит в том, что он очень медленно работает в контексте программ, содержащих рекурсивные цикла.

Предложенная модификация алгоритма, основанного на использовании ЧТФ, ускоряет этот алгоритм до уровня, позволяющего использовать его в промышленных компиляторах. Основная идея модификации состоит в том, что сильно-связные компоненты графа вызовов рассматривается как единое целое. Для всех процедур, входящих в цикл используется одна и та же ЧТФ. В результате этого теряется степень детализации результатов анализа, но происходит выигрыш в скорости его работы. Другими новшествами данной работы следует считать:

- механизм наследования результатов анализа при таких межпроцедурных преобразованиях, как inline-подстановка и cloning.
- механизм наследования результатов, выраженных в терминах пользовательских имен.
- гибкий механизм управления чувствительности анализа к контексту (context-sensitivity level).
- алгоритм сравнения вызывающих контекстов и принятия решения о переприменимости ЧТФ.
- система эвристик, ускоряющая сходимость алгоритма.

Предлагаемый алгоритм тестировался на задачах из пакетов SPECint92 и SPECint95 [7]. На входящих в эти пакеты приложениях модифицированный алгоритм показал хорошее время работы с минимальными потерями в точности вычисляемой информации.

Список литературы

1. **Steven S. Muchnik** “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers, San Francisco CA.
2. **Brian R. Murphy, Monica S. Lam** “Program Analysis with Partial Transfer Function”, Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, January, 1999, pages 94-103.
3. **Robert P. Wilson, Monica S. Lam** “Efficient Context-Sensitive Pointer Analysis C Programs”, ACM SIGPLAN, Conference on Programming Language Design and Implementation, June 1995.
4. **Maryam Emami** “A practical interprocedural alias analysis for optimizing/parallelizing C compiler”. Master’s thesis, School of Computer Science, McGill University, August 1993.
5. **P. Cousot, R. Cousot**, “Abstract interpretation framework”. Journal of logic and Computation, 2(4) 511-547, August 1992.
6. **Marc Shapiro, Susan Horwitz** “Fast and Flow-Insensitive Points-To Analysis”. Proceedings of 24th ACM SIGPLAN-SIGACT symposium of programming language, pp.1-14, Paris, France, January (1997).
7. **Standard** Performance Evaluation Corporation (www.spec.org).