

编译原理 (A) 大作业

算符优先分析表和卷积优化

李子龙 518070910095 [单人完成]

2021 年 6 月 26 日

目录

第一部分 算符优先分析表	2
1 题目分析	3
2 算法描述	4
2.1 集合依赖关系	4
2.2 三进 DFS	4
2.3 识别关系	6
3 函数接口	6
 第二部分 卷积优化	 7

第一部分 算符优先分析表

运行环境

操作系统 Windows

语言 Rust[1]

```
opg [filename]
```

程序输出

样例 1 输入

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | i
```

样例 1 输出

	*	()	+	i	\$
*	>	<	>	>	<	>
(<	<	=	<	<	
)	>		>	>		>
+	<	<	>	>	<	>
i	>		>	>		>
\$	<	<		<	<	=

样例 2 输入

```
E -> E + E | E * E | ( E ) | id
```

样例 2 输出

The grammar is ambiguous.

1 题目分析

首先需要声明在算符优先语法中算符优先级的定义 [2]。

定义 1 对于两个终结符 T_1 和 T_2 ，有下面的算符优先级定义（其中 U_1 是非终结符）

1. $T_1 = T_2$ 如果存在产生式 $U \rightarrow xT_1T_2y$ 或 $U \rightarrow xT_1U_1T_2y$ 。
2. $T_1 < T_2$ 如果存在产生式 $U \rightarrow xT_1U_1y$ 而且存在一个推导 $U_1 \Rightarrow z$ 使得 T_2 是 z 的最左终结符。
3. $T_1 > T_2$ 如果存在产生式 $U \rightarrow xU_1T_2y$ 而且存在一个推导 $U_1 \Rightarrow z$ 使得 T_1 是 z 的最右终结符。

本部分即针对一个上下文无关文法，输出算符优先分析表。如果文法是有二义性的，将会报错。如果两个终结符之间没有上述三个关系的其中一个，将会留空，意为没有优先关系。

根据定义 1，可以对符号构造下面两个集合以判断情况 2 和 3：

定义 2 假设 V_T 是该文法终结符号对应的集合， V_N 是该文法非终结符号对应的集合。对符号 U_1 定义下面两个集合：

$$FIRSTVT(U_1) = \{T | (U_1 \Rightarrow Ty \vee U_1 \Rightarrow U_2Ty) \wedge T \in V_T \wedge U_2 \in V_N\} \quad (1)$$

$$LASTVT(U_1) = \{T | (U_1 \Rightarrow xT \vee U_1 \Rightarrow xTU_2) \wedge T \in V_T \wedge U_2 \in V_N\} \quad (2)$$

除了上面的定义方法，对于这样的集合，还有这样的性质：

$$(U_1 \Rightarrow U_2y) \Rightarrow FIRSTVT(U_2) \subseteq FIRSTVT(U_1) \quad (3)$$

$$(U_1 \Rightarrow xU_2) \Rightarrow LASTVT(U_2) \subseteq LASTVT(U_1) \quad (4)$$

这样定义 1 就有了如下的等价定义：

定义 3 对于两个终结符 T_1 和 T_2 ，有下面的算符优先级定义（其中 U_1 是非终结符）

1. 如果找到了这样的产生式右部 T_1T_2 或 $T_1U_1T_2$ ，那么 $T_1 = T_2$ 。
2. 如果找到了这样的产生式右部 T_1U_1 且 $T_2 \in FIRSTVT(U_1)$ ，那么 $T_1 < T_2$ 。
3. 如果找到了这样的产生式右部 U_1T_2 且 $T_1 \in LASTVT(U_1)$ ，那么 $T_1 > T_2$ 。

2 算法描述

2.1 集合依赖关系

首要任务就是对于每一个非终结符求出 $FIRSTVT$ 和 $LASTVT$ 。式 (1) 和 (2) 所对应的终结符为图中的盲端，式 (3) 和 (4) 所对应的非终结符导出的依赖节点为中间节点。这样就可以构造出集合的依赖关系有向图。

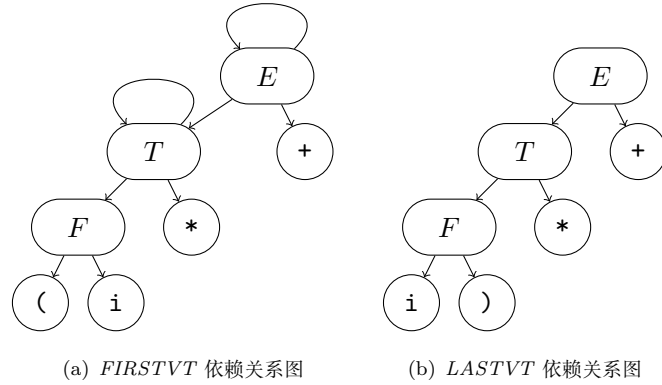


图 1: 依赖关系图

2.2 三进 DFS

一进 DFS: 归并分类 现在的依赖关系图依然是有向有环图。根据包含关系的特性，如果出现

$$S_1 \subseteq S_2 \subseteq \cdots \subseteq S_n \subseteq S_1$$

所对应的环路

$$S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n \rightarrow S_1$$

则这些集合都是相等的：

$$S_1 = S_2 = \cdots = S_n$$

使用 DFS 检测环路，并采用类似并查集的方法归并同类的元素，记录每个非终结符所对应的集合号码，以及每一类的非终结符集合。

表 1: 类别号码

(a) <i>FIRSTVT</i>			(b) <i>LASTVT</i>		
<i>F</i>	<i>E</i>	<i>T</i>	<i>F</i>	<i>E</i>	<i>T</i>
0	2	4	1	2	0

二进 DFS: 连接消环 通过再一次的 DFS 构造类别之间的 DAG (有向无环图), 因为此时同类型内部的环边已经被消除。

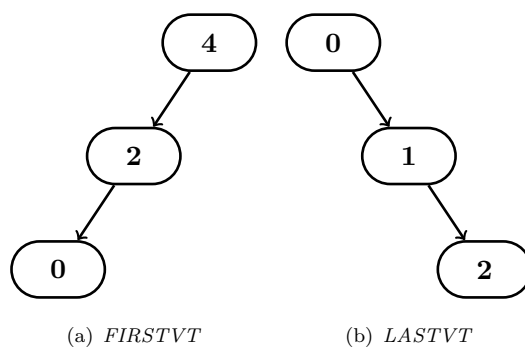


图 2: 类别连接

三进 DFS: 映射输出 对于每一个非终结符, 查类别号码表 1 得到其对应类别, 然后在类别连接图 2 中从该类别节点进行 DFS, 合并路上对应的终结符节点, 就可以得到该非终结符对应的集合。

表 2: 集合

(a) <i>FIRSTVT</i>		(b) <i>LASTVT</i>	
<i>E</i>	i + (*	<i>E</i>	i +) *
<i>T</i>	i (*	<i>T</i>	i) *
<i>F</i>	i (<i>F</i>	i)

2.3 识别关系

最后一步，就是根据定义 3 来遍历产生式检测终结符号之间的优先关系了。在识别之前，需要现在产生式的集合中加入对于开始符号的输入首尾标记：

$$E \rightarrow \$E\$$$

因为该符号比较特殊，其优先级是额外定义的 [3]：

$$\$ < T, T > \$, \$ = \$, \quad \forall T \in V_T$$

并不属于原有的语法，所以要在这个地方再加入。

最终就可以得到本部分开头的程序输出。

3 函数接口

以下是运行样例 1 时由 ufttrace[4] 生成的函数调用列表。

