Implementing $\lambda\underline{\omega}$ in Coq

Parnian Naderi

July 23, 2023

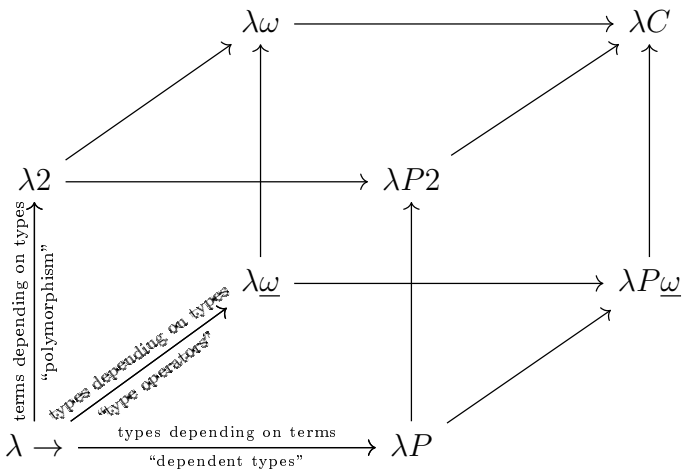
Abstract

In this report, we will survey a system of λ-Cube, called $\lambda\underline{\omega}$, which enriches *simply typed λ-calculus* with the ability to define *types which depend on types*. We will first define $\lambda\underline{\omega}$ in the style of [**?**], prove some of its properties, including the *free-variable Lemma*, and then implement our definitions and results in the proof .assistant *Coq*.

1 What is $\lambda\underline{\omega}$

λ-Cube is a collection of formal systems built atop of simply typed λ-calculus, or $\lambda\rightarrow$. Three dimensions of the "*cube*" represent three ways that we can augument $\lambda\rightarrow$ with the ability to have dependency between terms and types. So each system on a corner of the cube is a generalization of $\lambda\rightarrow$ with dependency, namely *terms dependending on terms*, which is $\lambda\rightarrow$ itself, *terms dependending on types*, which is $\lambda 2$, *types dependending on terms*, which is λP, and *types dependending on types*, which gives us $\lambda\underline{\omega}$, the system which we are going to investigate.

Figure 1: The λ-cube



$\lambda\underline{\omega}$ was first investigated by J.-Y. Girard in his PhD thesis [?], where he studied the class of provably total funcions in higher-order arithmetic, which turned to be the functions definable in an extension of $\lambda\underline{\omega}$ with $\lambda 2$.

In addition to the proper types of $\lambda \rightarrow$, there is another construction in system $\lambda\underline{\omega}$ called *type constructor*, which is a generalized notion of type that can depend on other types. A proper type is a concrete type which can have terms, similar to the types in $\lambda \rightarrow$. A type constructor is a construction which can take one or more types or type contstructors as arguments, and becomes a proper type when applied to the right number of arguments. This distinction allows us to introduce dependency between types. So we would have generic types, for example the type constructor `list` which can be applied to a proprt type `nat` to deliver the type `list nat` of all finite sequneces of natural numbers.

Any proper type or type contstructor have a super-type, which is called its *kind*. A kind can either be the kind of proper types, which is denoted by $*$, or that of a type constructor, which is an inductive construction of the binary connective $\rightarrow$ over $\{*\}$. So, for example, types like σ and $\sigma \rightarrow \tau$ are of the kind $*$, and a type constructor which takes two types is of the kimd $* \rightarrow * \rightarrow *$.

Now that we have different super-types, which need a formation rule in their own right, we also need a super-super-type to contain the previous level of kinds, which is denoted by $\Box$.

2 Definitions

2.1 Language of $\lambda\underline{\omega}$

The language of $\lambda\underline{\omega}$ has four level. The last level contains only $\Box$, which contains the level of kinds.

Kind: Kinds are the super-type of all types. $*$ denotes the kind of proper types, and the kinds for type contstructors are denoted by a combination of $*$ and $\rightarrow$..

$$\mathbb{K} ::= * \mid \mathbb{K} \rightarrow \mathbb{K}$$

Type: Types are also similar to the types in $\lambda \rightarrow$, plus two more operators: type variables, closed under operations of function type, type application and type abstraction.

$$\mathbb{T}_{\underline{\omega}} ::= \mathbb{V} \mid \mathbb{T}_{\underline{\omega}} \rightarrow \mathbb{T}_{\underline{\omega}} \mid \mathbb{T}_{\underline{\omega}}\mathbb{T}_{\underline{\omega}} \mid \lambda \mathbb{V} : \mathbb{K}.\mathbb{T}_{\underline{\omega}}$$

where $\mathbb{V}$ is chosen from a countable set of type variables and $\mathbb{K}$ is a kind. The set of free type variables of a type σ is denoted by $FV(\sigma)$ and contains all type variables occured in σ, except those that are bound by a λ.

