J Autom Reasoning (2012) 49:327–362
DOI 10.1007/s10817-011-9230-5

A Solution to the PoplMark Challenge Based on de Bruijn Indices

Jérôme Vouillon

Received: 28 July 2009 / Accepted: 16 May 2011 / Published online: 4 June 2011

Abstract The PoplMark challenge proposes a set of benchmarks intended to assess the usability of proof assistants in the context of research on programming languages. It is based on the metatheory of System $F_{<:}$. We present a solution to the challenge using de Bruijn indices, developed with the Coq proof assistant.

Keywords Proof assistants · Theorem proving · Metatheory · Variable binding · De Bruijn indices · Coq

1 Introduction

The PoplMark challenge [5] is a set of benchmarks designed to evaluate the state of mechanization in the metatheory of programming languages. The challenge was timely: proof assistants are becoming mature and there is a raising interest among the programming language research community towards mechanized proofs.

In our experience [3, 27, 28], we are still at a stage where writing a full polished mechanized proof for a conference paper remains a huge undertaking and is usually not worthwhile. Still, we have found it very useful to have mechanized proofs of key results. Of course, it gives high confidence on the results. But, above all, it forces to think about every details: there can be no fuzzy definitions, no proofs by hand waving. As a consequence, one gets a much better understanding of what makes the proofs go through. Without the help of a proof assistant, we believe the papers mentioned above would have been very different, if written at all.

Mechanized proofs have other advantages compared to usual paper proofs. Writing paper proofs requires a lot of attention. As the formalization evolves, a lot of work is required to update the proofs properly and then convince oneself that they remain correct. Some obvious lemma whose proof was only sketched may well

J. Vouillon (✉)
PPS, UMR 7126, CNRS, Univ Paris Diderot, Sorbonne Paris Cité, 75205 Paris, France
e-mail: jerome.vouillon@pps.jussieu.fr

become wrong. We find this work tedious and stressful. On the other hand, with a mechanized proof, the proof assistant indicates directly which parts of the proof have to be adjusted. And once the whole proof goes through, we know it is correct. As an illustration, we can consider the paper proofs in the PoplMark challenge paper. These proofs are of a much higher standard than what is usually done. They are well-written and very detailed. We find it significant that, despite this, the proof sketch of one of the lemmas is incorrect (see Section 4.8), and a significant lemma is missing[1] (see Section 5.2).

We present a solution to the PoplMark challenge based on de Bruijn indices. The PoplMark team was critical of this technique:

> *The technology should impose reasonable overheads. [...] our experience is that explicit de Bruijn-indexed representations of variable binding structure fail this test. [...] In our experience, [...] de Bruijn representations have two major flaws. First, the statements of theorems require complicated clauses involving "shifted" terms and contexts. These extra clauses make it difficult to see the correspondence between informal and formal versions of the same theorem–there is no question of simply typesetting the formal statement and pasting it into a paper. Second, while the notational clutter is manageable for "toy" examples of the size of the simply-typed lambda calculus, we have found it becomes quite a heavy burden even for fairly small languages like F<:.*

We hope the present paper, and the corresponding proof development, will convince the reader that de Bruijn indices actually incurs reasonable overheads, at least with the help of a proof assistant (we would not use them for paper proofs!).

Our solution [26] to the PoplMark challenge is written in Coq [23]. The challenge is composed of three parts. The solution addresses challenges 1 (transitivity of $F_{<:}$ subtyping) and 2 (type safety of $F_{<:}$). Each of these challenges are divided in two parts, starting with properties of pure $F_{<:}$ and then asking that the same properties be proved for $F_{<:}$ enriched with records and pattern matching. Our solution to the first part of these challenges is about 1,300 lines,[2] omitting empty lines and comments). This appears to be fairly competitive, given that we have made no attempt at minimizing the size of our solution. In particular, we make little use of automation. All lemmas are explicitly applied: the proofs do not take advantage of the hint database mechanism provided by Coq. This makes the proofs longer, but probably more readable as the dependencies between the different lemmas can be explicitly read from the proof script.

We have tried to follow as closely as possible the paper proofs provided in the challenge paper. We refer to these proofs when describing our proof development. In this paper, we provide the Coq definitions and lemma statements. Proof details, as well as a few lemmas of little importance, are omitted. We start by providing an explanation of de Bruijn indices (Section 2). Then, we present our solution to challenges 1A (Section 3) and 2A (Section 4) and sketch the solution to challenges 1B

[1]For solution 2B, the proof of lemma A.15 (Progress) relies on the fact that a well-typed pattern can always match a well-typed value of the same type (that is, rule E-LetV can be applied). This is not obvious, but this issue is not even mentioned.

[2]The solution to challenges 1B and 2B is about 2,200 lines.

and 2B (Section 5). Finally, we rapidly present how challenge 3 could be addressed (Section 6).

2 De Bruijn Indices

2.1 Syntax

De Bruijn indices [11] is a notation for representing terms with binders up to α-equivalence. The key idea is that variable occurrences in a term are not represented by a name, but by an *index*, which is a natural number. This index counts the number of binders between this occurrence of the variable and the corresponding binder.

We illustrate de Bruijn indices using the types of System $F_{<:}$. The standard syntax is the following.

$\texttt{T} ::=$	*types*
$\texttt{X}$	*type variable*
$\texttt{Top}$	*maximum type*
$\texttt{T} \to \texttt{T}$	*type of functions*
$\forall \texttt{X<:T.T}$	*universal type*

A universal type $\forall \texttt{X<:T}_1\texttt{.T}_2$ contains a binder for variable $\texttt{X}$ in type $\texttt{T}_2$. In the corresponding de Bruijn syntax, type variables $\texttt{X}$ are replaced by indices $\texttt{N}$ and bound variables are implicit:

$\texttt{T} ::=$	*de Bruijn types*
$\texttt{N}$	*type index*
$\texttt{Top}$	
$\texttt{T} \to \texttt{T}$	
$\forall_{<:}\texttt{T.T}$	

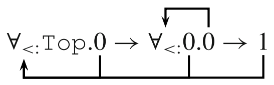
For instance, the type:

$$\forall \texttt{X<:Top.X} \to \forall \texttt{Y<:X.Y} \to \texttt{X}$$

is represented using the following syntax:

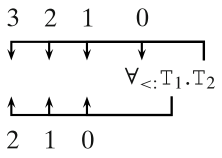
$$\forall_{<:}\texttt{Top}.0 \to \forall_{<:}0.0 \to 1\ .$$

Clearly, de Bruijn notation is hard to read, as a same number can stand for different variables, and conversely. For the sake of clarity, it is worthwhile to draw arrows from indices to the corresponding binder.



2.2 Shifting and Substitution

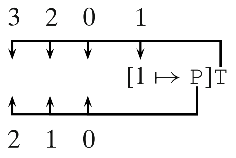
What makes de Bruijn indices practical is that, usually, few terms are explicitly written in proofs. One rather manipulates abstract types (written with just a letter $\texttt{T}$) or type schemes (written, for instance, $\forall_{<:}\texttt{T}_1\texttt{.T}_2$). It is then more fruitful to consider a dual point of view, focusing on contexts rather than on index occurrences in terms. For instance, let us consider the term below.



We have represented the outer context below the type. This is also the context of type $\mathtt{T}_1$. The context of type $\mathtt{T}_2$ is above. When a type $\mathtt{T}$ is moved from the global context into type $\mathtt{T}_2$ (typically, by substitution), all its indices have to be shifted by 1 in order to make room for the additional bound variable. We write $\uparrow^0 \mathtt{T}$ for the result of this *shift* operation. This operation is also called *lift* or *bump* [22]. One must actually be able to make room at any index $\mathtt{N}$, hence the more general shifting operator $\uparrow^{\mathtt{N}}$.

As a more concrete example, given a type $\mathtt{T}$, in order to build a type $\forall\mathtt{X}{<:}\mathtt{Top}.\mathtt{T}$ (where variable $\mathtt{X}$ does not occur in $\mathtt{T}$), one need to make room for variable $\mathtt{X}$ at rank 0 in $\mathtt{T}$. Hence the representation $\forall_{<:}\mathtt{Top}.\uparrow^0 \mathtt{T}$.

Substitution has a dual effect on the context. In a substitution $[\mathtt{N} \mapsto \mathtt{P}]\mathtt{T}$, the type $\mathtt{T}$ is in a context with an additional variable $\mathtt{N}$. Type $\mathtt{P}$ is in the global context. These two contexts are illustrated below.



After substitution, variable $\mathtt{N}$ has been replaced by type $\mathtt{P}$ in type $\mathtt{T}$ and the variables $\mathtt{I} > \mathtt{N}$ occurring in $\mathtt{T}$ has been decremented by 1 to fill the gap. This corresponds to moving from the upper context to the lower context in the figure above.

The shifting and substitution operators are defined recursively in Fig. 1. Note that when a binder is crossed (universal type), the cut-off index is increased by one.

Shifting

$$\begin{array}{lll}
\uparrow^{\mathtt{I}} \mathtt{N} & = \mathtt{N}+1 & \text{when } \mathtt{I} \leq \mathtt{N} \\
 & = \mathtt{N} & \text{when } \mathtt{I} > \mathtt{N} \\
\uparrow^{\mathtt{I}} \mathtt{Top} & = \mathtt{Top} & \\
\uparrow^{\mathtt{I}} (\mathtt{T} \to \mathtt{T}) & = (\uparrow^{\mathtt{I}} \mathtt{T}) \to (\uparrow^{\mathtt{I}} \mathtt{T}) & \\
\uparrow^{\mathtt{I}} \forall_{<:}\mathtt{T}.\mathtt{T} & = \forall_{<:} \uparrow^{\mathtt{I}} \mathtt{T}. \uparrow^{\mathtt{I}+1} \mathtt{T} &
\end{array}$$

Substitution

$$\begin{array}{lll}
[\mathtt{I} \mapsto \mathtt{P}]\mathtt{N} & = \mathtt{N} & \text{when } \mathtt{N} < \mathtt{I} \\
 & = \mathtt{P} & \text{when } \mathtt{N} = \mathtt{I} \\
 & = \mathtt{N}-1 & \text{when } \mathtt{N} \geq \mathtt{I} \\
[\mathtt{I} \mapsto \mathtt{P}]\mathtt{Top} & = \mathtt{Top} & \\
[\mathtt{I} \mapsto \mathtt{P}](\mathtt{T}_1 \to \mathtt{T}_2) & = ([\mathtt{I} \mapsto \mathtt{P}]\mathtt{T}_1) \to ([\mathtt{I} \mapsto \mathtt{P}]\mathtt{T}_2) & \\
[\mathtt{I} \mapsto \mathtt{P}]\forall_{<:}\mathtt{T}_1.\mathtt{T}_2 & = \forall_{<:}[\mathtt{I} \mapsto \mathtt{P}]\mathtt{T}_1.[\mathtt{I}+1 \mapsto \uparrow^0 \mathtt{P}]\mathtt{T}_2 &
\end{array}$$

Fig. 1 Shifting and substitution

$$\begin{array}{lcl}
\mathtt{T} & = & [\mathtt{N} \mapsto \mathtt{P}] \uparrow^{\mathtt{N}} \mathtt{T} \\
\uparrow^{\mathtt{N}} \uparrow^{\mathtt{N+I}} \mathtt{T} & = & \uparrow^{\mathtt{N+I+1}} \uparrow^{\mathtt{N}} \mathtt{T} \\
\uparrow^{\mathtt{N}} [\mathtt{N}+\mathtt{I} \mapsto \mathtt{P}]\mathtt{T} & = & [\mathtt{N}+\mathtt{I}+1 \mapsto \uparrow^{\mathtt{N}} \mathtt{P}] \uparrow^{\mathtt{N}} \mathtt{T} \\
\uparrow^{\mathtt{N+I}} [\mathtt{N} \mapsto \mathtt{P}]\mathtt{T} & = & [\mathtt{N} \mapsto \uparrow^{\mathtt{N+I}} \mathtt{P}] \uparrow^{\mathtt{N+I+1}} \mathtt{T} \\
[\mathtt{N}+\mathtt{I} \mapsto \mathtt{Q}][\mathtt{N} \mapsto \mathtt{P}]\mathtt{T} & = & [\mathtt{N} \mapsto [\mathtt{N}+\mathtt{I} \mapsto \mathtt{Q}]\mathtt{P}][\mathtt{N}+\mathtt{I}+1 \mapsto \uparrow^{\mathtt{N}} \mathtt{Q}]\mathtt{T}
\end{array}$$

Fig. 2 Interaction rules

Similarly, when a type P crosses a binder, its indices are shifted. This use of the shifting operator is an instance of a general rule that we follow:

Whenever a type (or a term) crosses a binder, its indices must be shifted to fit the new context.

