Bachelor project: Type safety for system F with Coq

Wène Kouarfate

February 2024

Contents

1 Introduction

Modern programming languages need to ensure program security, especially regarding typing. As type systems become more complex, formal methods are essential for their verification. This bachelor's work aims to use Coq to prove the safety of the System F typing system. It serves as a practical introduction to type systems formalization and computer-assisted proof methods while enhancing understanding of type theory in programming.

We will use a formal system, the λ calculus, and its extensions as our abstract programming language. This approach eliminates technical and implementation details. Our programs will be expressions in this language, and their execution will be computation processes until they yield an output.

The lambda calculus is also central to our second major topic: computer-assisted proofs. We will find that the terms of our language can serve as proofs, and their types as propositions. This is known as the Curry-Howard isomorphism. It creates a duality between the lambda calculus type systems and logical systems like propositional logic. Simply put, it allows us to use programs and their type systems in the same way as proofs and assertions.

[1]

2 State of the art

We will genuinely start with one of the simplest forms of the λ-calculus as initially formulated by Church to build more sophisticated extensions through our experience of this functional language from a programmer's point of view.

2.1 Type-free lambda calculus

Let V be the set of variables. The set Λ of expressions of the λ-calculus or λ-terms is the smallest following set satisfying :

$$V ::= x \mid y \mid z \mid \ldots$$
$$\Lambda ::= V \mid \lambda V.\Lambda \mid (\Lambda\Lambda)$$

λ-terms can respectively be variables, abstraction of terms into a function and application of one term to another λ-terms. Applications are left-associative and abstractions are right-associative.

2.1.1 Substitutions

A variable is bounded to an abstraction $\lambda x.M$ if it corresponds to the formal variable x used to capture the argument. Bound variables can be seen as placeholders and substituting them does not change the sens of the term e.g $\lambda x : x$ and $\lambda y.y$ does the same thing. Free variables are those that are not bound in

[1]The Coq implementation of the following work can be found at https://gitlab.unige.ch/Wene.Kouarfate/type-safety-for-system-f-with-Coq

a term. They correspond in a subprogram to constants and may be subject to renaming via substitution.Their definition is recursively extended to Λ as the function $FV : \Lambda \to \mathscr{P}(V)$

$$\begin{aligned} x \in V &\Rightarrow FV(x) = \{x\} \\ x \in V, M \in \Lambda &\Rightarrow FV(\lambda x.M) = FV(M) \setminus \{x\} \\ M, N \in \Lambda &\Rightarrow FV((MN)) = FV(M) \cup FV(N) \end{aligned}$$

Naive substitution of a variable x by a term N in another term M (annotated $M[x := N]$) consists of replacing all free occurrences of x in M by N.

The problem with this is illustrated by *variable capture* happening when a variable that is free in N ends up being bound in $M[x := N]$. It feels like breaking the semantic of the λ-calculus regarding the following example : substituting y to x in $\lambda x.xy$ ruins the argument capture property :

$$((\lambda x.xy)[y := x])N \text{ is } (\lambda x.xx)N \text{ results in } (NN)$$

while the same operation with $z, z', ...$ results in $(Nz), (Nz'), ...$
Formally :

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \ (x \neq y) \\ (\lambda x.M)[x := N] &\equiv \lambda x.M \\ (\lambda y.M)[x := N] &\equiv \lambda y.M[x := N](x \neq y, y \notin FV(N)) \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_1[x := N]) \end{aligned}$$

For the rest of the work, we will assume that variables are always chosen so that all bound variables of terms involved in substitution are chosen to be different from free ones. This convenient way to avoid variable capture troubles is called the variable convention or the Barendregt [1] convention and will play a central role in further proofs.

2.1.2 Reduction

$$(\lambda x.M)N \to_\beta M[x := N]$$

Reduction featuring the so-called β-redexes (The application of an abstraction $\lambda x.M$ to another term N) and corresponding reduction is the core concept of the λ-calculus as an abstract programming language, it introduces the idea of execution. Abstraction can be seen as programs waiting for arguments, with their bounded variables as formal parameters. It is replaced by the execution parameter and the evaluation can keep going into the substituted body.

2.2 Simply typed lambda calculus

An issue with the type-free λ-calculus is its indiscriminate nature. It is like doing physical computations without caring about dimensions. If a function

R associated with a specific resistor aims to return the resulting tension when a current (represented by the λ-term I) flows through it (RI), it should not then be applied to a λ-term R' representing another resistor: do not compare oranges and apples! R will then be a function from ampers to volts, annotated $R : \mathtt{A} \to \mathtt{V}$. Formally, the set of types T is defined as the smallest following set :

$$B = \mathtt{Bool} \mid \mathtt{Nat} \mid ...$$
$$T = B \mid T \to T$$

Types can be base types or arrows from any type to another one.We write $M : \sigma$ to mean M of type σ. Hence abstractions become

$$\lambda x : \sigma.M$$

so that one knows a term has to be of type σ (e.g. $\mathtt{A}$ for ampers) to be involved with it in a β-reduction. This verification belongs to type-checkers according to the following rules :

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash M : (\sigma_1 \to \sigma_2) \quad \Gamma \vdash N : \sigma_1}{\Gamma \vdash (MN) : \sigma_2}\ (\to\text{E})$$

$$\frac{\Gamma; (x : \sigma_1) \vdash M : \sigma_2}{\Gamma \vdash \lambda x : \sigma_1.M : (\sigma_1 \to \sigma_2)}\ (\to\text{I})$$

With the context Γ inductively defined as : $(x : \sigma_1) \in \Gamma :: (y : \sigma_2)$ if $(x : \sigma_1) \equiv (y : \sigma_2)$ or $(x : \sigma_1) \in \Gamma$ and $(x : \sigma_1) \notin \emptyset$ where " :: " is the appending operator and $\emptyset$ the empty context.

2.3 Polymorphic lambda calculus (System F or $\lambda 2$)

2.3.1 Intuition

It is a common thing in algorithmic to have some problems that are strongly independent from involved types like BFS or DFS traversals or shortest path algorithms on graphs of numbers or strings that would be pretty similar modulo the types.

This is one of the two motivations of John C. Reynolds in [5] to introduce *an extension of the λ-calculus which permits user-defined types and polymorphic functions* illustrated by the *problem of polymorphic sort functions.* A program in which many types of arrays must be sorted so that for any type σ, it accepts an array of that type and a binary ordering predicate whose elements must also be of that type. Where we would have no choice but to write separate but similar sort functions for each type with our previous functional language, Reynolds suggests the possibility for *types themselves to be passed as a special kind of parameter whose usage is restricted in a way that permits the syntactic checking of type correctness for some σ* .

The second motivation is the idea of languages where the semantics of correct programs should never depend upon the implementation of primitives. The primitive types like integers would in other words be polymorphic to their implementations (e.g. binary representation as our hardware or rather Peano's arithmetic like Coq's `nat`).

2.3.2 Formal syntax and semantics of System F

This results in system F, whose set of types T is the smallest fitting the following rules :

$$\begin{aligned} V_T &= \alpha \mid \mu \mid ... && \text{(type variables)} \\ B &= \texttt{Bool} \mid \texttt{Nat} \mid ... && \text{(base types)} \\ T &= B \mid T \to T \mid \forall V_T.T && \text{(types)} \end{aligned}$$

Types can be made out of base types, arrow types, and quantification of type variables out of other types and we are now able in our programs to abstract type variables α out of terms M (annotated $\lambda\alpha.M$)

For example $\texttt{Id} \equiv (\lambda\alpha.\lambda x : \alpha.x)$ is an abstraction of the type variable α out of a classical identity function $\lambda x.x$. When applied to a type, e.g. $\texttt{Id[Nat]} \to_\beta \lambda x : \texttt{Nat}.x$, it results in the identity function for elements of those types. The type-checker will derive types of those abstractions or applications of types according to the following rules :

$$\frac{\Gamma; \alpha \vdash M : \sigma}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\sigma}\ (\forall\text{I}) \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma_1}{\Gamma \vdash M[\sigma_2] : \sigma_1[\alpha := \sigma_2]}\ (\forall\text{E})$$

Example 1.

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in (\alpha; (x : \alpha)) \qquad \alpha \notin FV(\alpha; (x : \alpha))}{\alpha; (x : \alpha) \vdash x : \alpha}\ \text{Var}}{\alpha \vdash (\lambda x : \alpha.x) : \alpha \to \alpha}\ \to\text{I}}{\vdash (\lambda\alpha.\lambda x : \alpha.x) : \forall\alpha.\alpha \to \alpha}\ \forall\text{I}}{\vdash (\lambda\alpha.\lambda x : \alpha.x)\texttt{[Nat]} : (\forall\alpha.\alpha \to \alpha)[\alpha := \texttt{Nat}]}\ \forall\text{E}$$

2.4 The Coq proof assistant

Coq as a software is built upon a functional language called Gallina. The language has a common syntax similar to those of the ML family as OCaml and will not be formally presented (for more information please refer to [6]).