Term: Terms are exactly the same as terms in $\lambda \rightarrow$: closure of term variables by operations of term application and term abstraction.

$$\Lambda_{\underline{\omega}} ::= V \mid \Lambda_{\underline{\omega}}\Lambda_{\underline{\omega}} \mid \lambda V : \mathbb{T}_{\underline{\omega}}.\Lambda_{\underline{\omega}}$$

where $\mathbb{V}$ is chosen from a countable set of term variables and $\mathbb{T}_{\underline{\omega}}$ is a type. We also denote the set of all free term variables of a term M by $FV(M)$, which contains all variables occured in M which are not bound by a λ.

2.2 Typability

In order to determine what terms are typable in $\lambda\underline{\omega}$, we need the notions of typing judgements, which is a relation between a term, a type, and arbitrary declarations of term and type variables. Hence we will judge that term to be of that specific type, called a *typing statement*, in presence of a sequnece of term or type variables with predetermined types or kinds, called a *context*.

A typing statement gathers the four levels of the language to form a hierarchy.

Statement: A statement is any pair of the form

$$M : \sigma \quad \text{or} \quad \sigma : K \quad \text{or} \quad K : \Box$$

where M is a term, σ is a type and K is a kind. Or more formally

$$\texttt{stat} ::= \Lambda_{\underline{\omega}} : \mathbb{T}_{\underline{\omega}} \mid \mathbb{T}_{\underline{\omega}} : \mathbb{K} \mid \mathbb{K} : \Box$$

Declaration: A declaration is a statement of the first two forms, where it's left part is a term variable or a type variable.

$$\texttt{decl} ::= V : \mathbb{T}_{\underline{\omega}} \mid \mathbb{V} : \mathbb{K}$$

Context: A finite sequnece of declarations is called a context. So a context is either empty, or is made up of a declaration joined to another context.

$$\mathbb{C} ::= [] \mid \texttt{decl} :: \mathbb{C}$$

We denote the set of term or type variables in a context Γ by $dom(\Gamma)$.

Judgement: A pair of a context Γ and a statement $A : B$ is called a judgement, and is denoted by

$$\Gamma \vdash A : B$$

Typability in $\lambda\underline{\omega}$ is defined inductively, using rules of inference, as is shown in the next table, where $s \in \{*, \Box\}$. Notice that all rules in this table, except for rule *(sort)*, have two versions for terms and types, by our choice of s.

Figure 2: Typability rules for $\lambda\underline{\omega}$

(sort)	$\vdash * : \Box$
(var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ if $x \notin \Gamma$
(weak)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$ if $x \notin \Gamma$
(form)	$\dfrac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}$
(appl)	$\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$
(abst)	$\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}$
(conv)	$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ if $B =_\beta B'$

A typing judgement $\Gamma \vdash A : B$ is *valid* in $\lambda\underline{\omega}$ if a tree with root $\Gamma \vdash A : B$ can be constructed using these rules.

3 Properties

Now we are ready to state and prove a result about the system $\lambda\underline{\omega}$, namely *free variables Lemma*, as is stated in [**?**]. Free variables Lemma states that a typing statement can be validated in the presence of no more variable than its own free variables in the context.

Free Variables Lemma: $\Gamma \vdash M : \sigma$ *implies* $FV(M) \subseteq dom(\Gamma)$.

Proof. The proof uses induction on the height of a tree that validates $\Gamma \vdash M : \sigma$. Observe that the case for rule *(sort)* is ruled out, since M can not be $*$.

(var) Trivial.

(weak) By induction hypothesis.

(form) Infeasible, since this rule can only judge statements for types and kinds.

(appl) Let $M = PQ$. By the definition of free variables, we know that $FV(M) = FV(P) \cup FV(Q)$. Induction hypothesis states that both $FV(P)$ and $FV(Q)$ are included in $dom(\Gamma)$, which suffices to conclude that also $FV(M) \subseteq dom(\Gamma)$.

