УДК 004.4'4

Алгоритм восстановления позиций выражений в исходном коде Cloud Sisal программ

Гордеев Д. С. (Институт систем информатики им. А.П. Ершова СО РАН)

В статье предложен алгоритм восстановления позиций выражений в исходном коде программ на языке Cloud Sisal. Актуальность исследования обусловлена важностью точного сопоставления элементов абстрактного синтаксического дерева с фрагментами исходного текста для построения инструментов разработки, таких как редактор исходного кода, визуальный отладчик, средства диагностики ошибок. Предлагаемый подход решает проблему неполной информации о позициях в выходных данных синтаксических анализаторов, модификация которых затруднительна. В работе описан разработанный трехфазный алгоритм, включающий этапы восстановления последовательности лексем, вычисления позиций лексем и вычисления позиций вершин абстрактного синтаксического дерева. Асимптотическая оценка времени выполнения алгоритма линейно зависит от объема входных данных и не превышает O(n), где n — количество символов в исходной программе.

Ключевые слова: *CPPS, Cloud Sisal, внутреннее представление, абстрактное синтаксическое дерево, AST, лексема, синтаксический анализатор.*

1. Введение

В лаборатории конструирования и оптимизации программ ИСИ СО РАН ведется разработка системы облачного параллельного программирования CPPS [4], являющейся интегрированной средой разработки и исполнения программ на языке функционального программирования Cloud Sisal [5]. Одним из ключевых компонентов CPPS является визуальный отладчик, который реализует визуальное изображение внутреннего представления Cloud Sisal программ [3]. Внутреннее представление IR является иерархическим атрибутированным ориентированным ациклическим графом с портами. Данное представление состоит из вершин, содержащих входные и выходные порты. Вершины IR соответствуют функциям и операциям, входные порты соответствуют аргументам, а выходные порты соответствуют результатам вычислений. Для повышения удобства визуальной отладки требуется синхронизация между визуальным представлением

IR графа и исходным текстом Cloud Sisal программы. Возникает задача вычисления позиций выражений в исходном тексте по соответствующим им вершинам IR графа. Решение данной задачи необходимо для реализации таких функций, как: выделение выражений в исходном коде при выделении вершин в IR графе, отображение визуальных эффектов для фрагментов исходного текста программы при распространении данных по дугам IR графа, динамическая установка и снятие точек останова в визуальном редакторе IR графа, динамическая установка и снятие точек останова редакторе исходного кода. Поскольку вершины IR тесно связаны с узлами абстрактного синтаксического дерева AST [1], исходная задача сводится к задаче вычисление соответствия между узлами дерева AST и их позициями в исходном тексте Cloud Sisal программы.

Задача точного сопоставления элементов абстрактного синтаксического дерева с фрагментами исходного текста программ является актуальной и рассматривается в не только в работах, связанных с визуализацией программ. В работе об автоматической верификации C-программ на основе смешанной аксиоматической семантики [10] приведено описание протокола обратной трансляции из промежуточного C-kernel-представления на исходный язык C-light. При трансляции из C-light в C-kernel модифицированные конструкции обогащаются метаинформацией. Эта метаинформация включает данные о примененных при трансляции правилах и о позициях данных конструкций в исходной программе, включая номера строк. Таким образом, данный протокол обратной трансляции является примером решения задачи о поддержке обратного отображения программы из промежуточного представления в исходное. Недостатком данного протокола обратной трансляции является необходимость модификации транслятора для поддержки работы с метаинформацией. В работе о системе IF1-Viewer [8], предназначенной для отображения графовых представлений Sisal-программ в виде промежуточной формы IF1, реализована функциональность подсветки строк в исходном коде Sisal-программы при нажатии на соответствующий узел графового представления в графическом редакторе. Недостатком системы IF1-Viewer является подсветка строк в целом без детализации до конкретных столбцов или позиций в строке, а также отсутствие устойчивости к небольшим модификациям исходного кода, таким как добавление комментариев и скобок. В системе HASKEU [7], предназначенной для разработки программ на функциональном языке Haskell, реализована функциональность одновременных графического и текстового представлений Haskell-программ, а также функциональность синхронизации между данными представлениями. Недостатком данной работы является отсутствие детального описания реализации синхронизации между графическим и текстовым представлениями Haskell-программ. В работах [2, 6] описаны

компоненты трансляции и компиляции программ на языках SISAL 3.*, которые применяются для визуализации внутреннего представления Cloud Sisal программ [3], однако возвращаемая информация о позициях выражений в исходном тексте программы является неполной.