2.4.1 Prior

We approach in the following points some of the theoretical foundations (among many others) of the system Coq that we found necessary to the comprehension of everything that would be implemented.

- **Constructive logic**: a variant of the classical logic that rejects the rule of excluded middle tierce ($\Gamma \vdash A \wedge \neg A$) or equivalent deduction schema such as the double negation elimination ($\Gamma \vdash \neg\neg A \iff A$). These principles would allow non-constructive existential proofs: proofs of the existence of a mathematical object that could not provide a way to find such an object [2] [3]. Constructive proofs of existence can automatically be derived into an algorithm for building such objects via the next point :

- **The Curry-Howard isomorphism**: (CHI) A one-to-one correspondence between types with their inhabitants (expressions of that type) and logical assertions with their proofs. Let us compare derivation and proof trees :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A; x : A; y : B \vdash x : A}\,(\text{Var})}{u : A; x : A \vdash \lambda y : B.x : B \to A}\,(\to \text{I})}{u : A \vdash \lambda x : A.\lambda y : B.x : A \to B \to A}\,(\to \text{E}) \qquad \dfrac{}{u : A \vdash u : A}\,(\text{Var})}{u : A \vdash (\lambda x : A.\lambda y : B.x)(u) : B \to A}\,(\to \text{E})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A; B \vdash A}\,(\text{Var})}{A \vdash B \Rightarrow A}\,(\Rightarrow \text{I})}{A \vdash A \Rightarrow B \Rightarrow A}\,(\Rightarrow \text{E}) \qquad \dfrac{}{A \vdash A}\,(\text{Var})}{A \vdash B \Rightarrow A}\,(\Rightarrow \text{E})$$

The latter has the same structure as the former but with terms erased out (plus replacing $\to$ by $\Rightarrow$) . Coq uses this property notably to extract verified programs out of correctness proofs. In Coq, `Set` is the sort of program's type and `Prop` is the sort of assertions (The type of proposition which are themselves types of their proofs). `Prop` and `Set` are themselves of type `Type`

- **The (inductive) calculus of constrution**: the CHI establishes a parallel among extensions of the λ-calculus type system and extensions built upon its corresponding logical system: the minimal propositional logic[4]. Barendregt presents these extensions in [1] among three axes :

  – Polymorphism: abstraction of types out of terms. It corresponds to second-order propositional logic where quantification can be made over the range of propositions, e.g. polymorphic lists on the parametrized type `A`:

[2] Such as a proof that there exist two irrational numbers a and b such that a^b is rational without providing a pair of a and b nor an algorithm to approach them (or without knowing if a and b can be either $\sqrt{2}$ and $\sqrt{2}$ or $\sqrt{2}^{\sqrt{2}}$ and $\sqrt{2}$)

[3] Not like Cantor's diagonal method that constructs for any sequence of numbers in the interval $[0, 1]$ a real number that could not fit a one-to-one correspondence with the set of natural numbers, proving its non-countability.

[4] These systems are generally minimal with only the $\Rightarrow$ and $\forall$ operators

```
Inductive list (A : Set) :=
    | nil : list A
    | cons : A -> list A -> list A.
```

Expressions of that type can be made by two constructors that are declared with their types.

– Dependent types: types parametrized by terms. They correspond to the predicate logic which features the quantification of individual variables over predicates. Dependent types are a powerful expressive tool frequently used in inductive definitions to parameterize (or quantify since Coq uses the keyword `forall`) the types of some constructors by variables followed by one or more assertions on them. Example of a list of natural numbers lower or equal to 4

```
Inductive list4 :=
    | nil4 : list4
    | cons4 : forall (n : nat), (n <= 4) -> list4 -> list4.
```

To build such a list using the second constructor, it must be provided not only a natural number but also an expression of type $\texttt{n} \leq \texttt{4}$: a proof that the number is lower or equal to 4. Another example of dependent type can be found on the way to build such a proof expression, by applying theorems as simple functions :

```
le_S : forall (n m : nat), (n <= m) -> (n <= S m)
le_n : forall (n : nat), n <= n
```

They can be cleverly applied to numbers to obtain the following term and its type: `(le_S 2 3 (le_S 2 2 (le_n 2))): 2 <= 4`

– Type operators: functions of types into types. It corresponds to higher-order propositional logic. For example, let us alternatively build the (`Set`) type of lists of natural numbers lower or equal to 4 by first defining a dedicated type and then providing it as the `A` type parameter to `list`:

```
Inductive nat_leq_4 : Set :=
  | of_nat : forall (n : nat), (n <= 4) -> nat_leq_4.

Definition list4len := list nat_leq_4.
```

– The calculus of construction (CoC): Barendregt presents each one of those extensions as moving along one of a cube axes where the simply typed lambda calculus is the origin: the lambda cube. The CoC[5] is simply the diagonal move on this cube for the origin to its opposite point which can be seen as moving on the three axes. The calculus of inductive construction extends CoC notably with inductive types.

[5]A clue on the origin of why Coq is called Coq!

2.4.2 Commands

Coq organizes the execution flow of programs through a wide range of commands (keywords starting with uppercase whose statements end with a dot). Results are printed in a dedicated panel. Let us review the most common and useful ones in our further implementations :

- `Check` : type-checks its argument. Any expression manipulated by Coq has its type prealably derivated. This also holds for functions, inductive definitions, or even theorems. `Print` prints the definition of an object if available[6] including its type and `Compute` also reduce it (more on that in 2.4.3). E.g. the Peano arithmetic style implementation of natural numbers in Coq :

```
 Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat.
```

- `Inductive`: define inductive types via constructors and their signature. Coq automatically generates induction principles that can be used to prove properties by induction about those types e.g. `list4`. The `Fixpoint` command defines recursive functions about inductive definitions using `match` construct to capture their recursive structure :

```
Fixpoint list4len (l : list4) :=
   match l with
   | nil4 => 0
   | (cons4 h _ t) => 1 + (list4len t)
end.
```

  Normal functions can be defined as any other standard term with the `Definition` command[7]. Recursive functions as well as classical functions can also be defined in an anonymous function style :

```
Definition list4sum := (fix f (l:list4) := match l with
   | nil4 => 0
   | (cons4 h _ t) => h + (f t)
end).
```

- `Notation` : defines custom notations. `Notation "A /\ B" := (and A B).`

- `Theorem, Lemma, Corollary ...` equivalent commands that start a proof. They take the assertion as an argument and switch into proof mode.`Proof` and `Qed` starts and stops the body of the demonstration[8].It assists in interactively building proofs using tactics which can be seen asdeduction patterns like *modus ponens*, by induction or by contradiction ...

[6]The object is said to be transparent or else it is opaque

[7]Functions are first-class citizens!

[8]More than visual delimitations of proofs in a script, the former can be used to provide general directives for the whole proof like theorem databases for automated proof and the later try to build the proof term from the interactive proof.

Let us begin by presenting the proof state: a set of yet unproven goals. A goal consists of a conclusion and its local context separated by a line. The conclusion is the current to-be-proven statement. The local context is made out of direct variables involved in the proofs, stated with their type. They could be variable with their type (of sort `Set`) or hypotheses as terms of the type of the asserted assumptions (of sort `Prop`). The following example shows a `Theorem` statement entering proof mode and presenting the proof state. It is about the sum of elements in a `list4` being lower or equal to four times its length.

```
Theorem list4sum_le_list4len_mul_4 :
        forall (l:list4), (list4sum l) <= 4 * (list4len l).
1 goal
  ============================
  forall l : list4, list4sum l <= 4 * list4len l
Proof.
```

2.4.3 Tactics

Tactics can generate subgoals: to prove the current conclusion, one applies a tactic that transforms it into one or more (simpler) subgoals that have to be proven and so on until there remains none. They can be combined with the semicolon operator (`tactic1;tactic2`): the latter tactic is applied to the subgoals generated by the former.

- `intros`: introduce the leftmost parts of an implication as hypotheses and also quantified variables as defined in the context. It corresponds to the inference rule that proves implications: to show that `A` implies `B`, prove the latter under the hypothesis of the former. The same holds for the $\forall$ introduction rule: you have to prove the quantified proposition with the variable added to the context.

  ```
  intros l.
    l : list4
    ============================
    list4sum l <= 4 * list4len l
  ```

- `induction`: proof by structural induction or induction on a relation. It generates from the current goal as many subgoals as constructors of the inductive type and adds an induction hypothesis by default based on the induction principle generated by the `Inductive` command.

  ```
  induction l.
  2 goals
    ============================
    list4sum nil4 <= 4 * list4len nil4
  goal 2 is:
   list4sum (cons4 n l l0) <= 4 * list4len (cons4 n l l0)
  ```

You also have a `destruct` tactic that does not generate induction hypothesis and `inversion` that directly solves self-contradictory subgoals.

