

Mechanizing Type Environments in weak HOAS

Alberto Ciaffaglione, Ivan Scagnetto

*Università di Udine
Dipartimento di Matematica e Informatica
via delle Scienze, 206 - 33100 Udine, Italia*

Abstract

We provide a paradigmatic case study, about the formalization of System $F_{<}$'s type language in the proof assistant **Coq**. Our approach relies on *weak HOAS*, for the sake of producing a readable and concise representation of the object language. Actually, we present and discuss two encoding strategies for *typing environments* which yield a remarkable influence on the whole formalization. Then, on the one hand we develop System $F_{<}$'s metatheory, on the other hand we address the equivalence of the two approaches internally to **Coq**.

Keywords: Type Theory, Logical Frameworks, HOAS, POPLmark Challenge

1. Introduction

Encoding in a sound way an object language and developing its metatheory are not the only goals in the field of Computer-Aided Formal Reasoning. In fact, if the formal representation of a system is too cumbersome or too far away from its “informal” counterpart, using the computer to prove theorems is not compelling enough for the casual user, compared to carrying out proofs with paper and pencil. Therefore, since the dawn of the first logical frameworks and proof assistants, there is an ongoing debate about different encoding techniques and tools for a convenient and “user friendly” activity of formal proof development.

Type theory-based logical frameworks (LFs) provide several useful mechanisms which are automatically made available by the underlying metalanguage: unification, pattern matching, recursive functions definition, natural deduction-style reasoning, etc. Moreover, some systems like, *e.g.*, the Edinburgh LF [1], go a little further, suggesting an encoding methodology known as *Higher-Order Abstract Syntax (HOAS)*, where the variables of the object language are identified with the metavariables of the underlying typed λ -calculus, and the binders are represented by functional constants. In this way, the basic notions of α -conversion and capture-avoiding substitution are delegated to the metalanguage of the framework, with the consequence that the resulting encodings are rather concise, elegant and reminiscent of the original counterparts on paper.

However, it is well-known that the advantages of encodings based on HOAS often thin out as soon as the proof development process starts. In particular,

this happens when one wants to reason formally about the *metatheory* of the object language, so that it is necessary to handle at the proof level some of the notions delegated to the underlying metalanguage (*e.g.*, bound variables and capture-avoiding substitution). In the literature, there is a lot of work which is devoted to recover some degree of expressivity for HOAS-based encodings, in several settings: namely, layered approaches [2], well-formedness (*a.k.a.* validity) predicates [3], nominal calculi [4], axiomatic theories [5, 6], new frameworks with built-in support for programming with HOAS [7], and so on.

In this paper, we adopt the *weak* variant [8] of the HOAS methodology and work in the inductive setting of the **Coq** proof assistant [9], to focus on a common problem encountered in encoding and reasoning formally about a wide range of formal systems: that is, the representation of a *typing environment* (*i.e.*, the data structure recording the associations between the free variables occurring in a proof and their types). In order to provide a significative, sufficiently general¹ and, we hope, compelling case study, we take as object system the *type language* of System $F_{<}$: (already used as a test-bed for the famous POPLmark Challenge [10]), and in particular we address its algorithmic subtyping.

Our *first* achievement is a weak HOAS formalization of System $F_{<}$ ’s subtyping system and the subsequent solutions to the POPLmark Challenge parts 1A (*i.e.*, reflexivity and transitivity of subtyping) and 1B (extension to the language enriched with record types). This contribution is carried out by adopting the “traditional” encoding of type environments as *lists* of pairs, with the result that the subtyping is managed via a *deep*, sequent-style encoding.

On the other hand, we observe that the type systems beneath type theory-based logical frameworks are usually given in natural deduction-style; hence, their implementations give rise to natural deduction proof systems. Since these systems help the user in finding the proof term by means of a top-down process, it may be convenient to encode also the object language by following this pattern. Moreover, a natural deduction representation favors a smoother treatment of the hypotheses used in proof developments (such as, *e.g.*, the assumptions related to the typing environment), allowing one to delegate them to the underlying metalanguage. Thus, we rephrase System $F_{<}$ ’s subtyping in natural deduction-style, by providing an alternative representation of the typing environment. Precisely, we render the typing assumptions contained in the environment by means of an auxiliary “bookkeeping” judgment [11, 12, 13, 14, 15], which simply records the existence of such assumptions. Thus, as a *second*, independent result, we prove formally that the consequent *shallow* encoding of System $F_{<}$ ’s subtyping is *adequate* (*i.e.*, both sound and complete) *w.r.t.* the deep one used in the first part to address the POPLmark Challenge 1A. Even if such a correspondence between encodings based on “explicit” and “implicit” derivation contexts seems to be known in the “folklore”, we are not aware of any mechanization within logical frameworks.

Throughout the whole formal development, we use the *Theory of Contexts*

¹From the point of view of the typical issues to be faced when mechanizing a formal system.

(*ToC*) to be able to reason formally in weak HOAS about variables, binders, etc. The *ToC*, introduced in [5] as a set of axioms about basic properties of names/variables, was proved sound by means of a categorical model [16].

Despite the peculiar nature of System $F_{<}$, we believe that our achievements about the two encoding strategies are portable to other settings as well, since their applicability is rather independent from the particular object system taken as a case study. Notice also that the role of the type environment is very significant, as it is involved in both static and dynamic properties of languages.

We consider the present work as a contribution to the ongoing *deep vs. shallow* debate raised by the seminal paper [17], where the authors introduced the dichotomy between deep and shallow approaches in the quest for the most *concise/elegant/usable/etc.* adequate encoding. Originally, a deep encoding was defined as “representing syntax as a type within a mechanized logic”. Today, the difference between the two approaches is measured according to the amount of machinery delegated to the metalanguage, *i.e.*, *how close* (how shallow), or *how far* (how deep) the encoding is *w.r.t.* the logical framework considered [18]. Thus, a “shallow encoding” aims at delegating to the framework as much as possible the notions and mechanisms of the object language. The benefit of this approach is twofold. From the practical point of view, it yields more concise and elegant encodings, freeing the user from the burden of representing and handling explicitly extra machinery. Moreover, it often offers a deeper insight on the object system itself, because it entails a “standardization process” on the object language constructs. This is indeed the case with the use of HOAS for encoding binders (and the related α -conversion and capture-avoiding substitution).

Synopsis. Sections 2.1 and 6.1 contain the presentation of material from [10], therefore can be skipped by the reader familiar with the POPLmark Challenge. In Section 2 we introduce System $F_{<}$ ’s type language on paper: its syntax, the subtyping relation (both in sequent and in natural deduction-style) and the statement of the first task of the Challenge. In Section 3 we present the deep encoding of the object system in *Coq*. We devote the Section 4 to a brief excursus about the Theory of Contexts, which we use in the formal development throughout the rest of the paper. The solution to the POPLmark Challenge 1A is described in Section 5, and extended to record types (version 1B) in Section 6. In Section 7 we discuss the limits of the deep encoding approach, paving the way to the alternative shallow encoding presented in Section 8. The internal adequacy and the tradeoffs between the two encodings are carried out in Section 9. Concluding remarks, related and future work are discussed in Section 10. A major part of the present paper is based on the publications [19, 20], while the development in *Coq* is available as a web appendix [21].

2. A paradigmatic case study: System $F_{<}$

In this section we illustrate the object system taken as case study in this paper, *i.e.*, System $F_{<}$ ’s type language, and the development of its metatheory *on paper*. In Section 2.1 we define System $F_{<}$ ’s syntax and subtyping, then we

focus on the transitivity property of the latter, which forms the core of the first task of the POPLmark Challenge [10]. In the following Section 2.2 we rephrase the Challenge by addressing System $F_{<}$ ’s *well-scoping* discipline, which is left implicit, *on purpose*, in the reference paper [10]. In the last Section 2.3 we reformulate again System $F_{<}$ ’s subtyping, by pursuing a *shallow* encoding approach, to be exploited in the final part of the paper, starting from Section 8.

We remark that the other mechanisms left implicit in [10], *i.e.*, α -conversion and capture-avoiding substitution of variables for variables, will be addressed directly through Coq’s metalanguage, and therefore discussed in later sections.

2.1. The POPLmark Challenge 1A

The first task of the Challenge focuses on System $F_{<}$ ’s *type language*, that we consider in its *pure* version in this section, *i.e.*, without record types (Challenge 1A). The syntax of *types* features variables (taken, as usual, from an infinite set of distinct symbols Var), the constant Top (the supertype of any type), functions, and bounded quantification (*i.e.*, universal types):

$$\begin{array}{llll} \text{Type : } S, T ::= & X & \text{type variable} & Top \quad \text{maximal type} \\ & S \rightarrow T & \text{function type} & \forall X<:S. T \quad \text{universal type} \end{array}$$

Universal types, which form in fact the individual characteristic of $F_{<}$, arise by combining polymorphism and subtyping: on the one hand types such as $\forall X. T$ are intended to specify the type of *polymorphic* functions; on the other hand bounded universal quantifiers such as $\forall X<:S$ carry *subtyping* constraints. Actually, the universal type $\forall X<:S. T$ has the effect of binding the occurrence of X in T , but not in S . The *type environments* are formed by subtyping constraints too, involving type variables and types:

$$\begin{array}{ll} \text{Env : } \Gamma ::= & \emptyset \quad \text{empty type environment} \\ & \Gamma, X<:T \quad \text{type variable binding} \end{array}$$

Type variables within environments have to respect a *scoping* discipline: only fresh variables can be introduced, that is, $X \notin \text{dom}(\Gamma)$; moreover, such variables cannot occur free in the type they are bound to, *i.e.*, $X \notin \text{fv}(T)$; finally, the variables that appear free in T must be already collected in the environment Γ . Hence, a typical two-variable well-scoped environment is, *e.g.*, $X<:Top, Y<:X$ (notice that we will give formal definitions in the next Section 2.2).