В данной работе описывается трехфазный алгоритм восстановления позиций вершин дерева AST в исходном тексте. В разделе 2 описана формальная постановка задачи, приведены обозначения и основные термины. В разделе 3 описан алгоритм, состоящий из трёх этапов: восстановления последовательности лексем слова из соответствующего дерева AST, восстановление позиций лексем во входном слове с помощью посимвольного сканирования входного слова и восстановление позиций вершин AST с помощью восстановленных позиций лексем. Для каждого из этапов приведён псевдокод. В разделе 4 приведены доказательства утверждений о том, что алгоритм всегда завершается, имеет линейную сложность относительно длины входного слова, и для всех лексем слова вычисляются позиции во входном слове. В разделе 4 также приводится обсуждение применимости предложенного алгоритма в случае возможности модифицировать синтаксический анализатор, возвращающий AST.

2. Постановка задачи

Пусть задан контекстно-свободный язык L. Пусть G это однозначная контекстно-свободная грамматика, порождающая язык L. Пусть w это слово (исходный текст программы) из языка L. Пусть SA это синтаксический анализатор, построенный для грамматики G. Пусть AST это абстрактное синтаксическое дерево, построенное для слова w анализатором SA. Пусть V_{AST} это множество вершин из дерева AST. Позицией подстроки в тексте (лексемы или выражения) будем называть четвёрку натуральных чисел (l_S, c_S, l_E, c_E) из N^4, где l_S и c_S это номера строки и столбца начала подстроки соответственно, l_E и c_E это номера строки и столбца окончания подстроки соответственно. Требуется построить отображение F: $V_{AST} \rightarrow N^4$, которое каждой вершине дерева AST сопоставляет позицию в исходном тексте w.

3. Описание алгоритма

Входные данные алгоритма: слово w из языка L и дерево AST = SA(w).

Выходные данные алгоритма: отображение $F : V_{AST} \rightarrow N^4.V_{AST}$

Алгоритм состоит из трех этапов:

1) Восстановление последовательности лексем слова w из дерева AST.

2) Восстановление позиций лексем в слове w.

3) Восстановление позиций вершин дерева AST.

3.1. Восстановление последовательности лексем

Вход: Корень дерева AST.

Выход: Упорядоченный список лексем, соответствующих листьям дерева AST.

Алгоритм представляет собой рекурсивный обход дерева в глубину, в процессе которого посещаются все листовые узлы. Лексемы, хранящиеся в этих узлах, последовательно добавляются в выходной список в порядке их обхода. Данный порядок соответствует порядку следования лексем в исходном тексте для корректно построенного AST.

```
function ExtractLexemes(AstNode node) → Lexeme[]
begin
   if node is leaf then
      begin
         return [node.lexeme]
      end
   else
      begin
         lexemes = []
         for childNode in node.children
            lexemes.append(ExtractLexemes(childNode))
         return lexemes
      end
end
```

3.2. Восстановление позиций лексем

Вход: Исходный текст w, список лексем Lexeme[].

Выход: Отображение P: N→ N^4, сопоставляющее каждой лексеме из L ее позицию в w.

Алгоритм выполняет сканирование слова w и для каждой лексемы l находит ее первое вхождение в тексте, начиная с текущей позиции сканирования. Алгоритм руководствуется следующими правилами:

а) Обработка комментариев: Алгоритм распознает и пропускает однострочные и многострочные комментарии, определенные в грамматике G. При подсчете номеров строк и столбцов длина комментариев учитывается, но сами комментарии в список лексем не входят.