We have found that using this rule consistently is very effective at keeping the de Bruijn notation manageable.

2.3 Interaction Rules Between Shifting and Substitution

A last crucial ingredient for dealing effectively with de Bruijn indices is a characterization of how shifting and substitution interact. The five rules of interest are given in Fig. 2. They cover all possible interactions between the two operators. In practice, it is always clear when to apply these rules: when stuck with two operators applied to the same term, apply the corresponding rule. We believe it is important to explain these rules. For this, we illustrate graphically the effect of the shift and substitution operators on the context by diagrams as in Fig. 3. The upper lines stand for the contexts before the operation is performed while the lower lines stands for the resulting contexts. The black rectangle is the variable added or removed.

The first rule is illustrated in Fig. 4. Inserting a variable at index N and then replacing it by a type P is the same as leaving the term unchanged.

The second rule is illustrated in Fig. 5. On the left hand side, two variables are inserted, first at index $\mathtt{N}+\mathtt{I}$, then at index N. The second insertion shifts the first variable to index $\mathtt{N}+\mathtt{I}+1$. This is the same as inserting variables first at index N, then at index $\mathtt{N}+\mathtt{I}+1$ (right hand side of the equation).

In the third rule, the term $[\mathtt{N}+\mathtt{I} \mapsto \mathtt{P}]\mathtt{T}$ is shifted at index N (left hand side of the equality). This can also be performed as follows. As the context of term P is the outer context, is should be shifted at the same index N. Term T is in a context with a variable added at index $\mathtt{N}+\mathtt{I}$. This is greater or equal to N, thus term T is also shifted at index N. On the other hand, index $\mathtt{N}+\mathtt{I}$ becomes index $\mathtt{N}+\mathtt{I}+1$ after shifting.

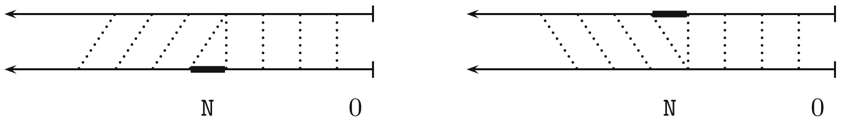


Fig. 3 Effect of shifting (*left*) and substitution (*right*) on the context

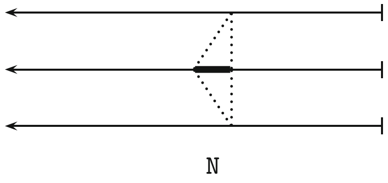


Fig. 4 Illustration of the equality $\mathtt{T} = [\mathtt{N} \mapsto \mathtt{P}] \uparrow^{\mathtt{N}} \mathtt{T}$

So, finally, type $\uparrow^{\mathtt{N}} \mathtt{P}$ is substituted for index $\mathtt{N} + \mathtt{I} + 1$ in type $\uparrow^{\mathtt{N}} \mathtt{T}$ (right hand side of the equality).

In the fourth rule, the term $[\mathtt{N} \mapsto \mathtt{P}]\mathtt{T}$ is shifted at index $\mathtt{N} + \mathtt{I}$. This can also be performed as follows. As the context of term $\mathtt{P}$ is the outer context, is should be shifted at the same index $\mathtt{N} + \mathtt{I}$. Term $\mathtt{T}$ is in a context with a variable added at index $\mathtt{N}$. It should be shifted at index $\mathtt{N} + \mathtt{I} + 1$ which corresponds to index $\mathtt{N} + \mathtt{I}$ in the outer context. Index $\mathtt{N}$ is not affected by the shifting, as it is smaller than $\mathtt{N} + \mathtt{I} + 1$. So, finally, type $\uparrow^{\mathtt{N+I}} \mathtt{P}$ is substituted for index $\mathtt{N}$ in type $\uparrow^{\mathtt{N+I+1}} \mathtt{T}$.

In the last rule, type $\mathtt{Q}$ is substituted at index $\mathtt{N} + \mathtt{I}$ in $[\mathtt{N} \mapsto \mathtt{P}]\mathtt{T}$. This can also be performed as follows. The substitution is applied to term $\mathtt{P}$ which is in the outer context. Then, the substitution has to be adjusted before being applied to type $\mathtt{T}$: index $\mathtt{N} + \mathtt{I}$ in the outer contexts becomes index $\mathtt{N} + \mathtt{I} + 1$ in the context of type $\mathtt{T}$; type $\mathtt{Q}$ becomes type $\uparrow^{\mathtt{N}} \mathtt{Q}$. Hence the right hand side of the equation.

2.4 An Alternative Definition of Substitution

The definition of substitution we have given is standard [18, 21, 22]. It turns out there is an alternative way to define substitution, also widely used [6, 9, 14]. In fact, both definitions appear in the seminal paper [11]. The idea is to defer the application of the shifting operator $\uparrow^{\mathtt{I}}$: rather than applying it each time a binder is crossed, an iterated version $\uparrow^{\mathtt{I}}_{\mathtt{J}}$ of the shifting operator is applied once at each occurrence of the variable substituted for (the natural number $\mathtt{J}$ is the number of shiftings to perform). This is more efficient when the number of occurrences of the variable is less than the number of binders in the term. We write $\{\mathtt{N} \mapsto \mathtt{P}\}\mathtt{T}$ for this other substitution, whose definition is given in Fig. 6. The two definitions of substitutions are related as follows:

$$\{\mathtt{N} \mapsto \mathtt{P}\}\mathtt{T} = [\mathtt{N} \mapsto \uparrow^{0}_{\mathtt{N}} \mathtt{P}]\mathtt{T}$$

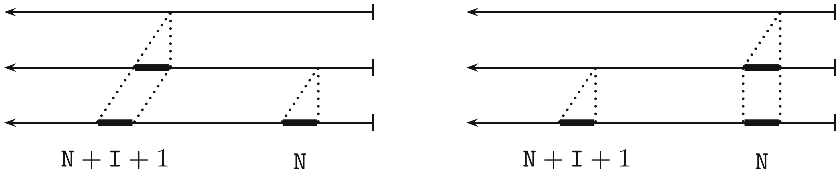


Fig. 5 Illustration of the equality $\uparrow^{\mathtt{N}}\uparrow^{\mathtt{N+I}} \mathtt{T} = \uparrow^{\mathtt{N+I+1}}\uparrow^{\mathtt{N}} \mathtt{T}$

Iterated shifting

$$
\begin{array}{lll}
\uparrow^{\mathtt{I}}_{\mathtt{J}}\ \mathtt{N} & = \mathtt{N} + \mathtt{J} & \text{when } \mathtt{I} \le \mathtt{N} \\
 & = \mathtt{N} & \text{when } \mathtt{I} > \mathtt{N} \\
\uparrow^{\mathtt{I}}_{\mathtt{J}}\ \mathtt{Top} & = \mathtt{Top} & \\
\uparrow^{\mathtt{I}}_{\mathtt{J}}\ (\mathtt{T} \to \mathtt{T}) & = (\uparrow^{\mathtt{I}}_{\mathtt{J}}\ \mathtt{T}) \to (\uparrow^{\mathtt{I}}_{\mathtt{J}}\ \mathtt{T}) & \\
\uparrow^{\mathtt{I}}_{\mathtt{J}}\ \forall_{<:}\mathtt{T}.\mathtt{T} & = \forall_{<:}\ \uparrow^{\mathtt{I}}_{\mathtt{J}}\ \mathtt{T}.\ \uparrow^{\mathtt{I}+1}_{\mathtt{J}}\ \mathtt{T} &
\end{array}
$$

Alternative substitution

$$
\begin{array}{lll}
\{\mathtt{I} \mapsto \mathtt{P}\}\mathtt{N} & = \mathtt{N} & \text{when } \mathtt{N} < \mathtt{I} \\
 & = \uparrow^{0}_{\mathtt{I}}\ \mathtt{P} & \text{when } \mathtt{N} = \mathtt{I} \\
 & = \mathtt{N} - 1 & \text{when } \mathtt{N} \ge \mathtt{I} \\
\{\mathtt{I} \mapsto \mathtt{P}\}\mathtt{Top} & = \mathtt{Top} & \\
\{\mathtt{I} \mapsto \mathtt{P}\}(\mathtt{T}_1 \to \mathtt{T}_2) & = (\{\mathtt{I} \mapsto \mathtt{P}\}\mathtt{T}_1) \to (\{\mathtt{I} \mapsto \mathtt{P}\}\mathtt{T}_2) & \\
\{\mathtt{I} \mapsto \mathtt{P}\}\forall_{<:}\mathtt{T}_1.\mathtt{T}_2 & = \forall_{<:}\{\mathtt{I} \mapsto \mathtt{P}\}\mathtt{T}_1.\{\mathtt{I}+1 \mapsto \mathtt{P}\}\mathtt{T}_2 &
\end{array}
$$

Fig. 6 Alternative definition of substitution

The two definitions coincide when $\mathtt{N}$ is 0.

Rasmussen notes that the definition of substitution we have adopted is significantly easier to use [21]. Indeed, it relies on a simpler shifting operator, with a single parameter. We also find it more intuitive, as the substituted type is in the current context, rather than in a possibly deeper context. Finally, the counterparts to the interaction rules of Fig. 2 for the alternative definition of substitution are more complicated and harder to understand [10].

3 Challenge 1A: Transitivity of $F_{<:}$ Subtyping

This challenge consists in proving the transitivity of the subtyping relation of System $F_{<:}$ starting from an algorithmic specification of the relation.