Algorithmic subtyping $\Gamma \vdash S<:T$, *i.e.*, “ S is a subtype of T under assumptions Γ ”, captures the intuition that an instance of S may be safely used wherever an instance of T is expected. It is defined by induction and it is intended to concern only *well-scoped* types (*i.e.*, when $\Gamma \vdash S<:T$ is derived, all the type variables that occur free in S and T have to be in the domain of Γ):

$$\begin{array}{c} \overline{\Gamma \vdash S <: Top} \text{ (Top)} \quad \overline{\Gamma \vdash X <: X} \text{ (Refl)} \quad \frac{X <: U \in \Gamma \quad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \text{ (Trans)} \\ \\ \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \text{ (Arr)} \quad \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma, X<:T_1 \vdash S_2 <: T_2}{\Gamma \vdash \forall X<:S_1. S_2 <: \forall X<:T_1. T_2} \text{ (All)} \end{array}$$

The Challenge focuses on the algorithmic version of subtyping because its ultimate goal is the experimentation of *implementations* of the formalized definitions. Actually, being syntax-directed, algorithmic subtyping is easier *to reason with* than its equivalent, more familiar *declarative* presentation, where the rules (Refl) and (Trans) are replaced by the following ones:

$$\frac{X <: U \in \Gamma}{\Gamma \vdash X <: U} \quad (1) \quad \frac{}{\Gamma \vdash S <: S} \quad (2) \quad \frac{\Gamma \vdash S <: T \quad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (3)$$

In fact, the first task of the Challenge addresses the relationship between the two subtyping versions, as it consists to prove that the *transitivity* property (3) is a derivable property within the algorithmic system (the same is required for reflexivity (2), a goal which is less problematic, though).

The proof of the transitivity is challenging essentially in two respects: it has to be proved *together* with the *narrowing* property, and the whole proof requires a *mutual and nested induction* proof argument².

Proposition 1 (Transitivity). *Let be $\Gamma \in Env$, and $S, Q, T, X, M, N, P \in Type$:*

- 1) *If $\Gamma \vdash S <: Q$ and $\Gamma \vdash Q <: T$, then $\Gamma \vdash S <: T$.*
- 2) *If $\Gamma, X <: Q, \Delta \vdash M <: N$ and $\Gamma \vdash P <: Q$, then $\Gamma, X <: P, \Delta \vdash M <: N$.*

Proof. 1), 2) are proved together, by induction on the structure of the type Q .

1) The proof for *transitivity* proceeds by an inner induction on the structure of the derivation $\Gamma \vdash S <: Q$, with a case analysis on the final rule of such a derivation and on that of the second hypothesis $\Gamma \vdash Q <: T$. We illustrate the crucial case when both the derivations end with an application of the (All) rule (where it holds $S \equiv \forall X <: S_1.S_2$, $Q \equiv \forall X <: Q_1.Q_2$, $T \equiv \forall X <: T_1.T_2$):

$$\frac{\frac{\vdots}{\Gamma \vdash Q_1 <: S_1} \quad \frac{\vdots}{\Gamma, X <: Q_1 \vdash S_2 <: Q_2}}{\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: Q_1.Q_2} \quad (All) \quad \frac{\frac{\vdots}{\Gamma \vdash T_1 <: Q_1} \quad \frac{\vdots}{\Gamma, X <: T_1 \vdash Q_2 <: T_2}}{\Gamma \vdash \forall X <: Q_1.Q_2 <: \forall X <: T_1.T_2} \quad (All)$$

To conclude $\Gamma \vdash \forall X <: S_1.S_2 <: \forall X <: T_1.T_2$ via the (All) rule, two premises are needed: first, $\Gamma \vdash T_1 <: S_1$ may be derived by induction hypothesis from the third and the first subderivations; however, the induction hypothesis cannot be applied to the second and fourth ones (to deduce $\Gamma, X <: T_1 \vdash S_2 <: T_2$), because their environments are different. Hence, the narrowing property, *i.e.*, the *outer* induction hypothesis (being Q_1 *structurally smaller* than Q) has to be exploited, to derive $\Gamma, X <: T_1 \vdash S_2 <: Q_2$ from the second and the third subderivations. To construct the required derivation $\Gamma, X <: T_1 \vdash S_2 <: T_2$ from this last hypothesis and the fourth subderivation, it is necessary to apply again the *outer* induction hypothesis (*i.e.*, the transitivity itself, with Q_2 structurally smaller than Q).

2) Similarly, the proof for *narrowing* proceeds by an inner induction on the structure of the derivation $\Gamma, X <: Q, \Delta \vdash M <: N$, again with a case analysis on

²The proof of transitivity is reported in [10, 22], albeit not in a fully detailed fashion.

the final rule applied. The treatment of this “twin” property is even subtler when the last rule applied is (Trans), and M is exactly X :

$$\frac{\displaystyle \frac{\vdots}{\Gamma, X <: Q, \Delta \vdash Q <: N}}{\Gamma, X <: Q, \Delta \vdash M \equiv X <: N} \text{ (Trans)}$$

Now, $\Gamma, X <: P, \Delta \vdash Q <: N$ is derived by induction hypothesis, and $\Gamma, X <: P, \Delta \vdash P <: Q$ via a straightforward *weakening* property. This time, the *outer* induction hypothesis has to be exploited *with the same* Q ; that is, the transitivity property is used to deduce $\Gamma, X <: P, \Delta \vdash P <: N$ from the two inferred derivations. In the end, an application of the (Trans) rule allows to obtain $\Gamma, X <: P, \Delta \vdash X <: N$. \square

2.2. System $F_{<}$ in sequent-style

We give now an alternative presentation of System $F_{<}$ ’s subtyping, by making explicit the *scoping* discipline, which is implicit in the formulation [10] reported in the previous Section 2.1. While carrying out this step, we are mainly inspired by the features provided by logical frameworks based on type theory.

On the one hand, we keep using the same syntax for *types* of Section 2.1; on the other hand, we perform small changes on the *subtyping* system, and we prove that the new version is equivalent to the original one. Afterwards, we update the statement and the proof of Challenge 1A. The formalization in Coq of the resulting system and metatheory will be discussed in Sections 3, 4, 5.

We manage the *type environment* as a concrete collection, made of pairs variable-type in the form $\langle X, T \rangle \in Var \times Type$; therefore, we have to capture *formally* two concepts related to the environment itself. First, we define the *closure* of types T w.r.t. environments Γ (a sort of compatibility) via the relation $closed \subseteq Type \times Env$, to state that the free variables of T have to appear in the domain of Γ . Second, the *well-formedness* of environments $ok \subseteq Env$ prescribes that, when a new pair $\langle X, T \rangle$ makes an environment Γ grow, X must both be fresh w.r.t. Γ and not appear in T , and T has to be closed w.r.t. Γ . In what follows, we write $fv(T)$ for the type variables occurring free in a type T , and overload the symbols “ \in , \notin ” in a way which is clear from the context.

Definition 1 (Closure, Well-formedness). *For $\Gamma = \langle X_1, T_1 \rangle, \dots, \langle X_n, T_n \rangle \in Env$, $T \in Type$, the domain of Γ and the predicates $closed$, ok are defined as follows:*

$$\begin{aligned} dom(\Gamma) &\triangleq \{X_1, \dots, X_n\} & closed(T, \Gamma) &\triangleq \forall Y. Y \in fv(T) \Rightarrow \exists U. \langle Y, U \rangle \in \Gamma \\ \frac{}{ok(\emptyset)} & (ok \cdot \emptyset) & \frac{ok(\Gamma) \quad X \notin dom(\Gamma) \quad closed(T, \Gamma)}{ok(\Gamma, \langle X, T \rangle)} & (ok \cdot pair) \end{aligned}$$

Notice that we do not need the condition $X \notin fv(T)$ among the premises of the (*ok*·pair) rule, because it can be derived from the second and the third hypotheses. Finally, the main subtype judgment $\Gamma \vdash S <: T$ is rendered as $sub(\Gamma, S, T)$, where sub is a predicate defined on 3-tuples, $sub \subseteq Env \times Type \times Type$.

Definition 2 (Subtyping). Assume $\Gamma \in Env$, $S, T, U, S_1, S_2, T_1, T_2, X, Y \in Type$. Then, the predicate *sub* is defined by induction, as follows³:

$$\begin{array}{c}
\frac{ok(\Gamma) \quad closed(S, \Gamma)}{sub(\Gamma, S, Top)} (top) \quad \frac{ok(\Gamma) \quad \langle X, U \rangle \in \Gamma}{sub(\Gamma, X, X)} (var) \\
\\
\frac{\langle X, U \rangle \in \Gamma \quad sub(\Gamma, U, T)}{sub(\Gamma, X, T)} (trs) \quad \frac{sub(\Gamma, T_1, S_1) \quad sub(\Gamma, S_2, T_2)}{sub(\Gamma, S_1 \rightarrow S_2, T_1 \rightarrow T_2)} (arr) \\
\\
\frac{sub(\Gamma, T_1, S_1) \quad ok(\Gamma, \langle Y, T_1 \rangle) \Rightarrow sub((\Gamma, \langle Y, T_1 \rangle), S_2\{Y/X\}, T_2\{Y/X\})}{sub(\Gamma, \forall X <: S_1.S_2, \forall X <: T_1.T_2)} (all)
\end{array}$$

It is apparent that we have obtained a *sequent-style* encoding of subtyping, i.e., a formal system whose set of inference rules manipulate *derivation assertions* made of *premises* (Γ) and *conclusions* ($S <: T$). It is also immediate that our presentation of subtyping is equivalent to the original one of Section 2.1: informally arguing, we observe that we are using the *same* type environments and that we have formalized their *well-formedness* and a kind of *compatibility* between them and the types. In fact, we have enriched the subtyping rules with side-conditions to make them fully formal, as required by the Challenge [10].

Now we can address the Challenge 1A (Proposition 2 below), ensuring that our version of subtyping fulfills the required properties. Notice that it is necessary to add two premises to the reflexivity statement *w.r.t.* [10], because this is a property whose proof, in absence of hypothetical subtyping derivations, is carried out on the structure of a type; therefore, we have to assume the consistency of both the environment and the type considered. Before the main result we state two preliminary lemmas, to connect each other the three judgments defined in this section, and to address the environment. In the following, given an environment Γ , $perm(\Gamma)$ stands for any permutation of its components.

Lemma 1 (Auxiliary judgments). Let be $\Gamma \in Env$, and $S, T \in Type$:

- 1) $sub(\Gamma, S, T) \Rightarrow ok(\Gamma)$;
- 2) $sub(\Gamma, S, T) \Rightarrow closed(S, \Gamma) \wedge closed(T, \Gamma)$.

