为 RISC- V 形式化验证工具 Sail 添加代码风格检查

关天扬

2026 年 1 月 14 日

背景：为什么代码风格很重要？

- **一致性 (Consistency):** 在大型项目中，统一的命名风格可以使代码库看起来像是单人完成的，极大地降低了认知负荷。
- **可读性 (Readability):** 清晰的命名能够自解释代码的意图。例如，`MAX_XLEN` 明显是一个常量，而 `MemoryAccess` 是一个类型。
- **可维护性 (Maintainability):** 新成员可以更快地理解和融入项目。代码审查的重点可以放在逻辑上，而非风格问题上。
- **Sail 语言:** 作为一种用于描述指令集体系结构（ISA）的语言，其模型（如 RISC-V, ARM）通常非常庞大和复杂，代码规范性至关重要。

现有问题

在引入命名检查器之前：

- **缺乏自动化**：Sail 编译器本身不强制任何命名约定。
- **依赖人工**：命名规范完全依赖于开发者的自觉性和代码审查（Code Review）。
- **效率低下**：人工审查耗时耗力，且容易出现疏漏。
- **风格不一**：不同的开发者可能有不同的偏好，导致代码库风格混乱。

```
// main.sail
type My_Int_Type = int;
function Get_Value(input_param : My_Int_Type) -> int {
    let result_value = input_param + 1;
    return result_value;
}
```

命名约定：`My_Int_Type` 应该使用 `TitleCase`，比如 `MyIntType`。`Get_Value` 应该使用 `snake_case`，比如 `get_value`。

图：人工审查风格问题

我们的解决方案：naming_check 模块

核心思想

在 Sail 编译流程中集成一个新的静态分析遍（Pass），使其能够自动地、系统地检查代码中所有标识符的命名是否符合预设规范。

主要优势：

- **自动化**：无需人工干预，编译时自动执行检查。
- **可配置**：提供“警告”和“严格”两种模式。
- **集成性**：作为编译器的一部分，无缝集成到现有开发流程中。
- **一致性保障**：从根本上保证了整个项目代码风格的一致性。

核心功能：四类标识符检查

`naming_check` 模块主要针对以下四种类型的标识符进行风格检查：

- **类型 (Types):** 包括结构体（structs）、枚举（enums）、联合（unions）等。
- **函数 (Functions):** 包括全局函数、散布式定义的函数（scattered function）等。
- **变量 (Variables):** 包括 let 绑定的局部变量、函数参数等。
- **常量 (Constants):** 在顶层 let 绑定中，根据其表达式推断出的常量。

命名风格规则

我们设定了一套默认的命名规范，这套规范在大型项目中被广泛证明是行之有效的。

标识符类型	命名风格 (Style)	示例 (Example)
类型	`TitleCase`	`MemoryAccess`, `Privilege`
函数	`snake_case`	`execute_load`, `get_csr_value`
变量	`snake_case`	`mem_addr`, `reg_value`
常量	`SCREAMING_-SNAKE_CASE`	`MAX_XLEN`, `DEFAULT_VALUE`

表: 默认命名规范

如何启用和配置

通过两个新的命令行参数来控制命名检查器的行为：

警告模式 (Warning Mode - 默认)

```
--naming-check
```

在此模式下，不符合规范的命名会以警告（Warning）的形式输出，但不会中断编译过程。适合在迁移老旧代码库时使用。

严格模式 (Strict Mode)

```
--naming-check-strict
```

在此模式下，任何命名违规都会被视为一个错误（Error），并立即中断编译。这能强制所有新代码都遵循规范。

模块概览：.mli 和.ml

新功能由两个文件组成：

- `naming_check.mli` **(接口文件)**
  - 定义了核心的数据类型，如 `naming_style` 和 `naming_config`。
  - 暴露了公共函数和配置项，如 `opt_enabled` 和 `run` 函数。
  - 包含了详细的文档注释，是理解模块功能的入口。
- `naming_check.ml` **(实现文件)**
  - 包含了所有检查逻辑的具体实现。
  - 通过遍历 Sail 的抽象语法树（AST）来访问每一个标识符。
  - 实现了风格判断、常量检测、错误报告等核心算法。

核心数据结构

我们定义了命名风格的代数数据类型和用于保存配置的记录类型。

```
type naming_style =
  | TitleCase
  | Snake_case
  | Screaming_snake_case
  | Any

```

Listing 1: OCaml: 定义命名风格 (naming_style)

```
type naming_config = {
  type_style : naming_style;
  function_style : naming_style;
  variable_style : naming_style;
  constant_style : naming_style;
}

(* Default configuration *)
let default_config = {
  type_style = TitleCase;
  function_style = Snake_case;
  variable_style = Snake_case;
  constant_style = Screaming_snake_case;
}

```

Listing 2: OCaml: 存储检查配置 (naming_config)

风格匹配函数

我们为每种命名风格都实现了一个简单的检查函数，它们通过字符级别的规则来判断字符串是否合规。

```ocaml
let is_title_case s =
  if String.length s = 0 then false
  else
    let first_char = s.[0] in
    first_char >= 'A' && first_char <= 'Z' &&
    not (String.contains s '_')