3.1 Syntax

The definition of types follows closely the grammar in Section 2. We define an inductive type *typ* with four constructors, one for each type construct. The constructor *tvar*, corresponding to indices, takes a natural number and returns a type. The constructor *top* is a type. The constructors *arrow* and *all* both take two types and return a type.

```
Inductive typ : Set :=
  | tvar : nat → typ
  | top : typ
  | arrow : typ → typ → typ
  | all : typ → typ → typ.
```

3.2 Shifting and Substitution

Shifting and substitution are specified as functions defined by recursion on types. This is basically the same definition as in Section 2. The type $\uparrow^{\mathtt{T}}$ `N` is written *tshift* `N T`. The type `[N ↦ P]T` is written *tsubst* `T N P`.

```
Fixpoint tshift (I : nat) (T : typ) : typ :=
  match T with
  | tvar N ⇒ tvar (if less_or_equal I N then 1 + N else N)
  | top ⇒ top
  | arrow T1 T2 ⇒ arrow (tshift I T1) (tshift I T2)
  | all T1 T2 ⇒ all (tshift I T1) (tshift (1 + I) T2)
  end.
Fixpoint tsubst (T : typ) (I : nat) (P : typ) : typ :=
  match T with
  | tvar N ⇒
      match compare_nat N I with
      | Nat_less _ ⇒ tvar N
      | Nat_equal _ ⇒ P
      | Nat_greater _ ⇒ tvar (N - 1)
      end
  | top ⇒ top
  | arrow T1 T2 ⇒ arrow (tsubst T1 I P) (tsubst T2 I P)
  | all T1 T2 ⇒ all (tsubst T1 I P) (tsubst T2 (1 + I) (tshift 0 P))
  end.
```

We prove the five interaction rules between shifting and substitution of Fig. 2.

```
Lemma tsubst_tshift_prop :
  ∀ (N : nat) (T P : typ), T = tsubst (tshift N T) N P.
Lemma tshift_tshift_prop :
  ∀ (N I : nat) (T : typ),
  tshift N (tshift (N + I) T) = tshift (1 + N + I) (tshift N T).
Lemma tshift_tsubst_prop_1 :
  ∀ (N I : nat) (T P : typ),
  tshift N (tsubst T (N + I) P) =
  tsubst (tshift N T) (1 + N + I) (tshift N P).
Lemma tshift_tsubst_prop_2 :
  ∀ (N I : nat) (T P : typ),
  tshift (N + I) (tsubst T N P) =
  tsubst (tshift (1 + N + I) T) N (tshift (N + I) P).
Lemma tsubst_tsubst_prop :
  ∀ (N I : nat) (T P Q : typ),
  tsubst (tsubst T N P) (N + I) Q =
  tsubst (tsubst T (1 + N + I) (tshift N Q)) N (tsubst P (N + I) Q).
```

3.3 Environments

An environment is a sequence of bindings. We include term variable bindings even though they are not used for this challenge problem. They will be needed for challenge 2A (Section 4).

`E ::=`	*type environments*
$\emptyset$	*empty type environment*
`E, x : T`	*term variable binding*
`E, X<:T`	*type variable binding*

Environments are defined inductively. There are three constructors: *empty* for the empty environment, *evar* for term variable bindings and *etvar* for type variable bindings.

```
Inductive env : Set :=
  | empty : env
  | evar : env → typ → env
  | etvar : env → typ → env.
```

Following the de Bruijn notation, variable names are omitted in environment bindings. Basically, a variable index is its depth in the environment. To be more precise, we use different namespaces for type variables and term variables. Thus, the type variable with index `N` is the `N`-th *type variable* bound in the environment, term variables being skipped. Indeed, using different namespaces makes the definition of substitutions much clearer: substituting a term has no effect on types, and we never have to deal with the case where a term variable is the same as the type variable substituted for.

We define two functions *get_tvar* and *get_var* for accessing the environment, both of type

$$env \rightarrow nat \rightarrow option\ typ.$$

The first function retrieves the bound associated to a type variable while the second one returns the type of a term variable. They both return *None* when the variable is unbound.

Function *get_tvar* is defined by induction on environments. If the end of an environment is reached, the value *None* is returned as the variable is not bound (case *empty*). Term variable bindings are skipped (case *evar*). If the correct index is reached (that is, $X = 0$), the function returns its bound. Otherwise, the index X is the successor of some natural number X' (which is written $X = S\ X'$), and the function is called recursively on X'. The type returned is shifted each time a type variable binding is crossed (the *opt_map* function applied to some function f maps *None* to *None* and *Some v* to *Some* (f v)).

```
Fixpoint get_tvar (E : env) (X : nat) : option typ :=
  match E with
  | empty ⇒ None
  | evar E' _ ⇒ get_tvar E' X
  | etvar E' T ⇒
      opt_map (tshift 0)
        (match X with
         | 0 ⇒ Some T
         | S X' ⇒ get_tvar E' X'
         end)
  end.
```

The function *get_var* is defined similarly.

```
Fixpoint get_var (E : env) (x : nat) : option typ :=
  match E with
  | empty ⇒ None
  | evar E' T ⇒
      match x with
      | 0 ⇒ Some T
      | S x' ⇒ get_var E' x'
      end
  | etvar E' _ ⇒ opt_map (tshift 0) (get_var E' x)
  end.
```

A flat environment, where all types lie in the same context, may seem simpler at first sight than our choice of a nested environment. But many operations would be more complicated. For instance, when extending a flat environment with a new variable, all types in the environment have to be shifted. Well-formedness conditions (Section 3.4) are also much simpler to state with nested environments.

3.4 Well-Formedness Conditions

The challenge problem states that types and environments should be well-scoped. Enforcing this condition represents a significant part of our proofs (which is completely left out of the paper proofs). It is tempting to do without these conditions. The proofs may well go through. However, one would then be considering a different calculus.

The well-formedness conditions are as follows. In an environment `E,X<:T`, the type variable `X` must not be in the domain of environment `E` and the type `T` must be well-formed, that is, the free variables of type `T` must all be in the domain of environment `E`. The first condition is ensured by our definition of the environment. The second will have to be stated explicitly.

We write the type well-formedness predicate *wf_typ* `E T` as a recursive function. The interesting cases are the case of indices *tvar*, where we check the variable is bound in the current environment `E`, and the case of universal types *all*, where the inner type *T2* is checked in an extended environment.

```
Fixpoint wf_typ (E : env) (T : typ) : Prop :=
  match T with
  | tvar X ⇒ get_tvar E X ≠ None
  | top ⇒ True
  | arrow T1 T2 ⇒ wf_typ E T1 ∧ wf_typ E T2
  | all T1 T2 ⇒ wf_typ E T1 ∧ wf_typ (etvar E T1) T2
  end.
```

The predicate could also have been defined using inductive rules. But, in practice, we never reason by induction on the predicate. Using a fixpoint definition, one can take advantage of the computational power of Coq. Assumptions *wf_typ* `E T` typically occur in proof hypotheses for some known types `T`. For instance, suppose we know that `T` is of the shape *arrow* T_1 T_2. Then, with the definition above, the assumption is directly equivalent (by reduction) to *wf_typ* E $T_1 \wedge$ *wf_typ* E T_2. With an inductive definition, the use of an inversion tactic would be required instead.[3]

The environment well-formedness condition is straightforward to state. We use a recursive function as well.

```
Fixpoint wf_env (E : env) : Prop :=
  match E with
  | empty ⇒ True
  | evar E T ⇒ wf_typ E T ∧ wf_env E
  | etvar E T ⇒ wf_typ E T ∧ wf_env E
  end.
```

We prove that type well-formedness is preserved when the environment is weakened (more precisely, when more type variables are bound). These two lemma are useful for proving environment weakening and strengthening lemmas below.

```
Lemma wf_typ_env_weaken :
  ∀ (T : typ) (E E' : env),
  (∀ (X : nat), get_tvar E' X = None → get_tvar E X = None) →
  wf_typ E T → wf_typ E' T.
Lemma wf_typ_extensionality :
  ∀ (T : typ) (E E' : env),
  (∀ (X : nat), get_tvar E X = get_tvar E' X) →
  wf_typ E T → wf_typ E' T.
```

3.5 Subtyping Relation

The specification of the subtyping relation matches closely the challenge problem. We only insert some well-formedness assertions in rules *SA_Top* and *SA_Refl_TVar*.

```
Inductive sub : env → typ → typ → Prop :=
  | SA_Top :
      ∀ (E : env) (S : typ), wf_env E → wf_typ E S → sub E S top
  | SA_Refl_TVar :
      ∀ (E : env) (X : nat),
      wf_env E → wf_typ E (tvar X) → sub E (tvar X) (tvar X)
  | SA_Trans_TVar :
      ∀ (E : env) (X : nat) (T U : typ),
      get_tvar E X = Some U → sub E U T → sub E (tvar X) T
```

[3]Another possibility could be to reason by induction on the well-formedness hypothesis when one would reason by induction on terms in the paper proof. We have not investigated this.

```
  | SA_Arrow :
      ∀ (E : env) (T1 T2 S1 S2 : typ),
      sub E T1 S1 → sub E S2 T2 → sub E (arrow S1 S2) (arrow T1 T2)
  | SA_All :
      ∀ (E : env) (T1 T2 S1 S2 : typ),
      sub E T1 S1 → sub (etvar E T1) S2 T2 →
      sub E (all S1 S2) (all T1 T2).
```

The well-formedness conditions are inserted in this definition so as to ensure that all environments and types in subtyping relation are well-formed.

```
Lemma sub_wf :
  ∀ (E : env) (T U : typ),
  sub E T U → wf_env E ∧ wf_typ E T ∧ wf_typ E U.
```

Thanks to this lemma, most well-formedness hypotheses can be omitted in lemma statements. Note that as few well-formedness conditions as possible were inserted in the definition of subtyping, so as to reduce the number of times a well-formedness condition has to be proven when building a subtyping derivation.

3.6 Reflexivity of Subtyping

This is Lemma A.1 in the paper proofs. The proof mirrors the paper proof. We believe it is useful to show at least once what a Coq proof looks like. The proof is performed by using tactics. It is hardly readable. Still, there is some structure. The proof starts by swapping the two quantifiers *E* and *T*. It is by induction on type *T*. In all cases, the environment *E* and two hypotheses *H1* (environment well-formedness) and *H2* (type well-formedness) are moved to the hypotheses. Then, each case is resolved by using one of the subtyping rules. In cases the type is of shape *arrow T1 T2* or *all T1 T2*, the induction hypotheses *IHT1* and *IHT2* are applied to conclude. As one can see, the structure of the proof is apparent here. This is not however the style of proof which is encouraged by existing tools, which rather force a linear and unstructured style.

```
Lemma sub_reflexivity :
  ∀ (E : env) (T : typ), wf_env E → wf_typ E T → sub E T T.
intros E T; generalize E; clear E; induction T; intros E H1 H2;
  [ apply SA_Refl_TVar; trivial
  | apply SA_Top; trivial
  | apply SA_Arrow;
      [ exact (IHT1 _ H1 (proj1 H2))
      | exact (IHT2 _ H1 (proj2 H2)) ]
  | apply SA_All;
      [ exact (IHT1 _ H1 (proj1 H2))
      | apply IHT2 with (2 := (proj2 H2));
        simpl; simpl in H2; tauto ] ].
Qed.
```

3.7 Permutation and Weakening

This section corresponds in the paper proofs to Lemma A.2 (Permutation and Weakening). As we have term variables in the environment, we also need to prove part of Lemma A.5 (Weakening for Subtyping and Typing) now. The paper proofs suggest to show the two following properties. First, subtyping is preserved by any environment permutation that preserves well-formedness. Second, inserting multiple bindings to the right of an environment preserves subtyping.

It does not appear cost-effective to specify environment permutation and simultaneous insertion of multiple bindings. Instead, we have chosen to prove a one-step weakening lemma, where a variable is inserted anywhere in the environment. This is strong enough for the remainder of the proofs.

