Extension part: closure & higher order support

Lucas Franceschino

May 15, 2017

1 My plans

I want my compiler to have higher order function support and closure support. To do so, I have to improve or add the following:

1.1 Grammar

Change my grammar to allow function definition in function bodies. Then a function would begin by a - possible empty - list of variable or function declarations.

I would also like to have anonymous functions, so I have to find a syntax for them and add them to the grammar definition.

1.2 Blur the line between functions and variables, lambda functions

In order to make functions ok to store into "normal" variables, I need to let the user define variables by assignation of "function like" values.

I also need to let the user use lambda functions. I plan to use the exact same mechanism that I use for normal functions, but with a different syntax and no name (the name will be actually replaced by "lambda_{id}").

1.3 Type inference

Improve my type inference: at the moment, my type inference struggles with some cases of recursive functions. At the moment I have a short analysis of function dependencies, but not something strong enough. I plan to rebuild that to make a full dependency graph of all functions and variables and make it able to compute variable liveness.

1.4 Closure handling strategy

As I said previously, by using a dependency graph, I'll be able to see where a variable is used. If a closure λ uses a variable (or function) x and if λ can be called outside of its parent scope p (i.e. returned by parent scope), then, we lift the variable x to the heap. It means that if some statements in p need to access or to set x, then, they will refer to some place in the heap. Then, if multiple lambda functions using x are returned from p (via a list or a tuple for instance), then x will exist (uniquely) even if the scope p is done.

1.5 Closure code generation

Let suppose we have the following code:

```
f(a){
        g(c){
                return c * a;
        }
        return g;
}
f(10)(2);
```

Then we can't just return g, we need (runtime) information concerning some scope of f. Using a dependency analysis, we know $n = 1$ variables (a) are used in g. Then, we make a $n + 1$ dimentional tuple in the heap with the hard address of g and the addresses of all different linked variables. There is no need to actually know the number of linked variables of g: on calling, we will let g unpacks theses variables addresses.