(abst) Let $M = \lambda x : \sigma.P$. We know that $FV(M) = FV(P) - \{x\}$ by the definition of free variables. On the other hand, induction hypothesis states that $FV(P) \subseteq dom(\Gamma) \cup \{x\}$. Thus $FV(M) \subseteq dom(\Gamma)$.

conv By induction hypothesis.

4 Implementation

In this section, we are going to implement the $\lambda\underline{\omega}$ system in Coq. To reach this goal, we have to implement every element, from atomic to complex. The smallest element of the system is an atomic term and type - which are included in infinite countable sets. So we will be needing Nat - which we import below.

```
Import Nat.
```

To define context, we must know its structure first. List is an ideal one, especially that it helps us to keep in line with one of the most important features of the system - the importance of order in a context. This order also helps us with induction on the context, Hence the import of list and list notations.

```
Require Import List.
Import List.ListNotations.
```

These two folks are to help us keep in line with the order-sensitivity of declarations in lambda weak omega.

```
Definition termorder := nat.
Definition typeorder := nat.
```

The highest level of assignment is assigning a kind to a type, and kind is based on the definition of chapter 4 in Nederpelt's book. kinds are either star, or kind -> kind. Star itself is a kind, and kind -> kind is also a kind (so we must receive two kinds and return one), hence the definition below.

```
Inductive kind :=
| star : kind
| karrow : kind → kind → kind
.
```

Remark: kinds, types and terms are defined as inductives, since we use induction on their structure to prove theorems.

Just some notations for neat coding. The levels are auxiliary and merely to satisfy coq!

```
Notation "*" := star.
```

```
Infix "-»" := karrow (at level 100).
```

Let's define our types. Types, based on the definitions in the previous sections, are either type variables, type -> type type applied on type, and abstraction of type over a type. A type variable is only concerned with the order of declaration, hence it should only receive an order. Arrows and applications need two types and return a type.
The type abstraction case is a bit tricky: the general from is λtypevariable:kind.type. Thus we need a typeorder (not a type, because we do not abstract over the other types like abstraction), a kind that helps with declaring the type variable, and a type. It must return a type, hence the definition below.

```
Inductive type :=
| tyvar : typeorder → type
| tyarrow : type → type → type
| tyappl : type → type → type
| tyabst : typeorder → kind → type → type
.
```

An infix notation (_"infix notation" _) for neat coding. The level is quite auxiliary, to satisfy Coq.

```
Infix "->*" := tyarrow (at level 95).
```

Now that we are done with types, let's move on to terms. Terms are either variables, application or abstraction (just like simply typed lambda calculus). Variables are again concerned with their declaration order, thus they need a termorder to become a term. application needs two terms, and abstraction needs a termorder (so that the binded term becomes exclusively aa variable and not an application or abstraction), a type for the binding variable, and a term. It returns a term.

```
Inductive term :=
| tvar : termorder → term
| tappl : term → term → term
| tabst : termorder → type → term → term
.
```

The following fixpoint is going to check whether a variable belongs to the free variables of a term. Since the definition of free variable is recursive, "Fixpoint" helps us with making the definition recursive. It receives a termorder and a term and returns a boolean. the term is either a single variable, application of two terms, or an abstraction. For the first case, it suffices to check the equality of the variables. For the second case, x might either be in the applicant or the term the applicant is applied on. For the final case, if the variable is equal to the binding variables of the abstraction, it is not a free variable. Else, it might be free in the term that an abstraction is done over it.

```
Fixpoint freevar (x : termorder) (r: term) : bool :=
```

```
match r with
| tvar y ⇒ eqb x y
| tappl t1 t2 ⇒ (freevar x t1) || (freevar x t2)
| tabst v _ t1 ⇒ if eqb v x then false else (freevar x t1)
end.
```

A statement is implemented according to the definition: either we declare that a term is of a certain type, or a type is of a certain kind, or a kind is of the box sort. So we either receive a term and a type and get a statement, or a type and kind, or a kind (because there is nothing above *sort*
)

```
Inductive statement :=
| term_type : term → type → statement
| type_kind : type → kind → statement
| kind_box : kind → statement
.
```

Declaration involves variables, and we only have two types of variables, term and type.

```
Inductive declaration :=
| dectm : termorder → type → declaration
| decty : typeorder → kind → declaration
.
```

Contexts are ordered lists of declarations, and the list that coq defines has an order.

```
Definition context : Type := list declaration.
```