```

Listing 3: OCaml: 检查 TitleCase

```ocaml
let is_snake_case s =
  if String.length s = 0 then false
  else
    let first_char = s.[0] in
    (first_char >= 'a' && first_char <= 'z') &&
    String.for_all (fun c ->
      (c >= 'a' && c <= 'z') ||
      (c >= '0' && c <= '9') || c = '_'
    ) s

```

Listing 4: OCaml: 检查 snake_case

错误与警告报告

`report_naming_issue` 是报告问题的核心函数。它会检查 `opt_strict` 标志来决定是抛出错误还是打印警告。

```
let report_naming_issue l id expected_style category =
  let name = string_of_id id in
  if not (matches_style name expected_style) then begin
    let message = Printf.sprintf "%s '%s' should use %s"
      (string_of_category category)
      name
      (string_of_style expected_style)
    in
    if !opt_strict then
      raise (Reporting.err_general l message)
    else
      Reporting.warn "Naming convention" l message

```

Listing 5: OCaml: 报告命名问题

抽象语法树（AST）遍历

检查的核心是递归地遍历 Sail 的 AST。我们为不同类型的定义（如 `DEF_type`, `DEF_fundef`）编写了专门的检查函数。

```
let check_type_def config (TD_aux (td, annot)) =
  let l = fst annot in
  match td with
  | TD_abbrev (id, _, _)
  | TD_record (id, _, _, _)
  | TD_variant (id, _, _, _)
  | TD_enum (id, _, _) ->
      report_naming_issue l id config.type_style Category_type
  | ...

```

Listing 6: OCaml: 检查类型定义 (Type Definitions)

常量与变量的区分

在 Sail 中，顶层 `let` 绑定既可以定义常量也可以定义全局变量。我们通过一个启发式函数 `is_constant_binding` 来区分它们。

判断逻辑

如果一个绑定的右侧表达式完全由字面量（literals）、`sizeof`、或其他简单纯函数（如 `add_int`）构成，我们便认为它是一个常量。

```ocaml
let is_constant_binding (LB_aux (LB_val (_, exp), _)) =
  let rec is_constant_exp (E_aux (e, _)) =
    match e with
    | E_lit _ -> true
    | E_sizeof _ -> true
    | E_tuple exps -> List.for_all is_constant_exp exps
    | E_app (id, args) when is_constant_function id ->
        List.for_all is_constant_exp args
    | _ -> false
  in
  is_constant_exp exp

```

Listing 7: OCaml: 判断是否为常量绑定

实例：不规范的 Sail 代码

下面是一段包含多种命名风格问题的 Sail 代码。

```
1  // Type should be TitleCase
2  type My_Address = 64
3
4  // Constant should be SCREAMING_SNAKE_CASE
5  let default_permission = 0b11
6
7  // Function should be snake_case
8  function GetPermission(addr : My_Address) -> bits(2) {
9    // Variable should be snake_case
10   let RetVal = default_permission;
11   RetVal
12 }
13
```

演示：警告模式 (--naming-check)

使用 --naming-check 参数进行编译，会输出以下警告信息，但编译会成功。

编译器输出

```
Warning: Naming convention at line 2:
Type 'My_Address' should use TitleCase (e.g., MemoryAccess)

Warning: Naming convention at line 5:
Constant 'default_permission' should use
SCREAMING_SNAKE_CASE (e.g., MAX_XLEN)

Warning: Naming convention at line 8:
Function 'GetPermission' should use snake_case
(e.g., execute_load)

Warning: Naming convention at line 10:
Variable 'RetVal' should use snake_case (e.g., mem_addr)
```

演示：严格模式 (--naming-check-strict)

使用 --naming-check-strict 参数，第一个命名问题就会导致编译失败并报错。

编译器输出

```
Error: at line 2:
Type 'My_Address' should use TitleCase (e.g., MemoryAccess)

[1] > exit 1
```

行为

编译过程在遇到第一个错误后立即中止。这强制开发者必须修复所有命名问题。

修正后的代码

修正所有命名问题后，代码如下。现在它可以通过严格模式检查。

```
// Correct: TitleCase
type MyAddress = 64

// Correct: SCREAMING_SNAKE_CASE
let DEFAULT_PERMISSION = 0b11

// Correct: snake_case
function get_permission(addr : MyAddress) -> bits(2) {
  // Correct: snake_case
  let ret_val = DEFAULT_PERMISSION;
  ret_val
}

```

总结与展望

总结

- **实现了什么**: 一个集成在 Sail 编译器中的自动化命名规范检查器。
- **解决了什么问题**: 消除了手动检查命名规范的负担，保证了大型代码库的风格一致性。
- **带来的好处**: 提高了代码质量、可读性和长期可维护性。

未来工作

- **更强的可配置性**: 允许用户通过配置文件（如 `.sail-lint`）自定义命名规则。
- **更多检查项**: 扩展检查器以覆盖更多的代码风格问题（如行长度、注释风格等）。
- **IDE 集成**: 为 VSCode 等编辑器提供实时 linting 支持。

感谢观看

Questions?