实验遇到的问题&解决方案

对于条件判断的短路实现

在处理逻辑运算符 `&&` 和 `||` 的时候，我们不能将其简单地视为一般的二元运算符来处理。一般的处理方法就是使用控制流的短路操作来处理。具体如何进行短路操作在理论课上也有介绍，这里我们使用的方法是 `backpatching`。

在具体的处理过程中，我们不需要像理论课上讲的那样使用一个 `hole` 来表示还没有出现过的基本块，而是直接创建出所有所需的基本块，为短路的跳转提供具体的基本块（此时这些基本块都是空的，其实也是某种意义上的 `hole`）。然后在处理完前面的语句之后，我们可以使用 `Builder.SetInsertPoint` 函数来在我们预先声明的基本块中插入语句。

> 这里有一个小坑，就是如果我们不显式地调整基本块的位置的话，一个函数中基本块是按照它创建的顺序排列的。虽然每个基本块都会有一个终结语句，具体的顺序不会用太大的影响，但是我们还是使用 `BasicBlock.moveAfter(BB)` 来调整一下基本块的位置

由于在 clang 生成的语法树中，`&&` 和 `||` 都是解析成二元运算符，因此我们也就将短路运算集成在了处理二元运算符的函数（`buildBinaryOperator()`）中。在处理这两个短路运算时，函数的返回值是最后一个表达式的真值。

while 中声明的变量

由于我们生成的 ir 只会在某一个函数执行完毕之后才会释放分配给该函数的栈内存，这与我们在高级语言中理解的变量出了作用域就会被释放掉是有所区别的。这就意味着如果我们在一个循环中的声明的变量，那么最终声明的变量个数很有可能等于循环执行的次数。当循环执行次数过高的时候，可能会发生爆栈的问题。

较早期的 C 语言标准，变量声明必须都放在函数的最开始位置（比如使用 tcc 编译时就会遇到这个问题）。这里我们也使用类似的方法进行处理。我们将函数中所有的变量声明都抽取处理，然后放在函数的第一个基本块中执行 `alloca` 语句进行声明（第一个块不是循环体，这由我们循环的实现决定）。

这一个过程听起来十分的麻烦，但是在实现的过程中出人意料地简单，我们只需要在 `Builder.CreateAlloca` 之前将 `Builder` 的当前插入位置设置为函数入口块的终结语句之前就可以了。如果函数的入口块没有终结语句，则说明我们当前还处于函数的入口块中生成语句，因此这种情况不用处理。实现的部分代码如下：

```
1  llvm::BasicBlock *currBlock = Builder.GetInsertBlock();
2  if (currBlock->getParent()->getEntryBlock().getTerminator() != nullptr) //
   当前块不是最初始的块
3     Builder.SetInsertPoint(currBlock->getParent()-
   >getEntryBlock().getTerminator());
4
5  llvm::AllocaInst * Alloca = Builder.CreateAlloca(type, 0, *id);
6  Builder.SetInsertPoint(currBlock); // 生成完alloca语句之后我们回到原来的位置
```

变量重名的问题

这里我们偷了一个懒，巧妙地避开了变量重名的问题。在查看 llvm 生成的 AST 树的时候，我们发现每一个节点都被赋予了一个全局唯一的 `id`。然后我们就利用这个全局唯一的 `id` 作为变量名，然后变量重名的问题就被很巧妙地解决了:)。

字符串处理的问题

最后一个测例是一个打印一个水野爱图片的程序，这里面涉及到了字符串的处理问题。

> 除了字符串，最后一个样例还新增了 `long long` 和 `char` 类型，`do while` 语句，不同类型变量之间的类型转换问题。如果跳过最后一个测例摆烂不做的话，按照老师课上说的根据通过测例个数来计算成绩的话，我们将会在实验3痛失 $\frac{1}{355} \times 100 \approx 0.28$ 分

字符串的声明问题

例如，对于以下的字符串声明语句：

```
const char r[9] = "\\\\\\\"", t[9] = "\n\\\"\\n\\\\\\";
```

通过查看官方生成的 ir：

```
@__const.wk_puts.r = private unnamed_addr constant [9 x i8] c"\\\\\\\22\00\00\00\00\00", align 1
@__const.wk_puts.t = private unnamed_addr constant [9 x i8] c"\0A\\\22\\n\\\\\\\00", align 1

; other statement

define i64 @wk_puts(i8* %s) #0 {
    %r = alloca [9 x i8], align 1
    %t = alloca [9 x i8], align 1
    %10 = bitcast [9 x i8]* %r to i8*
    call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 1 %10, i8* align 1 getelementptr inbounds ([9 x i8], [9 x i8]* @__const.wk_puts.r, i32 0, i32 0), i64 9, i1 false)
    %11 = bitcast [9 x i8]* %t to i8*
    call void @llvm.memcpy.p0i8.p0i8.i64(i8* align 1 %11, i8* align 1 getelementptr inbounds ([9 x i8], [9 x i8]* @__const.wk_puts.t, i32 0, i32 0), i64 9, i1 false)

}
```