- `simpl` : performs a 'light' normalization. It results in a simpler form of the expression making it eligible for other tactics.

  ```
  simpl.
    ============================
    0 <= 0
  ```

- `reflexivity` : resolve goals of the form `R t t'` if `R` is a reflexive relation and if `t` and `t'` are definitionally equal or convertible (more on that with the next point). It ends the current goal and presents the next if there is one. We can also see the induction hypothesis `IHl`.

  ```
  reflexivity.
  1 goal
    n : nat
    l : n <= 4
    l0 : list4
    IHl : list4sum l0 <= 4 * list4len l0
    ============================
    list4sum (cons4 n l l0) <= 4 * list4len (cons4 n l l0)
  ```

  Similar tactics such as `trivial` (resolves goals that correspond to an accessible lemma), `contradiction` (solve the current goal by finding contradiction in the conclusion and hypotheses) [9] with `reflexivity` are all attempted with the `easy` tactic.

- `cbv` : call-by-value-oriented conversion[10]. Conversion rules check if two terms are convertible i.e. if they both convert to normal forms that are syntactically equal. `simpl` is another conversion tactic that aims for more readable results by unfolding only constants that lead to 'simplification'.

  Briefly we have α-equivalence for syntactic equality modulo bound variables, β-reduction of β-redexes, δ-reduction replacing context variables by their values, ι-reduction of inductive objects (`match, fix,...`) , ζ-reduction of `let ... in ...` definition and η-expansion of term $M : (\forall x : T, U)$ by $\lambda x : T.(Mx)$. For more details please refer to [6]. They can parameterize the conversion tactic :

  ```
  cbv beta iota delta [list4len list4sum];fold list4len list4sum.
    ...
    ============================
    n + list4sum l0 <= 4 * (1 + list4len l0)
  ```

[9] `inversion` or also `symmetry`

[10] Also `lazy` for call-by-need or `cbn` for call-by-name.`hnf` for weak-head normal form

- `rewrite` rewrite a side of equality from one expression to another[11].

```
Nat.mul_add_distr_l
     : forall n m p : nat, n * (m + p) = n * m + n * p

rewrite Nat.mul_add_distr_l.
  ...
  ============================
  n + list4sum l0 <= 4 * 1 + 4 * list4len l0
```

- `apply` : the *modus ponen* inference rule, if `A` implies `B` and `A` is stated we can deduce `B`.[12].To prove an assertion `B` while disposing of an assertion `A`⇒`B`, it is sufficient to prove `A` so that *modus ponen* would deduce `B` (and so on for `A`⇒`A'`⇒...`B` that will generate subgoals `A`, `A'`...). Two subgoals are generated and will be solved by `easy`.

```
Nat.add_le_mono
     : forall n m p q : nat, n <= m -> p <= q -> n + p <= m + q

apply Nat.add_le_mono.
2 goals
  ...
  l : n <= 4
  IHl : list4sum l0 <= 4 * list4len l0
  ============================
  n <= 4 * 1
goal 2 is:
 list4sum l0 <= 4 * list4len l0.

easy.
easy.
(* Or simply: apply Nat.add_le_mono;easy. *)
No more goals.
Qed.
```

3 The theory and its implementation

Let us define and implement a more concrete functional language in Coq and work on it. This language is expected to have the $\lambda 2$ property we want to prove and study but also usual programming language primitives like naturals, booleans, zero-predicate, and if-then-else statements.

[11]Theorem eligible to a rewriting pattern can be found with the `SearchRewrite` command
[12]Theorem eligible for `apply` can be found with `SearchPattern` command, `Search` Command simply found theorem matching a given pattern

3.1 Syntax

Let Λ, V, and T respectively be the smallest Sets of terms, values and typesdefined by :

$$
\begin{aligned}
\Lambda := {} & x && \text{(variable)}\\
& \mid \lambda x : T.\Lambda && \text{(term abstraction)}\\
& \mid \lambda \alpha.\Lambda && \text{(type abstraction)}\\
& \mid (\Lambda\ \Lambda) && \text{(term application)}\\
& \mid \Lambda[T] && \text{(type application)}\\
& \mid true && \text{(true)}\\
& \mid false && \text{(false)}\\
& \mid if\ \Lambda\ then\ \Lambda\ else\ \Lambda && \text{(If then else)}\\
& \mid 0 && \text{(Zero)}\\
& \mid S\ \Lambda && \text{(Successor)}\\
& \mid P\ \Lambda && \text{(Predecessor)}\\
& \mid Z\ \Lambda && \text{(Zero predicate)}
\end{aligned}
$$

$$
\begin{aligned}
V := {} & \lambda x : T.\Lambda && \text{(term abstraction)}\\
& \mid \lambda \alpha.\Lambda && \text{(type abstraction)}\\
& \mid true && \text{(Boolean value : true)}\\
& \mid false && \text{(Boolean value : false)}\\
& \mid 0 && \text{(Natural value : Zero)}\\
& \mid S\ V && \text{(Natural value : Non-zero natural number)}
\end{aligned}
$$

$$
\begin{aligned}
T := {} & \alpha && \text{(Type variables)}\\
& \mid \texttt{Bool} \mid \texttt{Nat} && \text{(Base types)}\\
& \mid T \to T && \text{(Arrow types)}\\
& \mid \forall \alpha.T && \text{(Quantified types)}
\end{aligned}
$$

We can exemplify the types implementation by:

```
    Inductive typ : Set :=
  | typ_bool : typ
  | typ_nat : typ
  | typ_var : nat -> typ
  | typ_arrow : typ -> typ -> typ
  | typ_all : nat -> typ -> typ.
```

Thanks to the `Notation` command, the implementation uses a user-defined, friendly syntax close to the one defined previously that we will be alternatively using or mixing in definitions with classical theoretical notations.

Another more technical comment is about our choice to represent type variables with the Coq's native `nat` type as we also implement term variables using Coq `string` type. This convenient choice to choose Coq datatypes and to better differentiate types and term variables will also have its importance in further proofs on those variables that will be at the machine level on those datatypes.

3.2 Substitutions

First, let us extend the definitions at 2.1.1 to our fully defined language exemplifying our implementation here:

```
  Fixpoint sub_trm ($x$:string)($s$ $t$:trm) struct $t$:  trm := match t with
| $y$ => if $x \equiv y$ then $s$ else $t$
| $\lambda y : T.t1$ => if $x \equiv y$ then $t$ else $\lambda y : T.(t1[x \rightarrow s])$
| $\lambda k.t_1$ => $\lambda k.(t1[x \rightarrow s])$
| $t_1$ $t_2$ => $(t_1[x \rightarrow s])(t_2[x \rightarrow s])$
| $t_1[T]$ => $(t1[x \rightarrow s])[T]$
| $true$ => $true$
| $false$ => $false$
| $if$ $t_1$ $then$ $t_2$ $else$ $t_3$ => $if$ $t_1[x \rightarrow s]$ $then$ $t_2[x \rightarrow s]$ $else$ $t_3[x \rightarrow s]$
| $0$ => $0$
| $S$ $t_1$ => $S$ $(t1[x \rightarrow s])$
| $P$ $t_1$ => $P$ $(t1[x \rightarrow s])$
| $Z$ $t_1$ => $Z$ $(t1[x \rightarrow s])$
end
```

Substitution is fundamental to both evaluation and typing which represent the execution and the related type-checking of our programs. Moving from simple λ-calculus to its polymorphic form brings a new concept to the table to consider: types as concrete constructions that can be manipulated almost the same as terms. They can be abstracted and applied to lambda terms and then need their proper substitutions. We will introduce two substitutions due to our choice in annotation convention: the church style instead of the curry style.

- Typing *à la Church* where terms are (partially) annotated together with types. Every abstraction on terms must specify the type of their formal argument. This approach we follow with our simply typed λ-calculus implies that an initial substitution of types into terms may result in a sub-substitution of types into annotated types.[13]

- Typing *à la curry*[14] where terms are syntactically the same as in the untyped λ-calculus. It corresponds to implicit typing programming languages where the type-checker tries to infer types and succeeds if they

[13] In such systems, each term has at most one type

[14] Terms in those type systems can have more than one type.

exist instead in the previous case of verifying them and succeeding if they match with the declaration [15] .

