А. Е. Москаль

ПРЕДСТАВЛЕНИЕ СВЯЗИ ИСХОДНОГО И ВЫХОДНОГО ТЕКСТОВ ПРИ АВТОМАТИЧЕСКОМ ПОРОЖДЕНИИ ТЕКСТОВ ПРОГРАММ

Введение

При использовании макропроцессоров существует проблема представления связи порожденного текста с исходным. Такая привязка необходима для корректной выдачи диагностики об ошибках, работы отладчика, browser'a и т. п.

Следует отметить, что автоматическая генерация текста программы является очень распространенным механизмом, используемым не только в макропроцессорах в собственном смысле этого слова. Аналогичные проблемы возникают во многих других системах, например в широко известной системе построения трансляторов YACC [1]. Кроме того, близкие проблемы возникают в системах с расширяемым синтаксисом [2, 3], в системах для автоматического построения кодогенераторов ([4] и другие).

Хотя проблема является общей для всех макропроцессоров, рассмотрение будет производиться на примере макропроцессора языка C [5], который является наиболее известным.

Другие подходы к проблеме

Обычным решением проблемы является привязка текста с точностью до строки. Например, в препроцессорах языка C она осуществляется при помощи вставки в порожденный текст специальных операторов, задающих имя файла и номер строки в этом файле для каждой строки порожденного текста (так называемый оператор #line[1]).

Это является удовлетворительным решением проблемы в случае языка C (имеющего крайне слабые макровозможности, которые к тому же ориентированы на построчную обработку), но гораздо менее приемлемо для более развитых макроязыков (например, для языка широко используемого макропроцессора M4).

Даже в случае, если построчная привязка считается удовлетворительным решением проблемы, при таком подходе трансляторы с языка C, как правило, неспособны точно указать позицию в строке, приведшую к возникновению ошибки.

Однако кроме этих недостатков, существуют более серьезные. Рассмотрим следующий фрагмент программы на языке C++:

```
#define for_all(ptr, list) \
    for (List * ptr = list; \
      ptr != NUL; ptr = ptr -> next)

  int f (List * list)
  {
    int s = 0;
    for_all (plist, lis)
      s += plist -> info;
    return s;
  }
```

В данном примере содержится две ошибки: использование NUL вместо NULL и неверно написанное имя переменной list в параметре макроса.

Обычно применяемый подход даст нам оба сообщения о неописанных именах, которые будут привязаны к строке, содержащей вызов макроса. Если применительно к имени переменной list это вполне разумная привязка, то для имени NUL это не слишком удачный вариант.

---

[1] Вставка этой информации непосредственно в порожденный текст не является необходимой. Того же результата можно достичь, задав отдельно таблицу соответствия для строк выходного файла.

В случае более сложного макроса поиск причины такой ошибки может быть источником серьезных трудностей (особенно если ошибка имеет не столь очевидный характер, либо если в одной строке вызывается несколько макросов).

Первый вариант, который приходит в голову, — это для каждого символа выходного текста хранить указатель на байт в одном из исходных файлов, из которого он получен. Этот способ даст удовлетворительные результаты в данном примере, но будет вести себя совершенно неудовлетворительно, например, в таком случае:

```
#define isIdLetter(c)\
   (isalpha (c) || (c) == '_')
```

Для корректной работы этого макроса должно быть доступно описание функции isaplha. Если это не так, то данный подход приведет к привязке сообщения об ошибке в строке с описанием макроса, тогда как причину ошибки следует искать в месте его использования. Более того, при таком способе привязки место вызова макроса, приведшее к ошибке, установить почти невозможно.

В случае работы browser'a или отладчика, а также в случае использования более развитых макросредств (формирования новых идентификаторов, определения макросов внутри других макросов и т. п.) проблемы только усугубляются.

Становится ясным, что попытка однозначной привязки позиции выходного текста к исходному тексту является принципиально недостаточной. Разумным выходом представляется хранение полной информации о происхождении каждого байта выходного файла, позволяющей проследить историю его формирования.

