Project 5 - Translation (v2)

2025 年 12 月 2 日

我们于 Dec 2 发布了 v2 版本的文档，在 Project 5 的项目假设中新增了一条：函数的参数中不能有数组类型。

1 项目要求

在 Project 5 中，你需要将一个通过了语义检查的 Splc 源代码翻译为 LLVM IR。

1.1 项目假设

在 Project 5 中，我们拥有如下假设：

- 样例不存在**语法错误**以及**语义错误**。
- 函数一定被 `return` 返回。
- 不存在**指针相减**[1]。
- （v2：）函数的参数的类型不能为 array type（但是能是 pointer to array）。[2]

并继承了如下假设：

- `specifier` 中的**完整结构体**规则出现时，它在语法树中一定是**全局变量定义**或**全局结构体声明**的任意次子节点，即它不会出现在函数参数（包含函数声明和函数定义）、函数体内；
- `specifier` 不再能够被推导为 `CHAR`，即 `char` 类型不再会出现；同时，表达式中的字符常量也不会出现。
- 样例中出现的结构体均是被完整声明过的，即不存在 incomplete structure。
- **函数声明**与 **函数定义**一定相符，即函数的返回类型与参数列表一定相同，你不需要进行额外检查。
- 函数的返回类型**只能**为 `int`。

1.2 扩展要求

> 扩展部分在每个 Project 之间都是独立计分的，在后续的 Project 中移除对某扩展部分的支持不影响前序 Project 的分数。也就是说，你可以选择在前面的 Project 中完成较为简单的扩展任务；如果你发现在后续的 Project 中完成扩展部分过于困难，你可以选择不完成后续 Project 的扩展部分，这样不会影响你前面 Project 的分数。

[1] 由于指针相减需要计算一个类型的长度，而我们目前的类型系统尚未支持这一点。

[2] 移除数组类型的原因是：C 语言中，函数的参数会被进行调整（adjusted, 6.7.5.3 Function declarators, Rule 7），数组类型会被调整为指针类型；然后，调用者处会发生 Array decay，数组会变成指针。因此，在 C 语言中传递数组实际上是传递的数组基地址的指针。我们的 Splc 语言不想支持 Array Decay 这一复杂情况，故也将函数参数设定为避免数组类型。

本项目的扩展部分与前序 Project 保持一致，你需要确保你的类型检查能够正确处理**结构体**和**指针**相关的语法。

2 程序的运行环境

你的程序可以调用以下函数，这些函数将以函数声明的方式存在于每个 Splc 源文件开头：

- `int readint()`：从标准输入流读取一个 `int` 并返回。
- `int writeint(int)`：向标准输出流打印一个 `int`，该函数的返回值永远为 0。
- `int setseed(int)`：设置伪随机数生成器的种子，该函数的返回值永远为 0。
- `int getrand()`：从伪随机数生成器中生成一个 `int` 并返回。
- `int assert_eq(int, int, int)`：对比第二个参数与第三个参数是否一致，不一致则退出程序。

你的程序的**主函数入口**为 `main0`。

Splc stdlib

```
int readint();
int writeint(int out);
int setseed(int seed);
int getrand();
int assert_eq(int where, int given, int expected);

int main0() {
    // TODO ...
    return 0;
}
```

3 framework 说明

初始代码位于 https://github.com/sqlab-sustech/CS323-Compilers-2025F-Projects 的 project5-base 分支。

你需要实例化一个 framework.llvm.IRBuilder 类，以此来构建 LLVM IR；构建完成后，使用 llvm.AbstractGrader::printIR(IRBuilder) 方法来打印生成的 IR。

若你需要修改 framework 包下的文件，请确保你的程序行为不依赖于你所修改的部分。在测评时，framework 包下所有文件均会被删除，然后替换为我们提供的版本。

3.1 表示 LLVM IR

framework.llvm 包中存在以下类用于表示 LLVM IR 中的实体：

- IRType：LLVM IR 的类型系统。支持 i1（逻辑值）、i32（普通 int）、数组、结构体、指针。
- IRValue：LLVM IR 中的一个 Identifier，分为 Named Values 与 Constants 两种，后者只能通过以下方式创建：
  - IRValue::constNull()：返回一个 ptr 类型的常量 null。
  - IRValue::consti32(int)：返回一个 i32 类型的常量。
  - IRValue::constTrue()、IRValue::constFalse()：返回一个 i1 类型的常量，表示逻辑中的 true 与 false。
- Inst：表示 LLVM IR 中的一条指令，目前我们仅支持以下指令：
  - Terminator Instructions: ret, br（包括无条件跳转与有条件跳转）
  - Binary Operations: add、sub、mul、sdiv 与 srem（有符号除法、余数）
  - Memory Access and Addressing Operations：alloca、load、store、getelementptr
  - Other Operations：icmp、call、zext（用于扩展 i1 到 i32）