```
std::rt::lang_start() {
  std::rt::lang_start::_{{closure}}() {
    std::sys_common::backtrace::_rust_begin_short_backtrace() {
      core::ops::function::FnOnce::call_once() {
        main::main() {
          // 程序入口
          main::opg_generate() {
            // 生成函数
            main::gen_productions();
            // 生成产生式
            main::gen_non_terminals();
            // 生成非终结符
            main::gen_firstvt() {
              // 构造 FIRSTVT 集
              main::dfs::compose_elements() {
                // 聚合元素
                main::dfs::Dfs::dfs() {
                  // DFS 函数
                  main::dfs::Dfs::dfs_merge() {
                    // 一进 DFS
                    main::dfs::Dfs::merge();
                    // 一进 DFS 子函数
                  } /* main::dfs::Dfs::dfs_merge */
                  main::dfs::Dfs::dfs_conn();
                  // 二进 DFS
                  main::dfs::Dfs::dfs_map();
                  // 三进 DFS
                } /* main::dfs::Dfs::dfs */
              } /* main::dfs::compose_elements */
            } /* main::gen_firstvt */
          main::gen_lastvt() {
            // 构造 LASTVT 集
            main::dfs::compose_elements() {
              main::dfs::Dfs::dfs();
            } /* main::dfs::compose_elements */
          } /* main::gen_lastvt */
          main::get_terminals();
          // 生成终结符
          main::find_eq();
          // 寻找相等关系
          main::find_less();
          // 寻找小于关系
          main::find_greater();
          // 寻找大于关系
          _<main::table::OpTable as core::fmt::Display>::fmt() { // 输出
            main::table::OpTable::to_string();
          }
        }
      }
    }
  }
}
```