Another notation for neat coding.

```
Reserved Notation "x ⊢ y" (at level 110).
```

Infix notations, for neat coding

```
Infix ":#" := dectm (at level 105).
Infix ":!" := decty (at level 105).
```

This definition, and the coercion afterwards, is also for neat coding: We want to tell coq to see termorders as variables. Coercion makes it possible to use it in inductive props and theorems, without the need to feeding input.

```
Definition atomic_term_as_term (tmo : termorder) : term := tvar tmo.
Coercion atat (tmo : termorder) : term := atomic_term_as_term tmo.
```

This definition helps us to tell coq that when we are trying to make a judgement, we do not see much difference between declaration and statement on paper. But coq sees these different since we have defined them separately. Therefore it is desirable to define this and

then use Coercion to ensure the possibility of usage without the need to feed a parameter.

```
Definition dec2stat (d : declaration) : statement :=
    match d with
    | dectm tmo t ⇒ term_type tmo t
    | decty to k ⇒ type_kind (tyvar to) k
    end.
Coercion dec_as_stat (d :declaration) : statement := dec2stat d.
```

Now we are going to have some recursive definitions to help us with a few things:

- Checking whether a term variable of a type exists in our context
- Checking whether a type variable of a kind exists in our context
- Replacing bounded variables of a term with another variable (while satisfying conditions)

This one helps with us check the existence of term variables. We only need to check the term variable declarations and pass over the type variable declarations. The empty context contains no variables.

```
Fixpoint check_term (v : termorder) (con : context) : bool :=
    match con with
    | nil ⇒ false
    | x :: l ⇒ match x with
                | dectm a b ⇒ (eqb v a) || (check_term v l)
                | decty _ _⇒ check_term v l
    end
    end.
```

Checking the existence of type variables just like above, with passing over the term variable declarations. It does not matter whether the type is of what kind, so we use a wildcard when checking type declarations.

```
Fixpoint check_type (to : typeorder) (con : context) : bool :=
    match con with
    | nil ⇒ false
    | x :: l ⇒ match x with
                | dectm a b ⇒ check_type to l
                | decty a _ ⇒ (eqb a to) || (check_type to l)
    end
    end.
```

Substitutes term B in A. x is the free variable we want to substitute B in its place. Different cases are considered:

- If A is a single variable, we check whether it equals the free variable x. If equal, do the substitution. If not, do nothing and return A.
- If A is equivalent to XY, where X and Y are terms, replace B in X and Y. Note that the substitution does not change the form of A.
- If A is equivalent to λy:X.Y, consider two cases: if x is the binding variable, do nothing. If not, substitute B in Y.

```
Fixpoint termreplacement (A : term) (B : term) (x : termorder) : term :=
  match A with
  | tvar y ⇒ if eqb y x then B else A
  | tappl X Y ⇒ tappl (termreplacement X B x) (termreplacement Y B x)
  | tabst y X Y ⇒ if eqb y x then A else tabst y X (termreplacement Y B x)
  end.
```

Substitutes type B in A. α is the free type variable we want to substitute B in its place. Different cases are considered:

- If A is a single type variable, we check whether it equals the free variable α. If equal, do the substitution. If not, do nothing and return A.
- If A is in form X -> Y, where X and Y are types, substitute B in each of X and Y. Note that the structure of A should not change.
- If A is equivalent to XY, where X and Y are types, replace B in X and Y. Note that the substitution does not change the form of A.
- If A is equivalent to λx:X.Y, consider two cases: if α is the binding type variable, do nothing. If not, substitute B in Y.

```
Fixpoint typereplacement (A : type) (B : type) (α : typeorder) : type :=
  match A with
  | tyvar β ⇒ if eqb β α then B else A
  | X ->* Y ⇒ (typereplacement X B α) ->* (typereplacement Y B α)
  | tyappl X Y ⇒ tyappl (typereplacement X B α) (typereplacement Y B α)
  | tyabst x X Y ⇒ if eqb x α then A else tyabst x X (typereplacement Y B α)
  end.
```

This notation is important: it saves us a lot of writing and is set for β-reduction. The inductive prop immediately after, is to compute reductions for terms, which works as follows:

- redexT computes one-step abstraction elimination.
- compAppl1T and compAppl2T say that right and left application preserver reduction, respectively.