3.7.1 Inserting a Type Variable Binding in an Environment

In order to state the weakening lemma, we first specify what it means to insert a type variable binding in an environment. The proposition *insert_tvar* `N` E_1 E_2 asserts that environment E_2 can be built from environment E_1 by inserting a single binding at index `N`, that is, environment E_1 is of shape `E`, E' while environment E_2 is of shape `E, X<:T,` E' where variable `X` corresponds to index `N`. Note that all type variables in environment E' are shifted when the binder is inserted (rules *ib_var* and *ib_tvar*).

```
Inductive insert_tvar : nat → env → env → Prop :=
  | ib_here :
      ∀ (T : typ) (E : env), wf_typ E T → insert_tvar 0 E (etvar E T)
  | ib_var :
      ∀ (X : nat) (T : typ) (E E' : env),
      insert_tvar X E E' →
      insert_tvar X (evar E T) (evar E' (tshift X T))
  | ib_tvar :
      ∀ (X : nat) (T : typ) (E E' : env),
      insert_tvar X E E' →
      insert_tvar (1 + X) (etvar E T) (etvar E' (tshift X T)).
```

We now present the properties of the *insert_tvar* relation with respect to *get_var* and *get_tvar*. Basically, when *insert_tvar* X' `E` E' holds, environment E' is environment `E` with a type variable binding added at index X' and all types shifted accordingly.

```
Lemma get_tvar_insert_tvar_ge :
  ∀ (X X' : nat) (E E' : env),
  insert_tvar X' E E' → X' ≤ X →
  get_tvar E' (1 + X) = opt_map (tshift X') (get_tvar E X).
Lemma get_tvar_insert_tvar_lt :
  ∀ (X X' : nat) (E E' : env),
  insert_tvar X' E E' → X < X' →
  get_tvar E' X = opt_map (tshift X') (get_tvar E X).
Lemma get_var_insert_tvar :
  ∀ (x X' : nat) (E E' : env),
  insert_tvar X' E E' →
  get_var E' x = opt_map (tshift X') (get_var E x).
```

Finally, we state the properties of type variable binding insertion with respect to well-formedness.

```
Lemma insert_tvar_wf_typ :
  ∀ (T : typ) (X : nat) (E E' : env),
  insert_tvar X E E' → wf_typ E T → wf_typ E' (tshift X T).
Lemma insert_tvar_wf_env :
  ∀ (X : nat) (E E' : env), insert_tvar X E E' → wf_env E → wf_env E'.
```

The same proof pattern is used whenever specifying a relation between environments: first the relation is defined, then we prove its properties with respect to environment access functions, and finally we prove its properties on well-formedness.

3.7.2 Actual Proofs

We can now state and prove the weakening lemma. The occurrence of a shifting operator in the lemma statement reflects the fact that a type variable binding is inserted at depth *X* in the environment.

```
Lemma sub_weakening_tvar_ind :
  ∀ (E E' : env) (X : nat) (U V : typ),
  insert_tvar X E E' → sub E U V → sub E' (tshift X U) (tshift X V).
```

This is the instance of the lemma where *X* is index 0.

```
Lemma sub_weakening_tvar :
  ∀ (E : env) (T U V : typ),
  wf_typ E V → sub E T U → sub (etvar E V) (tshift 0 T) (tshift 0 U).
```

We now prove a weakening lemma for term variable bindings. Rather than proving the property by induction on a derivation (as in the paper proofs), we rely on a more general result: the subtyping relation does not depend on term variable bindings. This intermediate result is also used to prove strengthening (Section 4.6.4).

```
Lemma sub_extensionality :
  ∀ (E E' : env) (U V : typ),
  (∀ (X : nat), get_tvar E X = get_tvar E' X) →
  wf_env E' → sub E U V → sub E' U V.
Lemma sub_weakening_var :
  ∀ (E : env) (T U V : typ),
  wf_typ E V → sub E T U → sub (evar E V) T U.
```

3.8 Transitivity and Narrowing

Transitivity and narrowing (Lemma A.3) must be proved simultaneously. The proof is by induction on the size of terms, defined below. In order to state the narrowing lemma, we also specify what it means for an environment $\mathtt{E}'$ to be a narrow of an environment $\mathtt{E}$.

3.8.1 Size of Types

The size of a type T is defined as follows, by structural recursion.

```
Fixpoint size (T : typ) : nat :=
  match T with
  | tvar _ ⇒ 0
  | top ⇒ 0
  | arrow T1 T2 ⇒ 1 + size T1 + size T2
  | all T1 T2 ⇒ 1 + size T1 + size T2
  end.
```

A simple induction shows that shifting preserves the size of types.

```
Lemma shift_preserves_size : ∀ (T : typ) (X : nat), size (tshift X T) = size T.
```

3.8.2 Narrowing Relation

The environments $\mathtt{E}_1 = \mathtt{E}, \mathtt{X}{<:}\mathtt{Q}, \mathtt{E}'$ and $\mathtt{E}_2 = \mathtt{E}, \mathtt{X}{<:}\mathtt{P}, \mathtt{E}'$ are in a narrowing relation (written *narrow* $\mathtt{N}\ \mathtt{E}_1\ \mathtt{E}_2$) if $\mathtt{E} \vdash \mathtt{P}{<:}\mathtt{Q}$.

```
Inductive narrow : nat → env → env → Prop :=
  | narrow_0 :
      ∀ (E : env) (T T' : typ),
      sub E T' T → narrow 0 (etvar E T) (etvar E T')
  | narrow_extend_var :
      ∀ (E E' : env) (T : typ) (X : nat),
      wf_typ E' T → narrow X E E' → narrow X (evar E T) (evar E' T)
  | narrow_extend_tvar :
      ∀ (E E' : env) (T : typ) (X : nat),
      wf_typ E' T → narrow X E E' →
      narrow (1 + X) (etvar E T) (etvar E' T).
```

The environments satisfy the following properties. They are identical for all variables distinct from variable X.

```
Lemma get_tvar_narrow_ne :
  ∀ (X X' : nat) (E E' : env),
  narrow X E E' → X' ≠ X → get_tvar E X' = get_tvar E' X'.
Lemma get_var_narrow :
  ∀ (X x' : nat) (E E' : env),
  narrow X E E' → get_var E x' = get_var E' x'.
```

The two bounds T and T' of variable X are in subtyping relation.

```
Lemma get_tvar_narrow_eq :
  ∀ (X : nat) (E E' : env),
  narrow X E E' →
  ∃ T, ∃ T',
  get_tvar E X = Some T ∧ get_tvar E' X = Some T' ∧ sub E' T' T.
```

Finally, narrowing preserves well-formedness.

```
Lemma narrow_wf_typ :
  ∀ (E E' : env) (T : typ) (X : nat),
  narrow X E E' → wf_typ E T → wf_typ E' T.
Lemma narrow_wf_env :
  ∀ (E E' : env) (X : nat), narrow X E E' → wf_env E → wf_env E'.
```

3.8.3 Actual Proof

We first state the properties of transitivity and narrowing. These statements are used to formulate in a succinct way intermediate lemmas.

```
Definition transitivity_prop (Q : typ) :=
  ∀ (E : env) (S T : typ), sub E S Q → sub E Q T → sub E S T.
Definition narrowing_prop (Q : typ) :=
  ∀ (E E' : env) (X : nat) (S T : typ),
  narrow X E E' → get_tvar E X = Some Q →
  sub E S T → sub E' S T.
```

The proof follows closely the paper proofs. However, we cannot perform a proof on the distinguished type Q as the induction in the paper proof is on *Q up to alpha conversion* (shifting). Instead, we perform a proof by induction on the size of types. Note that it is actually not that clear what a proof by induction on Q defined up to alpha-conversion means. The extensive literature on nominal reasoning techniques [20, 24] shows that this is indeed far from obvious. If one is not careful, this kind of reasoning by induction up to α-conversion can actually be used to prove false [25]!

First, we give the crucial step in the proof of transitivity, showing that transitivity holds if we assume that both transitivity and narrowing hold for smaller cut types Q'.

```
Lemma transitivity_case :
  ∀ Q : typ,
  (∀ Q' : typ,
    size Q' < size Q → transitivity_prop Q' ∧ narrowing_prop Q') →
  transitivity_prop Q.
```

Next we give the crucial step in the proof of narrowing, showing that narrowing for Q holds if we assume transitivity for types of the same size as Q. This cannot be for Q itself as in the paper proof, as the hypothesis is applied not just to type Q, but also to type *tshift* 0 Q.

```
Lemma narrowing_case :
  ∀ Q : typ,
  (∀ Q' : typ, size Q' = size Q → transitivity_prop Q') →
  narrowing_prop Q.
```

Finally, we combine the above lemmas into the full proof of transitivity and narrowing, by induction on the size of Q.

```
Lemma transitivity_and_narrowing :
  ∀ Q : typ, transitivity_prop Q ∧ narrowing_prop Q.
```

```
Lemma sub_transitivity :
  ∀ (E : env) (T U V : typ), sub E T U → sub E U V → sub E T V.
Lemma sub_narrowing :
  ∀ (E E' : env) (X : nat) (S T : typ),
  narrow X E E' → sub E S T → sub E' S T.
```

4 Challenge 2A: Type Safety of $F_{<:}$

This challenge consists in proving the type safety of System $F_{<:}$, that is, the usual progress and preservation lemmas.

4.1 Syntax

The syntax of $F_{<:}$ terms is the following.

```
t ::=                      term
      x                      variable
      λx:T.t                 abstraction
      t t                    application
      λX<:X.t                type abstraction
      t[T]                   type application
```

With de Bruijn notation, variables are replaced by an index (constructor *var*). Binder names are implicit: variable names are omitted from abstraction *abs* and type abstraction *tabs*.

```
Inductive term : Set :=
  | var : nat → term
  | abs : typ → term → term
  | app : term → term → term
  | tabs : typ → term → term
  | tapp : term → typ → term.
```

As explained in Section 3.3, type and term variables belong to different namespaces: type variable *tvar* 0 and term variable *var* 0 refer to distinct binders.

4.2 Shiftings and Substitutions

We define two additional shifting operators, for shifting term indices and type indices in terms, and two substitutions operators, for substituting terms and types in terms.

```
Fixpoint shift (x : nat) (t : term) : term :=
  match t with
  | var y ⇒ var (if less_or_equal x y then 1 + y else y)
  | abs T1 t2 ⇒ abs T1 (shift (1 + x) t2)
  | app t1 t2 ⇒ app (shift x t1) (shift x t2)
  | tabs T1 t2 ⇒ tabs T1 (shift x t2)
  | tapp t1 T2 ⇒ tapp (shift x t1) T2
  end.
```

```
Fixpoint shift_typ (X : nat) (t : term) : term :=
  match t with
  | var y ⇒ var y
  | abs T1 t2 ⇒ abs (tshift X T1) (shift_typ X t2)
  | app t1 t2 ⇒ app (shift_typ X t1) (shift_typ X t2)
  | tabs T1 t2 ⇒ tabs (tshift X T1) (shift_typ (1 + X) t2)
  | tapp t1 T2 ⇒ tapp (shift_typ X t1) (tshift X T2)
  end.
Fixpoint subst (t : term) (x : nat) (t' : term) : term :=
  match t with
  | var y ⇒
      match compare_nat y x with
      | Nat_less _ ⇒ var y
      | Nat_equal _ ⇒ t'
      | Nat_greater _ ⇒ var (y - 1)
      end
  | abs T1 t2 ⇒ abs T1 (subst t2 (1 + x) (shift 0 t'))
  | app t1 t2 ⇒ app (subst t1 x t') (subst t2 x t')
  | tabs T1 t2 ⇒ tabs T1 (subst t2 x (shift_typ 0 t'))
  | tapp t1 T2 ⇒ tapp (subst t1 x t') T2
  end.
Fixpoint subst_typ (t : term) (X : nat) (T : typ) : term :=
  match t with
  | var y ⇒ var y
  | abs T1 t2 ⇒ abs (tsubst T1 X T) (subst_typ t2 X T)
  | app e1 e2 ⇒ app (subst_typ e1 X T) (subst_typ e2 X T)
  | tabs T1 e1 ⇒ tabs (tsubst T1 X T) (subst_typ e1 (1 + X) (tshift 0 T))
  | tapp e1 T2 ⇒ tapp (subst_typ e1 X T) (tsubst T2 X T)
  end.
```

We do not prove any interaction rules involving these substitution and shifting operators, as we did for types (Section 3.2). Indeed, this is not needed for the proofs.