б) Поиск вхождений: Для лексемы l алгоритм ищет подстроку, совпадающую с l, начиная с текущей позиции в тексте. После успешного сопоставления текущая позиция сканирования перемещается на символ, следующий за окончанием лексемы l, и начинается поиск следующей лексемы.

в) Обработка лишних скобок: Алгоритм корректно обрабатывает области текста, не соответствующие ни одной лексеме из входного списка пропуская их. Это обеспечивает устойчивость к форматированию исходного кода.

Приведённый ниже псевдокод алгоритма опирается на семантику конструкции switch-case, при которой выполняется только одна case-ветвь, и, соответственно, не требуется добавлять инструкцию break для каждой case-ветви. Заметим, что в switch из цикла while только две case-ветви не увеличивают индекс i, указывающий на текущий символ в слове w. Первая такая case-ветвь (*) соответствует переходу в состояние обработки потенциально избыточных открывающих скобок, в котором начинается накапливание позиций открывающих скобок. Вторая такая case-ветвь (**) соответствует возврату из состояния открывающих скобок, при котором используются позиции только самых правых открывающих скобок. При этом соответствующие лексемы открывающих скобок уже обработаны в процессе накапливания позиций открывающих скобок.

```
function BuildLexemePositions(string w, Lexeme[] lexemes) → (N → N^4)
P = {}
current_line = 1
current_column = 1
lexeme_index = -1
mode = text
sub_mode = none
foreach lexeme in lexemes
begin
   lexeme_index += 1
   length = len(lexeme)
   tokenScanCompleted = false
   while (tokenScanCompleted = false && i < len(w))
      switch (mode, sub_mode)
         case (text, _) when w[i..i+2] = "//":
            mode = single_line_comment; current_column += 2; i += 2;
```

```
case (text, _) when w[i..i+2] = “/*”:
   mode = multi_line_comment; current_column += 2; i += 2;
case (text, _) when w[i..i+2] = “\r\n”:
   current_line += 1; current_column = 0; i += 1;
case (text, none) when w[i..i+len] = lexeme:
   tokenScanCompleted = true; current_column += len; i += len;
   position = (current_line, current_column, current_line, current_column + len)
   P = P U (lexeme_index, position)
case (text, none) when w[i..i+1] = “(”: // (*)
   j = lexeme_index
   while (lexemes[j] = “(”) begin  j += 1 end
   lb_count = j – lexeme_index
   fixedSizeQueue.setSize(count)
   sub_mode = lb
case (text, none):
   current_column += 1; i += 1;
case (text, lb) when lb_count > 0 and w[i..i+1] = “(” :
   tokenScanCompleted = true
   fixedSizeQueue ← (current_line, current_column, current_line, current_column + len)
   lb_count -= 1
   current_column += 1; i += 1;
case (text, lb) when lb_count > 0:
   current_column += 1; i += 1;
case (text, lb) when lb_count = 0 and w[i..i+len] = lexeme:
   fixedSizeQueue ← (current_line, current_column, current_line, current_column + len)
   current_column += 1; i += 1;
case (text, lb) when lb_count = 0 and w[i..i+1] = “(”: // (**)
   for(int k = 0; k < fixedSizeQueue.Size; k++)
      P = P U (lexeme_index + k – fixedSizeQueue.Size, fixedSizeQueue[k])
   sub_mode = none
case (text, lb) when lb_count = 0:
   current_column += 1; i += 1;
case (multi_line, _) when w[i..i+2] = “*/”:
   mode = text
```

```
            current_column += 2; i += 2;
        case (multi_line_comment, _) when w[i..i+2] = "\r\n":
            current_column = 0; i += 1; current_line += 1
        case (multi_line_comment, _):
            current_column += 1; i += 1
        case (single_line_comment, _) when w[i..i+2] = "\r\n":
            mode = text
            current_column = 0; i += 1; current_line += 1
        case (single_line_comment, _):
            current_column += 1; i += 1
    end
    return P
```

3.3. Восстановление позиций вершин дерева AST

Вход: Дерево AST, отображение P.