Предлагаемая структура

Предлагаемое решение основывается на наблюдении, что порождаемый текст может быть представлен как результат последовательности замен подстрок в тексте: #include заменяется на содержимое включаемого файла, оператор условной трансляции — на ветвь then или else, вызов макроса — на его тело, вхождения формальных параметров в тело макроса — на значения фактических параметров и т. п.

Таким образом, мы получаем следующее представление порождаемого текста, состоящее из четырех типов записей, каждый из которых описывает некоторый текст, реально (или «виртуально») существовавший во время генерации текста. Все эти записи содержат в качестве первого поля длину описываемого ими текста (поле length).

- FILE <length> <file_name> <file_contents> описывает содержимое файла, который принимал участие в порождении текста (т. е. главного файла либо файла, подключенного при помощи операции вставки текста #include); поле <file_name> задает имя файла, поле <file_contents> — содержимое файла.
- STR <length> <string> — литеральная строка символов. Поле <string> содержит текст строки. Запись типа STR необходима для представления строк, не имеющих образа в исходных текстах: например, даты/времени трансляции, номера строки и т. п. Тип STR, как правило, не используется для представления текстов, имеющих сложную структуру (т. е. возникших в результате макроподстановки).
- SEG <length> <base> <start> — результат выборки из строки, описываемой записью <base>, подстроки длиной <length> символов, начиная с позиции <start>.
- CHANGE <length> <base> <beg> <end> <new> <tag> — результат замены группы символов в строке, описываемой записью <base>, начиная с позиции <beg> до позиции <end>, на содержимое строки <new>. Элемент <tag> обозначает причину замены: вызов макроса, оператор #include, подстановка параметров и т. д.

Легко видеть, что по любой из таких записей соответствующий ей текст (или любой фрагмент этого текста)[2] может быть однозначно восстановлен при помощи несложной

[2] Более того, при реконструкции наперед заданного отрезка текста необходимо обойти только те фрагменты нашего представления, которые имели отношение к его возникновению. При этом для хранения фрагментов формируемого текста достаточно одного буфера с длиной, равной длине результата, который может быть размещен перед вызовом функции (и памяти для организации стека вызовов). Это позволяет весьма эффективно использовать вышеописанные структуры как представление текста при написании самого макрогенератора.

рекурсивной процедуры (текст этой процедуры может быть найден в Приложении — см. функцию repr).

Результатом макрогенерации является одна из вышеупомянутых записей, представляющая текст, полученный в результате работы макропроцессора. Она может быть конвертирована в текстовый файл (для «отдельно стоящего» макропроцессора) либо непосредственно использована как представление текста (для макропроцессора, встроенного в транслятор). Полезным побочным результатом использования такого представления является то, что существует представление для фрагментов текста, появившихся только в процессе макрогенерации (например, фактический аргумент макрокоманды может быть представлен ссылкой на узел, который представлял его во время генерации).

Для получения массива привязок отрезка выходного текста (в общем случае желательно иметь дело именно с отрезком — одна только начальная позиция не дает возможности полноценно справиться с такими ситуациями, как, например, формирование имени переменной путем конкатенации строк) следует просто выполнить процедуру обхода, близкую к процедуре, используемой для выборки подстроки, с учетом следующих замечаний:

- Если в записях CHANGE мы попадаем в заменяемый фрагмент, то наряду с происхождением нового текста надо проследить и происхождение этого фрагмента. При этом следует учитывать причину замены: например, в языке C нас, скорее всего, не интересуют замены, связанные с операторами условной трансляции.
- Непосредственным источником информации о привязке являются записи типа FILE. Записи типа STR, как правило, не несут содержательной информации. Записи типа CHANGE могут быть использованы для детального описания происхождения текста, однако, по нашему опыту, попытки их показа на пользовательском уровне приводят только к нагромождению ненужной информации.

Улучшенное представление

Описанное выше представление имеет серьезный недостаток: оно приводит к очень длинным цепочкам замен (например, если в тексте программы производится 1000 подстановок простого макроса без параметров, то чтобы добраться до привязки фрагмента из начала текста, придется просмотреть цепочку из 1000 записей CHANGE). Это, вообще говоря, может приводить к квадратичной сложности некоторых операций поиска.