4.3 Well-Formedness

The predicate *wf_term* `E t` asserts that all the variables in term `t` are bound in environment `E`.

```
Fixpoint wf_term (E : env) (t : term) : Prop :=
  match t with
  | var x ⇒ get_var E x ≠ None
  | abs T1 t2 ⇒ wf_typ E T1 ∧ wf_term (evar E T1) t2
  | app t1 t2 ⇒ wf_term E t1 ∧ wf_term E t2
  | tabs T1 t2 ⇒ wf_typ E T1 ∧ wf_term (etvar E T1) t2
  | tapp t1 T2 ⇒ wf_term E t1 ∧ wf_typ E T2
  end.
```

4.4 Typing Relation

The typing relation is as defined in the challenge problem. We have only added a single well-formedness condition in rule *T_Var*.

```
Inductive typing : env → term → typ → Prop :=
  | T_Var :
       ∀ (E : env) (x : nat) (T : typ),
       wf_env E → get_var E x = Some T → typing E (var x) T
  | T_Abs :
       ∀ (E : env) (t : term) (T1 T2 : typ),
       typing (evar E T1) t T2 → typing E (abs T1 t) (arrow T1 T2)
  | T_App :
       ∀ (E : env) (t1 t2 : term) (T11 T12 : typ),
       typing E t1 (arrow T11 T12) →
       typing E t2 T11 → typing E (app t1 t2) T12
  | T_Tabs :
       ∀ (E : env) (t : term) (T1 T2 : typ),
       typing (etvar E T1) t T2 → typing E (tabs T1 t) (all T1 T2)
  | T_Tapp :
       ∀ (E : env) (t1 : term) (T11 T12 T2 : typ),
       typing E t1 (all T11 T12) → sub E T2 T11 →
       typing E (tapp t1 T2) (tsubst T12 0 T2)
  | T_Sub :
       ∀ (E : env) (t : term) (T1 T2 : typ),
       typing E t T1 → sub E T1 T2 → typing E t T2.
```

We prove that assertion *typing E t T* implies the well-formedness assertions *wf_env E*, *wf_term t* and *wf_typ T* in Section 4.6.3.

4.5 Reduction Rules

Values are abstractions and type abstractions.

```
Definition value (t : term) :=
  match t with
  | abs _ _ ⇒ True
  | tabs _ _ ⇒ True
  | _ ⇒ False
  end.
```

We define the syntax of contexts, as in the challenge problem.

```
Inductive ctx : Set :=
  | c_hole : ctx
  | c_appfun : ctx → term → ctx
  | c_apparg : ∀ (t : term), value t → ctx → ctx
  | c_typefun : ctx → typ → ctx.
```

We specify what it means to insert a term *t* in a context *c*.

```
Fixpoint ctx_app (c : ctx) (t : term) : term :=
  match c with
  | c_hole ⇒ t
  | c_appfun c' t' ⇒ app (ctx_app c' t) t'
  | c_apparg t' _ c' ⇒ app t' (ctx_app c' t)
  | c_typefun c' T ⇒ tapp (ctx_app c' t) T
  end.
```

These are the three evaluation rules.

```
Inductive red : term → term → Prop :=
  | E_AppAbs :
      ∀ (t11 : typ) (t12 t2 : term),
      value t2 → red (app (abs t11 t12) t2) (subst t12 0 t2)
  | E_TappTabs :
      ∀ (t11 t2 : typ) (t12 : term),
      red (tapp (tabs t11 t12) t2) (subst_typ t12 0 t2)
  | E_Ctx :
      ∀ (c : ctx) (t1 t1' : term),
      red t1 t1' → red (ctx_app c t1) (ctx_app c t1').
```

4.6 Some Properties of Typing and Subtyping

We first prove some useful properties of the typing and subtyping relations. In order to state these properties, we define two relations on environments, corresponding respectively to term variable and type variable substitution.

4.6.1 Removal of a Term Variable Binding

We define the operation of removing a term variable binding from the environment:

$$\mathtt{E, x : T, E'} \overset{\mathtt{x}}{\longmapsto} \mathtt{E, E'}.$$

This is what is needed for showing type preservation when substituting a term. As this is a functional relation, we have chosen to specify it using a recursive function rather than an inductive definition.

```
Fixpoint remove_var (E : env) (x : nat) : env :=
  match E with
  | empty ⇒ empty
  | etvar E' T ⇒ etvar (remove_var E' x) T
  | evar E' T ⇒
      match x with
      | 0 ⇒ E'
      | S x ⇒ evar (remove_var E' x) T
      end
  end.
```

Here are the properties of the function with respect to functions *get_var* and *get_tvar*.

```
Lemma get_var_remove_var_lt :
  ∀ (E : env) (x x' : nat),
  x < x' → get_var (remove_var E x') x = get_var E x.
Lemma get_var_remove_var_ge :
  ∀ (E : env) (x x' : nat),
  x ≥ x' → get_var (remove_var E x') x = get_var E (1 + x).
Lemma get_tvar_remove_var :
  ∀ (E : env) (X x': nat),
  get_tvar E X = get_tvar (remove_var E x') X.
```

As previously, we also prove three lemmas regarding well-formedness: inserting or removing a binding preserves type well-formedness; removing a binding preserves environment well-formedness.

4.6.2 Substitution of a Type for a Type Variable in an Environment

We define the operation of substituting a type $\mathtt{T}'$ for some variable $\mathtt{X}$ in an environment:

$$\mathtt{E}, \mathtt{X}{<:}\mathtt{T}, \mathtt{E}' \mapsto \mathtt{E}, [\mathtt{X} \mapsto \mathtt{T}']\mathtt{E}',$$

where $\mathtt{E} \vdash \mathtt{T}'{<:}\mathtt{T}$ (definition A.9 in the paper proofs). The relation *env_subst* $\mathtt{X}$ $\mathtt{T}'$ $\mathtt{E}_1$ $\mathtt{E}_2$ holds when $\mathtt{E}_1$ and $\mathtt{E}_2$ are the two environments in relation above, and the subtyping relation holds. To be precise, the type $\mathtt{T}'$ in the *env_subst* relation is the type $\mathtt{T}'$ in the subtyping relation above, but properly shifted to the context corresponding to environment $\mathtt{E}_2$.

```
Inductive env_subst : nat → typ → env → env → Prop :=
  | es_here :
      ∀ (E : env) (T T' : typ),
      sub E T' T → env_subst 0 T' (etvar E T) E
  | es_var :
      ∀ (X : nat) (T T' : typ) (E E' : env),
      env_subst X T' E E' →
      env_subst X T' (evar E T) (evar E' (tsubst T X T'))
  | es_tvar :
      ∀ (X : nat) (T T' : typ) (E E' : env),
      env_subst X T' E E' →
      env_subst (1 + X) (tshift 0 T') (etvar E T)
        (etvar E' (tsubst T X T')).
```

Here are the properties of the relation with respect to functions *get_var* and *get_tvar*. Basically, when *env_subst* $\mathtt{X}'$ $\mathtt{T}$ $\mathtt{E}$ $\mathtt{E}'$ holds, the environment $\mathtt{E}'$ is the environment $\mathtt{E}$ where the type $\mathtt{T}$ has been substituted for the variable at index $\mathtt{X}'$.

```
Lemma env_subst_get_var :
  ∀ (x X' : nat) (E E' : env) (T : typ),
  env_subst X' T E E' →
  get_var E' x = opt_map (fun T' ⇒ tsubst T' X' T) (get_var E x).
```

Lemma *env_subst_get_tvar_lt* :
  $\forall$ (*X X'* : *nat*) (*E E'* : *env*) (*T* : *typ*),
  env_subst X' T E E' $\rightarrow$ *X* < *X'* $\rightarrow$
  get_tvar E' X = *opt_map* (fun *T'* $\Rightarrow$ *tsubst T' X' T*) (*get_tvar E X*).
Lemma *env_subst_get_tvar_ge* :
  $\forall$ (*X X'* : *nat*) (*E E'* : *env*) (*T* : *typ*),
  env_subst X' T E E' $\rightarrow$ *X'* < *X* $\rightarrow$
  get_tvar E' (*X* - 1) =
  opt_map (fun *T'* $\Rightarrow$ *tsubst T' X' T*) (*get_tvar E X*).

A crucial lemma we can now state is that well-formedness is preserved by substitution.

Lemma *env_subst_wf_typ* :
  $\forall$ (*E E'* : *env*) (*S T* : *typ*) (*X* : *nat*),
  env_subst X T E E' $\rightarrow$ *wf_typ E S* $\rightarrow$ *wf_env E'* $\rightarrow$
  wf_typ E' (*tsubst S X T*).

4.6.3 Typing Relation Well-Formedness

We now show that the typing relation only relates well-formed environments, terms and types. The proof is by induction on the typing derivation. For case *T_Tapp*, we rely on lemma *env_subst_wf_typ* above.

Lemma *typing_wf* :
  $\forall$ (*E* : *env*) (*t* : *term*) (*U* : *typ*),
  typing E t U $\rightarrow$ *wf_env E* $\wedge$ *wf_term E t* $\wedge$ *wf_typ E U*.

4.6.4 Permutation, Weakening, Strengthening and Narrowing

As with subtyping (Section 3.7), we do not prove a general permutation lemma such as Lemma A.4 (Permutation for Typing) in the paper proofs. Rather, we prove weakening and strengthening lemmas where variable insertion and removal may occur anywhere in the environment.

First, we finish proving Lemma A.5 (Weakening for Subtyping and Typing).

Lemma *typing_weakening_tvar* :
  $\forall$ (*E* : *env*) (*t* : *term*) (*U V* : *typ*),
  wf_typ E V $\rightarrow$ *typing E t U* $\rightarrow$
  typing (*etvar E V*) (*shift_typ* 0 *t*) (*tshift* 0 *U*).
Lemma *typing_weakening_var* :
  $\forall$ (*E* : *env*) (*t* : *term*) (*U V* : *typ*),
  wf_typ E V $\rightarrow$ *typing E t U* $\rightarrow$ *typing* (*evar E V*) (*shift* 0 *t*) *U*.

We prove Lemma A.6 (Strengthening).

Lemma *sub_strengthening_var* :
  $\forall$ (*E* : *env*) (*x* : *nat*) (*U V* : *typ*),
  sub E U V $\rightarrow$ *sub* (*remove_var E x*) *U V*.