Выход: Отображение F: $V_{AST} \rightarrow N^4$

Алгоритм представляет собой рекурсивный обход AST. Для каждой вершины $v \in V_{AST}$ позиция вычисляется как объединение позиций всех ее дочерних вершин. Обход вершин AST производится в том же порядке, что и в алгоритме из раздела 3.1, при этом значение index соответствует количеству обработанных лексем из слова w.

```
procedure BuildAstNodePosition(AstNode node, N → N^4 P, integer index = 0, F = {})
begin
if node is leaf then
    index += 1
    F = F U {(node, P(index))}
else
    begin
        start_index = index
        for childNode in node.children
            BuildAstNodePosition(childNode, P, index, F)
        first = P(start_index + 1)
        last = P(index)
        position = (first.start_line, first.start_column, last.end_line, last.end_column)
```

```
        F = F U {(node, position)}
    end
end
```

4. Обсуждение

Предложенный трехфазный алгоритм представляет решение для случая, когда существующий синтаксический анализатор возвращает дерево AST с недостаточной информацией о позициях выражений в исходном тексте программы или без неё и недоступен для модификации. Если существует возможность использовать лексический анализатор отдельно от синтаксического, то вместо рекурсивного обхода дерева AST на первом этапе предложенного алгоритма, на котором восстанавливается последовательность лексем, возможно использовать вызов лексического анализатора. Далее, если лексический анализатор доступен для модификации, то второй этап предложенного алгоритма возможно исключить, если модифицировать лексический анализатор так, чтобы для каждой извлекаемой лексема вычислялась и сохранялась её позиция в исходном слове w. Далее, если синтаксический анализатор доступен для модификации, третий этап предложенного алгоритма можно исключить, если модифицировать синтаксический анализатор так, чтобы при построении дерева AST каждая вершина снабжалась информацией о позиции соответствующего выражения в исходном слове w, наследуемой от позиций порождающих его лексем. Если же модификация синтаксического или лексического анализаторов невозможна или трудоёмка, то предложенный трехфазный алгоритм представляет собой полное и эффективное решение поставленной задачи.

4.1. Свойства алгоритма

Предложенный трехфазный алгоритм обладает следующими свойствами.

Утверждение 1. Алгоритм всегда завершается.

Доказательство:

Этап 1: Обход конечного дерева AST гарантированно завершается, так как количество узлов $|V_{AST}|$ конечно, и каждый узел посещается ровно один раз.

Этап 2: Длина исходного текста |w|=n. Каждая итерация цикла либо увеличивает индекс i, который указывает на позицию сканирования слова w, либо соответствует переключению в режим обработки открывающих скобок, либо обратному переключению из режима обработки открывающих скобок. Причём, если группа открывающих скобок разделена только пробельными символами или комментариями, то для такой группы переключение в

режим обработки открывающих скобок произойдёт ровно один раз. Таким образом, количество итераций, не увеличивающих индекс i не превышает 2/3*n в случае максимально возможного количества открывающих скобок. И, поскольку $i \leq n$, алгоритм завершится за не более чем 2n шагов.

Этап 3: Аналогично этапу 1.

Утверждение 2. Сложность алгоритма составляет O(n), где n — длина слова w.

Доказательство:

Этап 1: Сложность обхода дерева в глубину равна $O(|V_{AST}|)$, где V_{AST} множество вершин. Известно, что количество вершин в дереве AST пропорционально количеству лексем для многих однозначных грамматик [9]. При этом количество лексем не превышает количество символов в слове w. Соответственно, сложность обхода дерева AST равна O(n).