Back to substitutions, the complete rules for a certain type σ by a variable type α are the following (we will now use conventional mathematical notations):

$$\begin{aligned}
x[\alpha := \sigma] &\equiv x \\
(\lambda x : \delta.M)[\alpha := \sigma] &\equiv \lambda x : \delta[\alpha := \sigma].M[\alpha := \sigma] \\
(\lambda \alpha.M)[\alpha := \sigma] &\equiv \lambda \alpha.M[x := M] \\
(M_1 M_2)[\alpha := \sigma] &\equiv (M_1[\alpha := \sigma])(M_1[\alpha := \sigma]) \\
(M[\delta])[\alpha := \sigma] &\equiv (M[\alpha := \sigma])[\delta[\alpha := \sigma]] \\
true[\alpha := \sigma] &\equiv true \\
false[\alpha := \sigma] &\equiv false \\
(if\ M_1\ then\ M_2\ else\ M_3)[\alpha := \sigma] &\equiv if\ M_1[\alpha := \sigma]\ then\ M_2[\alpha := \sigma]\ else\ M_3[\alpha := \sigma] \\
0[\alpha := \sigma] &\equiv 0 \\
(SM)[\alpha := \sigma] &\equiv SM[\alpha := \sigma] \\
(PM)[\alpha := \sigma] &\equiv SM[\alpha := \sigma] \\
(ZM)[\alpha := \sigma] &\equiv SM[\alpha := \sigma]
\end{aligned}$$

As stated previously, this definition involves a second type of substitution of types into the type δ where $\delta[\alpha := \sigma]$ is recursively defined in the following way :

$$\begin{aligned}
\alpha[\alpha := \sigma] &\equiv \sigma \\
\gamma[\alpha := \sigma] &\equiv \gamma\ (\alpha \neq \gamma) \\
(\forall \alpha.T)[\alpha := \sigma] &\equiv \forall \alpha.T \\
(\forall \gamma.T)[\alpha := \sigma] &\equiv \forall \gamma.(T[\alpha := \sigma]), (\alpha \neq \gamma) \\
(T_1 \to T_2)[\alpha := \sigma] &\equiv (T_1[\alpha := \sigma]) \to (T_2[\alpha := \sigma]) \\
\texttt{Bool}[\alpha := \sigma] &\equiv \texttt{Bool} \\
\texttt{Nat}[\alpha := \sigma] &\equiv \texttt{Nat}
\end{aligned}$$

One last unspecified construction is the notion of free and bound type variables as important as with terms: programs are closed terms with closed types. Unlike in 2.1.1, where we state FV as a map from terms into sets of variables, we choose in our implementation approach to define the set of possible closed types that can be built from the current type context (a sequence of type variables, more on that with typing 3.4).

```
  Inductive BV (Δ:ctxT) : typ -> Prop :=
| bv_bool :  BV Δ Bool
```

[15] Illustrating why type checking problems ($\Gamma \vdash M : \sigma?$) and type synthesis problems ($\Gamma \vdash M :?$) are usually the same. To solve $MN : \sigma$ one has to solve $N :?$ And if it gives an answer τ, solve $M : \tau \to \sigma$?

```
| bv_nat :  $BV\ \Delta$ Nat
| bv_varT : forall ($\alpha$:nat), $\alpha \in \Delta$ -> $BV\ \Delta\ \alpha$
| bv_arr :  forall ($\sigma\ \delta$:typ),($BV\ \Delta\ \sigma$)->($BV\ \Delta\ \delta$)->($BV\ \Delta$ ($\sigma \to \delta$))
| bv_all :  forall ($\alpha$:nat)($\sigma$:typ), ($BV\ \Delta :: \alpha\ \sigma$) -> ($BV\ \Delta\ \forall\alpha.\sigma$).
```

We will sometimes write the statement $BV\ \Delta\ \sigma$ as $\sigma \in BV(\Delta)$.

3.3 The evaluation

$$\frac{v \in V}{(\lambda x:\sigma.M)v \to M[x := v]}\ (1) \quad \frac{M \to M'}{(M\ N) \to (M'\ N)}\ (2) \quad \frac{v \in V \qquad M \to M'}{(v\ M) \to (v\ M')}\ (3)$$

$$\frac{\sigma \in T}{(\lambda\alpha.M)[\sigma] \to M[\alpha := \sigma]}\ (4) \qquad \frac{M \to M'}{(M\ [T]) \to (M'\ [T])}\ (5)$$

$$\frac{}{if\ true\ then\ M\ else\ N \to M}\ (6) \quad \frac{}{if\ false\ then\ M\ else\ N \to N}\ (7)$$

$$\frac{M \to M'}{if\ M\ then\ N\ else\ N' \to if\ M'\ then\ N\ else\ N'}\ (8)$$

$$\frac{M \to M'}{S\ M \to SM'}\ (9) \qquad \frac{M \to M'}{P\ M \to PM'}\ (10) \qquad \frac{}{P\ 0 \to 0}\ (11)$$

$$\frac{v \in \text{Natural values}}{P\ (S\ v) \to v}\ (12) \quad \frac{M \to M'}{Z\ M \to ZM'}\ (13) \quad \frac{}{Z\ 0 \to true}\ (14) \quad \frac{v \in \text{Natural values}}{Z\ (S\ v) \to false}\ (15)$$

Since the core of evaluation consists of rewriting β-redexes into their contracts, there are as many evaluation strategies as argument consumption policies by functions.

- Call by value (CBV): each argument is evaluated before substitution in the function's body. It is the strategy followed in our implementation and most programming languages. The purpose is to evaluate the argument once and for all to avoid re-evaluating the same term in case it appears more than one time in the body (which is reasonably probable while it can happen too that CBV processes unnecessary computing).Two simple examples with

$$M := (\lambda x : \mathtt{Nat}.if\ (Zx)\ then\ (Sx)\ else\ (Px))(P0)$$

$$N := ((\lambda x : \mathtt{Nat}.\lambda y : \mathtt{Nat}.if\ (Zx)\ then\ x\ else\ y)0)(P(S(P(S0))))$$

$$\begin{aligned}
M &\to (\lambda x : \texttt{Nat}.if\ (Zx)\ then\ (Sx)\ else\ (Px))0 \\
&\to if\ (Z0)\ then\ (S0)\ else\ (P0) \\
&\to if\ true\ then\ (S0)\ else\ (P0) \\
&\to (S0)
\end{aligned}$$

$$\begin{aligned}
N &\to (\lambda y : \texttt{Nat}.if\ (Z0)\ then\ 0\ else\ y)(P(S(P(S0)))) \\
&\to (\lambda y : \texttt{Nat}.if\ (Z0)\ then\ 0\ else\ y)(P(S0)) \\
&\to (\lambda y : \texttt{Nat}.if\ (Z0)\ then\ 0\ else\ y)0 \\
&\to (Z0)\ then\ 0\ else\ 0 \\
&\to true\ then\ 0\ else\ 0 \\
&\to 0
\end{aligned}$$

- Call by name (CBN): the arguments are directly substituted into the body and are evaluated only if necessary. [16] On the contrary, this strategy avoids useless computations but not redundant ones (like if the body is of the form $x + x$, x would have to be evaluated twice before the addition). The two previous examples show CBN is a worse choice in the first case but a better choice in the second.

$$\begin{aligned}
M &\to_\beta if\ (Z(P0))\ then\ (S(P0))\ else(P(P0)) \\
&\to_\beta if\ (Z0)\ then\ (S(P0))\ else(P(P0)) \\
&\to_\beta if\ true\ then\ (S(P0))\ else(P(P0)) \\
&\to_\beta (S(P0)) \\
&\to_\beta (S0)
\end{aligned}$$

$$\begin{aligned}
N &\to_\beta (\lambda y : \texttt{Nat}.if\ (Z0)\ then\ 0\ elsey)(P(S(P(S0)))) \\
&\to_\beta (Z0)\ then\ 0\ else(P(S(P(S0)))) \\
&\to_\beta true\ then\ 0\ else(P(S(P(S0)))) \\
&\to_\beta 0
\end{aligned}$$

3.4 Typing

Some implementation precisions: we have separated the 'informal' context we have used since the beginning into two different types of contexts: the first (usually Γ of type `ctxV` in implementation) for bindings of term variables to their types and the second (usually Δ of type `ctxT` in implementation) for bound type variables. The appending operator on contexts is " :: ".

$$\frac{x : \sigma \in \Gamma \qquad \sigma \in BV(\Delta)}{\Gamma, \Delta \vdash x : \sigma}\ (\text{Var}) \qquad \frac{\Gamma :: (x : \sigma), \Delta \vdash M : \delta \qquad \sigma \in BV(\Delta)}{\Gamma, \Delta \vdash \lambda x : \sigma.M : (\sigma \to \delta)}\ (\to\text{I})$$

