Type Systems and Functional Programming

Type inference

The objective of this activity is the implementation of a **type inference** mechanism for the polymorphic lambda calculus (System F), seen as a programming language.

The language specification can be found on page 3 of this document.

Example

The typing output of a previous example is given below:

Prog.in

```
a
(a b)
true=\x.\y.x
false=\x.\y.y

not=\x.((x false) true)
(not true)
(not false)

and=\x.\y.((x y) false)
((and true) true)
((and true) false)
((and false) true)
((and false) false)

or=\x.\y.((x true) y)
((or true) true)
((or true) false)
((or false) true)
((or false) false)

nil=\x.true
null=\l.(l \x.\y.false)
cons=\x.\y.\z.((z x) y)
car=\l.(l true)
cdr=\l.(l false)
if=\p.\then.\else.((p then) else)
```

Type inference

```
t0
t4
(t6->(t7->t6))
(t9->(t10->t10))

(((t13->(t14->t14))->((t17->(t18->t17))->t20))->t20)
(t21->(t22->t22))
(t32->(t33->t32))

((t41->((t44->(t45->t45))->t47))->(t41->t47))
(t56->(t57->t56))
(t68->(t69->t69))
(t73->(t74->t74))
(t85->(t86->t86))

(((t99->(t100->t99))->(t98->t104))->(t98->t104))
(t105->(t106->t105))
(t117->(t118->t117))
(t137->(t138->t137))
(t149->(t150->t150))

(t154->(t155->(t156->t155)))
(((t159->(t160->(t161->(t162->t162))))->t164)->t164)
(t166->(t167->((t166->(t167->t172))->t172)))
(((t175->(t176->t175))->t178)->t178)
(((t181->(t182->t182))->t184)->t184)
((t187->(t188->t192))->(t187->(t188->t192)))
```

The numbers above are generated by a **counter**, used to spawn **unique** type variables.

Strong/weak polymorphism

Assume the following Haskell snippets:

Strong polymorphism

```
f x y = ( (id x), (id y) ) -- id is identity function in Haskell

> :t id
a -> a

> :t f
f :: a -> b -> (a, b)
```

```
> f 2 3
(2, 3)

> f 2 True
(2, True)
```

Weak polymorphism

```
f1 g x y = ( (g x), (g y) )

> :t f1
f1 :: (a -> b) -> a -> a -> (b, b)

> f1 id 2 3
(2, 3)

> f1 id 2 True
<type error>
```

The **difference** between the two snippets resides in the way the applied function is **supplied**:

- In the **first** case, the function is mentioned **explicitly**. It is polymorphic and adapts to the type of its actual argument, on **each** application within the definition. We say that the top-level variables are **strongly polymorphic** or, equivalently, that the type variables from within their signature are **universally** quantified. Thus, we may read the type of `f` as `∀a.(a → a)`.
- In the **second** case, the function is passed as an **argument**. In contrast to the first case, this can be thought of as having a single function "instance". When the function is **first** applied, within the definition, the type variables in its **signature** become bound. In the snippets above, when `g` is applied onto `2`, the type of its argument remains bound to `Int`, which leads to an error when attempting to apply `g` once again, but this time onto `True`. We say that the formal function arguments are **weakly polymorphic** or, equivalently, that the type variables from within their signature are **free**. Thus, we may read the type of `g`, when the actual argument is `id`, as simply `(a → a)`, with no quantification involved.

The behavior from the first case is obtained by **copying** the expression signature, for **each** of its occurrences. This way, the type variables are replaced with new ones, independent from the former. For example, in the first snippet above, we may think that the first occurrence of `id` has the type `(a → a)` and the second one, `(b → b)`. This is the purpose of the `copy` function in the `TVar'` synthesis rule, in `Synthesis Example.pdf`.

Implementation

Use your **unification** mechanism as a starting point. The following are recommended **steps** for implementing the inference rules:

1. `Typing.Inference`: Implement the functions in the order in which they are provided. Some remarks:
    - While contexts map **program variables** to their types, substitutions map **type variables** to other types!
    - While the lexical contexts are enriched while recursing **forward**, substitutions are enriched on the way **back**.
    - The typing rules for **variables** may be stated as below:
        - If a variable is located within the lexical context, its type is returned **as is**.
        - If a variable is located within the dynamic context, its type is returned as a **unique copy**.
        - If a variable cannot be located within either of the two contexts, a new type variable is **generated**.
    - The typing rules for functional abstractions and definitions involve **hypothesizing** about the type of the formal argument, and the variable being defined, respectively. For generating new type variables, a **counter** may be employed, which gets incremented after each generation.
