Enum of Rust used in Lisp interpreter

yuchengye

日期：December 6, 2021

目录

1 Rust 的枚举类型和模式匹配在解释器泛型中的运用

1.1 Problems

在实现解释器时（无论是 JSON 还是 Lisp 等），一个必须要考虑的问题是多种类型的如何处理、以及嵌套子表达式应该如何表示。下面谈一谈我在用 C++ 和 Rust 处理这两个问题时的感慨

1.1.1 Generic type

```
using i32 = int32_t;
using f64 = double;
struct Value{
    value_t type;
    union{i32 Integer; f64 Fp;} data;
    template<class T>
    T get(){
^^Iif(type==Int) return data.Integer;
^^Ireturn data.Fp;
    }
    Value(f64 _data):type(Double){data.Fp=_data;}
    Value(i32 _data):type(Int){data.Integer=_data;}
    Value():type(Int){data.Integer=0;}
};
// store
auto a = Value(10);
```

```
auto b = Value(3.14);

// get
assert(a.get<i32>()==10);
assert(a.get<f64>()==3.14);
```

以上是我在ILisp中的实现：使用 **type** 来记录所存储值的类型；使用 **union** 存储数据来使用尽量少的空间。这种方式是权衡后的选择。在取值的时候虚要根据 type 判断 get<>() 的类型。而且虽然在 C++11 之后，union 可以存储非 POD 类型的数据了，但是对 std::string 的支持依然不是很好。导致我只能存储 i32 和 f64 两种简单的类型。相比之下，Rust 的枚举类型在应对这种场景时显得更加得心应手：

```
#[derive(Debug, Clone)]
pub enum Atomic {
    Number(i32),
    Float(f64),
    Symbol(String)  // TODO: refactor to &str
}

#[derive(Debug, Clone)]
pub enum LispType {
    Atom(Atomic),
    List(Vec<LispType>)
}

pub fn eval(expr: &LispType) {
    match expr {
^^ILispType::List(subexpr) => {
^^I    // do something
^^I    // calculate the value of sub-expression
^^I},
^^ILispType::Atom(Atomic::Symbol(symbol_name)) => {
^^I    // do something
^^I    // get value from Env
^^I},
^^ILispType::Atom(i32_or_f64) => {
^^I    match i32_or_f64 {
^^I^^IAtomic::Number(num) => {/* do something */},
^^I^^IAtomic::Float(fp) => {/* do something */}
^^I    }
^^I}
    }
```

}

以上是我在rlisp的实现，可以看到，由于

1. Rust 的 enum 中每种枚举都可以和一个已有类型的值绑定
2. 通过 match 表达式强大的模式匹配功能，可以 **优雅**（至少我觉得比较优雅）地取出所绑定的值。

1.1.2 Nested sub-expression

对比上面的代码 (cpp rust)：

1. 在 c++ 实现中，我没有将类似 vector<Value> 的结构放在结构体 Value 的定义中，因为我觉得这使得 Value 的体积过于膨胀了，一个值类型应该是 **简洁**。因此在 c++ 实现中，我保留了 ["(", ")"]，将其作为边界字符，在 eval() 函数遇到"(" 时则进入下一层递归，在遇到")" 计算并返回子表达式的值。
2. 在 rust 实现中，由于 enum 足够简洁，我直接将子表达式（一个 List，和 rust 的 Vec<Lisptype> 绑定）也作为一种枚举中的一种类型（实际上 List 本来就是 Lisp 的一种类型，这种设计更贴合 Lisp）。类似地，eval() 函数在 match 到 LispType::List(Vec<LispType>) 的时候进入下一层递归，离开作用域时计算并返回子表达式的值即可