Поэтому описанную ранее структуру имеет смысл изменить следующим образом: вместо записи CHANGE будем использовать запись REPLACE следующего вида:

```
REPLACE <length> <base> <changes>:
  vector of (<new_text>, <new_start>, <start>, <end>)
```

где каждый элемент массива <changes> описывает замены фрагмента текста <base> в диапазоне <start>...<end> на <new_text>.

Поле <new_start> задает начало замененного фрагмента в результирующем тексте.

Эта информация является избыточной (она может быть легко восстановлена путем просмотра всех элементов массива замен), но необходима для обеспечения быстрого поиска нужных замен по диапазону позиций в выходном тексте.

Остальные координаты заменяемых фрагментов даются в терминах исходного фрагмента <base> (и не должны иметь взаимных пересечений). Массив <changes> упорядочен по возрастанию позиций фрагментов, что позволяет использовать в нем двоичный поиск. Это приводит к снижению сложности алгоритма (с квадратичной до $n \log n$).[3]

Замечания о способе применения

Для систем, в которых используется описанный формат, естественным способом использования привязки является просто вызов соответствующих процедур просмотра привязки в тот момент, когда им необходима информация (например, в момент выдачи диагностики об ошибках). Примерный текст этой процедуры приводится в Приложении (см. функцию links).

В случае, если результат работы макрогенератора подается на вход программе, написанной без учета возможности использования этого формата (например, уже существующему

[3] Строго говоря, это неверно: в случае, если начало следующей замены содержится в конце результата предыдущей, усовершенствованное представление вырождается в описанное ранее, но на практике такое бывает редко.

транслятору), возможно написание постпроцессора, который обработает сообщения об ошибках, оттранслировав координаты текста (их следует получить в терминах порожденного текста, т. е. использование конструкций типа #line противопоказано) в координаты в исходном тексте.

Очевидным возражением против использования описанной выше структуры в качестве основы для представления текста в макрогенераторе являются соображения эффективности. Однако на практике оказывается, что при кэшировании значений функций, используемых для выборки текста из описанной структуры (так называемом *memoisation*), потери скорости снижаются практически до нуля.

Заключение

Описанная выше схема была реализована в рамках работы над проектом See2K и использовалась для установления связи между конструкциями, обнаруженными анализатором программ на языке C/C++ в препроцессированном тексте, и исходным текстом.

Для получения этой информации был реализован препроцессор ANSI C, который в качестве внутренних структур данных использовал описанное представление. Некоторой неожиданностью для автора было то, что использование этого представления привело к существенному прояснению внутренней логики работы макропроцессора. Поэтому можно считать, что описанная техника полезна безотносительно к задаче получения привязки.

Кроме того, ценным свойством этой структуры оказалось то, что она дает естественное представление для промежуточных фрагментов текста, которые не имеют образа в выходном тексте, например для вызова макроса внутри условия оператора #if. Это позволяет естественным и информативным образом отразить ссылки на такие макросы в информации об использовании идентификаторов (и очень полезно для диагностики ошибок в таких фрагментах).

Аналогичная структура используется в разрабатываемом в настоящее время компиляторе с языка C, где она применяется для выдачи диагностики и порождения отладочной информации.

Разработанный препроцессор проигрывает по скорости препроцессору GNU C примерно в два раза, что в основном следует отнести на отсутствие некоторых специфических оптимизаций процесса макрогенерации: например, макропроцессор компилятора GNU C избегает повторного чтения и просмотра текста файлов, имеющих вид

```
#ifndef NAME
#define NAME
    ...
#endif
```

в случае, если имя NAME уже определено.

Затраты памяти под хранение структур данных макропроцессора также сравнительно невелики и обычно превосходят размер выходного текста не более чем в 2-3 раза, что не представляет никакой проблемы при объемах памяти, имеющихся у современных компьютеров.