Proof. Both the points are proved by induction on the structure of the derivation of $sub(\Gamma, S, T)$; the proof of point 2) requires point 1). \square

Lemma 2 (Environment). Let be $\Gamma, \Delta \in Env$, and $X, P, Q, S, T \in Type$:

- 1) *Well-formedness*: $ok(\Gamma, \langle X, Q \rangle, \Delta) \wedge sub(\Gamma, P, Q) \Rightarrow ok(\Gamma, \langle X, P \rangle, \Delta)$;
- 2) *Permutation*: $sub(\Gamma, S, T) \wedge ok(perm(\Gamma)) \Rightarrow sub(perm(\Gamma), S, T)$;
- 3) *Weakening*: $sub(\Gamma, S, T) \wedge ok(\Gamma, \Delta) \Rightarrow sub((\Gamma, \Delta), S, T)$.

Proof. 1) By induction on the structure of Δ , and Lemma 1(2). 2) By induction on the derivation of $sub(\Gamma, S, T)$, and Lemma 1(1). 3) By induction on the derivation of $sub(\Gamma, S, T)$, and point 2). \square

³Notice in the (all) rule the substitution of Y for the free occurrences of X , written $\{Y/X\}$.

Proposition 2 (Challenge 1A). *Let be $\Gamma, \Delta \in Env$, $S, Q, T, X, M, N, P \in Type$:*

Reflexivity: $ok(\Gamma) \wedge closed(S, \Gamma) \Rightarrow sub(\Gamma, S, S)$.

Transitivity: $sub(\Gamma, S, Q) \wedge sub(\Gamma, Q, T) \Rightarrow sub(\Gamma, S, T)$.

Narrowing: $sub((\Gamma, \langle X, Q \rangle, \Delta), M, N) \wedge sub(\Gamma, P, Q) \Rightarrow sub((\Gamma, \langle X, P \rangle, \Delta), M, N)$.

Proof. (*Reflexivity*) By induction on the structure of S . (*Transitivity and Narrowing*) Simultaneously, by induction on the structure of Q ; we point out here some extra details *w.r.t.* Proposition 1, depending on the cases of Q .

(*Transitivity*) [$Q=Top$]: via Lemma 1(2). [$Q=Y$]: by inner induction on the derivation of $sub(\Gamma, S, Y)$. [$Q=U \rightarrow V$]: by inner induction on the derivation of $sub(\Gamma, S, U \rightarrow V)$, Lemma 1(2), and the outer induction hypothesis, *i.e.*, the transitivity statement itself, used twice with U and V , which are structurally smaller than Q . [$Q=\forall Y <: U.V$]: by inner induction on the derivation of $sub(\Gamma, S, \forall Y <: U.V)$, Lemma 1(2), and the outer induction hypothesis, this time both the narrowing statement with U and the transitivity with V , where, again, both U and V are structurally smaller than Q (see also Proposition 1).

(*Narrowing*) All the cases require an inner induction on the derivation of $sub((\Gamma, \langle X, Q \rangle, \Delta), M, N)$, and Lemmas 1(1), 2(1). When the *(trs)* rule is matched by such an induction, all the cases but the $Q=Top$ one need the application of the outer induction hypothesis, *i.e.*, the transitivity with the starting Q (see Proposition 1). Moreover, when *(trs)* is matched, the $Q=Top$ case requires the Lemma 1(2), and the other cases the Weakening property (Lemma 2(3)). \square

2.3. System $F_{<}$: in natural deduction-style

In this section we rephrase System $F_{<}$'s algorithmic subtyping of Section 2.1 in a slightly different way than in Section 2.2, by adopting a *natural deduction* encoding approach. The introduction of such a variant, as discussed in Section 1, is an attempt to pursue a (more) *shallow* representation of the object system.

As it is well-known, a formal system in natural deduction-style is defined by a set of inference rules which manipulate *conclusions*, such as \mathcal{A} , whereas sequent-style systems handle derivation assertions made of *premises and conclusions*, such as $\Gamma \vdash \mathcal{A}$. Therefore, a natural deduction formal proof of $\Gamma \vdash \mathcal{A}$ is represented by a tree whose *root* is labeled with \mathcal{A} and whose set of *leaves* form the derivation context Γ . In this section we render System $F_{<}$'s subtyping $\Gamma \vdash S <: T$ via the judgment $sub_N(S, T)$, where sub_N is a predicate defined on pairs, $sub_N \subseteq Type \times Type$. Actually, the typing environment Γ is represented through *associations* between variables and types, by means of the *bookkeeping* predicate $book \subseteq Var \times Type$, where we keep using Var as the set of variables. A suitable collection of these kinds of associations forms in fact our “global” derivation context Δ , that is intended to play the role of Γ . To address the *scoping* discipline, we add the predicate $closed_N \subseteq Type$, which states that the free variables appearing in a type T must be recorded in the context Δ :

$$closed_N(T) \triangleq \forall Y. Y \in fv(T) \Rightarrow \exists U. book(Y, U)$$

It is then possible to define System $F_{<}$'s subtyping in natural deduction-style, which actually looks more concise than in the previous Section 2.2.

Definition 3 (Subtyping, bis). Assume $S, T, U, S_1, S_2, T_1, T_2, X, Y \in \text{Type}$. In the (all_N) rule below, let $\text{fresh}(Y)$ stand for the two conditions $Y \notin \text{fv}(S_1) \cup \text{fv}(T_1) \cup (\text{fv}(S_2) \setminus \{X\}) \cup (\text{fv}(T_2) \setminus \{X\})$ and there does not exist any $V \in \text{Type}$ different from T_1 such that $\text{book}(Y, V)$ ⁴. Then, the predicate sub_N is defined by:

$$\begin{array}{c}
\frac{\text{closed}_N(S)}{\text{sub}_N(S, \text{Top})} (top_N) \quad \frac{\text{book}(X, U) \quad \text{sub}_N(U, T)}{\text{sub}_N(X, T)} (trs_N) \\
\\
\frac{\text{book}(X, U)}{\text{sub}_N(X, X)} (var_N) \quad \frac{\text{sub}_N(T_1, S_1) \quad \text{sub}_N(S_2, T_2)}{\text{sub}_N(S_1 \rightarrow S_2, T_1 \rightarrow T_2)} (arr_N) \\
\\
\frac{
\begin{array}{c}
[\text{fresh}(Y) \wedge \text{book}(Y, T_1)] \\
\vdots \\
\text{sub}_N(T_1, S_1) \quad \text{sub}_N(S_2 \{Y/X\}, T_2 \{Y/X\})
\end{array}
}{\text{sub}_N(\forall X <: S_1.S_2, \forall X <: T_1.T_2)} (all_N)
\end{array}$$

The (all_N) rule is a *conditional* one, as it depends on a premise which is formed, in turn, by a premise and a conclusion. We have written such an hypothetical premise within square brackets, according to Gentzen's original notation, to bear in mind that it must be *discharged*, *i.e.*, cancelled, in the course of a formal proof because it represents a *local* hypothesis. To convey to the reader the meaning of the (all_N) rule, we display here the derivation of $\text{sub}_N(\forall X <: \text{Top}.X, \forall X <: Z.Y)$ in the context $\text{book}(Y, \text{Top}), \text{book}(Z, Y)$ ⁵:

$$\frac{
\frac{
\frac{\text{book}(Z, Y)}{\text{closed}(Z)} (def.) \quad \frac{\text{book}(X, Z)}{[\text{book}(X, Z)]_{(1)}}
}{\text{sub}_N(Z, \text{Top})} (top_N) \quad \frac{
\frac{\text{book}(Z, Y) \quad \frac{\text{book}(Y, \text{Top})}{\text{sub}_N(Y, Y)} (var_N)}{\text{sub}_N(Z, Y)} (trs_N)
}{\text{sub}_N(X, Y)} (trs_N)
}{\text{sub}_N(\forall X <: \text{Top}.X, \forall X <: Z.Y)} (all_N)(1)$$

We can address the well-scoping by proving that, when $\text{sub}_N(S, T)$ is derived, the free variables in S and T have to be “booked” in the derivation context Δ .

Lemma 3 (Scoping). Let be $S, T \in \text{Type}$: $\text{sub}_N(S, T) \Rightarrow \text{closed}_N(S) \wedge \text{closed}_N(T)$

Proof. By induction on the derivation of $\text{sub}_N(S, T)$. □

The formalization in **Coq** of the present, natural deduction-style subtyping will be discussed in Section 8, while in Section 9 will be addressed its relationship *w.r.t.* the sequent-style subtyping version introduced in Section 2.2.

⁴Notice that we need the first condition to capture that “ \forall ” binds the occurrence of X in S_2, T_2 but not in S_1, T_1 , and the second one to guarantee the consistency of the context Δ .

⁵In natural deduction, local hypotheses are indexed with the rules they are discharged by.

3. A first encoding with explicit type environments

In this section we encode in **Coq** the *sequent-style* formulation of System $F_{<}$'s type language, that we have presented on paper in Section 2.2.

3.1. Higher-order representation of syntax and binders

To avoid an explicit representation of the notion of free and bound variables (and the related machinery of α -conversion and capture-avoiding substitution of variables for variables), we adopt a *weak HOAS* encoding approach [3, 8, 5], using a distinct non-inductive type for variables, and representing binders with constants of functional type. In the following, **Var** will be the *non-inductive* type encoding System $F_{<}$'s *variables*; hence, variables like X , Y , ... will be represented by **Coq**'s metalanguage variables **X**, **Y**, ... of type **Var**. Next, we define the *inductive* type **Tp** to encode System $F_{<}$'s *types*, with four constructors for the maximal type, variables⁶, function and universal types (see Section 2.1):

```
Parameter Var: Set.
Inductive Tp: Set := var: Var -> Tp          | top: Tp
                  | arr: Tp -> Tp -> Tp      | fa : Tp -> (Var -> Tp) -> Tp.
Coercion var: Var >-> Tp.
```

As anticipated, the “ \forall ” binder is rendered by the constructor **fa**, which is *higher-order* (as it takes as second argument a function from **Var** to **Tp**). This choice allows us to represent the binder correctly, by delegating to **Coq** the management of the bound variable X in the expression $\forall X<:S. T$. To be more precise, if we denote with **S** the encoding of S and with **T[X]** the encoding of T (where the occurrence of the bound variable **X**, corresponding to X , is explicitly denoted by the square brackets), the representation of $\forall X<:S. T$ is given by **(fa S (fun X:Var => T[X]))**. In fact, the variable **X** is bound by the metalanguage functional construct **fun**; hence, α -conversion and capture-avoiding substitution of variables for variables are automatically dealt with by **Coq**'s metalanguage.