- compAbstT says that abstraction preserves reduction.

```
Reserved Notation "a →βt b" (at level 102).
Inductive βreductionTerm : term → term → Prop :=
| redexT : ∀ (x : termorder) (α : type) (M N : term),
    (tappl (tabst x α M) N) →βt (termreplacement M N x)
| compAppl1T : ∀ (M N P : term), (M →βt N) → ((tappl M P) →βt (tappl N P))
| compAppl2T : ∀ (M N P : term), (M →βt N) → ((tappl P M) →βt (tappl P N))
| compAbstT : ∀ (M N : term) (x : termorder) (α : type), (M →βt N) → ((tabst x α M)
→βt (tabst x α N))
where "X →βt Y" := (βreductionTerm X Y).
```

This notation and its following inductive prop help us with things related to the properties of β-equivalence in terms:

- β-equivalence is reflexive: every term is equivalent with itself.
- If M is equivalent with P and P is reduced to N (or N is reduced to P), M is equivalent with N

```
Reserved Notation "X =βt Y" (at level 107).
Inductive βequivTerm : term → term → Prop :=
| reflBT : ∀ (M : term), M =βt M
| rightBT : ∀ (M P N : term), (M =βt P) → (P →βt N) → (M =βt N)
| leftBT : ∀ (M P N : term), (M =βt P) → (N →βt P) → (M =βt N)
where "X =βt Y" := (βequivTerm X Y).
```

This notation is also important: it saves us a lot of writing and is set for β-reduction. The inductive prop immediately after, is to compute reductions for types, which works as follows:

- redexTy computes one-step abstraction elimination.
- compAppl1Ty and compAppl2Ty say that right and left application preserver reduction, respectively.
- compAbstTy says that abstraction preserves reduction.

```
Reserved Notation "a →β b" (at level 102).
Inductive βreductionType : type → type → Prop :=
| redexTy : ∀ (x : typeorder) (α : kind) (M N : type),
    (tyappl (tyabst x α M) N) →β (typereplacement M N x)
| compAppl1Ty : ∀ (M N P : type), (M →β N) → ((tyappl M P) →β (tyappl N P))
| compAppl2Ty : ∀ (M N P : type), (M →β N) → ((tyappl P M) →β (tyappl P N))
| compAbstTy : ∀ (M N : type) (x : typeorder) (α : kind), (M →β N) → ((tyabst x α M)
→β (tyabst x α N))
```

```
where "X →β Y" := (βreductionType X  Y).
```

This notation and its following inductive prop help us with things related to the properties of β-equivalence in types:

- β-equivalence is reflexive: every type is equivalent with itself.
- If M is equivalent with P and P is reduced to N (or N is reduced to P), M is equivalent with N

```
Reserved Notation "X =β Y" (at level 107).
Inductive βequivType : type → type → Prop :=
| reflBTy : ∀ (M : type), M =β M
| rightBTy : ∀ (M  P  N : type), (M =β P) → (P →β N) → (M =β N)
| leftBTy : ∀ (M  P  N : type), (M =β P) → (N →β P) → (M =β N)
where "X =β Y" := (βequivType X  Y).
```

Just so have something to keep in the armory for unpredicted times

```
Parameter βeqkind : kind → kind → Prop.
Infix "=βκ" := βeqkind (at level 107).
```

The heart of the work, at last, the inference rules. The six cases, that with the exception of sort, each one has a star version and a box version.

Figure 3: Our implementation of rules of the system $\lambda\underline{\omega}$. In all rules below, Γ is `context`, x is `termorder`, α is `typeorder`, M and N are `term`, A and B are `type`, K and J are `kind`, and *state* is an arbitrary `statement`.