Приложение. Текст процедур для чтения и просмотра привязки

В качестве примера приведем тексты процедур на Standard ML [6][4]:

```
datatype sg = SG of int * int; (* text segment beg:end (end not included) *)
infix 0 :: SG

datatype text =
  STR    of string
| FILE   of string * string (* name contents *)
| SEG    of text * sg
| CHANGE of text * sg * text; (* base seg new *)

(* вычисление длины `text' *)
fun len (STR     (   contents )) = String.size contents
  | len (FILE    (_, contents )) = String.size contents
```

[4] Использовалась версия SML/NJ для выборки подстроки и формирования массива привязок фрагмента. Для простоты приводится вариант, в котором каждая подстановка представляется отдельной записью.

```
  | len (SEG     (_, fr SG to )) = to - fr
  | len (CHANGE  (base, fr SG to, new)) =
      len base - (to - fr) + len new;

(* вычисление строки, определенной узлом представления и диапазоном в нем *)
fun repr (fr SG to) (STR  (   contents)) =
     substring (contents, fr, to - fr)
  | repr (fr SG to) (FILE (_, contents)) =
     substring (contents, fr, to - fr)
  | repr (fr SG to) (SEG (text, fr' SG _)) =
     repr (fr+fr' SG to+fr') text
  | repr (segm as (fr SG to))
    (src as CHANGE (base, fr' SG to', new)) =
    let val to'' = fr' + len new;
        val d = to' - fr'  - len new
    in
        if      to <= fr'  then repr segm base
        else if fr <  fr'  then repr (fr SG fr')
           base ^ repr (fr' SG to) src
        else if to <= to''
             then repr (fr - fr' SG to - fr') new
        else if fr < to'' then repr (fr SG to'')
                src ^repr (to'' SG to) src
        else repr (fr + d   SG to + d  ) base
    end

(* сборка списка ссылок в исходный текст, соответствующий отрезку
порожденного текста *)
fun links _ (STR _) = []
  | links segm (file as FILE (name, _)) =
     [(name, segm, repr segm file)]
  | links (fr SG to) (SEG (text, fr' SG _)) =
     links (fr+fr' SG to+fr') text
  | links (segm as (fr SG to))
     (src as CHANGE (base, fr' SG to', new)) =
    let val to'' = fr' + len new;
        val d = to' - fr'  - len new
    in
        if      to <=  fr'  then links segm base
        else if fr < fr'
             then links (fr SG fr')
                  base @ links (fr'  SG to ) src
        else if to <=  to''
             then links(fr - fr' SG to  - fr')
                  new @ links (fr' SG to') base
        else if fr < to''
             then links (fr - fr' SG to''- fr')
                  new @ links (to'' SG to) src
        else
             links (fr + d   SG to + d   ) base
    end
;
```

Файл "t1.cc" (входной файл препроцессора):

```
# define M 123
++
# include "t1.inc"
--
```

Файл "t1.inc":

```
[M]
```

Файл, получившийся на выходе препроцессора[5]:

```
# 1 "/home/msk/cpp-sml/t1.cc"

++
# 1 "/home/msk/cpp-sml/t1.inc"
[123]
# 4 "/home/msk/cpp-sml/t1.cc"
--
```

Результат вычисления формулы "links (68 SG 71) text" (диапазон соответствует фрагменту "123"):

```
  [("/home/msk/cpp-sml/t1.cc",11 SG 14,"123"),
   ("/home/msk/cpp-sml/t1.inc",1 SG 2,"M"),
   ("/home/msk/cpp-sml/t1.cc",18 SG 36,
    "# include \"t1.inc\"")]
```

Указатель литературы

1. **Johnson S. C.** Yacc — yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey.
2. **De Rauglaudre D.** CamlP4. Version 2.00.1. November 18, 1998. Institut National de Recherche en Infomatique et Automatique http://cristal.inria.fr
3. **Cardelli L.** An Implementation of $F_{<:}$. Digital Equipment Corporation, SRC Research Report 97, February 1993 (Revised June 1997).
4. **Emmelmann H.** BEG — a Back End Generator. User Manual, November 9, 1989, GMD Foschungsstelle an der Univesität Karlsruhe.
5. **Kernighan B.W., Ritchie D.M.** The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall, 1978. 274 p.
6. **Milner R., Tofte M., Harper R.** The Definition of Standard ML. MIT Press, 1990.

[5] Операторы #line сохранены для обеспечения работы стандартного механизма привязки сообщений об ошибках в трансляторе с языка C