3.2. Type environments as lists of pairs

According to the sequent-style approach to the representation of the subtyping relation, introduced in Section 2.2, *type environments* are managed as “explicit”, concrete structures. Therefore we encode them as *lists* of pairs, whose components belong to the types **Var** and **Tp**, respectively:

```
Definition envTp: Set := (list (Var * Tp)).
```

This choice is quite intuitive and natural, except for the fact that now, obviously, the encoded environments grow “toward the left” (*i.e.*, the head of the list), while environments grow on paper toward the right.

To reason about variables, types and type environments, we need to formalize the auxiliary predicates introduced in Section 2.2, *i.e.*, the (non)occurrence

⁶Notice that **var** is declared as a coercion operator, which avoids to type explicitly the constructor, where a variable should stand for a term of type **Tp**.

of variables into types, the freshness of variables/presence of pairs inside environments, and the well-scoping of types *w.r.t.* the environments themselves. First, we introduce the inductive predicates `isin`⁷ and `notin`:

```
Inductive isin (X:Var): Tp -> Prop :=
  isin_var: isin X X
| isin_arr: forall S T:Tp, isin X S \/ isin X T -> isin X (arr S T)
| isin_fa : forall S:Tp, forall U:Var->Tp, isin X S \/
  (forall Y:Var, X<>Y -> isin X (U Y)) -> isin X (fa S U).
Inductive notin (X:Var): Tp -> Prop :=
  notin_top: notin X top
| notin_var: forall Y:Var, X<>Y -> notin X Y
| notin_arr: forall S T:Tp, notin X S -> notin X T -> notin X (arr S T)
| notin_fa : forall S:Tp, forall U:Var->Tp, notin X S ->
  (forall Y:Var, X<>Y -> notin X (U Y)) -> notin X (fa S U).
```

The intuitive meaning of `(isin X T)` is that the variable `X` occurs free in `T`, $X \in fv(T)$ in Section 2.2, while `(notin X T)` stands for the opposite concept, $X \notin fv(T)$. The two definitions are syntax-driven, with just one introduction rule for each constructor of type `Tp` (apart from the `top` case for `isin`). Thus, a legitimate question is whether our encodings of `isin` and `notin` are indeed sound, *e.g.*, if `isin` is the opposite of `notin` and vice versa. Since the type of variables is open (*i.e.*, non-inductive), we will be able to carry out such a formal check in Coq once we will have introduced the Theory of Contexts (in Section 4).

In the following we will use `notin_ho`, built on top of predicate `notin`, stating that a variable does not occur in a term with a hole (see Section 4):

```
Definition notin_ho:= fun X: Var => fun S: Var->Tp =>
  forall Y: Var, X<>Y -> (notin X (S Y)).
```

Concerning the environments, we proceed by encoding the freshness of a variable $X \notin dom(\Gamma)$ (`Gfresh`), the presence of a constraint $\langle X, T \rangle \in \Gamma$ (`isinG`), and the closure of a type $closed(T, \Gamma)$ (`Gclosed`) *w.r.t.* them:

```
Inductive Gfresh (X:Var): envTp -> Prop :=
  GfVoid: Gfresh X nil
| GfGrow: forall G:envTp, forall Y:Var, forall T:Tp,
  Gfresh X G -> X<>Y -> Gfresh X (cons (Y,T) G).
Inductive isinG (X:Var) (T:Tp): envTp -> Prop :=
  checkG: forall G:envTp, forall y:Var, forall U:Tp,
  (X=y /\ T=U) \/ isinG X T G -> isinG X T (cons (Y,U) G).
Definition Gclosed (T:Tp) (G:envTp): Prop :=
  forall X:Var, (isin X T) -> exists U:Tp, isinG X U G.
```

We can then state inductively the well-formedness of environments:

⁷It is worth noticing that the clause `X<>Y` is not strictly needed in the `isin_fa` constructor in order to have a sound encoding. However, it is useful during the proof development, in particular when using inversion tactics, to provide “freshness” information about the `Y` variable used to “fill the hole” of type `U`.

```

Inductive okEnv: envTp -> Prop :=
  okVoid: okEnv nil
| okGrow: forall G:envTp, forall x:Var, forall T:Tp,
    okEnv G -> Gfresh X G -> Gclosed T G -> okEnv (cons (X,T) G).

```

3.3. Encoding the subtyping system

The representation of the subtyping relation, *sub* in Section 2.2, follows closely its counterpart on paper, apart from the constructor for the universal type `subG_fa`, which is accommodated via an hypothetical premise about a locally quantified variable, which makes the encoding higher-order:

```

Inductive subGTp: envTp -> Tp -> Tp -> Prop :=
  subG_top: forall G:envTp, forall S:Tp,
    okEnv G -> Gclosed S G -> subGTp G S top
| subG_var: forall G:envTp, forall X:Var, forall U:Tp,
    okEnv G -> isinG X U G -> subGTp G X X
| subG_trs: forall G:envTp, forall X:Var, forall U T:Tp,
    isinG X U G -> subGTp G U T -> subGTp G X T
| subG_arr: forall G:envTp, forall S1 S2 T1 T2:Tp,
    subGTp G T1 S1 -> subGTp G S2 T2 ->
    subGTp G (arr S1 S2) (arr T1 T2)
| subG_fa : forall G:envTp, forall S1 T1:Tp,
    forall S2 T2:Var->Tp, subGTp G T1 S1 ->
    (forall X:Var, okEnv (cons (X,T1) G) ->
      subGTp (cons (X,T1) G) (S2 X) (T2 X)) ->
    subGTp G (fa S1 S2) (fa T1 T2).

```

In the statement of the constructor `subG_fa`, which formalizes the (*all*) rule of Section 2.2, a new⁸ variable *X* is generated by the metalanguage, and the hypothetical premise that such *X*, coupled with *T1*, extends correctly the current environment *G*, is provided. These grant that *X* is really fresh *w.r.t.* *G*, and that *X* does not appear in *T1* (remember that *X* is not bound in *T1* in universal types such as $\forall X <: T_1. T_2$ encoded by `(fa T1 T2)`). Finally, it is necessary to bind the occurrences of *X*, potentially appearing in the second-order types *S2* and *T2*: this is grasped by instantiating *S2* and *T2* with the fresh variable *X*.

4. The Theory of Contexts

Having chosen a weak HOAS approach for the representation of System $F_{<}$'s “ \forall ” binder, we cannot rely on Coq's support for inductive types to deal

⁸The freshness of *X* is granted by the scoping rules of Coq. This means that if, by chance, *S2* or *T2* contained an occurrence of a free variable with the same name, the locally bound *X* would be automatically renamed by the system, in order to avoid captures. However, such notion of freshness, being delegated to the metalanguage, is not available at the object level. When we need to handle it within a proof development, we must explicitly add some premises using the `notin/notin.ho` predicates (see, *e.g.*, the shallow encoding of the subtyping relation in Section 8.2 and, in particular, the encoding of the universal binder rule).

with variable-related mechanisms and properties (see, *e.g.*, [8, 5]). Indeed, since binders are represented by constants of functional type, **Var** (*i.e.*, the type of variables) cannot be inductive, otherwise *exotic terms*⁹ may rise. Hence, we adopt and instantiate the *Theory of Contexts* (ToC [23, 16]), namely, a type-theoretic axiomatization which has been proposed to give a metalogical account of the fundamental notions of *variable* and *context*¹⁰ as they appear in HOAS. Remarkably, when the ToC is instantiated in a weak HOAS setting, it is still compatible with recursive and inductive environments of popular type theory-based logical frameworks and proof assistants (*e.g.*, the **Coq** system).

In the following, the expression $M[\cdot]$ will denote a *context*, *i.e.*, a term with *holes*, like, *e.g.*, $M[\cdot] \equiv (\cdot \rightarrow \cdot) \rightarrow \text{Top}$ for a context with one kind of hole with two occurrences of the latter. Then, the context $M[\cdot]$ filled in by a variable X will be denoted by $M[X] \equiv (X \rightarrow X) \rightarrow \text{Top}$ (*i.e.*, all the occurrences of the hole will be filled in by X). Contexts, like $M[\cdot]$ above, are represented in a weak HOAS setting by functional terms of type $\text{Var} \rightarrow \mathbf{T}$ (where **Var** is the type representing variables and **T** is the type representing the syntactic category of terms). Hence, the instantiation $M[X]$ is rendered as the application $(M\ X)$. We can have of course more than one kind of hole (each kind with its set of occurrences) like, *e.g.*, $N[\cdot][\cdot] \equiv \forall Y <: T. (\cdot \rightarrow *) \rightarrow (\cdot \rightarrow Y)$; in this case $N[X][Z]$ gives rise to the term $\forall Y <: T. (X \rightarrow Z) \rightarrow (X \rightarrow Y)$. Again, in a weak HOAS setting, contexts with two holes like $N[\cdot][\cdot]$ above are represented by functional terms of type $\text{Var} \rightarrow \text{Var} \rightarrow \mathbf{T}$, and the application $N[X][Z]$ is rendered as $(N\ X\ Z)$.

The notion of context can be easily extended to typing environments and, more in general, lists or sets of terms. We present now the informal intended meaning of ToC's axioms, together with their instantiation in our encoding.

Decidability of equality over variables. Given any variables X and Y , it is always possible to decide whether $X = Y$ or $X \neq Y$ (the symbol “=” stands for Leibniz equality). In our case, the instantiation in **Coq** is:

Axiom LEM_Var: forall X Y:Var, X=Y \/ X<>Y.

where **LEM** stands for *Law of Excluded Middle*¹¹.

Freshness/Unsaturation. Given any term M , there exists a variable X which does not occur free in it (*i.e.*, there are infinitely many variables). We need this property for syntactical terms of type **Tp**:

Axiom unsat: forall T:Tp, exists X:Var, notin X T.

An informal justification of this axiom is that, being the syntax of the encoded language finitary (*i.e.*, terms are built via a finite number of constructors), one

⁹Exotic terms are legal terms derivable in the LF at hand, which do not correspond to any entity of the object language. Hence, they hinder the adequacy of the encoding [3].

¹⁰Contexts are “terms with holes”, where the holes can be filled in by variables.