Finally, we prove Lemma A.7 (Narrowing for the Typing Relation), an analog for the typing relation to the narrowing lemma for subtyping.

```
Lemma typing_narrowing_ind :
  ∀ (E E' : env) (X : nat) (t : term) (U : typ),
  narrow X E E' → typing E t U → typing E' t U.
Lemma typing_narrowing :
 ∀ (E : env) (t : term) (U V1 V2 : typ),
 typing (etvar E V1) t U → sub E V2 V1 → typing (etvar E V2) t U.
```

4.6.5 Substitution and Typing

We show that substitution and type substitution preserves typing. These are Lemma A.8 (Substitution preserves typing), Lemma A.10 (Type substitution preserves subtyping) and Lemma A.11 (Type substitution preserves typing) in the paper proofs.

Compared to the lemma in the paper proofs, the first lemma is slightly stronger: term *q* is typed in the final environment. This makes it possible to use our one-step weakening lemmas rather that the stronger lemmas of the paper proofs.

```
Lemma subst_preserves_typing :
  ∀ (E : env) (x : nat) (t q : term) (T Q : typ),
  typing E t T → get_var E x = Some Q →
  typing (remove_var E x) q Q → typing (remove_var E x) (subst t x q) T.
Lemma tsubst_preserves_subtyping :
  ∀ (E E' : env) (X : nat) (T U V : typ),
  env_subst X T E E' →
  sub E U V → sub E' (tsubst U X T) (tsubst V X T).
Lemma subst_typ_preserves_typing_ind :
  ∀ (E E' : env) (t : term) (U P : typ) (X : nat),
  env_subst X P E E' →
  typing E t U → typing E' (subst_typ t X P) (tsubst U X P).
Lemma subst_typ_preserves_typing :
  ∀ (E : env) (t : term) (U P Q : typ),
  typing (etvar E Q) t U → sub E P Q →
  typing E (subst_typ t 0 P) (tsubst U 0 P).
```

4.6.6 Inversion Lemmas

We don't prove Lemma A.12 (Inversion of subtyping) explicitly. We rely on Coq inversion tactics instead. On the other hand, we prove inversion lemmas for the typing relation (Lemma A.13), as Coq tactics are not sufficient in this case.

```
Lemma t_abs_inversion :
  ∀ (E : env) (t : term) (T0 T1 T2 T3 : typ),
  typing E (abs T1 t) T0 → sub E T0 (arrow T2 T3) →
  sub E T2 T1 ∧ (∃ T4, sub E T4 T3 ∧ typing (evar E T1) t T4).
Lemma t_tabs_inversion :
  ∀ (E : env) (t : term) (T0 T1 T2 T3 : typ),
  typing E (tabs T1 t) T0 → sub E T0 (all T2 T3) →
  sub E T2 T1 ∧
  (∃ T4, sub (etvar E T2) T4 T3 ∧ typing (etvar E T2) t T4).
```

4.7 Progress

We begin by describing the shape of closed values of arrow and quantifier types. This is Lemma A.14 (Canonical Forms) in the paper proofs.

```
Lemma fun_value :
  ∀ (t : term) (T1 T2 : typ),
  value t → typing empty t (arrow T1 T2) → ∃ t' , ∃ T1' , t = abs T1' t'.
Lemma typefun_value :
  ∀ (t : term) (T1 T2 : typ),
  value t → typing empty t (all T1 T2) → ∃ t', ∃ T1', t = tabs T1' t'.
```

We then prove that any non-value can be decomposed into an evaluation context and a subterm which can take a step (Lemma A.15).

```
Lemma local_progress :
  ∀ (t : term) (U : typ),
  typing empty t U →
  value t ∨ ∃ c, ∃ t0, ∃ t0', red t0 t0' ∧ t = ctx_app c t0.
```

The proof of Theorem A.16 (Progress) is then straightforward.

```
Theorem progress :
  ∀ (t : term) (U : typ), typing empty t U → value t ∨ ∃ t', red t t'.
```

4.8 Preservation

We only prove the first part of Lemma A.18, which relates evaluation contexts and the typing relation. Indeed, the second part is not needed: the paper proof of the preservation theorem refers to this part to derive a fact, but then does not make any use of this fact!

The challenge paper only sketches the proof: reason by induction on the structure of evaluation contexts, and then by case on the last rule used in the typing derivation. This proof does not go through when the last rule is *T_Sub*. Instead, our proof is by induction on the typing derivation and case on the evaluation context.

```
Lemma context_replacement :
   ∀ (e : env) (c : ctx) (t t' : term) (T : typ),
   (∀ (T' : typ), typing e t T' → typing e t' T') →
   typing e (ctx_app c t) T → typing e (ctx_app c t') T.
```

We now prove that immediate reduction preserves the type of terms (Lemma A.19).

```
Lemma local_preservation_app :
  ∀ (E : env) (t12 t2 : term) (T11 U : typ),
  typing E (app (abs T11 t12) t2) U → typing E (subst t12 0 t2) U.
Lemma local_preservation_tapp :
  ∀ (E : env) (t12 : term) (T11 T2 U : typ),
  typing E (tapp (tabs T11 t12) T2) U → typing E (subst_typ t12 0 T2) U.
```

Finally, we prove Theorem A.20 (Preservation).

```
Theorem preservation :
  ∀ (E : env) (t t' : term) (U : typ),
  typing E t U → red t t' → typing E t' U.
```

4.9 An Alternative Reduction Relation

Instead of using contexts, one can use explicit closure rules for the evaluation relation. This yields slightly simpler proofs of progress and preservation, both by induction on the typing derivation.

```
Inductive red' : term → term → Prop :=
  | appabs :
      ∀ (t11 : typ) (t12 t2 : term),
      value t2 → red' (app (abs t11 t12) t2) (subst t12 0 t2)
  | tapptabs :
      ∀ (t11 t2 : typ) (t12 : term),
      red' (tapp (tabs t11 t12) t2) (subst_typ t12 0 t2)
  | appfun :
      ∀ t1 t1' t2 : term, red' t1 t1' → red' (app t1 t2) (app t1' t2)
  | apparg :
      ∀ t1 t2 t2' : term,
      value t1 → red' t2 t2' → red' (app t1 t2) (app t1 t2')
  | typefun :
      ∀ (t1 t1' : term) (t2 : typ),
      red' t1 t1' → red' (tapp t1 t2) (tapp t1' t2).
```

5 Challenges 1B and 2B: Adding Records

Challenges 1B and 2B extend the two previous challenges by enriching the language with records. The structure of our solution remains the same. We present the required changes to definitions and lemmas.

5.1 Challenge 1B: Transitivity of Subtyping with Records

Challenge 1A is extended by enriching the type language with record types.

$\mathtt{T} ::= \ldots$	*types*
$\{\mathtt{l}_i = \mathtt{T}_i\}$	*type of records*

This does not add significant difficulties to the transitivity proof, as record types do not contain binders.

A record constructor *trecord* is added to the definition of types of Section 3.1. This assumes given a set of labels *lab*, with decidable equality. A record is a list of pairs of a label and a type. We do not use the predefined list datatype, as we would then have to prove explicitly an induction principle rather than having Coq generate it for us.

```
Inductive typ : Set :=
  ...
  | trecord : trec → typ
with trec : Set :=
  | tcons : lab → typ → trec → trec
  | tnil : trec.
```

We then explicitly generate induction principles for these types, as Coq does not do it by default for mutually defined inductive types.

```
Scheme typ_induction := Induction for typ Sort Prop
with trec_induction := Induction for trec Sort Prop.
```

We define a function *trec_get* : *trec* → *lab* → *option typ* that takes a record *R* and a label *l* and returns the type associated to label *l* in record *R*, if any. The function is declared as a coercion, which means that it is automatically inserted by Coq when needed during typechecking: concretely, one can then write the expression *R l* for the expression *trec_get R l*.

The shifting and substitution functions *tshift* and *tsubst* of Section 3.2 are extended in a straightforward way by recursively traversing patterns. Environments (Section 3.3) are unchanged.

The type well-formedness condition of Section 3.4 is adjusted as follows: a record *R* is well-formed if it contains no repeated label (condition *R2 l* = *None* below) and the types it contains are well-formed.

```
Fixpoint wf_typ (E : env) (T : typ) : Prop :=
  match T with
  ...
  | trecord R1 ⇒ wf_trec E R1
  end
with wf_trec (E : env) (R : trec) : Prop :=
  match R with
  | tcons l T1 R2 ⇒ wf_typ E T1 ∧ wf_trec E R2 ∧ R2 l = None
  | tnil ⇒ True
  end.
```

Finally, the subtyping rule for records is added to the rules of Section 3.5.

```
Inductive sub : env → typ → typ → Prop :=
  ...
  | SA_Rcd :
      ∀ (e : env) (R1 R2 : trec),
      wf_env e → wf_trec e R1 → wf_trec e R2 →
      (∀ (l : lab), R1 l = None → R2 l = None) →
      (∀ (l : lab) (T1 T2 : typ),
        R1 l = Some T1 → R2 l = Some T2 → sub e T1 T2) →
      sub e (trecord R1) (trecord R2).
```

The statements of the different lemmas are then unchanged. Their proofs are adjusted to deal with the additional syntactic construction.

5.2 Challenge 2B: Type Safety with Records and Pattern Matching

The preservation and progress results of Challenge 2A are extended to cover records and pattern matching.

5.2.1 Terms

Terms are enriched with the following constructions.

$\mathtt{t} ::=$	$\ldots$	*term*
	$\{\mathtt{l}_i = \mathtt{t}_i\}$	*record*
	$\mathtt{t.l}$	*projection*
	$\mathtt{let\ p = t\ in\ t}$	*pattern binding*
$\mathtt{p} ::=$		*patterns*
	$\mathtt{x : T}$	*variable pattern*
	$\{\mathtt{l}_i = \mathtt{p}_i\}$	*record pattern*

We thus extend the definition of terms of Section 4.1. We first define patterns. As for record types, we do not make use of the predefined list datatype.

```
Inductive pat : Set :=
  | pvar : typ → pat
  | precord : prec → pat
with prec : Set :=
  | pcons : lab → pat → prec → prec
  | pnil : prec.
```

Binder names are implicit: variable names are omitted in constructor *pvar*. Patterns may bind an arbitrary number of variables. In order to refer to each binder, patterns are read in a sequential fashion: the leftmost occurrence of the *pvar* constructor is the outermost binder, while the rightmost one is the innermost binder.