3.2 翻译源代码

首先，你需要创建一个全局唯一的 IRBuilder 实例。

然后，你可以使用 defineStructure、defineGlobalVar、declareFunction 来处理结构体声明、全局变量定义和函数声明。

随后，使用 defineFunction 来开始翻译函数定义（包含函数体）。该方法会返回一个 FunctionBuilder，专门用于构建一个函数体；它会创建一个 BasicBlockBuilder 作为 root Block (entry basic block)，你也可以使用 FunctionBuilder::newBasicBlock 来创建一个新的 BasicBlockBuilder。

在每个 BasicBlockBuilder 中，你可以开始添加指令。其方式为调用 BasicBlockBuilder 下预定义的 ret、alloca 等函数。这些函数会进行基本的检查，以帮助你尽早定位问题，例如：

- 不能向一个 Terminated Basic Block 中添加新指令。
- add、sub 等指令要求两个操作数类型相同。
- load、store 等指令要求传入一个指针。

所有产生一个值的指令，其在 BasicBlockBuilder 中对应的创建函数都会返回一个 IRValue 对象，它的名字可以通过创建函数的最后一个参数 String name 传入。如果你不想为它取名，你也可以传入 null，FunctionBuilder 将为你取一个函数体内唯一的名字。

4 测试你的编译器

我们为 Project 5 创建了一套测试框架，它位于 `project5_testcases` 目录下：

- 形如 `text[0-9]*` 的文件夹表示一个测试样例，里面可以包含 n 对测试点（一个输入文件，对应 `readint` 方法；一个输出文件，用于比对 `writeint` 的输出）。
- `Makefile` 以及一些辅助脚本。

`Makefile` 有如下功能：

1. `make refs`：将所有测试样例使用 `clang-19` 与 `splc_crt.c` 编译为参考可执行文件，每个样例将被编译为三份：普通、带 UBSan 与带 ASan 的二进制。
2. `check_testcases.sh`：运行所有测试样例的三种二进制，重定向 `stdin` 为每个样例的输入文件，并比 `stdout` 和每个样例的输出文件。你可以使用这两个工具测试你的测试样例是否正确（见 5.2 章节）。
3. `make genir`：通过 IDEA 产生的 Java class 文件，调用你的编译器，将每个测试样例编译到 LLVM IR。

   你需要在 Makefile 中的 `IDEA_TARGET` 配置 IDEA 编译产物路径。

   `genir` 的每个 Target 都自动设置了 `IDEA_TARGET` 下所有 `*.class` 文件作为依赖：如果你修改了你的编译器代码，在 IDEA 中重新编译一次，再执行 `Make genir`，Makefile 会自动地重新生成所有 IR。
4. `make compileir`：使用 `clang-19` 编译你的所有 IR，每个 IR 将被编译为普通版本和带 ASan 的版本。
5. `testir.sh` 与第二条类似，运行由你的编译器产生的 LLVM IR，对比每个测试点的输出与标准输出。

5 评分

Project 5 将采用互相评分的模式，简而言之：你的每份提交不仅需要携带编译器实现，还需要携带**一些测试样例**；你的编译器实现将会使用其他同学的测试样例进行评测，并以此作为重要的评分依据。

5.1 测试样例

每一份测试样例均满足以下要求：

- 由一份源代码与 5 对标准输入和标准输出组成。
- 源代码不存在词法错误、语法错误、语义错误，且满足上述项目假设。（我们将通过一个独立的程序来验证它）。
- 不存在 Undefined Behavior 与内存错误。
- 程序的行为是确定性的（Deterministic）：对于一样的输入，它总是输出一样的内容。

  我们在运行环境中支持了伪随机数生成器，你可以按需指定其种子。

5.2 测试样例要求

你的提交需要额外携带 10 份测试样例，其中 6 份不包含结构体与指针扩展、2 份仅包含结构体扩展、2 份为包含结构体和指针扩展。

你所提交的每份测试样例都必须满足上述要求。你所提交的测试样例将进行如下步骤进行验证：

1. 以 `clang-19 -Wall -Werror -g -O0` 进行编译，必须不存在编译错误。
2. 增加编译参数 `-fsanitize=undefined` (UBSan)，运行所有标准输入，所得到的结果必须与标准输出一致，并且 UBSan 没有报错。
3. 增加编译参数 `-fsanitize=address` (ASan)，运行所有标准输入，所得到的结果必须与标准输出一致，并且 ASan 没有报错。

若完成以上步骤，你的测试样例（包含源程序与 5 个测试点）将被视为一个合法的测试样例，并被纳入**共享测试集合**（Shared Tests）。

5.3 评测

我们会对你的编译器进行两部分评测。对于每个测试样例，通过所有测试点才认为通过。

- **Official Tests**：由教学团队撰写的测试集合，**我们会在测试前会完整公布**。其中，基础部分、结构体扩展与指针扩展的比例为 6:2:2。
- **Shared Tests**：由所有同学提交的测试样例所组成的测试集合，我们不会公布这一部分。

5.4 评分

你的分数是以下三部分之和：

- **Official Tests**：该项满分 50 分，取决于通过测试样例的占比，每个测试样例分值相同。
- 你提交的 10 份测试样例中合法样例的个数：该项满分 10 分，每个测试样例为 1 分。
- **Shared Tests**：你的编译器在该集合上的通过率为 $p\%$，则得分为 $min(40, 45 \times p\%)$。