¹¹This is the minimal classical flavor that we require to reason about (free) occurrences of variables. Such an assumption is very close to the common practice, “on paper”, with nominal systems like, *e.g.*, process algebras or typing systems.

single term (of type Tp , in this case) can only contain a finite number of distinct variables. Hence, imagining to have an enumeration of variables (*i.e.*, a bijective correspondence between variables and natural numbers), you can always pick a fresh variable, choosing one associated to an index greater than the maximal natural number related to the free variables occurring in the term at hand.

Moreover, since we work in a setting with typed variables, we adopt also the following variant, whose pattern was introduced for the first time in [13]:

```
Axiom unsat': forall T U:Tp, exists X:Var,
  notin X T /\ notin X U /\ envBook X U.
```

where the fresh variable is required to be typed in the current environment (this is precisely the purpose of the predicate `envBook` which will be introduced in Section 8.1)¹². This axiom will be needed when we will have to deal with typing derivations in our shallow encoding of System $F_{<}$ (see Section 8). In fact, when the `sub_fa` constructor is involved (see Section 8.2) and we are required to reason by “going through” the binder `fa` (*i.e.*, we “access” the inner subterms), we have to instantiate the functional arguments and the schematic judgments over a suitably fresh variable which can be associated to the right type in an extension of the current environment. The axiom `unsat'` will provide such a variable for us, adding another appropriate `envBook`-hypothesis.

Extensionality. Two term contexts are equal if they are equal on a fresh variable; that is, if $M[X]=N[X]$ and $X \notin \text{fv}(M[\cdot]) \cup \text{fv}(N[\cdot])$, then $M=N$:

```
Axiom tp_ext: forall X:Var, forall S T:Var->Tp,
  notin_ho X S -> notin_ho X T -> (S X)=(T X) -> S=T.
```

Similarly, we need the extensionality also for environment contexts:

```
Axiom envTp_ext: forall X:Var, forall G G':Var->envTp,
  notin_envTp_ho X G -> notin_envTp_ho X G' ->
  G X = G' X -> G = G'.
```

β -expansion. It is always possible to split a term into a context applied to a variable¹³; that is, given a term M and a variable X , there is a context $N[\cdot]$ such that $N[X]=M$ and $X \notin \text{fv}(N[\cdot])$. We need the β -expansion both at the level of first-order contexts (*i.e.*, terms with one kind of hole) and at the level of second-order contexts (*i.e.*, terms with two kinds of holes):

```
Axiom tp_exp: forall S:Tp, forall X:Var,
  exists S':Var->Tp, notin_ho X S' /\ S=(S' X).
Axiom ho_tp_exp: forall S:Var->Tp, forall X:Var,
  exists S':Var->Var->Tp,
  notin_ho X (fun Y:Var => (fa top (S' Y))) /\ S=(S' X).
```

¹²If the `bottom` type (*i.e.*, the non-inhabited type representing *False*) is available, our axiom must be changed to exclude it, by enforcing `forall T U:Tp, U<>bottom -> exists X:Var...`

¹³In presence of binders, such a property is not derivable.

Starting from an idea of M. Hofmann [24], the consistency of ToC's axioms has been proved in [16], through the construction of a categorical model. As far as we know, a proof realization of the axioms in `Coq` does not exist; indeed, taking an inductive type like `nat` as the type of names/variables would give rise to "exotic terms" (*i.e.*, `Coq`'s canonical forms not corresponding to any entity of the object language), with a consequent loss of the encoding's adequacy. However, there is a mechanization of the ToC in `Isabelle/HOL` [34].

The properties formalized by the ToC have emerged from practical reasoning about process algebras, and have been proved to be quite useful in a number of situations. The scenario where they are exploited follows the general pattern of *fresh-renaming* lemmas. These allow to state that certain properties (subtyping relations in our case) are invariant under the substitution of variables with fresh ones. These kinds of properties cannot be derived in standard type theories using HOAS-based encodings, but need the use of β -expansion and extensionality.

Ultimately, the combined effect of ToC's axioms is that of recovering the capability of reasoning by *structural induction over contexts*. We explain this fact by means of an individual example, about the *monotonicity* of the predicate `isin`, which is needed in several cases within our formal development:

```
Lemma isin_mono: forall T:Var->Tp, forall X Y:Var,
  X<>Y -> isin X (T Y) -> (forall Z: Var, X<>Z -> isin X (T Z)).
```

A direct way to prove the lemma would be by higher-order induction on the structure of `T:Var->Tp`; however, `Coq` does not provide such a principle. Moreover, a naïve (*i.e.*, first-order) induction on `(T Y)` does not work, since there is no way to infer something on the structure of context `T` from the structure of `(T Y)` (notice that `Y` can occur free in `T`). Hence, we prove a preliminary lemma:

```
Lemma pre_isin_mono: forall n:nat, forall U:Tp,
  lntp U n -> forall V:Var, forall T:Var->Tp,
  notin_ho V T -> U=(T V) ->
  forall X Y:Var, X<>Y -> isin X (T Y) ->
  forall Z:Var, X<>Z -> isin X (T Z).
```

where `lntp` counts the number of constructors of type `Tp` occurring in a term:

```
Inductive lntp: Tp -> nat -> Prop :=
| lntp_top: lntp top 1
| lntp_var: forall X:Var, lntp X 1
| lntp_arr: forall T T':Tp, forall n1 n2:nat,
  lntp T n1 -> lntp T' n2 -> lntp (arr T T') (S (plus n1 n2))
| lntp_fa : forall T:Tp, forall U:Var->Tp, forall n1 n2:nat,
  lntp T n1 -> (forall X:Var, lntp (U X) n2) ->
  lntp (fa T U) (S (plus n1 n2)).
```

We introduce `lntp` because the plain induction principle, automatically provided by `Coq` for terms of type `Tp`, is not powerful enough. Indeed, the latter provides only the inductive hypothesis for proper subterms, while we need induction even on *fresh renamings* of proper subterms.

Therefore, `(lntp U n)` states that the term `U` is "built" using `n` constructors of the inductive type `Tp`. This fact allows us to argue by *complete* induction

on n in the proof of `pre_isin_mono`, thus recovering the structural information about U via inversion of the instance `(lntp U n)`. So far, we can apply β -expansion to infer the existence of a context $U' : \text{Var} \rightarrow \text{Tp}$ such that $U = (U' \ V)$, where V does not occur free in U' . Then, by applying the extensionality property, we can deduce that $T = U'$ and, since U' is not a variable but a concrete λ -abstraction, we “lift” structural information to the level of functional terms. Such an information can be finally used to solve the current goal, via rewriting.

To be more concrete, let us consider the case `(lntp U 1)`. By inverting this hypothesis, we get the case (among other ones) where the equalities $U = (T \ V)$ and $U = \text{top}$ hold. Then, by considering the context $U' = (\text{fun } X : \text{Tp} \Rightarrow \text{top})$, we can state that $(T \ V) = \text{top} = (U' \ V)$; hence we infer $(T \ V) = ((\text{fun } X : \text{Tp} \Rightarrow \text{top}) \ V)$. Finally, we “lift” such structural information to higher-order terms, via the extensionality axiom: namely, we deduce $T = (\text{fun } X : \text{Tp} \Rightarrow \text{top})$, *i.e.*, we get the structural information we need about T .

Capitalizing on the Theory of Contexts, we are now able to prove formally in `Coq` a bunch of useful lemmas about the predicates `isin` and `notin` and, in particular, that they are indeed opposite (see Section 3.2). First of all, we can prove the following *separation lemma*, stating that, given a type T , the set of variables occurring in T is disjoint from the set of variables not occurring in it:

Lemma Sep: `forall T:Tp, forall X Y:Var, isin X T -> notin Y T -> (X <> Y).`

The proof is carried out by structural induction on T using the `unsat` axiom in the case of the `fa` constructor; in this case the proof development is as follows:

```

T : Tp
t : Var -> Tp
H : forall v X Y : Var, isin X (t v) -> notin Y (t v) -> X <> Y
IHT : forall X Y : Var, isin X T -> notin Y T -> X <> Y
X : Var
Y : Var
H2 : notin Y T
H3 : forall y : Var, Y <> y -> notin Y (t y)
H1 : isin X T /\ (forall y : Var, X <> y -> isin X (t y))
=====
X <> Y

```

where $H2$ and $H3$ represent the fact that `notin Y (fa T t)` and $H1$ amounts to the hypothesis that `isin X (fa T t)`. Inverting $H1$, the first subcase (*i.e.*, when `isin X T`) is in fact trivial (a simple application of the inductive hypothesis `IHT`). Instead, the other subcase is more subtle:

```

...
H : forall v X Y : Var, isin X (t v) -> notin Y (t v) -> X <> Y
...
H3 : forall y : Var, Y <> y -> notin Y (t y)
H0 : forall y : Var, X <> y -> isin X (t y)
=====
X <> Y

```

It is apparent that, in order to use the higher-order inductive hypothesis H , we must exhibit a variable which must be *fresh w.r.t.* both X and Y . This is where the axiom `unsat` comes into play; indeed, by eliminating `(unsat (arr X Y))`, we obtain the needed fresh variable and we can conclude applying H (and then $H0, H3$). The only way to act without the `unsat` axiom would be to rephrase the lemma (and all the other statements depending on the axiom) by adding a suitable premise and stating the existence of suitably fresh variable(s)¹⁴. In our opinion, this would be very impractical and cumbersome for the user.

From the separation lemma `Sep` we can then derive the following results, characterizing the two predicates as opposite:

```
Lemma isin_not_notin: forall T:Tp, forall X:Var, isin X T -> ~notin X T.
Lemma notin_not_isin: forall T:Tp, forall X:Var, notin X T -> ~isin X T.
```

In the end, we may infer the following property, which turns out to be very handy when we consider occur-checking problems, playing the role of an instance of the “law of excluded middle”:

```
Lemma LEM_OC: forall T:Tp, forall X:Var, isin X T \/ notin X T.
```

5. A formal development of System $F_{<}$ ’s metatheory

In this section we illustrate the formal development carried out in the Coq system in order to achieve the POPLmark Challenge’s 1A task, *i.e.* reflexivity, transitivity and narrowing of subtyping (Section 2.2, Proposition 2).

