Functional Programming: Essay Course

Due on Jan 15, 2019

RUOPENG XU
18M38179

Problem 1

Answer either question. If you answer both questions the better answer will be chosen for your evaluation.

1. What's α -conversion for? What kind of problems we will see if α -conversion were not applied? Find Min-Caml programs that give incorrect answers in absence of proper α-conversion.

Note: You may want to consider the reason why inline.ml refers to Alpha.g.

2. Are optimization modules interdependent? Yes, but in what way? Find a pair of optimization modules A and B such that for a program P, B is effective when it is applied after A:

$\exists P.B(P) = P$ but $B(A(P)) \neq A(P)$

Answer for question 1

What's α -conversion for?

α -conversion is to assign different names to different variables. If the variables have same names, the process will be complicated.

In *alpha.ml* : *let find x env = try M.find x env with Not_found -> x* will find the current binding of x in env, if the binding not exists, it will return x. Therefore, in α -conversion, if the binding is already existed, it will find the binding of variables in env and return it, which means this variables need a new name. On the other hand, if can't find the binding, it will return the variable itself, which means it doesn't need a new name. In this way, we can make sure different variables have different names.
If it find an expression, it will call the *g env* again to assign different names for the variables in the expression.

For example, *let x = 123 in let x = 456 in x + x* will be translated to *let x1 = 123 in let x2 = 456 in x2 + x2*. In first let process, x will be bind to 123 firstly and 456 secondly, there are different bindings so they need have different names. *Id.genid* will count how many times x appears and call g env again and again in next expression. Finally we will get x1 and x2.

What kind of problems we will see if α -conversion were not applied?

If we didn't use α -conversion in inline expansion, this process may be camplicated and the names of variables will be confused.

Because in the process of inline expansion, min-caml replaces calls to small functions with their bodies. If the size of the cuntion is lesss than a threshold, min-caml replaces formal arguments with actual arguments, and need to copy the function body. Therefore, the variables may be duplicated and must be α -conversion again to make sure they are different.

Problem 2

Answer either question. If you answer both questions the better answer will be chosen for your evaluation.

1.Explain in detail the mechanism described in Figure 16, mincaml/overview.pdf.
- Compare this algorithm with two previous algorithms (Figure 14 and Figure 15). Present a Min-Caml code fragment examples that exhibits superiority of the last algorithm.
- For the examples you gave above, estimate the number of make_closure, apply_direct, and apply_closure executed at runtime when the generated code is executed.

2.Explain how min-caml compiler is organized to achieve the goal of being a multi-targeted native code compiler. (A multi-targeted native code compiler can generate native code targeted for different CPU architectures.)

Answer for qustion 1

In figure.14, function takes the value of its free variable x as an argument, and then the function is returned as a value, and its body is paired with the free variable. When the function is called, its body and the value of the FV are extracted from the closure. This approach generate a closure for every function and it's inefficient.

In figure.15, it separate the function which need closure from those can be called more conventional. It has a set of known functions that can be called dirctly, because they are known do not have free variables. If the function is closure-based, it calls *apply_closure*, otherwise it calls *apply_direct*. In this way it distinguish the type of labels from the type of normal variables.

In figure.16, for example, if the defination is *let rec x y1, y2, ... yn = e1 in e2*.Min-caml firstly assume the function has no free variable, and it can be added to known, and the funtion body e1 is converted. After that, if x doesn't have any free variables, it convert e2. If x has free variables, min-caml rewind the value of know and the reference of top function. Finally, if x never appears as a variable in e2, min-caml omit the closure creation for x.

For example, if the code is like this:

let rec make_adder x =
let rec adder y = x + y in
make_adder

In the first algorithm, all of the variables will be closure:
let rec adder y = x + y becomes (adder,x) as a value V1. Then *let rec make_adder x = V1* is closured to (make_adder,). It does a closure to make_adder which doesn't has any free variable, so it's inefficient.

What's more, in the second algorithm, it knows there are no free variables in make_adder. But make_adder still need a repreastation as a closure because a user who receives make_adder does not know in general if it has a free variable or not.

Therefore, in the optimized algorithm, *let rec adder y = x + y* becomes (adder,x) as a value V1 firstly as above. *let rec make_adder x = V1* is thought to be (make_adder,) as V2 temporarily. Beecause there are not any free variables in V2, min-caml omits the creation of the closure, which means we save the time of closure creation. Moreover, it is also represented as a closure, which will not confuse the users.

Problem 2 continued on next page. . .

In this example, the *make_closure* is called twice for adder and make_adder, *apply_closure* is called once for adder, and *apply_direct* is called once for make_adder.