[16]Call by need or Lazy evaluation: a variance of CBN which consists of storing the term's evaluation so that it could be used if the same term needs to be re-evaluated.

$$\frac{\Gamma, \Delta \vdash M : (\sigma \to \delta) \qquad \Gamma, \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash (MN) : \delta} (\to\text{E})$$

$$\frac{\Gamma, (\Delta :: \alpha) \vdash M : \sigma}{\Gamma, \Delta \vdash \lambda\alpha.M : \forall\alpha.\sigma} (\forall\text{I}) \qquad \frac{\Gamma, \Delta \vdash M : \forall\alpha.\sigma \qquad \delta \in BV(\Delta)}{\Gamma, \Delta \vdash M[\delta] : \sigma[\alpha := \delta]} (\forall\text{E})$$

$$\frac{}{\Gamma, \Delta \vdash \mathit{true} : \texttt{Bool}} (\text{TrueBool}) \qquad \frac{}{\Gamma, \Delta \vdash \mathit{false} : \texttt{Bool}} (\text{FalseBool})$$

$$\frac{\Gamma, \Delta \vdash b : \texttt{Bool} \qquad \Gamma, \Delta \vdash M : \sigma \qquad \Gamma, \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash \text{if } b \text{ then } M \text{ else } N : \sigma} (\text{IfBool})$$

$$\frac{}{\Gamma, \Delta \vdash 0 : \texttt{Nat}} (\text{ZeroNat}) \qquad \frac{\Gamma, \Delta \vdash M : \texttt{Nat}}{\Gamma, \Delta \vdash (SM) : \texttt{Nat}} (\text{SNat})$$

$$\frac{\Gamma, \Delta \vdash M : \texttt{Nat}}{\Gamma, \Delta \vdash (PM) : \texttt{Nat}} (\text{PNat}) \qquad \frac{\Gamma, \Delta \vdash M : \texttt{Nat}}{\Gamma, \Delta \vdash (ZM) : \texttt{Bool}} (\text{ZNat})$$

3.5 Type Safety

Here begins the process of proving a crucial property on type systems for $\lambda 2$ type safety. Type safety is the property of a type system to be free from a wide range of errors : type errors, while restricting in evaluating well-typed terms only [17]: *Well-typed terms do not go wrong.* This explains the propensity in almost every programming language to statically or dynamically type-check the most possible part of their program [18]. In fact, there is a phase in compilers called type-erasure in charge of the equivalent of translating well-type-checked terms into those of the untyped λ-calculus for faster computation. Such a function holds on the fact that any evaluation under the typed relation is possible in the corresponding untyped system. This illustrates why types that are not involved in any proper computation are essential to reliable programming language design.

But the road to type safety traverses two fortified castles guarded by a pair of incorruptible knights: progress and preservation!

3.5.1 Progress

Progress is about well-typed terms never being stuck (our conception of error: a term that is not a value but that cannot be reduced to it anymore): they are either value or can be evaluated into another term. But first, let us review the useful lemmas.

We will always consider that Γ is a term context, Δ is a type context, σ, δ, γ are types, α is type variables, x term variables, M, M', N, N', v are λ-terms.

Lemma 3.1 (Inversion of the typing relation)**.**

[17]Witch languages do not do since some untypable terms never go wrong like python's `if False: x = 3 + "l" else: x = 1 + 2`.

[18]The sooner possible at compilation

1. $\Gamma, \Delta \vdash true \ : \ \sigma \Rightarrow \sigma \equiv \texttt{Bool}$
2. $\Gamma, \Delta \vdash false \ : \ \sigma \Rightarrow \sigma \equiv \texttt{Bool}$
3. $\Gamma, \Delta \vdash 0 : \sigma \Rightarrow \sigma \equiv \texttt{Nat}$
4. $\Gamma, \Delta \vdash (SM) : \sigma \Rightarrow \Gamma, \Delta \vdash M : Nat \wedge \sigma \equiv \texttt{Nat}$
5. $\Gamma, \Delta \vdash (PM) : \sigma \Rightarrow \Gamma, \Delta \vdash M : Nat \wedge \sigma \equiv \texttt{Nat}$
6. $\Gamma, \Delta \vdash (ZM) : \sigma \Rightarrow \Gamma, \Delta \vdash M : Nat \wedge \sigma \equiv \texttt{Bool}$
7. $\Gamma, \Delta \vdash (If\ M\ then\ N\ else\ N') : \sigma \Rightarrow (\Gamma, \Delta \vdash M : \texttt{Bool}) \wedge (\Gamma, \Delta \vdash N : \sigma) \wedge (\Gamma, \Delta \vdash N' : \sigma)$
8. $\Gamma, \Delta \vdash x : \sigma \Rightarrow (x : \sigma) \in \Gamma$
9. $\Gamma, \Delta \vdash (\lambda x : \delta.M) : \sigma \Rightarrow \exists\gamma, (\Gamma :: (x : \delta), \Delta \vdash M : \gamma) \wedge (\sigma \equiv \delta \to \gamma)$
10. $\Gamma, \Delta \vdash (\lambda\alpha.M) : \sigma \Rightarrow \exists\gamma, (\Gamma, \Delta :: \alpha \vdash M : \gamma) \wedge (\sigma \equiv \forall\alpha.\gamma)$
11. $\Gamma, \Delta \vdash (MN) : \sigma \Rightarrow \exists\delta\gamma, (\Gamma, \Delta \vdash M : \delta \to \gamma) \wedge (\Gamma, \Delta \vdash N : \delta) \wedge (\sigma \equiv \gamma)$
12. $\Gamma, \Delta \vdash (M[\delta]) : \sigma \Rightarrow \exists\alpha\exists\gamma, (\Gamma, \Delta \vdash M : \forall\alpha.\gamma) \wedge (\sigma \equiv \gamma[\alpha := \delta])$

Proof. Direct consequences of inference rules for the typing relations. (In fact, the assertions could have been written as the typing inference rules but inversed: the premices as conclusion and vice-versa). Example case 4:

```
Lemma itr_succ :  forall (Γ:ctxV)(Δ :ctxT)(M:trm)(σ:typ),
(Γ,Δ ⊢ (SM):Nat) -> (Γ,Δ ⊢ M:Nat) ∧ (σ = Nat).
Proof.
intros Γ Δ M σ H;split;[inversion H | induction σ];easy.
Qed.
```

`split` will separate the conjunction we are proving into two subgoals for each side of the $\wedge$. In Coq one can write `tac;[tac_1 | ... | tac_n]` to respectively apply `tac_1`, ..., `tac_n` to the n subgoals generated by the tactic `tac`.

`inversion H` will try a proof by induction on $(\Gamma, \Delta \vdash (SM)$:`Nat`$)$ and will implicitly eliminate subgoals where the term does not match the structure of (SM) i.e. any typing inference rule except (SNat) which will yield induction hypothesis $(\Gamma, \Delta \vdash M$:Nat$)$. The same will happen with `induction` σ and result in subgoal `Nat=Nat`. □

Lemma 3.2 (Canonical forms)**.**

- if v is a value of type `Bool`, then v is either $true$ or $false$.
- if v is a value of type `Nat`, then v is a natural value.
- if v is a value of type $\sigma \to \delta$, then $v = \lambda x : \sigma.M$

- if v is a value of type $\forall\alpha.\sigma$, then $v = \lambda\alpha.M$

Proof. according to the grammar of values at 3.1 and based on the inversion lemma. The first case in Coq will look like this:

```
Lemma can_bool :  forall (Γ:ctxV)(Δ:ctxT)(M:trm),
(val M) -> (Γ,Δ ⊢ M :  Bool) -> (M = true ∨ M = false).
Proof.
intros Γ Δ M H H0.
induction H;try destruct H;inversion H0;auto.
Qed.
```

`Induction H` will launch a proof by induction on different types of values, replacing the sub-cases that are still inductive like `H : val` M by `H : value_bool` M or `H : value_nat` M. For the others like $M \equiv \lambda x.M$ it will just perform a rewriting with no remaining hypothesis `H`.

`try destruct H` will try its argument the tactic `destruct H` and will leave the goal unchanged if the tactic fails (for the hypothesis `H` not existing in some sub-goals). This result in destructing sub values like `value_bool` into *true* and $false$ and rewriting them (having `H0 :` $\Gamma, \Delta \vdash true$ `: Bool` for *true*).

The `inversion H0` tactic will try an induction on the typing relation and solve self-contradictory cases like $\Gamma, \Delta \vdash \lambda\alpha.M$ `:Bool`.

At this stage only remain subgoals $true \equiv true \ \vee \ true \equiv false$ and $false \equiv true \ \vee \ false \equiv false$ (thanks to renaming) that are solved by the tactic `auto` explained futher.