5.1. Basic properties

The auxiliary lemma mostly used throughout the script is the following:

```
Lemma Gclosed_lemma: forall G:envTp, forall S T:Tp,
  subGTp G S T -> Gclosed S G /\ Gclosed T G.
```

which is in fact the internal counterpart of Lemma 1.2, proved on paper. Its intuitive meaning is that, if we derive `(subGTp G S T)` (under such an hypothesis we are able to deduce first that G is a well-formed environment, Lemma 1.1), then all the variables occurring free in S and T belong to the domain of G . The proof is carried out by induction on the derivation of `(subGTp G S T)`, using `unsatG` when we need a fresh variable in the environment G :

```
Lemma unsatG: forall G:envTp, exists X:Var, Gfresh X G.
```

As the reader may guess, the proof of `unsatG` relies heavily upon the axiom `unsat` of the ToC (see Section 4). Actually, given an environment G , the idea is just to scan the variable declaration list $(X1, T1), \dots, (Xn, Tn)$ in G , to build an arrow type `(arr X1 (arr ... (arr Xn top) ...))`. Then, by eliminating `unsat` on this type, we can get a fresh variable not occurring into such type and, consequently, not appearing in the domain of G :

¹⁴Of course, the needed inequalities and/or non-occurrence statements can vary according to the lemma at hand.

```

Lemma domGtoT_notin: forall G:envTp, forall X:Var,
  notin X (domGtoT G) -> Gfresh X G.

```

where `domGtoT` is a function, defined by recursion on `G`, which builds the mentioned arrow type from the variables that belong to its domain:

```

Fixpoint domGtoT (G:envTp):= match G with
  | nil => top | (X,T)::G' => (arr X (domGtoT G')) end.

```

The proof of `domGtoT_notin` is performed by induction on the structure of `G`, using the axiom `LEM_Var` to discriminate between the occurrences of variables.

5.2. Reflexivity, transitivity and narrowing

Coming to the pursued task 1A of the POPLmark Challenge, the *reflexivity* property requires that the type under investigation is closed *w.r.t.* the starting environment, and the latter to be well-formed (see Proposition 2):

```

Lemma reflexivity: forall T:Tp, forall G:envTp,
  okEnv G -> Gclosed T G -> subGTp G T T.

```

The proof is a straightforward induction on the structure of `T`, resorting to `LEM_Var`¹⁵ when it is needed to discriminate between free variables, and using the monotonicity of the “occurrence” predicate `isin` (see Section 4).

Transitivity and narrowing are proved together, via an outer induction on the structure of the type `Q`, which is then isolated in front of them:

```

Theorem trans_narrow: forall Q:Tp,
  (forall S:Tp, forall G:envTp,
    (subGTp G S Q) -> forall T:Tp, (subGTp G Q T) -> (subGTp G S T)) /\
  (forall G':envTp, forall M N:Tp,
    (subGTp G' M N) -> forall D G:envTp, forall X:Var, forall P:Tp,
      G'=(app D (cons (X,Q) G)) -> subGTp G P Q ->
      subGTp (app D (cons (X,P) G)) M N).

```

The proof of *transitivity* is, apart from the use of the `unsat` axiom of the ToC (to handle the `fa` constructor), similar to that on paper, via an inner induction on the derivation of `(subGTp G S Q)`. However, we have suffered a little from the lack of “smart” support for nested inductions, having to rearrange the goal statement and to enrich it with suitable equalities to purge the inconsistent cases automatically generated by the nested application of the `induction` tactic.

The *narrowing* proof is carried out by an inner induction on the derivation of `(subGTp G' M N)`, where the environment `G'` is Coq’s list `(app D (cons (X,Q) G))`, which is built by means of the `append` function `app`. We have dealt with the `subG.var` and `subG.trs` subtyping rules by means of the `LEM_Var` axiom, but this proof requires extra care *w.r.t.* its statement on paper, in two respects.

First, as its formulation involves a structured environment, it is necessary to prove some technical lemmas involving Coq’s *lists* and their relationship with

¹⁵We recall from Section 4 that `LEM_Var` is the axiom stating that given any variables `X` and `Y`, it is always possible to decide whether `X = Y` or `X ≠ Y`.

the predicates `Gfresh`, `isinG`, `Gclosed`, `okEnv`. In carrying out such proofs, we have taken partial advantage of `Coq`'s built-in list library, especially about *permutations*, which are required by the Weakening property (Lemma 2.3).

Second, to master the sophisticated interdependence between the outer and the inner structural inductions, we have exploited a slight elaboration of “modus ponens”: $\forall A, B : Prop. A \wedge (A \Rightarrow B) \Rightarrow A \wedge B$ (where A and B are intended to play the role of transitivity and narrowing, respectively). In fact, when the inner induction hypothesis for narrowing matches the rule `subG_trs`, the outer induction hypothesis (*i.e.*, transitivity) has to be applied with the *starting* Q , not with a structurally smaller type. Therefore, to handle the involved cases within the outer induction (all but the $Q=top$ one), we reduce to prove the transitivity alone and the narrowing with the proof context enriched by the transitivity additional hypothesis, instead of merely splitting the two main proofs.

6. Records in System $F_{<}$

In this section we upgrade our solution to the Challenge 1A to deal with *record types*, *i.e.*, we address the Challenge 1B, first on paper and then in `Coq`.

6.1. Records in [10]

Record types are formed by zero or an arbitrary, finite number of pairs:

$$Type : S, T ::= \dots \{l_i : T_i\}^{i \in 1..n} \quad \text{record type } (l_i \text{ distinct, } n \in \mathbb{N})$$

Algorithmic subtyping of Section 2.1 is so augmented with a corresponding rule:

$$\frac{\{l_i\}^{i \in 1..n} \subseteq \{k_j\}^{j \in 1..m} \quad \text{if } k_j = l_i \text{ then } \Gamma \vdash S_j <: T_i}{\Gamma \vdash \{k_j : S_j\}^{j \in 1..m} <: \{l_i : T_i\}^{i \in 1..n}} \text{ (Rcd)}$$

6.2. Adding records on paper

We extend now our sequent-style formulation of System $F_{<}$'s subtyping presented in Section 2.2. To cope formally with record types, we have to ensure that their *labels* are pairwise distinct, by means of the (well-formation) predicate $wt \subseteq Type$. Then, the subtyping relation *sub* can be completed accordingly.

Definition 4 (Record labels). *For $\Gamma = \langle X_1, T_1 \rangle, \dots, \langle X_n, T_n \rangle \in Env$, $T \in Type$, the predicate *wt* is defined by induction, as follows:*

$$\begin{array}{l} \frac{}{wt(Top)} (wt\cdot top) \quad \frac{wt(S) \quad wt(T)}{wt(S \rightarrow T)} (wt\cdot arr) \quad \frac{wt(S) \quad wt(T)}{wt(\forall X <: S. T)} (wt\cdot all) \\[10pt] \frac{}{wt(X)} (wt\cdot var) \quad \frac{distinct\{l_1, \dots, l_n\} \quad \text{for all } i \in 1..n, wt(T_i)}{wt(\{l_i : T_i\}^{i \in 1..n})} (wt\cdot rcd) \end{array}$$

Definition 5 (Subtyping, with records). Assume $\Gamma \in Env$, $I=1..n$, $J=1..m$, $T_j, S_i \in Type \ \forall j \in J, i \in I$. The predicate *sub* of Definition 2 is augmented with:

$$\frac{ok(\Gamma) \quad closed(S, \Gamma) \quad wt(S) \quad \{l_i\}^{i \in I} \subseteq \{k_j\}^{j \in J} \quad (k_j = l_i) \Rightarrow sub(\Gamma, S_j, T_i)}{sub(\Gamma, S \equiv \{k_j : S_j\}^{j \in J}, T \equiv \{l_i : T_i\}^{i \in I})} \text{ (rcd)}$$

Notice that in this *formal* rule for record types (rcd) we have added three premises *w.r.t.* its above formulation (Rcd). The first two ones are necessary to extend the scope of Lemma 1 to the calculus *with* record types, while the third premise addresses the distinctness of labels. We remark that it is sufficient to require that the *closed* and *wt* conditions hold just for the “longer” record type S , being the same properties for T derivable (see Lemma 4 below).

In fact, Lemmas 1 and 2 of Section 2.2 can be promptly extended to the calculus with record types, provided the validity of the following Lemma.

Lemma 4 (Record types). Let be $\Gamma \in Env$, $S \equiv \{k_j : S_j\}^{j \in J}$, $T \equiv \{l_i : T_i\}^{i \in I} \in Type$:

- 1) $wt(S) \wedge \{l_i\}^{i \in I} \subseteq \{k_j\}^{j \in J} \Rightarrow wt(T)$;
- 2) $\{l_i\}^{i \in I} \subseteq \{k_j\}^{j \in J} \wedge (k_j = l_i \Rightarrow closed(T_i, \Gamma)) \Rightarrow closed(T, \Gamma)$.

Proof. 1), 2) By induction on the (list-like) structure of the record type T . \square

The only difference in the statement of the Challenge 1B *w.r.t.* the version without record types (Challenge 1A, Proposition 2) concerns reflexivity. To accommodate it with record types, we have actually added a third premise, which prescribes that record types cannot contain doubled labels.

Proposition 3 (Challenge 1B). Let be $\Gamma, \Delta \in Env$, $S, Q, T, X, M, N, P \in Type$:

Reflexivity: $ok(\Gamma) \wedge closed(S, \Gamma) \wedge wt(S) \Rightarrow sub(\Gamma, S, S)$.

Transitivity: $sub(\Gamma, S, Q) \wedge sub(\Gamma, Q, T) \Rightarrow sub(\Gamma, S, T)$.

Narrowing: $sub((\Gamma, \langle X, Q \rangle, \Delta), M, N) \wedge sub(\Gamma, P, Q) \Rightarrow sub((\Gamma, \langle X, P \rangle, \Delta), M, N)$.

Proof. (Reflexivity) By induction on S . The record case requires an inner induction on the structure of the (list-like) collection of its pairs. (*Transitivity and Narrowing*) Simultaneously, by induction on Q . When $Q = \{l_i : T_i\}^{i \in I}$, the transitivity requires an inner induction on the derivation of $sub(\Gamma, S, Q)$, while the narrowing on $sub((\Gamma, \langle X, Q \rangle, \Delta), M, N)$ and the use of Lemma 2(1). \square

6.3. Encoding records and subtyping

We enrich the inductive **Tp** of Section 3.1 to represent record types in **Coq**:

Definition `Lab := nat.`

Inductive `Tp: Set := ... | rcd: list (Lab * Tp) -> Tp.`

First, we define record’s labels via natural numbers, which provide us with the possibility of comparing such labels; consequently, we manage records as *lists* of pairs, formed by labels and, recursively, System $F_{<}$ ’s types (*i.e.*, terms in **Tp**).