(sort)	$\vdash * : \Box$	
(varstar)	$\dfrac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}$	`check_term` $x\ \Gamma = $ `false`
(varbox)	$\dfrac{\Gamma \vdash K : \Box}{\Gamma, \alpha : K \vdash \alpha : K}$	`check_type` $\alpha\ \Gamma = $ `false`
(weakstar)	$\dfrac{\Gamma \vdash stat \qquad \Gamma \vdash A : *}{\Gamma, x : A \vdash stat}$	`check_term` $x\ \Gamma = $ `false`
(weakbox)	$\dfrac{\Gamma \vdash stat \qquad \Gamma \vdash K : \Box}{\Gamma, \alpha : K \vdash stat}$	`check_type` $\alpha\ \Gamma = $ `false`
(formstar)	$\dfrac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *}$	
(formbox)	$\dfrac{\Gamma \vdash K : \Box \qquad \Gamma \vdash J : \Box}{\Gamma \vdash K \to J : \Box}$	
(applstar)	$\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$	
(applbox)	$\dfrac{\Gamma \vdash A : K \to J \qquad \Gamma \vdash B : K}{\Gamma \vdash AB : J}$	
(abststar)	$\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : *}{\Gamma \vdash \lambda x : A.M : A \to B}$	
(abstbox)	$\dfrac{\Gamma, \alpha : K \vdash A : J \qquad \Gamma \vdash K \to J : \Box}{\Gamma \vdash \lambda\alpha : K.A : K \to J}$	
(convkind)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : *}{\Gamma \vdash M : B}$	$A =_\beta B$
(convbox)	$\dfrac{\Gamma \vdash A : K \qquad \Gamma \vdash J : \Box}{\Gamma \vdash A : J}$	$K =_\beta J$

```
Inductive inferencerule : context → statement → Prop :=
| sort : [] ⊢ kind_box ×
| varstar : ∀ (Γ : context) (x : termorder) (A : type), (check_term x Γ) = false → (Γ ⊢
type_kind A *) → ((x :# A) :: Γ ⊢ (x :# A))
| varbox : ∀ (Γ : context) (s : typeorder) (A : kind), (check_type s Γ) = false → (Γ ⊢
```

```
kind_box A) → ((s :! *) :: Γ ⊢ (s :! *))
| weakkind : ∀ (Γ : context) (stat : statement) (x : termorder) (C : type), (check_term x
Γ) = false → (Γ ⊢ stat) → (Γ ⊢ type_kind C *)
                                    → ((x :# C) :: Γ ⊢ stat)

| weakbox : ∀ (Γ : context) (stat : statement) (s : typeorder) (C: kind), (check_type s Γ)
= false → (Γ ⊢ stat) → (Γ ⊢ kind_box C)
                                    → ((s :! C) :: Γ ⊢ stat)

| formstar : ∀ (Γ : context) (A B: type), (Γ ⊢ (type_kind A *)) → (Γ ⊢ (type_kind B *))
→ (Γ ⊢ (type_kind (A ->* B) *))
| formbox : ∀ (Γ : context) (A B: kind), (Γ ⊢ (kind_box A)) → (Γ ⊢ (kind_box B)) → (Γ
⊢ (kind_box (A -» B)))
| applstar : ∀ (Γ : context) (A B: type) (M N: term), (Γ ⊢ term_type M (A ->* B)) →
(Γ ⊢ term_type N A) → (Γ ⊢ term_type (tappl M N) B)
| applbox : ∀ (Γ : context) (A B: kind) (M N: type), (Γ ⊢ type_kind M (A -» B)) → (Γ
⊢ type_kind N A) → (Γ ⊢ type_kind (tyappl M N) B)
| abststar : ∀ (Γ : context) (A B: type) (M : term) (x : termorder), ((x :# A) :: Γ ⊢
term_type M B) → (Γ ⊢ type_kind (A ->* B) *)
                                    → (Γ ⊢ term_type (tabst x A M) (A ->* B))
| abstbox : ∀ (Γ : context) (A B: kind) (M : type) (x : typeorder), ((x :! A) :: Γ ⊢ type_kind
M B) → (Γ ⊢ kind_box (A -» B))
                                    → (Γ ⊢ type_kind (tyabst x A M) (A -» B))
| convkind: ∀ (Γ : context) (A : term) (B B' : type) (k : kind), (B =β B') → (Γ ⊢
term_type A B) → (Γ ⊢ type_kind B' k)
                                    → (Γ ⊢ term_type A B')
| convbox : ∀ (Γ : context) (A : type) (B B' : kind), (B =βκ B') → (Γ ⊢ type_kind A B)
→ (Γ ⊢ kind_box B')
                                    → (Γ ⊢ type_kind A B')
where "x ⊢ y" := (inferencerule x y).
```

Some helping lemma to prove the free variables Lemma. It states that, no matter how many valid declarations we add to the context, if a term exists in the context, will never disappear from it . The proof is with induction on Γ, with the base case being the empty context and the step being the non-empty context. The empty context contradicts our premise in the arrow, so we use inversion to solve this case. If the context is not empty, we have, from the induction hypothesis, that x exists in the smaller context, therefore we can prove the lemma for the extended context.

```
Lemma context_extension : ∀ (Γ : context) (x : termorder) (d : declaration),
    check_term x Γ = true → check_term x (d :: Γ) = true.
    Proof. intros Γ. induction Γ.
    - intros x d H. inversion H.
```

```
- intros. simpl. simpl in H. rewrite → H. destruct d.
    + apply Bool.orb_comm.
    + reflexivity. Qed.
```

