# Fundamentals of Programming Languages

# Assignment 5

# Linear Types and Session Types

Mestrado em (Engenharia) Informática
Faculdade de Ciências da Universidade de Lisboa

2022/2023

## 1 Linear Types

1. The linear type system does not enjoy the property of Linear Weakening. Linear Weakening states that if Γ ⊢ t : T, then Γ, x : Q ⊢ t : T for any type Q. However, in the linear type system, this is not necessarily the case.

   For example, consider the following linear type system:

   Γ ⊢ t : T

   where T is a linear type. If we try to add a new variable x : Q to the context Γ with Q being a linear type, we would not be able to do so because the linear type system only allows us to use a linear type once. Thus, the linear type system does not enjoy the property of Linear Weakening.

   Here is a counterexample:

   Let Γ = x : A, T = A, and Q = A. Then, Γ ⊢ x : A. However, if we try to add x : A to the context, we get the following:

   Γ, x : A ⊢ x : A

   This is not valid in the linear type system, because A has already been used in the context Γ. Therefore, the linear type system does not enjoy the property of Linear Weakening.

2. By induction hypothesis on $\Gamma_1 \vdash t : T; \Gamma_2$

   - Case A-UVAR:
     $t = x$
     $T = unP$
     By induction hypothesis if $\Gamma_1, x : unP, \Gamma_2 \vdash t : T; \Gamma_1, \Gamma_2$ and $L(\Gamma_2) = \emptyset$ then $\Gamma_1, x : unP, \Gamma_2 \vdash t : T$. The theorem is proved to be true and we can observe that our conclusion looks similar to that of T-VAR.
   - Case A-LVAR:
     $t = x$
     $T = linP$
     By induction hypothesis if $\Gamma_1, x : linP, \Gamma_2 \vdash t : T; \Gamma_1, \Gamma_2$ and $L(\Gamma_2) = \emptyset$ then

$\Gamma_1, x : linP, \Gamma_2 \vdash t : T$. The theorem is proved to be true and we can observe that our conclusion looks similar to that of T-VAR.

- Case A-ABS:
  $t = q\lambda x : T_1.t_2$
  $T = qT_1 \rightarrow T_2$
  $\Gamma, x : T_1 \vdash t_2 : T_2; \Gamma_2$
  $q = un \implies \Gamma_1 = \Gamma_2\%(x : T_1)$
  By induction hypothesis if $\Gamma, x : T_1 \vdash t_2 : T_2; \Gamma_2$, $L(\Gamma_2) = \emptyset$ and $q = un \implies \Gamma_1 = \Gamma_2\%(x : T_1)$, then $\Gamma, x : T_1 \vdash t_2 : T_2$. By applying A-ABS we get $\Gamma_1 \vdash t : T$. The theorem is proved to be true and we can observe that our conclusion looks similar to that of T-ABS.
- Case A-APP:
  $t = t_1 t_2$
  $\Gamma_1 \vdash t_1 : qT_{11} \rightarrow T_{12}; \Gamma_2$
  $\Gamma_2 \vdash t_2 : T_{11}; \Gamma_3$
  - First subcase:
    By induction hypothesis, if $\Gamma_1 \vdash t_1 : qT_{11} \rightarrow T_{12}; \Gamma_2$ and $L(\Gamma_2) = \emptyset$, then we get $\Gamma_1 \vdash t_1 : qT_{11} \rightarrow T_{12}$.
  - Second subcase:
    By induction hypothesis, if $\Gamma_2 \vdash t_2 : T_{11}; \Gamma_3$ and $L(\Gamma_3) = \emptyset$, then we get $\Gamma_2 \vdash t_2 : qT_{11}$.

  By A-APP, we get $\Gamma_1 \vdash t : T_{12}$. The theorem is proved to be true and we can observe that our conclusion looks similar to that of T-APP.

3.

4. A sound type system can be defined by only allowing typable programs to be accepted. This means that if a program is accepted by the type system, it is guaranteed to be free of runtime typing errors.

   A complete type system can be defined by the hability of assigning a type to every possible program. This means that every program that can be written has a corresponding type in the type system. This can be useful for languages with type inference.

   It is generally preferable for a type system to be sound rather than complete for a few reasons:

   - Soundness ensures that typable programs behave correctly: Since the type system guarantees programs to be free of runtime typing errors, it helps to ensure the correctness and reliability of the program, accepting less but more correct programs.
   - Completeness can lead to unsound programs being accepted: It may allow programs that are not typable to be accepted and the type system may lead to errors in runtime.
   - For programmers it is more important to have soundness rather than completeness: Soundness will help catching and fixing mistakes early on in the development process. Indeed, completeness allows programmers to write programs with confidence, knowing that the type system will assign a type to the program, but it might lose relevance in the long term.

   As an example, we have a type system with a plus operation "+", that adds two integers. A sound type system might only allow the "+" operation to be applied to two integers and would reject programs that attempt to add values of other types (e.g. strings or booleans). This would ensure that all typable programs are guaranteed to behave correctly, because the type checker would catch any attempts to add incompatible types.

If we have the same type system to be complete, the same operation might be allowed to be applied to other types (e.g. integers, floats and strings). In this case, the type checker would not catch any errors related to adding incompatible types, and the programmer would need to handle these cases manually at runtime.

## 2 Session Types

Due to the derivations' length, I decided to solve them in paper. They can be found in the external images sent with this report.
I will be using () to represent the Unit constant/value for the sake of preserving space.

5. Solved in the attached image "5.jpg".

6. Solved in the attached image "6.jpg".

   R-NEW → R-THREAD → R-FORK → R-THREAD → R-THREAD → R-COMF → R-THREAD → R-CLOSE

   With this, we have that $n = 8$.

7. Solved in the attached pdf "7.pdf", for the following expression $e$:
   $\langle let(w, r) = new\ !unit.end_!\ in$
   $let(x, r) = receive\ r\ in\ wait\ r;$
   $fork\ (\backslash\_ : unit \to_1 close\ (send\ unit\ w));$
   $x\rangle$

   We can conclude that, since the process <x> can never end up in a "wait x" or "close y" format, the rule R-CLOSE will never apply and we don't have to reduce any further to show this.