1. 工具功能介绍与输入输出

1.1 工具概述

本工具名为 **SIPIS (Software IP Interface Synthesizer)**，是面向嵌入式系统软件 IP（SIP）提取框架中的核心组件之一。

其核心功能是自动识别目标函数涉及的所有变量在全系统范围内的物理角色，将模糊的接口定义精细化为 **5 类互斥的接口变量**。

1.2 核心功能

工具通过静态分析手段，实现了对以下五类接口变量的自动化分类与提取：

1. **参数变量 (Parameter)**：仅被读取且在全系统生命周期内只读的配置项。
2. **状态变量 (State)**：具有闭环反馈特征（既读又写）且满足全系统排他性（该变量不被其他函数读写）的内部状态维持变量。
3. **输入变量 (Input)**：从外部读取数据且运行期不被修改的变量。
4. **输出变量 (Output)**：承载计算结果且不依赖初始值的变量。
5. **输入输出变量 (InOut)**：既读取初始值又被修改，但不具备状态排他性的普通交互变量。

1.3 输入与输出

- **输入数据 (Input)**：
  - **IP源码**：包含完整的该IP及其依赖的 C 语言源代码（.c/.h 文件）。
  - **目标函数**：指定待提取为 SIP 的核心功能函数的名称（即 SIP 的入口）。
- **输出数据 (Output)**：一个标准化的 JSON 文档，表示对目标函数以及其子函数的接口变量进行的分类。
- **示例**：

```
{
  "function_name": "control_logic_update",
  "interface_semantics": {
    "parameter": [{"name": "G_LIMIT_THRESHOLD", "type": "int"}],
    "state": [{"name": "g_integrator_sum", "type": "float"}],
    "input": [{"name": "g_sensor_raw_data", "type": "short"}],
    "output": [{"name": "g_actuator_command", "type": "int"}],
    "inout": [...]
  }
}
```

1.4 第三方工具

libclang，它会将一个.c文件以及其相关的.h文件合并并构成一棵抽象语法树，这棵树基本包含了所有我们需要的信息。具体的使用方法，可以询问ai。

2. 工具的大致流程

本工具的分析流程遵循“预处理—行为提取—语义精化”的三阶段静态分析策略。

2.1 基础信息构建

此阶段旨在解析代码结构并确定分析的先后顺序。

1. **AST 构建**：利用解析器构建抽象语法树。
2. **提取全局变量**。
3. **调用图生成**：分析函数间的调用关系，建立函数调用图。
4. **逆拓扑排序**。

此阶段我们暂时不考虑**函数指针**带来的解析上的问题。

2.2 自底向上的行为提取

1. **顺序遍历**：按照第一阶段生成的序列依次遍历每个函数。
2. **过程内分析**：分析函数自身的代码行，识别对变量的直接读写操作。
3. **过程间分析**：将子函数的读写集合映射回当前函数的上下文。

2.3 语义精化

根据接口定义，使用**判断公式**对每一个函数的接口变量进行分类。

3. 核心定义

3.1 符号系统与分析域

首先定义静态分析涉及的基础集合与对象：

- **目标函数** f：当前正在分析的 SIP 入口函数。
- **全系统函数集** S：待分析软件中包含的所有函数集合。
- **变量全集** V_f：函数 `f` 可能涉及的所有变量集合，包括：
  - **局部变量集合** A_f：函数定义的形参。
  - **全局变量集合** G：在第一步中提取的所有全局变量。
  - **返回值ret**。

3.2 读写行为模型

为了区分“当前函数的直接操作”与“底层调用的间接操作”，定义了两层读写集合：

**1. 原子行为集合 **
仅考虑函数 f 自身代码行产生的直接读写行为。

- Rbase(f)：在 `f` 源码中被读取的变量集合。
  - *注意*：为识别对初始值的依赖，若变量在被读取前已在当前函数内被修改（被覆盖），则该次读取不计入原子读集合。
- Wbase(f)：在 f 源码中被修改（赋值/写操作）的变量集合。

这里的读写，我们暂时只考虑四则运算和位运算（a+b，a++，a-=b等），以及条件判断（if，while等）。

2. **传播行为集合**

通过自底向上策略，将深层调用链的副作用累积后的最终行为。

- Rtotal(f)：包含 Rbase(f) 以及所有子函数传递上来的读行为。
- Wtotal(f)：包含 Wbase(f) 以及所有子函数传递上来的写行为。

