FUNCTIONAL PROGRAMMING

Final Report

WANG BIYUAN

18M38156

Department of Computer Science

Lecturer: Ken Wakita

January 8, 2019

Project A

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.

Given a buffer argument, Min-Caml compiler applies the processing steps in the following order: lexical analysis (Lexer.token), parsing (Parser.exp), type inference (Typing.f), K-normalization (KNormal.f), α-conversion (Alpha.f), optimizations (iter), closure conversion (Closure.f), virtual machine code generation (Virtual.f), register allocation (RegAlloc.f), and code emission (Emit.f). α-conversion is an important step after K-Normalization, as an important transition stage between expression analysis and optimization.

α-conversion is aiming to assign different names to different variables. This is an indispensable conversion because different variables with the same name is confusing in various processes and cause unnecessary compiling errors. For example, if you repeatedly use a variable name x in a nested expression: *let x = 1 in let x =2 in x + x*, α-conversion will modify the expression to *let x1 = 1 in let x2 = 2 in x2 + x1* in order to distinguish the different assigned values to x. For example, if there is a function:

$$\begin{aligned} &\text{let } x = 123 - 456 \text{ in} \\ &\text{let } x = x - 789 \text{ in } -x \end{aligned} \tag{1}$$

The compiler will have no idea which value is pointed by x in each nested function if we do not apply α-conversion. If α-conversion is applied, the function will be rewritten as:

$$\begin{aligned} &\text{let } x1 = 123 - 456 \text{ in} \\ &\text{let } x2 = x1 - 789 \text{ in } -x2 \end{aligned} \tag{2}$$

Each x is specified to a unique name so that they can be recognised in each nested function.

In reality, it also affects the efficiency and complexity of other compiling steps. For example, in the optimization stage, Min-Caml will eliminate unnecessary definitions, it works in the way that searching for free variables in an expression to check whether a definition is unused or free to be eliminated. If there is no guarantee that each variable has its unique name, the elimination step will require a further one by one search in each part of the expression to make sure the usage of the variable. By implementing α-conversion before the elimination, it is unnecessary to do an exhaustive search and therefore simplify the code.

Moreover, in the inline expansion, we expand the body of the function if its body size is smaller than an integer reference threshold. During the expansion, we replace the function call by its arguments from the body mapping environment and the variable names may be repeated in different function bodies. Therefore, we have to apply α-conversion to eliminate this ambiguity in order to ensure the uniqueness of each variable.

Project B

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.

Figure 14 shows the simplest naive solution to generate closures for all functions. It does not check whether the function contains free variables or not but apply closure conversion to all. This is very time-consuming and computational inefficient.

Figure 15 classifies the functions into two cases. The first case is when we assume that the function has no free variable in e1 and closure convert e1. If there is indeed an empty set of free variable, we continue to process e2 with closure conversion. The second case is when the free variable set is not empty, the process should be rewound to the toplevel function and redo the closure conversion of e1. The drawback of this algorithm is exponential computational time with respect to the depth of nested function definitions.

Figure 16 is a more superior algorithm comparing to Figure 14 and Figure 15. It optimized the computation efficiency by classifying the functions into three cases. It considers the circumstance that even if x has no free variable but is returned as a value. This time we need to consider that both that whether x has an empty free variable set and x appears in e2 as free variable. Only if x satisfies both conditions, the closure conversion is applied. Otherwise, we can save the computation for x which does not appear in e2 and has no free variable and in this case the function can be called directly. Finally, the closure conversion is applied to x with non-empty free variable set as before. This algorithm carefully considers different cases in a function and only applies the conversion when necessary, which dramatically increases the computational efficiency.

```
| KNormal.LetRec({ KNormal.name = (x, t); KNormal.args = yts; KNormal.body = e1 }, e2) -> (* 関数定義の場合 *)
    (* 関数定義let rec x y1 ... yn = e1 in e2の場合は、
       xに自由変数がない(closureを介さずdirectに呼び出せる)
       と仮定し、knownに追加してe1をクロージャ変換してみる *)
    let toplevel_backup = !toplevel in
    let env' = M.add x t env in
    let known' = S.add x known in
    let e1' = g (M.add_list yts env') known' e1 in
    (* 本当に自由変数がなかったか、変換結果e1'を確認する *)
    (* 注意: e1'にx自身が変数として出現する場合はclosureが必要!
       (thanks to nuevo-namasute and azounoman; test/cls-bug2.ml参照) *)
    let zs = S.diff (fv e1') (S.of_list (List.map fst yts)) in
    let known', e1' =
      if S.is_empty zs then known', e1' else
      (* 駄目だったら状態(toplevelの値)を戻して、クロージャ変換をやり直す *)
      (Format.eprintf "free variable(s) %s found in function %s@." (Id.pp_list (S.elements zs)) x;
       Format.eprintf "function %s cannot be directly applied in fact@." x;
       toplevel := toplevel_backup;
       let e1' = g (M.add_list yts env') known e1 in
       known, e1') in
    let zs = S.elements (S.diff (fv e1') (S.add x (S.of_list (List.map fst yts)))) in (* 自由変数のリスト *)
    let zts = List.map (fun z -> (z, M.find z env')) zs in (* ここで自由変数zの型を引くために引数envが必要 *)
    toplevel := { name = (Id.L(x), t); args = yts; formal_fv = zts; body = e1' } :: !toplevel; (* トップレベル関数を追加 *)
    let e2' = g env' known' e2 in
    if S.mem x (fv e2') then (* xが変数としてe2'に出現するか *)
      MakeCls((x, t), { entry = Id.L(x); actual_fv = zs }, e2') (* 出現していたら削除しない *)
    else
      (Format.eprintf "eliminating closure(s) %s@." x;
       e2') (* 出現しなければMakeClsを削除 *)
| KNormal.App(x, ys) when S.mem x known -> (* 関数適用の場合 *)
    Format.eprintf "directly applying %s@." x;
    AppDir(Id.L(x), ys)
```

Figure 1: The example code fragment implement algorithm in Figure 16 is referred to closure.ml

If the example code fragment is executed, the number of make_closure should be greater than apply_closure which is greater than apply_direct.

N.B. Reference materials refer to [1] [2].

References

[1] A crash course for the mincaml compiler. `http://esumii.github.io/min-caml/index-e.html`. (Accessed on 01/08/2019).

[2] Eijiro Sumii. Mincaml: a simple and efficient compiler for a minimal functional language. In *Proceedings of the 2005 workshop on Functional and declarative programming in education*, pages 27–38. ACM, 2005.