An encoding via lists is the most intuitive and natural, even if this choice causes an immediate drawback. In fact, the recursive occurrence of types **Tp**

within lists is literally *ignored*¹⁶ by `Coq`, which therefore does not generate an appropriate induction principle for the record type constructor. In other words, when we come to prove properties by induction on the structure of types `Tp`, we cannot exploit any inductive hypothesis for the types which potentially appear, at some depth, inside records. An alternative approach, that we adopted in previous contributions [13, 14, 15], introduces a *mutual* recursive `Coq` type in place of `Tp`; in that case, all the constructors but the record one do not change, whereas the record constructor becomes a different type, say `Tp_rcd`, mutually defined with `Tp` via two constructors: one for the empty record and a second one for recursively defined records with at least one pair. Encoding the type `Tp` by mutual recursion would cause to have to carry out some proofs by *mutual induction*. This would add extra technicalities to the already challenging proofs picked out by the Challenge (that have to be performed by mutual and nested induction), with the risk of obscuring the focus of the Challenge itself.

In the present work we prefer to explore the encoding via lists, because we can delegate a major part of the formal development about records to the corresponding built-in library. Therefore, in Section 6.4 we will have to come up with a solution to the lacking of the induction principle for record types.

The representation of the subtyping *sub* (Section 3.3) is extended as follows:

```
Inductive subGTp: envTp -> Tp -> Tp -> Prop := ...
| subG_rcd: forall G: envTp, forall P Q: list (Lab*Tp),
  okEnv G -> Gclosed (rcd P) G ->
  NoDup (proj_lab P) -> incl (proj_lab Q) (proj_lab P) ->
  (forall p q:Lab*Tp, In p P /\ In q Q /\ (fst p = fst q) ->
    subGTp G (snd p) (snd q)) ->
  subGTp G (rcd P) (rcd Q).
```

The rule for records `subG_rcd` is rendered via several tools which we find already formalized in the built-in list library. `NoDup` is an inductive predicate checking whether a list does contain distinct elements; `incl` is the *set* inclusion between lists (in fact, there is no reason to adopt the “set” datatype, which would provide less benefits than the “list” datatype). `In` is a trivial list-membership (recursive) function, while `fst` and `snd` are the two destructors of the “pair” datatype. Therefore, we have to define ex novo just the straightforward recursive function `proj_lab`, which collects the list of labels from a record type.

6.4. Proving reflexivity, transitivity and narrowing

We state here some remarks about the extra difficulties that arise in upgrading the proofs in `Coq` of Section 5.2 to deal with record types.

As anticipated in the Section 6.3 above, when proving the *reflexivity* we have to face the lacking of induction principles for nested types, *i.e.*, types occurring, at any depth, in record types (the problem is caused by the encoding of records via lists, a choice that we have motivated, though). The solution we adopt, à

¹⁶These types, occurring recursively in a list, are named *nested types* by the literature.

la Chlipala [25], is to enrich the induction principle `Tp_ind`, provided by `Coq` to reason by structural induction on `Tp`, with the extra `Rcd_case` path:

```
Tp_ind: forall P: Tp->Prop, (P top) -> ...
      (forall L: list (Lab * Tp), P (rcd L)) -> forall T:Tp, P T.
Hypothesis Rcd_case: forall P:Tp->Prop, forall L:list (Lab*Tp),
      Rcd2Tp Lab Tp P L -> P (rcd L).
```

where we assume that a property `P` holds for a record type `(rcd L)` provided `P` holds for any type occurring in `L`. The `Rcd2Tp` component is a recursive function which scans the list of pairs `L` and applies the property `P` to its (right-hand) types. Therefore, for proving the reflexivity of a type in `Tp`, we do not argue through the induction principle `Tp_ind`, but via its extension `Tp_ind_ext`:

```
Lemma Tp_ind_ext: forall P:Tp->Prop, (P top) -> ...
      (forall L:list (Lab*Tp), forall p:Lab*Tp,
      In p L -> P (snd p) -> P (rcd L)) -> forall T:Tp, P T.
```

While the presence of records does not affect substantially the *narrowing* proof, it complicates the *transitivity* case. Actually, when the starting type `Q` is a record `(rcd L)`, the transitivity requires an application of the outer inductive hypothesis to any of the types appearing in `L`, which are structurally smaller than `Q`. Therefore, as for reflexivity above, we have to exploit the extended induction principle `Tp_ind_ext` (in place of `Tp_ind`) to deal with records.

In the end, the whole formal development to achieve the Challenge 1B has a size which is 60% greater than the script necessary for the Challenge 1A.

7. Practical remarks

With the formal development carried out so far in `Coq`, we have provided a `Coq` encoding of System $F_{<}$ ’s type language and a proof of the first task of the POPLmark Challenge. However, we are not completely satisfied with the mechanization of the subtyping judgment, since carrying around a representation of the *type environment* as a list is rather cumbersome. Actually, during the proofs of System $F_{<}$ ’s metatheory (we consider Challenge 1A from now on), several technical lemmas (corresponding to about the 33% of the `Coq` scripts and being larger than the script with the main proofs) have been devoted to manipulate such list, distracting us from the main theorem, as one can see in Table 1.

In the latter, the script about the properties of the Theory of Contexts (third line: 16.22 KB) should not deceive the reader into thinking that also the ToC is a substantial overhead. Indeed, all the proved properties (*e.g.*, monotonicity of `isin` and `notin`) could be assumed as axioms, like in [16], “freeing” about 8.83 KB, as their proofs are really routine. Moreover, while the derived properties about the ToC are intrinsically significant, the auxiliary lemmas about the manipulation of the type environments are really trivial properties, which would not be even mentioned in proofs with “pencil and paper”.

It is worth noticing that the choice of the logical framework is not completely transparent *w.r.t.* the issue of handling efficiently a “list-like” type environment.

Basic syntax definitions (34 lines)	1.42 KB
Subtyping encoding (50 lines)	2.37 KB
ToC (axioms, measure predicate and properties about variables like, e.g., monotonicity of <code>isin</code> and <code>notin</code> , 383 lines)	16.22 KB
Type environment properties (technical lemmas, 591 lines)	17.41 KB
Part 1A of the POPLmark Challenge (458 lines)	16.32 KB

Table 1: Coq scripts statistics.

Indeed, Gacek [31] developed in Abella a quite compact and elegant solution of the task 1A (and 2A) of the POPLmark Challenge, encoding the type environment with list-based structures and predicates.

Starting from these practical considerations, in the second part of the paper we will introduce an alternative encoding of System $F_{<}$, along the lines depicted in Section 2.3, *i.e.*, by adopting a natural deduction-encoding style, thus delegating the handling of the type environment directly to the Coq system.

8. A second encoding with implicit type environments

In this section we provide an alternative encoding of subtyping, by formalizing in Coq the *natural deduction-style* version introduced in Section 2.3.

8.1. The bookkeeping technique in Coq

The *book* predicate, introduced as a bookkeeping representation of the typing environment, is realized in Coq via the following declaration:

```
Parameter envBook: Var -> Tp -> Prop.
```

Again (as for `Var`, see Section 3.1), we define `envBook` as an *open*, *i.e.*, *non-inductive* type. This allows us to “mimic” the assumptions we make on paper, when we say “let us assume the constraints $X_1 <: T_1, \dots, X_n <: T_n$ ”, by introducing the following declarations, for suitable metavariables $X_i : \text{Var}$ and $T_i : \text{Tp}$:

```
Parameter d1:envBook X1 T1. ... Parameter dn:envBook Xn Tn.
```

The next step is to exploit the bookkeeping formalization to encode the closure of types, *i.e.*, the predicate *closed* of Section 2.3:

```
Definition closed (T:Tp): Prop := forall X:Var,
  isin X T -> exists U:Tp, envBook X U.
```

Obviously, our bookkeeping approach must be *regulated* in some way, otherwise it may be possible to introduce inconsistencies. Indeed, the user might feel free to declare “at will” new hypotheses of type `(envBook ...)` (*e.g.*, to enrich the current typing environment) and the latter could be in contradiction with some previous statements (*e.g.*, the user could end with a set of hypotheses stating that a given variable is associated to two distinct types). In fact, having delegated the representation of the typing environment to the metalanguage (via

the type (`envBook ...`)), we cannot enforce in `Coq` a reliable soundness discipline. However, we can capitalize on the approach adopted in the Edinburgh LF, where encodings were always accompanied by their *adequacy theorems* [1, 26]. In our case, we can proceed as follows.

Theorem 1. *Given an environment $\Gamma = X_1 <: T_1, \dots, X_n <: T_n$, there is a bijection between System $F_{<}$'s valid derivations of shape $\Gamma \vdash S <: T$ and canonical forms D of type $(\text{sub } S \ T)$, such that the following holds (we denote with $X_1, \dots, X_n, T_1, \dots, T_n, S, T$ the encodings of $X_1, \dots, X_n, T_1, \dots, T_n, S, T$, respectively):*

$X_1:\text{Var}, \dots, X_n:\text{Var}, H_1:(\text{envBook } X_1 \ T_1), \dots, H_n:(\text{envBook } X_n \ T_n) \vdash_{\text{Coq}} D:(\text{sub } S \ T)$

Proof. The bijection is defined by induction on System $F_{<}$'s formulas and derivations. The proof consists of a straightforward induction on derivations in System $F_{<}$ and on derivations of canonical forms of type $(\text{sub } S \ T)$ in `Coq`. \square

In this way, the bookkeeping assumptions are clearly tied to a valid environment in the object language, hence inconsistencies cannot arise. Note that we will return to this topic at the end of Section 9.3.