□

Theorem 3.3 (Progress). *For all well-typed terms $M : \sigma$, either M is a value or it can evaluate to another term M'*

$$\forall M, (\emptyset, \emptyset \vdash M : \sigma) \Rightarrow (M \in V) \vee \exists M' \mid M \rightarrow_\beta M'$$

Proof. Induction on the derivations of the typing relation $\emptyset, \emptyset \vdash t : \sigma$ [19] .

- The hypothesis is not respected in the (Var) case since for any x and σ, $(x : \sigma) \notin \emptyset$[20]

- Cases for values like (TrueBool), (FalseBool), (ZeroNat) or →I, ∀I are immediate. In Coq we just need to start our proof like this:

  ```
  intros M σ H.
  ...
  induction H;...;try solve [left;auto using val, value_bool, value_nat].
  ```

[19]Prove the assertion (that is the conclusion) for each inference rule of the relation assuming it is true for its premises.

[20]This is the famous *ex falso quodlibet*: $\bot \Rightarrow P$, any proposition is deducted from a contradiction

where the tactic `left` transform a subgoal of the form `A ∨ B` into simply `A`. To prove `A ∨ B`, you just need to prove `A` or to prove `B` (using `right`)[21]. `solve` try to resolve the goal with the provided tactic and leave it unchanged if it generates subgoals. Combining it with `try` helps to solve straightforward cases without changing non-trivial ones.

And finally one of the most useful tactics: `auto`. It recursively first tries to solve the goal by looking at the local context, applies `intros` otherwise, and tries to apply tactics that match the result in the increasing order of their cost. It can be provided some additional theorems to check with the `using` option or some whole databases of theorems by the `with` option. These databases can be user-defined (like our implementation's `sys_f_base`) our system-defined like `arith`, `zarith` or `datatypes`.

- For the (SNat) case $\emptyset, \emptyset \vdash (SM) : \texttt{Nat}$, the induction hypothesis tells us that M :
  - either is a value of type `Nat`, then it must be a natural value according to the 3.2 canonical forms lemma and so is SM which makes it a value.
  - or can evaluates to another term M' and so can SM to SM' according to the evaluation inference rules 9.

  The (PNat) and (ZNat) cases have similar cases except that even when the subterm M is a value 0 or (Sv), they can evaluate by inference rules 11,12 and 14,15 respectively. The second subcase corresponds to 10 and 13 .

- For the (IfBool) case $\emptyset, \emptyset \vdash if\ b\ then\ M\ else\ N : \sigma$, the induction hypothesis tells us that b :
  - either is a value: it is then $true$ or $false$ according to the canonical forms lemma so that the whole term can take a step by inference rule 6 or 7 of the evaluation rules.
  - or can evaluate to another term b'. The inference rule 8 holds and we have $t \to if\ b'\ then\ M\ else\ N$

- For the ($\to$E) case : $\emptyset, \emptyset \vdash (MN) : \sigma$, by induction, M :
  - either is a value of type $\sigma \to \delta$, then it must be an abstraction value $\lambda x : \sigma.M'$ according to 3.2 such that t is a β-redex. Depending on the other induction hypothesis, N :
    * which is either a value make that t reduces to $M'[x := N]$ by inference rule 1
    * or reduces to N' so that $t \to (MN')$ by inference rule 3

[21]introduction of $\vee$ inference rules

  - or can evaluate to another term M' and so can (MN) to $(M'N)$ according to inference rules 2.

- For the last case $(\forall E)$: $\emptyset, \emptyset \vdash M[\delta] : \sigma[\alpha := \delta]$, M by induction :
  - either is a value of type $\forall\alpha.\sigma$, then it must be an abstraction value $\lambda\alpha.M'$ according to 3.2 so that t reduces to $M'[\alpha := \delta]$ by inference rule 4.
  - or can reduce to another term M' and so can $M[\delta]$ to $M'[\delta]$ according to inference rules 5.

□

Remark.

As said before, constructive systems like Coq only accept concrete illustrations as proof of existential assertions. To prove a goal the form `exists (x:T), P x`, one must provide a term `t` the same type as `x` and built using terms in the local and global context. The tactic `exists t` then generates the subgoal `P t`. In fact `exists` is a defined notation for the inductive type `ex` owning only one constructor[22] :

```
>Locate "exists".
Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))

>Print ex.
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> exists y, P y.
```

The only way to inductively build an existential predicate `exists y, P y` is then to provide a variable `x` such that `P x` and the tactic `exists` can be seen here as `apply (ex_intro x)`.

3.5.2 Preservation

Preservation is about well-typed terms resulting in another well-typed one if they have to take a step into evaluation. We begin with preliminary definitions and lemmas.

And again $\Gamma, \Gamma_1, \Gamma_2$ are term contexts, $\Delta, \Delta_1, \Delta_2$ are type contexts, σ, δ, γ are types, α, μ type variables, x. term variables and M, M', N, N' are λ-terms.

Lemma 3.4 (Uniqueness of type)**.** *A lambda term can have at most one type in a given context.*

$$\forall\Gamma\ \forall\Delta\ \forall\sigma\delta\ \forall M,\ (\Gamma, \Delta \vdash M : \sigma) \wedge (\Gamma, \Delta \vdash M : \delta) \Rightarrow \sigma \equiv \delta$$

[22]The existential quantifier is modeled using universal quantifier

Proof. Structural induction on M [23] using corresponding inversion lemma assertion. □

Definition 3.1 ($\subseteq$)**.** *A context is included in any context that binds its variables to the same types. Those contexts are said to contain this context.*

$$\forall \Gamma_1 \Gamma_2,\ \Gamma_1 \subseteq \Gamma_2 \equiv \forall x, (x \in \Gamma_1) \Rightarrow (x \in \Gamma_2)$$

$$\forall \Delta_1 \Delta_2,\ \Delta_1 \subseteq \Delta_2 \equiv \forall \alpha, (\alpha \in \Delta_1) \Rightarrow (\alpha \in \Delta_2)$$

Remark.

- *Context extension preserves inclusion. The proof is based on the previous definition by comparing x and α respectively to the extending variable.*
- *Any context contains the empty context.*

Lemma 3.5 (Weakening)**.** $\subseteq$ *is invariant with respect to the typing and BV relation.*

$$\forall \Gamma_1 \Gamma_2\ \forall \Delta\ \forall M\ \forall \sigma, (\Gamma_1 \subseteq \Gamma_2) \Rightarrow (\Gamma_1, \Delta \vdash M : \sigma) \Rightarrow (\Gamma_2, \Delta \vdash M : \sigma)$$

$$\forall \Gamma\ \forall \Delta_1 \Delta_2\ \forall M\ \forall \sigma, (\Delta_1 \subseteq \Delta_2) \Rightarrow (\Gamma, \Delta_1 \vdash M : \sigma) \Rightarrow (\Gamma, \Delta_2 \vdash M : \sigma)$$

$$\forall \Gamma\ \forall \Delta_1 \Delta_2\ \forall \sigma, (\Delta_1 \subseteq \Delta_2) \Rightarrow (\sigma \in BV(\Delta_1)) \Rightarrow (\sigma \in BV(\Delta_2))$$

Proof. It is straightforward by induction into the said relations. Here is the third case in Coq :

```
Lemma weakening_bv :  forall (Δ Δ':ctxT)(σ:typ),
Δ ⊆ Δ' -> (BV Δ σ) -> (BV Δ' σ).
Proof.
intros Δ Δ' σ H H0.
generalize dependent Δ'.
induction H0;eauto using 3.5.2, BV.
Qed.
```

`generalize dependent` abstracts one or more terms out of hypotheses with a `forall`, especially induction hypothesis. This can help when Coq does not interpret some terms as parameters and not constants of some hypothesis.

`eauto` behaves `auto` in addition to trying resolution paths that would leave existential variables in the goal. Providing an inductive type like `BV` to the `using` option equals providing all its constructors.

The `induction H0` will launch a proof by induction on $\sigma \in BV(\Delta)$, the `apply constructuor` will probably be applied for each subcase with the right constructor inside of `eauto`. The induction hypothesis will then be applied and the resulting subgoal will be solved by 3.5.2. □

[23] Prove the assertion for each case of M assuming it is true for its subterms.