Этап 2: Длина исходного текста $|w|=n$. Каждая итерация цикла либо увеличивает индекс i, который указывает на позицию сканирования слова w, либо соответствует переключению в режим обработки открывающих скобок, либо обратному переключению из режима обработки открывающих скобок. Причём, если группа открывающих скобок разделена только пробельными символами или комментариями, то для такой группы переключение в режим обработки открывающих скобок произойдёт ровно один раз. Таким образом, количество итераций, не увеличивающих индекс i не превышает 2/3*n в случае максимально возможного количества открывающих скобок. И, поскольку $i \leq n$, то сложность алгоритма равна O(n).

Этап 3: Аналогично этапу 1.

Утверждение 3. Все лексемы из списка Lexeme[] получают позиции этапе 2.

Доказательство: Поскольку слово w принадлежит языку L, а список лексем получен из AST для w, последовательность лексем в L с точностью до пробельных символов и комментариев совпадает с последовательностью лексем в w. Следовательно, этап 2, пропуская комментарии и пробелы, гарантированно находит каждую следующую лексему, так как поиск начинается с текущей позиции и продолжается с позиции за текущей лексемой, поиск продолжается только в правом направлении, и все лексемы из восстановленной последовательности существуют в слове w.

5. Заключение

В работе представлен алгоритм восстановления позиций узлов AST в исходном коде. Алгоритм всегда завершается, имеет линейную сложность и не требует модификации существующего синтаксического анализатора. Для визуальной отладки программ на языке Cloud Sisal в рамках системы CPPS, где вершины IR наследуют структуру AST, и, соответственно, позиции выражений в исходном тексте, предложенный алгоритм полностью решает задачу соответствия изображений вершин внутреннего представления IR и языковых выражений в исходном тексте программы на языке Cloud Sisal. Это соответствие даёт возможность включать и выключать точки останова, не только используя визуальные изображения портов, дуг и вершин, а также фрагменты исходного текста, такие как имена переменных или параметров. Также это соответствие отрывает возможность добавлять отладочные визуальные эффекты непосредственно к изображению исходного текста Cloud Sisal программы в визуальном отладчике.

Список литературы

1. Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд. М.: Вильямс, 2008. 1184 с.
2. Глуханков М. П., Дортман П. А., Павлов А. А., Стасенко А. П. Транслирующие компоненты системы функционального программирования SFP // Современные проблемы конструирования программ. — Новосибирск: ИСИ СО РАН, 2002. — Вып. 9. — С. 69–87.
3. Гордеев Д. С. Визуализация внутреннего представления программ на языке Cloud SISAL, Научная визуализация, 2016. – Том 8, № 2. – С. 98–106.
4. Касьянов В. Н., Гордеев Д. С., Золотухин Т. А., Касьянова Е.В., Кондратьев Д.А. Система облачного параллельного программирования CPPS: визуализация и верификация Cloud Sisal программ // под ред. В. Н. Касьянова. Новосибирск: ИПЦ НГУ, 2020. 256 с.
5. Касьянов В. Н., Касьянова Е. В. Язык программирования Cloud Sisal, – Новосибирск, 2018. – 45 с. – (Препринт/РАН, Сиб. отд-ние, ИСИ; N181).
6. Стасенко А. П., Пыжов К. А., Идрисов Р. И. Компилятор в системе функционального программирования SFP // Вестник Новосибирского государственного университета. Серия: Информационные технологии. – 2008. – Т. 6, № 3. – С. 135-146.
7. Alam A., Bush V. HASKEU: An editor to support visual and textual programming in tandem // Proceeding of the 2016 SAI Computing Conference (SAI). London, UK, July 13-15, 2016. IEEE. pp. 805-814. DOI: https://doi.org/10.1109/sai.2016.7556071

8. Chen H., Shirazi B., Thrane S., Marquis J. IF1-Viewer: A Visual Tool for Graphical Display and Execution of SISAL Programs // Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications. Phoenix, USA, April 12-15, 1994. IEEE. pp. 206-212. DOI: https://doi.org/10.1109/PCCC.1994.504116
9. Grune D., Jacobs C. J. H. Parsing Techniques: A Practical Guide (Second Edition). Springer, 2008. — 662 p.