Terms are extended with records, projection and pattern binding.

```
Inductive term : Set :=
  ...
  | record : rec → term
  | proj : term → lab → term
  | tlet : pat → term → term → term
with rec : Set :=
  | rcons : lab → term → rec → rec
  | rnil : rec.
```

5.2.2 Substitution and Shifting

The shifting and substitution operators of Section 4.2 now have to deal with records and patterns. For this, it is convenient to define a function *offset* that takes as arguments a function *f*, a pattern *p* and some argument *x* and applies *f* to *x* one time for each binder occurring in pattern *p*.

```
Fixpoint offset (A : Set) (f : A → A) (p : pat) (x : A) : A :=
  match p with
  | pvar _ ⇒ f x
  | precord r ⇒ roffset f r x
  end
with roffset (A : Set) (f : A → A) (r : prec) (x : A) : A :=
  match r with
  | pcons _ p1 r2 ⇒ roffset f r2 (offset f p1 x)
  | pnil ⇒ x
  end.
```

When moving across a pattern, the cut-off argument *x* of the term shifting operator *shift* is increased by one for each binder occurring in the pattern. Other cases are straightforward.

```
Fixpoint shift (x : nat) (t : term) : term :=
  match t with
  ...
  | record t1 ⇒ record (rshift x t1)
  | proj t1 l ⇒ proj (shift x t1) l
  | tlet p t1 t2 ⇒ tlet p (shift x t1) (shift (offset (fun y ⇒ 1 + y) p x) t2)
  end
with rshift (x : nat) (t : rec) : rec :=
  match t with
  | rcons l t1 t2 ⇒ rcons l (shift x t1) (rshift x t2)
  | rnil ⇒ rnil
  end.
```

For term substitution *subst*, when moving across a pattern, the variable being substituted should be incremented and the substituted term should be shifted, once per binder occurring in the pattern.

```
Fixpoint subst (t : term) (x : nat) (t' : term) : term :=
  match t with
  ...
  | record t1 ⇒ record (rsubst t1 x t')
  | proj t1 l ⇒ proj (subst t1 x t') l
  | tlet p t1 t2 ⇒
        tlet p (subst t1 x t')
             (subst t2 (offset (fun y ⇒ 1 + y) p x) (offset (shift 0) p t'))
  end
with rsubst (t : rec) (x : nat) (t' : term) : rec :=
  match t with
  | rcons l t1 t2 ⇒ rcons l (subst t1 x t') (rsubst t2 x t')
  | rnil ⇒ rnil
  end.
```

Type shifting and substitution operators are straightforward to adapt.

5.2.3 Well-Formedness Conditions

We extend the well-formedness conditions of Section 4.3. We first define pattern well-formedness. The well-formedness of some parts of a pattern should be checked in an environment extended with previous binders in the pattern. This is also the case for term *t2* in a pattern binding term `let` *p = t1 in t2*. To implement this, function *wf_pat* takes as arguments not only the current environment *E* and a pattern *p* to be checked, but also a function *f* that checks the well-formedness of what is under the scope of the pattern, in the appropriate extended environment. For a variable pattern *pvar T*, we check that type *T* is well-formed and that everything under the scope of the pattern is also well-formed in the environment *E* extended with a term variable of type *T*. When checking a field *l* of a record pattern, we recursively check its associated pattern *p1*, and then the remainder of the pattern *r2* in the environment extended with the bindings in pattern *p1*.

```
Fixpoint wf_pat (E : env) (p : pat) (f : env → Prop) : Prop :=
  match p with
  | pvar T ⇒ wf_typ E T ∧ f (evar E T)
  | precord r ⇒ wf_prec E r f
  end
with wf_prec (E : env) (r : prec) (f : env → Prop) : Prop :=
  match r with
  | pcons l p1 r2 ⇒ r2 l = None ∧ wf_pat E p1 (fun E’ ⇒ wf_prec E’ r2 f)
  | pnil ⇒ f E
  end.

Fixpoint wf_term (E : env) (t : term) : Prop :=
  match t with
  ...
  | record t1 ⇒ wf_rec E t1
  | proj t1 l ⇒ wf_term E t1
  | tlet p t1 t2 ⇒ wf_term E t1 ∧ wf_pat E p (fun E’ ⇒ wf_term E’ t2)
  end
with wf_rec (E : env) (t : rec) : Prop :=
  match t with
  | rcons l t1 t2 ⇒ wf_term E t1 ∧ wf_rec E t2 ∧ t2 l = None
  | rnil ⇒ True
  end.
```

5.2.4 Typing Relation

We define the typing rules for patterns. They corresponds to the P-Var and P-Rcd rules of the challenge problem, except that we directly extend an environment instead of building a piece of environment to be later concatenated. The P-Rcd rule is decomposed in simpler definitions.

```
Inductive ptyping : env → pat → typ → env → Prop :=
  | P_Var :
        ∀ (E : env) (T : typ), ptyping E (pvar T) T (evar E T)
```

```
    | P_Rcd :
        ∀ (E E' : env) (r : prec) (U : trec),
        prtyping E r U E' → ptyping E (precord r) (trecord U) E'
  with prtyping : env → prec → trec → env → Prop :=
    | P_Rcd_Cons :
        ∀ (E1 E2 E3 : env),
        ∀ (l : lab) (t1 : pat) (t2 : prec) (U1 : typ) (U2 : trec),
        ptyping E1 t1 U1 E2 → prtyping E2 t2 U2 E3 → t2 l = None →
        prtyping E1 (pcons l t1 t2) (tcons l U1 U2) E3
    | P_Rcd_Nil :
        ∀ (E : env), prtyping E pnil tnil E.
```

Three typing rules are added to the rules of Section 4.4. Compared to the paper definition, the T-Rcd rule is decomposed in simpler definitions.

```
  Inductive typing : env → term → typ → Prop :=
    ...
    | T_Let :
        ∀ (E E' : env) (t1 t2 : term) (p : pat) (T1 T2 : typ),
        typing E t1 T1 → ptyping E p T1 E' → typing E' t2 T2 →
        typing E (tlet p t1 t2) T2
    | T_Rcd :
        ∀ (E : env) (t : rec) (T : trec),
        rtyping E t T → typing E (record t) (trecord T)
    | T_Proj :
        ∀ (E : env) (t1 : term) (l : lab) (T1 : trec) (T2 : typ),
        typing E t1 (trecord T1) → T1 l = Some T2 →
        typing E (proj t1 l) T2
  with rtyping : env → rec → trec → Prop :=
    | T_Rcd_Cons :
        ∀ (E : env) (l : lab) (t1 : term) (T1 : typ) (t2 : rec) (T2 : trec),
        typing E t1 T1 → rtyping E t2 T2 →
        t2 l = None → T2 l = None →
        rtyping E (rcons l t1 t2) (tcons l T1 T2)
    | T_Rcd_Nil :
        ∀ (E : env), wf_env E → rtyping E rnil tnil.
```

5.2.5 Semantics

The semantics of Section 4.5 is extended as follows. In the challenge paper, the pattern matching rules (rules M-Var and M-Rcd) specify an operator *match*($\mathtt{p}$, $\mathtt{t}_1$)$\mathtt{t}_2$, which matches term $\mathtt{t}_1$ against pattern $\mathtt{p}$ and performs the corresponding substitutions in $\mathtt{t}_2$. This operator is implemented as a function *pmatch*. An auxiliary function *prmatch* that deals specifically with the M-Rcd rule is simultaneously defined. There is one case per rule in the paper definition plus one case for failure. We maintain the following invariant: the value *t1* lives in the environment outside the pattern *p*, while the term *t2* lives in the environment containing the pattern bindings. This is exactly what is needed to perform the substitution in rule M-Var. In order to preserve this invariant, the matched value *t1* is shifted when crossing part of the pattern (call to the function *offset* is the body of function *prmatch* below). Substitution is performed from right to left, incrementally moving the term *t2* outside the bindings through substitutions. Another way to understand this is to see that the simultaneous substitution of terms t_0 to t_n in a term t can be decomposed into simple substitutions:

$$[0 \mapsto \uparrow^0_0 t_0][1 \mapsto \uparrow^0_1 t_1] \ldots [n \mapsto \uparrow^0_n t_n]t$$

where $\uparrow^0_k$ is operator $\uparrow^0$ iterated k times. The pattern matching function is only partially defined. We use the bind operator *opt_bind* of the *option* monad to handle possible failures in a concise way. One could have used an inductive definition here to avoid option types. However, one then need to use invertion tactics where one can here take advantage of the computational capabilities of Coq (see discussion in Section 3.4).

```
Fixpoint pmatch (p : pat) (t1 t2 : term) : option term :=
  match p, t1 with
  | pvar _, _ ⇒ Some (subst t2 0 t1)
  | precord p1, record r1 ⇒ prmatch p1 r1 t2
  | _, _ ⇒ None
  end
with prmatch (p : prec) (r1 : rec) (t2 : term) : option term :=
  match p with
  | pcons l p1 p2 ⇒
     opt_bind (prmatch p2 (offset (rshift 0) p1 r1) t2) (fun t2 ⇒
     opt_bind (r1 l) (fun t1 ⇒
     pmatch p1 t1 t2))
  | pnil ⇒ Some t2
  end.
```

Record values are records containing only values. Contexts are updated appropriately. There are two additional reduction rules, for pattern matching and for record projection.

```
Inductive red : term → term → Prop :=
  ...
  | E_LetV :
       ∀ (p : pat) (t1 t2 t: term),
       value t1 → pmatch p t1 t2 = Some t → red (tlet p t1 t2) t
  | E_ProjRcd :
       ∀ (l :lab) (t1 : rec) (t1' : term),
       rvalue t1 → t1 l = Some t1' → red (proj (record t1) l) t1'.
```

5.2.6 Proofs of Progress and Preservation

Several additional lemmas are required. The statement of other lemmas and theorems are unchanged, though their proofs are adjusted.

We start by some properties of typing and subtyping specific to records. Beside the weakening lemmas in Section 4.6.4 (corresponding to Lemma A.4 in the paper proof), we need an additional lemma for moving a term t under the scope of a pattern p.

Lemma *typing_weakening_ptyping* :
  $\forall$ (*E1 E2* : *env*) (*p* : *pat*) (*T U* : *typ*) (*t* : *term*),
  ptyping E1 p T E2 $\rightarrow$ *wf_typ E1 T* $\rightarrow$ *typing E1 t U* $\rightarrow$
  typing E2 (*offset* (*shift* 0) *p t*) *U*.

There are two inversion lemmas, for records, besides the one of Section 4.6.6. We did not have to prove Lemma A.12 (Inversion of subtyping) explicitly for challenge 2A, as we could rely on Coq inversion tactics instead. The case for record is less immediate and we prove the following lemma.

Lemma *record_subtyping_inversion* :
  $\forall$ (*e* : *env*) (*S* : *typ*) (*P* : *trec*),
  sub e S (*trecord P*) $\rightarrow$
  ($\exists$ *X*, *S* = *tvar X*) $\vee$
  ($\exists$ *Q*, *S* = *trecord Q* $\wedge$
    $\forall$ (*l* : *lab*) (*T* : *typ*), *P l* = *Some T* $\rightarrow$
    $\exists$ *U*, *Q l* = *Some U* $\wedge$ *sub e U T*).

We prove the third assertion of Lemma A.13, pertaining to records.

Lemma *t_record_inversion* :
  $\forall$ (*E* : *env*) (*t1* : *rec*) (*T* : *typ*) (*T1* : *trec*),
  typing E (*record t1*) *T* $\rightarrow$ *sub E T* (*trecord T1*) $\rightarrow$
  $\forall$ (*l* : *lab*) (*T2* : *typ*),
  T1 l = *Some T2* $\rightarrow$ $\exists$ *t2*, *t1 l* = *Some t2* $\wedge$ *typing E t2 T2*.

In order to prove progress (corresponding to Section 4.7), we show the second assertion of Lemma A.14 (Canonical Forms).

Lemma *record_value* :
  $\forall$ (*E* : *env*) (*t* : *term*) (*T* : *trec*),
  value t $\rightarrow$ *typing E t* (*trecord T*) $\rightarrow$
  $\exists$ *t'*, *t* = *record t'* $\wedge$ $\forall$ (*l* : *lab*), *t' l* = *None* $\rightarrow$ *T l* = *None*.

We also show that a value of some type T can be successfully matched again any pattern of type T, that is, rule E-LetV can be applied. This crucial lemma was overlooked in the paper proof provided in the challenge paper.