Lemma 3.6 (Substitution lemma for terms)**.** *Well-typedness is preserved by the substitution of variables by terms of the same type.*

$$\forall\Gamma\ \forall\Delta\ \forall\sigma\delta\ \forall x\ \forall tt', (\Gamma :: (x:\sigma), \Delta \vdash t : \delta) \wedge (\emptyset, \emptyset \vdash t' : \sigma) \Rightarrow (\Gamma, \Delta \vdash t[x := t'] : \delta)$$

Proof. By induction on the derivation of $\Gamma :: (x:\sigma), \Delta \vdash t : \delta$

- (Var), $\Gamma :: (x:\sigma), \Delta \vdash z : \delta$.
  - If $x \equiv z$ then by the uniqueness of type at 3.4 $\sigma \equiv \delta$. $z[x := t'] \equiv t'$ and we have $\Gamma, \Delta \vdash t' : \sigma$ because of $\emptyset, \emptyset \vdash t' : \sigma$ in the assumption and 3.5.2.
  - Or else $x \not\equiv z$ then $z[x := t'] \equiv z$ and $\Gamma :: (x:\sigma), \Delta \vdash z : \delta$.We can deduce $\Gamma, \Delta \vdash z : \delta$ because since $z : \delta \notin (x:\sigma)$ we must have $z : \delta \in \Gamma$.

  Remark.

  While Coq is a constructive system that rejects the excluded middle tierce as we have discussed at 2.4.1, there are still some situations implying similar assertions. It is the case of the `eqb` boolean equality on naturals (the datatype of our type variables) and strings (the datatype of our term variables). Such boolean operation denoted `x=?y` used with the equality proposition `x=y` builds a particular datatype beside the following `eqb_spec` lemma from the Coq standard library [7] :
  `Lemma eqb_spec s1 s2 : Bool.reflect (s1 = s2) (s1 =? s2).`
  and here is the inductive definition of `reflect` :

  ```
  Inductive reflect (P : Prop) : bool -> Set :=
    | ReflectT : P -> reflect P true
    | ReflectF : ~ P -> reflect P false.
  ```

  The use of this lemma is to help us build proofs of the form either $x \equiv z$ or $x \not\equiv z$ as in the previous point without breaking that constructive property of Coq.

  The tactic `destruct (eqb_spec x z).` will generate two subgoals with respective hypotheses `x=z` and `x<>z` that can be combined with `eqb_eq` and `eqb_neq` respectively:

  ```
  Lemma eqb_eq x y : (x =? y) = true <-> x = y.
  Lemma eqb_neq x y : (x =? y) = false <-> x <> y
  ```

- ($\to$I), $\Gamma :: (x:\sigma), \Delta \vdash \lambda z : \mu.M : \mu \to \tau$.
  - If $x \equiv z$, then $t[x := t'] \equiv \lambda z : \mu.M$, $x \notin FV(t)$ and t is of type δ.

  - Or else $t[x := t'] \equiv \lambda z : \mu.M[x := t']$. Using the induction hypothesis on $\Gamma :: (x : \sigma) :: (z : \mu), \Delta \vdash M : \tau$ given by the inversion lemma at 3.5.1 which becomes $\Gamma :: (z : \mu) :: (x : \sigma), \Delta \vdash M : \tau$ by weakening (guaranteed by variable convention) , plus $\Gamma :: (z : \mu), \Delta \vdash t' : \sigma$ given by the wekeaning lemma 3.5, the induction hypothesis told us that $\Gamma :: (z : \mu), \Delta \vdash M[x := t'] : \delta$. So we can deduce $\Gamma, \Delta \vdash \lambda z : \mu.M[x := t'] : \delta$ from rule ($\rightarrow$I).

- ($\forall$I), $\Gamma :: (x : \sigma), \Delta \vdash \lambda\alpha.M : \forall\alpha.\mu$.
  The inversion lemma gives $\Gamma :: (x : \sigma), \Delta :: \alpha \vdash M : \mu$. From the induction hypothesis we have $\Gamma, \Delta :: \alpha \vdash M[x := t'] : \mu$. But the previous is the premise by ($\forall$I) of $\Gamma, \Delta \vdash \lambda\alpha.M[x := t'] : \forall\alpha.\mu$. Finally, let us rewrite $t[x := t'] \equiv \lambda\alpha.M[x := t']$ by 3.2.

- (TrueBool), (FalseBool), and (ZeroNat) are straightforward by 3.2 since they remain unchanged to substitution and are of their types `Bool` and `Nat` (wich can be deducted via the inversion lemma) under any context.

- The (SNat), (PNat), (ZNat), (IfBool) and even ($\forall$E), ($\rightarrow$E) are barely less simpler: they are solved by the premise given by the 3.5.1 inversion lemma, paired with $\emptyset, \emptyset \vdash t' : \sigma$ and applied to the induction hypotheses plus some 3.2 substitution rewriting. Example with $\forall$E : using

$$t[x := t'] \equiv M[x := t'][\gamma]$$

  we have $(\forall\text{E})\ \dfrac{\Gamma, \Delta \vdash M[x := t'] : \forall\alpha.\mu \qquad \alpha \notin \Delta}{\Gamma, \Delta \vdash M[x := t'][\gamma] : \mu[\alpha := \gamma]}$

□

Remark. *By our variable convention, if $\alpha \not\equiv \mu$, $\mu \notin FV(\delta)$, the successive respective substitution of μ, α by τ, δ, $(T[\mu := \tau])[\alpha := \delta]$ can be rewritten as $(T[\alpha := \delta])[\mu := \tau[\alpha := \delta]]$.*

The convention variable which is the basis of a lot of assertions on the implementation has been postulated via an alias of the `Axiom` command: `Hypothesis`. What is interesting is that identifiers defined like that behave like local parameters regarding their section i.e. they are only accessible inside their section (like our `SystemF` module) and become undefined outside of it with every object depending on them being parametrized by a variable of their types. The variable is said to have been discharged. Here is an illustration of that phenomenon:

```
> Section sec.
> Variable a:nat.
> Definition f b:nat := a+b.
> Print f.
f = fun b : nat => a + b
     : nat -> nat
...
```

```
> End sec.
> Print f.
f = fun a b : nat => a + b
     : nat -> nat -> nat
```

The variable convention will be abstracted out of any theorem depending on it (like preservation) outside of the module the same way as the variable `a` with `f`.

Lemma 3.7 (Substitution lemma for types)**.**

$$\forall\Gamma\ \forall\Delta\ \forall\sigma\delta\ \forall\alpha\ \forall M, (\Gamma, \Delta :: \alpha \vdash M : \sigma) \Rightarrow (\Gamma[\alpha := \delta], \Delta \vdash M[\alpha := \delta] : \sigma[\alpha := \delta])$$

With $\delta \in BV(\Delta)$ and where inductively :

$$(\Gamma :: (x : \sigma))[\alpha := \delta] \equiv \Gamma[\alpha := \delta] :: (x : \sigma[\alpha := \delta])$$
$$\emptyset[\alpha := \delta] \equiv \emptyset$$

Proof. By induction on the derivation of $\Gamma, \Delta :: \alpha \vdash t : \sigma$. Also, let us denote $A[b := c]$ by A_{bc} for more readability.

- (Var) : $\Gamma, \Delta :: \alpha \vdash x : \sigma$. We have $x : \sigma \in \Gamma$ thus, $x : \sigma_{\alpha\delta} \in \Gamma_{\alpha\delta}$ by definition. Conclusion : $\Gamma_{\alpha\delta}, \Delta \vdash x_{\alpha\delta} : \sigma_{\alpha\delta}$. $(x[\alpha := \delta] \equiv x)$

- ($\to$I) : $\Gamma, \Delta :: \alpha \vdash \lambda x : \gamma.M : \gamma \to \tau$. By the inversion lemma, we get $\Gamma :: (x : \gamma), \Delta :: \alpha \vdash M : \tau$. Applying the induction hypothesis helps obtaining : $(\to\text{I})\ \dfrac{(\Gamma :: (x : \gamma))_{\alpha\delta}, \Delta \vdash M_{\alpha\delta} : \tau_{\alpha\delta}}{\Gamma_{\alpha\delta}, \Delta \vdash \lambda x : \gamma_{\alpha\delta}.M_{\alpha\delta} : (\gamma_{\alpha\delta} \to \tau_{\alpha\delta})}$. All that remains is to recall that $(\lambda x : \gamma.M)[\alpha := \delta] \equiv \lambda x : \sigma[\alpha := \delta].M[\alpha := \delta]$. The procedure is similar for ($\to$E).

- ($\forall$I) : $\Gamma, \Delta :: \alpha \vdash \lambda\mu.M : \forall\mu.\gamma$. By inversion $\Gamma, \Delta :: \alpha :: \mu \vdash M : \gamma$ and $\mu \notin (\Delta :: \alpha)$ by variable convention.