Some other helping lemma to prove the free variables Lemma. It states that, the existence of a declaration does not change with the removal of another declaration. We use a proof by cases: when we remove a declaration from Γ, either it is x or y (both are variables). If x is taken, we have a contradiction: the first premise in the arrows tell us that x and y must not be equal. If it is y that is taken, using the second premise in the second arrow, the goal is obvious.

```
Lemma context_shrink : ∀ (Γ : context) (x y : termorder) (A : type),
    (x =? y) = false → check_term x ((y :# A) :: Γ) = true → check_term x Γ = true.
    Proof. intros Γ x y A. intros. simpl in H0. apply Bool.orb_prop in H0.
    destruct H0.
    - rewrite → H0 in H. inversion H.
    - apply H0. Qed.
```

Now we are armed up to prove free variables Lemma.
This proof is by induction on the structure of the proof. We use generalized dependence to strengthen our induction hypothesis so that the contradictory cases could be resolved via inversion. In nearly all the cases, induction hypothesis is used to prove the goal directly, or with a mediatory step that changes the appearance of the hypothesis to help reach the goal easier (such as using orb_prop, or eqb_sym). In other cases (such as var, or all the rules that introduce a variable in their conclusion), we need to apply context extension lemma to include the additional variable in the context. In the cases that variables exist in the premises and not in the conclusions, we need to shrink the context in the induction hypothesis via context shrink lemma, so that we could use it to prove the goal.

```
Lemma freeVariable : ∀ (Γ : context) (L : term) (σ : type),
    (Γ ⊢ term_type L σ) →
        ∀ (x : termorder), freevar x L = true → check_term x Γ = true.
    Proof.
        intros Γ L σ.
        remember (term_type L σ) as s.
        intros H.
        generalize dependent σ.
        generalize dependent L.
        induction H; intros;
            inversion Heqs; subst.
            - simpl in H1. simpl. rewrite → H1. simpl. reflexivity.
            - apply context_extension. apply (IHinferencerule1 L σ).
                + apply H3.
                + apply H2.
            - apply context_extension. apply (IHinferencerule1 L σ).
```

```
        + apply H3.
        + apply H2.
    - simpl in H1. apply Bool.orb_prop in H1. destruct H1.
        + apply (IHinferencerule1 M (A ->* σ)).
            × reflexivity.
            × apply H1.
        + apply (IHinferencerule2 N A).
            × reflexivity.
            × apply H1.
    - simpl in H1. apply (context_shrink Γ x0 x A).
        + rewrite PeanoNat.Nat.eqb_sym. destruct (x =? x0).
            × inversion H1.
            × reflexivity.
        + apply (IHinferencerule1 M B).
            × reflexivity.
            × destruct (x =? x0).
                ** inversion H1.
                ** apply H1.
    - apply (IHinferencerule1 L B).
        + reflexivity.
        + apply H2. Qed.
```

The following lemmas are simply stated but not proved, due to little time and the exhaustive amount of proofs we have to write!

```
Lemma thinning : ∀ (Γ Γ' : context) (M : term) (σ : type),
    (∀ (d : declaration), In d Γ → In d Γ') →
    (Γ ⊢ term_type M σ) →
        (Γ' ⊢ term_type M σ).
Admitted.

Lemma substitution : ∀ (Γ Γ' : context) (x : termorder) (M N : term) (σ τ : type),
    (Γ' ++ (x :# σ) :: Γ ⊢ term_type M τ) →
    (Γ ⊢ term_type N σ) →
        (Γ' ⊢ term_type (termreplacement M N x) τ).
Admitted.

Lemma uniqueness : ∀ (Γ : context) (A : term) (B1 B2 : type),
    (Γ ⊢ term_type A B1 ) →
    (Γ ⊢ term_type A B2) →
        (B1 =β B2).
    Proof.
        intros Γ A B1 B2.
        remember (term_type A B1 ) as s1.
        remember (term_type A B2) as s2.
```

```
intros H1. intros H2.
generalize dependent A.
Abort.
```