Weekly assignment 3

Troels Thomsen - qvw203

8. December 2013

1 Week assignment 3

1.1 Exercise 1

1.1.1 1.a

Først printer vi 22, og derefter 5 og 2, fordi vi i call-by-value ikke ændrer funktionsparameter variablene.

1.1.2 1.b

Først printer vi 33, og derefter 14 og 5, fordi vi i call-by-reference ændrer funktionsparameterne videregivet i vores ydre scope. Vores "callee"ændre direkte på værdierne i vores "caller".

1.1.3 1.c

Først printer vi 22, og derefter kan vores x være enten 7 eller 12, og vores y er 3. Dette sker fordi vi i call-by-value-result returnere både vores a og c i x's position.

1.2 Exercise 2

1.2.1 2.a

Hvis vi kalder *f(4)* får vi printet *5*. Dette sker fordi funktionsparameteret int x i f, ikke overskriver det globale x, da funktionsparameteret har sit eget scope.
Hvis vi kalder f(7) printer vi igen 5, fordi vi inde i h laver et nyt x der kun eksiterer i h's scope. Havde vi kaldet *x=y+2*, i stedet for *int x=y+2*.

1.2.2 2.b

Hvis vi kalder *f(4)* får vi printet 4, fordi funktionsparameteret ikke har deres eget scope.
Hvis vi kalder *f(7)* får vi printet 9, af samme årsag.

1.2.3 2.c

Med dynamic scoping behøver vi ikke et inner vtable, fordi vi altid gerne vil opdatere vores outer vtable, der indeholder det yderste scope. Hvis vi kun ændrer vores yderste scope, vil det have samme effekt som dynamic scoping, hvor der ikke eksiterer inner scopes.

1.3 Exercise 3

1.3.1 3.a

Når funktionen *typeCheckExp* checker en funktion, gør den det rekursivt, ved at evaluere antallet og typerne af funktionsargumenterne. Først finder *typeCheckExp* de forventede argumenttyper for den givne funktion, og checker derefter om de matcher på input argumenternes typer.
Det rekursive kald sker på

```
val new_args = ListPair.map (fn (e,et)
        => typeCheckExp(vtab, e, et)) (args, exp_arg_tps)
```

Hvor hvert enkelt argument bliver type checket.
Hvis vi kører *chr(read()) = read()* igennem *typeCheckExp*, får vi at chr kræver en int som parameter, og returnerer en char. Så kalder vi *typeCheckExp* rekursivt på read, som må returnerer en int, hvis vi ikke skal have en typefejl.

1.3.2 3.b

For at finde de identiske typer må vi bruge *unify* som beskrevet i Most-General Unifier Algorithm.

1. *unify(*, *)* får os ind i step IV, fordi begge er type constructor.
2. *unify(list, alpha)* får os ind i step III fordi alpha er en type variable. Så får vi altså at *union(list, alpha) = list int*, fordi vi tager alle list's børn med i union.
3. I *unify(list, beta)* får os ind i step III fordi beta er en type variable. Så kører vi *union(list, beta) = list list int*, fordi vi igen tager alle list's børn med i union.

Alpha er altså en list int, og beta er en *list alpha = list list int*. Den samlede type bliver derfor en tuble *(list int, list list int)*.