官方的处理方式是先在全局中声明对应的字符串，然后使用 `memcpy` 函数将全局中的字符串复制到声明的变量中。因此我们也采用和官方一样的处理方式，在解析 AST 树的 `StringLiteral` 节点时，直接生成一个类型为 `[i8 x str_size]` 的全局数组变量。然后对于函数中的局部变量，我们直接使用 `Builder.CreateMemcpy()` 函数声明一条内存拷贝语句对局部变量进行一个初始化。

字符串中转义字符的处理问题

这里一开始处理的时候可会比较绕。在高级语言的代码中，对于一些特殊的符号，比如回车、`"`、`\` 这些特殊的符号，我们通常会使用转移符号表示，如 `\n`，`\"`，`\\`。但是在 llvm 解析之后的 AST 数中，这些转移字符都会被完整地表示出来。例如对于字符串 `\n\\\"`，它实际表示的是`['\n', '\\', '\"']` 这三个字符，但是在解析成 AST 树并作为 value 储存在树节点时，表示的却是 `['\', 'n', '\', '\', '\', '"']` 这六个字符。在搞清楚这一点时候，剩下的问题就比较好解决了。

另外一个需要注意的问题是如果初始字符串的长度小于声明的数组大小时，我们需要在后面填 `\0`。

函数重载问题

在评测机上的样例 `/workspace/SYsU-lang/tester/third_party/SYsU-lang-tester-perfermance/performance_test2021-public/fft0.sysu.c` 中定义了如下函数：

```
int memmove(int dst[], int dst_pos, int src[], int len){
    int i = 0;
    while (i < len){
        dst[dst_pos + i] = src[i];
        i = i + 1;
    }
    return i;
}
```

很不巧的是，LLVM 同样内置有一个 `memset` 函数。也就是说在这个样例中，我们会面临函数重载的问题。

通过查阅网上实现函数重载的相关方案，我们最终选定使用重命名函数来解决这一问题。

首先在声明函数的时候，我们先在当前模块查找该函数是否已有对应的实现，如果已经有相应的实现，我们则需要对这个函数进行重命名。重命名的方法可以是简单的在现有函数名的基础上加一个标号，但是在多文件编译的时候这种命名方式还是会出现命名冲突的问题（参考连接）。而在函数重载中，区别不同函数最本质还是要靠它的函数参数类型（和返回值），这个对应于 llvm 中的 `llvm::FunctionType`。因此我们使用函数的类型来重命名函数。

```
; before rename
define i32 @memmove(i32* %0, i32 %1, i32* %2, i32 %3) {
    ; other code
}

; after rename
define i32 @memmove_ri_ip_i_ip_i(i32* %0, i32 %1, i32* %2, i32 %3) {
    ; other code
}
```

> 这里 `ri` 表示函数返回的类型是 `i32`；`ip_i_ip_i` 分别表示函数的接受的参数依次为 `i32*`、`i32`、`i32*`、`i32` 类型。

但是在这里问题还没有结束。我们仅仅只是在声明的时候对函数进行重命名了，但是在函数调用时我们并不知道函数重命名之后的名字，或者说我们根本就不知道当前函数是不是一个重载的函数（输入的AST树并没有提供这些信息）。虽说我们可以不管函数是否是重载，而是对所有的函数都应用这条规则进行重命名，然后在函数调用的时候再使用相同的规则得到函数在声明时的名字，但是这种方法貌似有

一定的工程量（~~精力有限~~），而且一不小心就有可能影响到原来能过的测例，因此这里我们选用了一种算是偷懒的方法。

我们使用了一个新的表来记录重载的函数。为了使这一行为显得更有正当性（~~绝对不是特判~~），我们将这个表命名为重载函数表。我们将重载的函数都以`<key, value>`的方式记录到重载函数表中，key 值就简单粗暴地设置为`funcName+funcType`（这两个信息都可以在函数调用的时候获取），这样，在函数调用的时候就可以通过查找重载函数表来获取对应的函数。

```
// add the New function to overload function table
overloadFuncs.insert({TheName+type_str, TheFunction});

// refer the function from overload function table
auto type = O->getObject("type")->getString("qualType");
auto name_ol = ((*name) + (*type)).str();
if (overloadFuncs.find(name_ol) != overloadFuncs.end())
  return overloadFuncs[name_ol];
```

实验结果

如下图，实验通过了测试

```
      Start  9: benchmark_generator_and_optimizer_0
 9/10 Test  #9: benchmark_generator_and_optimizer_0 ...   Passed    1.05 sec
      Start 10: benchmark_generator_and_optimizer_1
10/10 Test #10: benchmark_generator_and_optimizer_1 ...   Passed  430.62 sec
```

评测机上通过提交的截图如下：