Lemma *matching_defined* :
  $\forall$ (*E E'* : *env*) (*p* : *pat*) (*T1* : *typ*) (*t1 t2* : *term*),
  ptyping E p T1 E' $\rightarrow$ *value t1* $\rightarrow$ *typing E t1 T1* $\rightarrow$
  $\exists$ *t*, *pmatch p t1 t2* = *Some t*.

For the proof of preservation (Section 4.8), we show Lemma A.17 (Matched patterns preserve typing).

Lemma *matched_patterns_preserve_typing* :
  $\forall$ (*E E'* : *env*) (*p* : *pat*) (*t1 t2 t* : *term*) (*T1 T2* : *typ*),
  ptyping E p T1 E' $\rightarrow$ *typing E t1 T1* $\rightarrow$ *typing E' t2 T2* $\rightarrow$
  pmatch p t1 t2 = *Some t* $\rightarrow$ *typing E t T2*.

Finally, we prove the cases corresponding to records of Lemma A.19, showing that immediate reduction preserves typing.

`Lemma` *local_preservation_proj* :
  ∀ (*E* : *env*) (*t1* : *rec*) (*t1'* : *term*) (*l* : *lab*) (*U* : *typ*),
  typing E (*proj* (*record t1*) *l*) *U* → *t1 l* = *Some t1'* → *typing E t1' U*.
`Lemma` *local_preservation_let* :
  ∀ (*E* : *env*) (*p* : *pat*) (*t1 t2 t* : *term*) (*U* : *typ*),
  typing E (*tlet p t1 t2*) *U* → *pmatch p t1 t2* = *Some t* → *typing E t U*.

6 Challenge 3 (Testing and Animating with Respect to the Semantics)

We have not addressed challenge 3. A solution would be to explicitly write a function *step* that performs one step of the reduction, when possible, and prove this function equivalent to the reduction relation. This corresponds precisely to task 3 of the challenge. Note that the relation *red* of Section 4.5 is not suitable for computation purposes, as it is defined inductively. Besides, it is not immediate to rephrase it as a function, as to implement case *E_Ctx*, one has to revert the context application *ctx_app c t*1. The alternative reduction relation *red'* of Section 4.9 is a better starting point.

Then, one can decide whether $t \longrightarrow t'$ for two given terms t and t' (task 1) by evaluating *step t* and comparing it to *Some* t', which is immediate when using de Bruijn indices as one does not have to deal with alpha-conversion.

Finally, checking whether $t \longrightarrow^* t' \not\longrightarrow$ for two given terms t and t' (task 2) can be performed by first defining a function that iterates function *step* a bounded number of times (in Coq, one can only define terminating functions), and then apply it with larger and larger integers until a normal form is reached.

It may also be convenient to write functions that converts between terms using named variables and terms using de Bruijn indices, as reading and writing actual terms directly with de Bruijn indices is harder than using named variables.

7 Conclusion

There appears to be four main approaches for dealing with binders. The *nominal* approach [20, 24] deals explicitly with named variables. This approach makes it possible to write mechanized proofs which are very close to paper proofs. However, it depends on the development of a large framework. *Higher-order abstract syntax* [19] is a technique consisting in using the bindings of the meta-language to encode the binding structure of the object language. This is a very powerful technique, but is only supported by some specific proof assistants. We have presented *de Bruijn indices* [11]. We believe the main advantage of this approach is that this representation technique is very simple and concrete: we are clearly dealing with terms up to α-equivalence. Finally, the *locally nameless* approach [4, 13, 15–17] is a hybrid approach: de Bruijn indices are used for bound variables and names are used for free variables. With this technique, one can avoid both the use of a shifting operator and the need of renaming variables during substitution. On the other hand, one has to deal with two kind of variables and a substitution for each.

De Bruijn indices have been used in many formal proofs developments. We list the most relevant references. Shankar [22] wrote a mechanical proof of the Church-Rosser theorem using the Boyer-Moore theorem prover. His definition of substitution is similar to ours, and he proves the same interaction lemmas (see Fig. 2). Altenkirch [1, 2] proved the strong normalization of System F in LEGO. He has to deal with both term and type variables and use two different namespaces as we do. Huet [14] also wrote a mechanical proof of the Church-Rosser theorem in Coq. He uses the alternative definition of substitution presented in Section 2.4, with computational reasons in mind. Rasmussen [21] ported this proof to Isabelle, reverting to the simpler definition of substitution. Barras [6–8] uses the definition of substitution of Huet for his formalization of the Calculus of Construction in Coq. This is justified, as one of the goals is to check an efficient implementation of the calculus.

Regarding the POPLmark challenge, the solution proposed by Berghofer [9] is the closest to ours. It covers all challenges. It is also based on de Bruijn indices, but using the alternative substitution presented in Section 2.4. This is the only other formalization of a calculus with record and pattern matching based on de Bruijn indices that we know of. Patterns are handled in a way similar to ours, though the simultaneous substitution is made explicit. A single namespace is used for type and term variables. As a consequence, there is a single shifting function. On the other hand, term substitution also changes indices in types.

There are several other solutions in Coq, most notably by Leroy and Charguéraud. Leroy [15] only addresses part A of the challenges. Charguéraud [12] has experimented both de Bruijn indices and the locally nameless approach. He only addresses challenge 1A but has put an impressive amount of effort in finding formulations that makes the proofs as simple as possible. In particular, rule SA-All is normally formulated with an implicit existential quantification on variable X:

$$\frac{\mathtt{E} \vdash \mathtt{T}_1\mathtt{<:}\mathtt{S}_1 \qquad \mathtt{E}, \mathtt{X}\mathtt{<:}\mathtt{T}_1 \vdash \mathtt{S}_2\mathtt{<:}\mathtt{T}_2}{\mathtt{E} \vdash \forall \mathtt{X}\mathtt{<:}\mathtt{S}_1.\mathtt{S}_2\mathtt{<:}\forall \mathtt{X}\mathtt{<:}\mathtt{T}_1.\mathtt{T}_2} \quad \text{(SA-ALL)}$$

In the locally nameless approach, he advocates to use a universal quantification instead, with variable X ranging over all possible names but a finite set L.

$$\frac{\mathtt{E} \vdash \mathtt{T}_1\mathtt{<:}\mathtt{S}_1 \qquad \forall \mathtt{X} \notin L,\ \mathtt{E}, \mathtt{X}\mathtt{<:}\mathtt{T}_1 \vdash \mathtt{S}_2\mathtt{<:}\mathtt{T}_2}{\mathtt{E} \vdash \forall \mathtt{X}\mathtt{<:}\mathtt{S}_1.\mathtt{S}_2\mathtt{<:}\forall \mathtt{X}\mathtt{<:}\mathtt{T}_1.\mathtt{T}_2}$$

Indeed, this rule provides a stronger elimination form, and can be proved equivalent to the previous one. His proofs are short but make extensive use of proof-search tactics.

We believe the simplicity and the relatively low overhead of de Bruijn indices makes it a worthwhile approach, especially for proof developments where binders are not omnipresent, and when starting from scratch.

References

1. Altenkirch, T.: Constructions, inductive types and strong normalization. Ph.D. thesis, University of Edinburgh (1993)

2. Altenkirch, T.: A formalization of the strong normalization proof for System F in LEGO. In: Bezem, J.G.M. (ed.) Typed Lambda Calculi and Applications. LNCS, vol. 664, pp. 13–28 (1993)
3. Appel, A.W., Melliès, P.A., Richards, C.D., Vouillon, J.: A very modal model of a modern, major, general type system. In: POPL '07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 109–122. ACM, New York, NY, USA (2007). doi:10.1145/1190216.1190235
4. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 3–15. ACM, New York, NY, USA (2008). doi:10.1145/1328438.1328443
5. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: the POPLmark challenge. In: International Conference on Theorem Proving in Higher Order Logics (TPHOLs) (2005)
6. Barras, B.: Coq en coq. Rapport de Recherche 3026, INRIA (1996)
7. Barras, B.: Verification of the interface of a small proof system in coq. In: Gimenez, E., Paulin-Mohring, C. (eds.) Proceedings of the 1996 Workshop on Types for Proofs and Programs. LNCS, vol. 1512, pp. 28–45. Springer, Aussois, France (1996)
8. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7 (1999)
9. Berghofer, S.: A solution to the PoplMark challenge in Isabelle/HOL (2006). http://www.in.tum.de/~berghofe/papers/Poplmark/
10. Berghofer, S., Urban, C.: A head-to-head comparison of de bruijn indices and names. Electron. Notes Theor. Comput. Sci. **174**(5), 53–67 (2007). doi:10.1016/j.entcs.2007.01.018
11. Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. Indag. Math. **34**(5), 381–392 (1972). http://alexandria.tue.nl/repository/freearticles/597619.pdf
12. Charguéraud, A.: Working with coq on the poplmark challenge. Available electronically at http://www.chargueraud.org/arthur/research/2006/poplmark/
13. Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In: HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications, pp. 413–425. Springer, London, UK (1994)
14. Huet, G.: Residual theory in λ-calculus: a formal development. J. Funct. Program. **4**(3), 371–394 (1994)
15. Leroy, X.: A locally nameless solution to the POPLmark challenge. Research Report RR-6098, INRIA (2007). http://hal.inria.fr/inria-00123945/en/
16. McBride, C., McKinna, J.: Functional pearl: i am not a number–i am a free variable. In: Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, pp. 1–9. ACM, New York, NY, USA (2004). doi:10.1145/1017472.1017477
17. Mckinna, J., Pollack, R.: Some lambda calculus and type theory formalized. J. Autom. Reason. **23**(3), 373–409 (1999). doi:10.1023/A:1006294005493
18. Nipkow, T.: More Church–Rosser proofs (in Isabelle/HOL. J. Autom. Reason. **26**(1), 51–66 (2001). doi:10.1023/A:1006496715975
19. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 199–208. ACM, New York, NY, USA (1988). doi:10.1145/53990.54010
20. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **186**, 165–193 (2003)
21. Rasmussen, O.: The Church–Rosser theorem in Isabelle: a proof porting experiment. Tech. Rep. 364, University of Cambridge, Computer Laboratory (1995). http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-364.ps.gz
22. Shankar, N.: A mechanical proof of the Church–Rosser theorem. J. ACM **35**(3), 475–522 (1988). doi:10.1145/44483.44484. http://portal.acm.org/citation.cfm?id=44484
23. The Coq Development Team: The coq proof assistant reference manual—version V8.2 (2008). http://coq.inria.fr
24. Urban, C.: Nominal techniques in isabelle/hol. J. Autom. Reason. **40**(4), 327–356 (2008). doi:10.1007/s10817-008-9097-2
25. Urban, C., Berghofer, S., Norrish, M.: Barendregt's variable convention in rule inductions. In: CADE-21: Proceedings of the 21st International Conference on Automated Deduction, pp. 35–50. Springer, Berlin, Heidelberg (2007). doi:10.1007/978-3-540-73595-3_4

26. Vouillon, J.: Solution to the poplmark challenge in coq (2005). Available electronically at http://alliance.seas.upenn.edu/∼plclub/cgi-bin/poplmark/
27. Vouillon, J.: Polymorphic regular tree types and patterns. In: POPL ’06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 103–114. ACM, New York, NY, USA (2006). doi:10.1145/1111037.1111047
28. Vouillon, J., Melliès, P.A.: Semantic types: a fresh look at the ideal model for types. In: POPL ’04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 52–63. ACM, New York, NY, USA (2004). doi:10.1145/964001.964006