在此模型中，针对 C 语言特性做了如下归一化处理：

- **结构体和数组**：对成员变量（struct.member）或数组单元（array[i]）的访问统一映射为对基变量（struct或array）的访问。
- **重要！指针**：对指针指向内存的读取/修改，不被视为对该指针的读/写。读取/修改指针指向的值，被视为对所有指向该地址的变量/指针的读/写。

3.3 核心逻辑定义：IsIO

- **定义**：一个变量 `v` 如果在函数 `f` 的执行路径中，既读取了初始值，又在计算后被修改，则满足 IsIO 特征。
- **公式**：$IsIO(f,v) \Leftrightarrow v \in Rtotal(f) \cap Wtotal(f)$

4. 核心算法详解

4.1 基础信息构建

此处略过。

4.2 自底向上的行为提取算法

本阶段的目标是计算目标函数 f 及其子函数的全量读写集合 Rtotal 和 Wtotal，形成对每个函数的摘要。

为了解决指针别名（Alias）和复杂的内存引用问题，算法通过维护两个数据结构来追踪变量与内存的关系：

1. **变量信息字典 (Variable Dict, VD)**：VD[name]→**Variable**
   i. **这里的类对象Variable**，见5.类设计概念。而在MS中，我们只存变量的名称。
   ii. 对于**pt**，结构体，数组可能会指向多个内存，所以PT应该使用字典，用来记录其具体的成员变量指向的内存（比如变量a自己就是a->o_1,它的成员变量就是a.b.c->o_2）。
2. **内存别名表 (Memory Set, MS)**：定义：MS[o_{id}]→{v_1,v_2,...}。映射抽象内存对象 o_{id} 到当前指向该内存的所有变量集合（别名集）。
   i. 这里需要注意一下，如果v不是一个单一变量，而是结构体或者数组，当只是对单一的某个内存进行修改，此时我们应该只记录这一个内存对应的是结构体或者数组的哪个单元变量（比如a.b.c , a[1]这样）

4.2.1 过程内分析 (Intra-procedural Analysis)

首先，提取函数涉及的所有可能与上级函数产生关联的局部变量（包括形参和返回值）。在对函数代码进行线性遍历时，算法执行以下三个阶段的逻辑：

**Phase 1: 惰性初始化 **
在处理每一行语句前，检查语句中涉及的所有变量 v。

- 若 `v` 尚未在 VD 表中记录（即 `v∉Keys(VD)`），且该语句是声明语句，需要对其进行初始化。
- 若 `v` 在 VD 表的中的pt是空的，且该语句是一个操作语句（四则运算），需要对其内存进行初始化。

**Phase 2: 指针指向更新
针对指针赋值操作（如 p = &x 或 p = q），执行“解绑旧关系—建立新关系”的流程：

1. **建立新关系**：如p = &x：我们需要去VD找到x，然后看它的pt，找到对应的内存，然后在该内存加入p，并在p的pt中加入该指向。
2. **解绑旧关系与垃圾回收**：获取 p 原本指向的旧对象 o_{old}。从旧对象的别名表中移除 ，并在 p 的 pt 中移除该指向。**GC 检测**：若 MS[o_{old}] 变为空集（无变量指向该内存），则从 `VD` 和 `MS` 中彻底销毁 o_{old}。

Phase 3: 间接读写处理 (Indirect Access)
针对指针解引用操作（如 *p）：

1. **获取实体**：查询 `VD[p]` 获取当前 `p` 指向的抽象对象 ocurr。
2. **解析别名**：查询 MS[ocurr] 获取该对象的所有别名变量集合 Aliases。
3. **记录行为**：若操作是写入（如 *p = expr）：将 `Aliases` 中的所有变量加入 Wtotal。若操作是读取（如 expr = *p）：将 `Aliases` 中**未被当前控制流覆盖写**的变量加入 `Rtotal` 。

4.2.2 过程间分析 (Inter-procedural Analysis)