  - if $\alpha \equiv \mu$ we have $\mu \in (\Delta :: \alpha)$ that would implies a contradiction in the hypothesis.
  - Or else $(\Delta :: \alpha :: \mu) \subseteq (\Delta :: \mu :: \alpha)$, the weakening lemma for types 3.5 validate $\Gamma, \Delta :: \mu :: \alpha \vdash M : \gamma$. After applying the induction hypothesis we get: $(\forall\text{I})\ \dfrac{\Gamma_{\alpha\delta}, \Delta :: \mu \vdash M_{\alpha\delta} : \gamma_{\alpha\delta}}{\Gamma_{\alpha\delta}, \Delta \vdash \lambda\mu.M_{\alpha\delta} : \forall\mu.\gamma_{\alpha\delta}}$ completed by the fact that if $\alpha \not\equiv \mu$ $(\lambda\mu.M)[\alpha := \delta] \equiv \lambda\mu.M[\alpha := \delta]$.

- ($\forall$E) : $\Gamma, \Delta :: \alpha \vdash M[\tau] : \gamma[\mu := \tau]$. By inversion M is of type $\forall\mu.\gamma$ and $\mu \notin (\Delta :: \alpha)$ still by variable convention meaning $\mu \not\equiv \alpha$. By induction : $M_{\alpha\delta}$ is under $\Gamma_{\alpha\delta}$ and Δ of type $\forall\mu.\gamma_{\alpha\delta}$. Resulting in : $(\forall\text{E})\ \dfrac{\Gamma_{\alpha\delta}, \Delta \vdash M_{\alpha\delta} : \forall\mu.\gamma_{\alpha\delta}}{\Gamma_{\alpha\delta}, \Delta \vdash M_{\alpha\delta}[\tau_{\alpha\delta}] : \gamma_{\alpha\delta}[\mu := \tau_{\alpha\delta}]}$. Recall of the 3.5.2 remark that $(\gamma[\mu := \tau])_{\alpha\delta} \equiv \gamma_{\alpha\delta}[\mu := \tau_{\alpha\delta}]$ since $\mu \not\equiv \alpha$ and $\mu \notin FV(\gamma)$ by the variable convention.

- (TrueBool),(FalseBool), and (ZeroNat) are straightforward cases since both these constants and their base types `Bool`,`Nat` remain unchanged after substitution. This can be achieved in Coq with

```
inversion H0; constructor.
```

  Where `H0` is the statement we are doing the induction on: this has the same effect as using 3.5.1 inversion lemma and will rewrite σ as `Bool` or `Nat`. `constructor` introduces hypotheses (`intros`), normalize the goal (`hnf`) and if it is an inductive type do `apply` with the appropriate constructor (e.g. (TrueBool) equivalent constructor) which will solve the goal.

- (SNat) : $\Gamma, \Delta :: \alpha \vdash (SM) :$ `Nat`.By inversion, induction, and substitution rewriting:

```
inversion H0; pose proof (IHM _ _ _ H H2);constructor;easy.
```

  `pose proof` add a new hypothesis to the local context whose type is the one of the provided instanciation of the `IHM` induction hypothesis. The underscores delegate to Coq the task of guessing the first dependent parameters (σ, Γ, Δ) from the last provided ones $(\delta \in BV(\Delta), \Gamma, \Delta :: \alpha \vdash t : \sigma)$.

The two following tactic chains can be combined and directly chained to the tactic that launches the induction proof:

```
Proof.
  ...
  induction M;intros;try solve
  [inversion H0;try pose proof (IHM _ _ _ H H2);constructor;easy].
```

This reduced the number of generated subgoals from 12 to 6. Also, we use induction on `M` instead of `H0` because of `generalize dependent` on σ, Γ, Δ which abstracts them out of the subgoal and remove hypothesis `H0` before it get reintroduced by `intros`. □

Theorem 3.8 (Preservation)**.** *If M is a well-typed term of type σ and it evaluates to M' then M' is a well-typed term of the same type.*

$$\forall\sigma\ \forall M, (\emptyset, \emptyset \vdash M : \sigma) \wedge (\exists M' \mid M \rightarrow_\beta M') \Rightarrow (\emptyset, \emptyset \vdash M' : \sigma)$$

Proof. By induction of the derivation of $\emptyset, \emptyset \vdash t : \sigma$

- (Var) : The hypotheses are not met for the same reasons as progress.

- ($\rightarrow$I), ($\forall$I) : The hypotheses are not met since abstraction values cannot take a step into evaluation.

- ($\rightarrow$E) : $\emptyset, \emptyset \vdash (MN) : \delta$. The inversion lemma rule 11 yields that there exist a type γ such that M is of type $\gamma \rightarrow \delta$ and N is of type γ.In other hands the statement $t \rightarrow t'$ would implies inference rules $1, 2$ or 3 :

- $(1) : (\lambda x : \gamma.M')v \to M'[x := v]$: The inversion lemma 3.5.1 9 gives us $(x : \gamma), \emptyset \vdash M' : \delta$ and $\emptyset, \emptyset \vdash v : \gamma$. We can now apply our 3.6 substitution lemma to obtain that $\emptyset, \emptyset \vdash M'[x := v] : \delta$.
- $(2) : (MN) \to (M'N)$ with $M \to M'$. The induction hypothesis applied to M results in $\emptyset, \emptyset \vdash M' : \gamma \to \delta$. Combined with the prior on the type of N, we can deduce by the same inference rule ($\to$E) that $(M'N)$ is of type δ
- $(3) : (vN) \to (vN')$ with $N \to N'$. As by the previous point, having N' of type γ by the induction hypothesis, we get (vN') of type δ.

* ($\forall$E) : $\emptyset, \emptyset \vdash M[\gamma] : \delta[\alpha := \gamma]$. M is of type $\forall\alpha.\delta$ by rule 12 of the inversion lemma. $t \to t'$ would imply the following evaluation rules :
  - $(4) : (\lambda\alpha.M')[\gamma] \to M'[\alpha := \gamma]$. The inversion lemma case 10 giving $\emptyset, \alpha \vdash M' : \delta$ applied to the substitution lemma for types 3.7 gives $\emptyset, \emptyset \vdash M'[\alpha := \gamma] : \delta[\alpha := \gamma]$.
  - $(5) : M[\gamma] \to M'[\gamma]$ with $M \to M'$. We have $\emptyset, \emptyset \vdash M'[\gamma] : \delta$ the same way as previous cases (2) and (3) : induction hypothesis on the premise plus ($\forall$E) inference rule.
* (TrueBool), (FalseBool) and (ZeroNat) cases are straightforward because they are values and do not evaluate the same as ($\to$I) or ($\forall$I).
* (SNat) : $\emptyset, \emptyset \vdash (SM) : \texttt{Nat}$. M is of type `Nat` by the inversion lemma and we only have case 9 implied into evaluation (if M was a value it would not be involved in evaluation and will be concerned by the previous point). $(SM) \to (SM')$ with $M \to M'$: M' of type `Nat` by induction hypthesis $\emptyset, \emptyset \vdash (SM') : \texttt{Nat}$ by (SNat). Similar arguments can justify (PNat) and (ZNat) cases.
* (IfBool) : $\emptyset, \emptyset \vdash (if\ M\ then\ N\ else\ N') : \delta$. By inversion lemma, M is of type `Bool`, N, N' of type δ. The cases :
  - (6) (and analogous (7) proved by replacing $true$ by $false$) : $t \to N$ and N is of type δ.
  - $(8) : t \to (if\ M'\ then\ N\ else\ N')$ with $M \to M'$. The induction hypothesis applied to the premise states that M' is also of type `Bool`. Thus t' is of type δ by (IfBool).

□

4 Conclusion

The completion of this bachelor project marks an achievement in demonstrating a Coq-based proof for the safety of the System F typing system. This exploration and implementation have enriched our comprehension of the core principles of type theory in programming. The project underscores the strength of the λ-calculus and the Curry-Howard isomorphism, drawing attention to the correspondence between type systems and logical systems. This has been useful in simplifying simpler cases of practical proofmaking by automation but also emphasizes the critical role of rigorous type-safety and formal methods in ensuring the security of programs.

References

[1] H. P. Barendregt. “Lambda calculi with types”. In: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures.* USA: Oxford University Press, Inc., 1993, pp. 117–309. ISBN: 0198537611.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5. URL: https://doi.org/10.1007/978-3-662-07964-5.

[3] Wène Olivier Kouarfate. *Type safety for system F with Coq.* https://gitlab.unige.ch/Wene.Kouarfate/type-safety-for-system-f-with-coq. 2024.

[4] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[5] John C. Reynolds. “Towards a theory of type structure”. In: *Symposium on Programming.* 1974. URL: https://api.semanticscholar.org/CorpusID:30450751.

[6] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* Accessed: 2024-06-02. Inria. France, 2024. URL: https://coq.inria.fr/documentation.

[7] *The Coq Standard Library.* Distributed with the Coq system. URL: https://coq.inria.fr/doc/V8.18.0/stdlib/.