```

    } /* _<main..table..OpTable as core..fmt..Display>::fmt */
    main::table::OpTable::to_string();
    } /* main::opg_generate */
    } /* main::main */
    } /* core::ops::function::FnOnce::call_once */
    } /* std::sys_common::backtrace::_rust_begin_short_backtrace */
    } /* std::rt::lang_start::_{{closure}} */
} /* std::rt::lang_start */

```

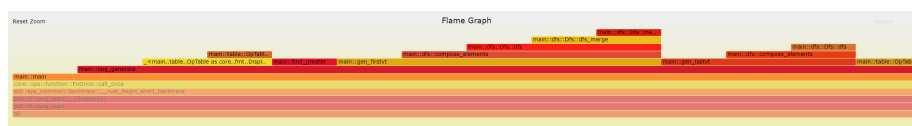


图 3: 火焰图 [5]

以上结果由 [该脚本](#) 生成。关于函数的更多信息，请访问 [API 文档](#)。

第二部分 卷积优化

运行截图

```

logcreative@logcreative:~/CompilerPrinciple$ /usr/bin/python3 /home/logcreative/CompilerPrinciple/cov2d/conv2d.py
primfn(A 1: handle, B 1: handle, compute 1: handle) -> ()
attr = {"global_symbol": "main", "tir.noalias": True}
buffers = {compute: Buffer(compute 2: Pointer(float32), float32, [1, 32, 32, 32], [1]),
           B: Buffer(B 2: Pointer(float32), float32, [32, 3, 3, 3], [1]),
           A: Buffer(A 2: Pointer(float32), float32, [1, 3, 32, 32], [1])}
buffer map = {A 1: A, B 1: B, compute 1: compute} {
  attr [pad temp: Pointer(float32)] "storage_scope" = "global";
  allocate(pad temp, float32, [3468]) {
    for (i1: int32, 0, 3) {
      for (i2: int32, 0, 34) {
        for (i3: int32, 0, 34) {
          pad temp[(((i1*1156) + (i2*34)) + i3)] = @tir.if then else((((i1 <= i2) && (i2 < 33)) && (1 <= i3)) && (i3 < 33)), (float32*)A_2[(((i1*1024) + (i2*32)) + i3) - 33]], 0f32, dtype=float32)
        }
      }
    }
    for (ff: int32, 0, 32) {
      for (yy: int32, 0, 32) {
        for (xx: int32, 0, 32) {
          compute 2[(((ff*1024) + (yy*32)) + xx)] = 0f32
          for (rc: int32, 0, 3) {
            for (ry: int32, 0, 3) {
              for (rx: int32, 0, 3) {
                compute 2[(((ff*1024) + (yy*32)) + xx)] = ((float32*)compute 2[(((ff*1024) + (yy*32)) + xx)] + ((float32*)pad temp[(((rc*1156) + (yy*34)) + (ry*34)) + xx] + rx])*(float32*)B_2[(((ff*27) + (rc*9)) + (ry*3)) + rx]])
              }
            }
          }
        }
      }
    }
  }
}
}

Conv: 0.187121 ms
logcreative@logcreative:~/CompilerPrinciple$

```

优化结果

输入大小和输出大小	优化前	优化后	提升效率
n, ic, ih, iw = 1, 3, 32, 32 oc, kh, kw = 32, 3, 3	0.187121 ms		
n, ic, ih, iw = 100, 512, 32, 32 oc, kh, kw = 1024, 3, 3	493806.119851 ms		

参考文献

- [1] C. N. Steve Klabnik, *The Rust Programming Language (Covers Rust 2018)*. Random House LCC US, 2019. [Online]. Available: https://www.ebook.de/de/product/37149179/steve_klabnik_carol_nichols_the_rust_programming_language_covers_rust_2018.html
- [2] R. W. Floyd, “Syntactic analysis and operator precedence,” *Journal of the ACM*, vol. 10, no. 3, pp. 316–333, Jul. 1963.
- [3] Operator-precedence grammar. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Operator-precedence_grammar
- [4] namhyung, “uftrace.” [Online]. Available: <https://github.com/namhyung/uftrace>
- [5] brendangregg, “FlameGraph.” [Online]. Available: <https://github.com/brendangregg/FlameGraph>