当函数调用子函数 sub 时，算法利用 sub的摘要信息更新 f 的状态：

1. **参数对象映射**：获取 `f` 中传入的实参。将该对象与 sub 中对应的形参建立映射关系。这里我们会在函数类function中的成员变量params来记录其形参。
2. **副作用传递**：根据 `sub` 的 Rtotal/Wtotal，通过上述映射关系，找到 f 中对应的对象。查找 `f` 中该对象对应的别名集合 (`MS[of]`)，将这些变量加入 `f` 的 `Rtotal` 和 `Wtotal` 。
3. **指向关系合并**：如果 `sub` 修改了指针的指向（例如通过双重指针参数），需要根据 `sub` 返回后的 `VD/MS` 状态，更新 `ff` 中的 VD 和 MS 表，以反映子函数对内存布局的改变。

通过以上步骤，为每一个函数生成的摘要包含：

1. 该函数的 `Rtotal` 和 `Wtotal` 集合。
2. 该函数结束时的 VD 和内 MS 。

过程间分析：

当函数 f 调用子函数时，算法执行以下映射：

1. **参数映射**：将 sub 的形参替换为 `f` 中传入的实参。
2. **别名解构**：如果实参是指针，利用当前上下文的 MS 将子函数对该指针的读写操作，还原为对 `f` 中具体变量的读写。
3. **集合合并**：将映射后的读写行为合并入 f 的 Rtotal 和 Wtotal。
4. **指向集表修改：**根据子函数的指向集表，修改主函数的指向集表。

通过以上步骤，我们会为每一个函数生成一份摘要，其中包括：

1.该函数的Rtotal和Wtotal集合。

2.该函数结束时的 VD 和内 MS 。

3.该函数的Af。包含在VD中

需要注意的是，会存在一些库函数级别的对函数指针进行操作的函数，这些函数可能需要我们单独对他们进行手写摘要。

4.3 语义精化

 image-20260113024258487

5. 类设计概览

我们需要两个核心类：

1. **Variable**：描述单个变量的静态属性（类型、域）和动态属性（指向关系 PT）。
2. **Function**：描述函数的元数据、行为摘要（R/W 集合）以及分析结束时的内存状态（VD 和 MS）。

Variable 类设计

这是 VD 字典中的 Value 对象。

属性名	类型	说明
name	str	变量全名。例如 "g_config" 或 "local_var".
raw_type	str	原始类型字符串（来自 Clang）。例如 "struct SensorData *" 或 "int [10]".
kind	str	类型类别。例如 "Pointer", "Record", "Array", "Builtin".
domain	Enum	变量的作用域："Global", "Param", "Local", "Return".
is_pointer	bool	快捷标志，判断该变量本身是否是指针。
pt	Dict[str, int]	**指向表 (Points-To)**。 **Key**: 成员访问路径。空字符串 "" 代表变量本身，".a" 代表成员 a，"[0]" 代表数组首元素。 **Value**: 抽象内存对象的 ID (Memory ID)。 *例如：struct {int* p} x; x.p 指向内存 5，则 pt = {".p": 5}*

Function 类设计

这是承载分析结果的核心类。

属性名	类型	说明
name	str	函数名称。
source_file	str	定义该函数的文件路径（用于调试和溯源）。
params	List[str]	**形参名称有序列表**。 存储 ["arg1", "arg2"]。这是调用映射的桥梁，用于在 VD 中查找对应的 Variable 对象。
ret_var	Variable	**返回值虚拟变量**。 用于处理 return x; 或 ptr = func(); 的情况。name 固定为 "**return**"。这里我也不太确定会不会用上，先设计出来
vd	Dict[str, Variable]	**变量信息字典 (Variable Dict)**。 **Key**: 变量名。 **Value**: Variable 对象。 包含该函数涉及的所有局部变量、形参和全局变量的快照。
ms	Dict[int, Set[str]]	**内存别名表 (Memory Set)**。 **Key**: 内存对象 ID (int)。

属性名	类型	说明
		Value: 指向该内存的变量路径集合 (Set[str])。 *例如：{101: {"a", "b.next"}} 表示变量 a 和 b.next 都指向内存 101。*
r_total	Set[str]	**全量读集合**。存储满足 IsIO 读条件的变量。
w_total	Set[str]	**全量写集合**。存储满足 IsIO 写条件的变量。
calls	List[CallSite]	**调用记录**（可选，用于调试）。 记录该函数调用了哪些子函数，以及当时传入的实参快照。