项目申请书

项目名称：llvm 与 gcc 兼容性增强——模板调构造函数使用 auto

项目主导师：张仁杰

申请人：陈冬燕

日期：2024.6.9

邮箱：2657726985@qq.com

1．项目背景

在 openEuler 生态中，GCC 作为历史悠久且功能强大的编译器，被广泛用于构建核心系统和应用程序。与此同时，LLVM/Clang 凭借其模块化架构、清晰的诊断信息和强大的工具链支持，正成为越来越多开发者的首选。

然而，在将项目从 GCC 迁移至 LLVM 的过程中，开发者常会遇到兼容性挑战。这些问题源于两方面：一是 GCC 支持一些非标准的 C++语法扩展；二是对于某些标准未明确定义的行为，二者有不同的实现。这些差异虽小，却会直接导致编译失败，增加了迁移成本和开发者的工作负担。

因此，系统性地识别并解决这些兼容性问题，增强 LLVM 对 GCC 常用特性的兼容，对于提升 openEuler 生态的编译环境多样性、降低项目迁移壁垒、优化开发者体验具有至关重要的价值。本项目旨在建立一套分析和解决此类问题的方法论，并以一个典型场景作为切入点，完成兼容性增强的完整实践。

2．技术方案

解决编译器兼容性问题的核心在于深入其内部工作流：**语法解析 (Parsing)** -> **语义分析 (Semantic Analysis)** -> **代码生成 (Code Generation)**。以 p->~auto() 为例，详解如何在 LLVM/Clang 的前端实现兼容。

```
template
void f (T* p)
{
p->~auto(); // p->~T();
}


int d;
struct A { ~A() { ++d; } };
```

```
int main()
{
f(new int(42));
f(new A);
if (d != 1)
throw;
return 0;
}
```

该场景 gcc 可以编译通过，llvm 编译会提示如下报错：

```
test.cpp:4:7: error: expected a class name after '~' to name a destructor
4 | p->~auto(); // p->~T();
| ^
1 error generated.
```

将 auto 改为对应类型名称后 LLVM 可以编译通过。

当前 LLVM 的报错 error: expected a class name after '~' to name a destructor，清晰地指明了问题出在语法解析阶段。Clang 的解析器在遇到 ~ 符号后，预期得到一个类型名（T），但它不认识 auto 在这个位置的特殊含义。要解决这个问题，需要依次修改语法解析和语义分析两个模块。

语法解析器 (Parser) 修改

- **目标**：让解析器能够“认识”并接受 ~auto 这个组合。
- **定位**：Clang 的 C++语法解析逻辑主要在 clang/lib/Parse/ParseCXX.cpp 文件中。我们需要找到处理“成员访问表达式”（IdExpression）或“析构函数名”（DestructorName）的相关函数，例如 Parser::ParseCXXIdExpression 或 Parser::ParseDestructorName。
- **具体操作**：
    1. 在解析 ~ 之后的名称时，当前的逻辑只检查是否为标识符（tok::identifier）或模板名等。
    2. 需要在这里增加一个分支判断：检查当前 Token 是否为 tok::kw_auto。
    3. 如果是，则消费（Consume）这个 auto Token，并构建一个临时的、特殊的 AST（抽象语法树）节点。这个节点可以是一个 UnresolvedLookupExpr 或一个专门为此场景设计的待定（placeholder）节点，它内部标记了“这是一个由 auto 推导的析构函数调用”。
    4. 如果否，则保持原有逻辑不变。

通过这一修改，源码就能通过语法解析阶段，生成一个包含了“待定析构函数”信息的 AST。

语义分析器 (Sema) 修改

- **目标**：对上一步生成的“待定”AST 节点进行类型检查和转换，赋予其正确的语义。
- **定位**：Clang 的 C++语义分析逻辑主要在 clang/lib/Sema/SemaExprCXX.cpp 等文件中，需要找到处理 AST 节点并进行类型推导和检查的地方。
- **具体操作**：
    1. 当 Sema 模块遍历 AST 并遇到我们新创建的“待定析构函数”节点时，它需要执行以下推导逻辑：
    2. **获取对象类型**：首先，获取 -> 或 . 运算符左侧表达式（即 p）的类型。对于 p->~auto()，Sema 会确定 p 的类型是 T*。
    3. **推导析构类型**：从对象指针类型 T* 中推导出实际的类类型 T。
    4. **查找析构函数**：在类型 T 的上下文中，查找其对应的析构函数（例如 ~T()）。

对于 int 等内置类型，这是一个伪析构函数调用（pseudo-destructor call），Sema 同样需要正确处理。

5. **构建最终 AST 节点**：使用查找到的真实析构函数信息，将临时的“待定”节点替换为一个标准的 CXXDestructorCallExpr 节点。这个最终节点与直接写 p->~T() 所生成的节点在结构和信息上是完全一致的。

3．项目计划

第一阶段（7 月 1 日 - 7 月 18 日）

搭建 AArch64 交叉编译环境，编译 openEuler LLVM，深入阅读 GCC 和 LLVM 源码，调研出 GCC 支持情况调研与测试用例迁移，开始撰写调研报告。

第二阶段（7 月 19 日 - 9 月 11 日）

Parser 修改：实现对~auto 的语法识别

Sema 修改：实现类型推导和 AST 节点转换

编写单元测试用例，进行全面的功能与回归测试

第三阶段（9 月 12 日 - 9 月 30 日）

准备最终的项目成果，按照 openEuler 社区规范提交，根据导师和社区的反馈进行修改，直至合入。