Programming Languages

Second Phase Report

José António Ribeiro da Silva Lopes
ist1103938

General behavior of the type checker

To implement type checking for the base L0 language, a new method for the `ASTNode` interface was created, called `typecheck`. Essentially, all it does is check whether or not all the nodes present in the program's AST can be properly typed according to what the language specifies as their typing rules.

Type checking recursive functions

A general problem regarding static type checking is how to type check recursive functions. In an initial implementation, when type checking an `ASTLet`, the type of the expression would only get assigned to the given name after type checking the expression itself, but in the case of recursive functions, the name of the function is referenced before it gets assigned any type, so it would throw a "Binding for <fun> not found not found". To solve this issue, the programmer has to explicitly declare the type of a recursive function, like in the following example:

```
let fact:int->int = fn x:int => {
    if (x==0) { 1 }
    else { x * (fact(x-1)) }
};
fact(5);;
```

As soon as the interpreter sees that a `let` binding as a type, it immediately assigns it to the given name, meaning that when `fact` is called inside it's own body, a type for it is already known.

This explicitly declaration of types can happen in any situation, not only when declaring recursive functions. If this is the case, after the whole expression is type checked, both types are compared to see if they are compatible, i.e. the type of the expression is a sub type of the explicitly declared type.

Behavior of product and union types

Product and union types are just ways of aggregating other types inside the same structure and associating them with labels. In the case of product types, which are called `structs` in this implementation, the type has "access" to all of it's inner fields via the dot (`.`) operator. As for union types, the concrete instance can only hold one of all the possible labels. For a more thorough use of union types, the `match` construct is the ideal solution. It allows for the destructuring of the type's labels, while being exhaustive in this same labels.

Sub-Typing and type checking recursive types

Sub-typing was implemented as a default method, part of the `ASTType` interface. Only the types with special sub-typing rules, like `ASTTStruct`, `ASTTArrow`, etc. overrode the default method. For all the concrete implementations of sub-typing, the first thing done was check if the other type was an instance of `ASTTId`. If so, that type would get unrolled and then passed again to the sub-type method. With this sort of “lazy unrolling”, recursive type checking is also achieved, because there will come a point where the type being compared is not a sub-type of the recursive type at all, or where it already is, or where future unrolls might result in this relation being observed.