8.2. A shallow encoding of the subtyping system

The representation of the natural deduction-style subtyping relation, sub_N in Section 2.3, is the following (to be compared with subGTp , Section 3.3):

```
Inductive subTp: Tp -> Tp -> Prop :=
| sub_top: forall S:Tp, closed S -> subTp S top
| sub_var: forall X:Var, forall U:Tp, envBook X U -> subTp X X
| sub_trs: forall X:Var, forall U T:Tp,
  envBook X U -> subTp U T -> subTp X T
| sub_arr: forall S1 S2 T1 T2:Tp, subTp T1 S1 -> subTp S2 T2 ->
  subTp (arr S1 S2) (arr T1 T2)
| sub_fa : forall S1 T1:Tp, forall S2 T2:Var->Tp, forall L:list(Var),
  subTp T1 S1 ->
  (forall X:Var, notin X S1 -> notin X T1 -> notin_list X L ->
    notin_ho X S2 -> notin_ho X T2 ->
    (envBook X T1 -> subTp (S2 X) (T2 X))
  ) -> subTp (fa S1 S2) (fa T1 T2).
```

The most notable difference *w.r.t.* the deep encoding of Section 3.3 (apart from the obvious disappearance of the occurrences of `envTp` objects) is in the constructor `sub_fa`, which formalizes the (all_N) rule. As for the constructor `subG_fa`, a new variable X is generated by the metalanguage, but in the present case the hypothetical premises about X look quite different (see Definition 3).

First of all, we remark that now the condition of *freshness* of X *w.r.t.* the derivation context Δ is not available at the object level. Therefore, we have to state *explicitly* that X must not appear neither in the type T_1 , to which it is associated in the derivation context Δ via the bookkeeping predicate `envBook`, nor in S_1 , through the premises `notin X S1` and `notin X T1` (notice that such constraints are *implicitly* assured by the `okEnv`-premise in the deep encoding).

The same condition has to be enforced for the second-order types **S2** and **T2**, by means of the second-order non-occurrence predicate **notin_ho**.

Correspondingly, we require that the new **X** is fresh *w.r.t.* the (names of the variables distributed in the) *whole* current derivation context Δ . Again, such constraint must be explicitly stated, and we capture it by the requirement that **X** does not appear in *some* list of variables **L** (formalized via the predicate **notin_list**, defined by a straightforward induction). This approach is borrowed from the encoding of Milner’s π -calculus in [8], where the list was used in schematic judgments to ensure the freshness of the locally quantified variables *w.r.t. all* (*i.e.*, not only those actually involved in the proof) the names occurring in the global environment. In fact, **L** is intended to be supplied, in the course of a formal proof, by inspecting the Δ at hand. Of course, such a list is usually the empty one, because in most cases it suffices just to care about the variables occurring in the object terms at hand. However, there is no risk of inconsistencies, whatever list we choose, because the schematic variable **X** is locally bound after **L**, and therefore **X** differs from all variables in **L**.

The technique of using schematic judgments where the locally bound variable is taken to be fresh *w.r.t.* a finite list of variables comes as a very handy tool also in other settings, like, *e.g.*, in locally nameless encodings, where it is called *cofinite quantification* [27]. Moreover, the ultimate gist of this approach relies on the notion of *finite support* of Nominal Logic [28, 4], where terms are assumed to use only finitely many variables.

9. Internal adequacy

The “non-standard” encoding of typing environments proposed in Section 8 raises urgently the question about its *consistency*. Indeed, the possibility of declaring “at will” hypotheses of type (**envBook** **X** **T**) induces an excessive degree of freedom, with the danger of yielding an inconsistent set of assumptions (*e.g.*, two **envBook**-judgments assigning different types to the same variable). Indeed, by comparing the definition of **subGTp** (Section 3.3) with **subTp** (Section 8.2), the reader can notice that all the conditions about the typing environment in the deep encoding (*e.g.*, well-formedness) do not have a direct counterpart in the shallow representation. In fact, the bookkeeping predicate **envBook** essentially delegates to the proof environment of the metalanguage the treatment of the object language typing environment. For instance, in the **sub_fa** rule we need to enforce explicitly the freshness conditions about the quantified variable **X**, by means of the **notin**, **notin_ho** and **notin_list**¹⁷ predicates; such constraints are instead provided in the deep encoding by the hypothetical premise about the environment well-formedness. Thus, to encode a given environment assumption like, *e.g.*, $\langle X <: T \rangle \in \Gamma$, all we can do is to introduce a constant **d** of a suitable type (**envBook** **X** **T**). The latter can then be used

¹⁷As already noticed in Section 8, the list **L** may take into account any variable in the current typing environment, not only those directly involved in the judgments at hand.

and possibly discharged at some point during the proof development process, according to the usual rules of the Coq system. However, such an assumption is not structured in a datatype (such as, *e.g.*, a list) nor handled by a mechanism available at object level. Hence, the user is actually free to represent arbitrary typing environments, simply introducing new constants of `envBook`-type. This is exactly the *excessive degree of freedom* we were speaking about at the beginning of this section. Therefore, to avoid the derivation of absurdities, we need a way to impose a kind of *discipline*, *i.e.*, we must define a formal mechanism to validate or discard subtyping derivations carried out in our shallow encoding.

In this section we achieve such a goal by establishing a formal correspondence in Coq (*i.e.*, an *internal* adequacy) between the shallow encoding presented in Section 8 and the corresponding deep version introduced in Section 3. Such an adequacy amounts to the following lemmas:

```

Lemma exp2imp: forall G:envTp, forall S T:Tp,
  subGTP G S T -> (book2Prop (envTp2envBook G)) -> subTp S T.
Lemma imp2exp: forall S T:Tp, subTp S T -> forall G:envTp, okEnv G ->
  (forall X:Var, forall U:Tp, envBook X U <-> isinG X U G) -> subGTP G S T.

```

In `exp2imp`, two recursive functions are used to “translate”, respectively, the typing environment `G`, involved in the hypothetical derivation (`subGTP G S T`), into a list of `envBook`-predicate instances (`envTp2envBook: envTp -> list Prop`) and the latter into a conjunction of hypotheses of the form (`envBook X T`). Their combined effect is, given a typing environment in list form, the generation of the equivalent bookkeeping assumptions, to deduce (`subTp S T`).

Dually, in lemma `imp2exp`, we may prove (`subGTP G S T`) starting from a derivation of (`subTp S T`) (in turn, deduced from a set of `envBook`-assumptions), provided that the explicit environment `G` is well-formed (*i.e.*, (`okEnv G`) holds) and it is equivalent to the following `envBook`-assumptions:

```
forall X:Var, forall U:Tp, envBook X U <-> isinG X U G
```

The intuitive meaning of this hypothesis is that (`envBook X U`) holds *if and only if* the pair (X, U) belongs to `G`.

9.1. Completeness

The proof of lemma `exp2imp` is easily carried out by induction on the derivation of (`subGTP G S T`), with the help of the following auxiliary properties:

1. $\Gamma \vdash S <: T$ implies that Γ is well-formed:


```
forall G S T, subGTP G S T -> okEnv G
```
2. The “closedness” (`Gclosed`) of a type *w.r.t.* a typing environment Γ in the deep encoding implies the closedness (`closed`) in the shallow encoding, when the bookkeeping assumptions are generated according to Γ :


```
forall S G, Gclosed S G ->
  (book2Prop (envTp2envBook G)) -> closed S
```
3. If $\langle X, U \rangle \in \Gamma$ and we generate our bookkeeping assumptions from such a Γ , there will be one of those stating that X has type U :


```
forall G X U, isinG X U G ->
  (book2Prop (envTp2envBook G)) -> (envBook X U)
```

We can conclude that the correspondence lemma relating a subtyping derivation in the deep encoding to its shallow counterpart is straightforward to prove.

9.2. Soundness

On the other hand, deriving in `Coq` the proof of lemma `imp2exp` is definitely more challenging. Actually, passing from a derivation of $(\text{subTp } S \ T)$ and the related `envBook`-assumptions to a derivation of $(\text{subGTp } G \ S \ T)$, where the environment G is determined by the `envBook`-hypotheses and must be well-formed, requires proving a suitable collection of auxiliary lemmas about occurrences of variables, and that all the auxiliary judgments are preserved by *fresh variable-renamings*. In fact, such renamings require a *complete* induction principle on the number of constructors used in a derivation, as `Coq`'s built-in `induction` scheme is not powerful enough. As already pointed out in Section 4, the reason for this fact is that the latter provides only the inductive hypothesis for proper subterms, while we need induction even on fresh renamings of proper subterms.

To convey to the reader what we mean by “fresh variable-renaming”, we list the main auxiliary lemmas we have proved (see the scripts in [21]). In what follows, we use the metavariables $\Gamma, \Gamma' \in Env$, $U, U', T, T', S \in Type$, $X, Y, Z \in Var$.

Lemma 5 (`Gfresh_rw`). *If $\Gamma = \Gamma'[Z]$ (with Z fresh in $\Gamma'[\cdot]$), $X \neq Z$ and $X \notin \text{dom}(\Gamma)$, then for all Y (such that $X \neq Y$) $X \notin \text{dom}(\Gamma'[Y])$.*

In other words, the previous lemma states that the non-occurrence of X in $\text{dom}(\Gamma)$ is preserved by renamings of different variables. Dually, the next lemma allows us to rename variables with fresh ones, preserving the occurrence conditions in typing environments.

Lemma 6 (`isinG_rw`). *If $X \notin \text{fv}(\Gamma'[\cdot])$, $\Gamma = \Gamma'[X]$ and $\langle X, U \rangle \in \Gamma$ (for a suitable U), then for each fresh variable Y ($X \neq Y$ and $Y \notin \text{fv}(\Gamma'[\cdot])$) there exists a suitable U' such that $\langle Y, U' \rangle \in \Gamma'[Y]$.*

Fresh renamings preserve also the *closed* property w.r.t. a given environment Γ .

Lemma 7 (`Gclosed_rw`). *If $\text{closed}(T, \Gamma)$, $X \notin \text{fv}(\Gamma'[\cdot]) \cup \text{fv}(T'[\cdot])$, $\Gamma = \Gamma'[X]$, $T = T'[X]$, then $\text{closed}(T'[Y], \Gamma'[Y])$, where Y is any variable such that $X \neq Y$ and $Y \notin \text{fv}(\Gamma'[\cdot]) \cup \text{fv}(T'[\cdot])$.*

The next lemma ensures that a well-formed environment is still well-formed when we rename some variables in it.

Lemma 8 (`okEnv_rw`). *If Γ is well-formed, $X \notin \text{fv}(\Gamma'[\cdot])$ and $\Gamma = \Gamma'[X]$, then $\Gamma'[Y]$ is well-formed for all variables Y such that $X \neq Y$ and $Y \notin \text{fv}(\Gamma'[\cdot])$.*

Finally, the lemma `subGTp_rw` states that renaming the last variable in the domain of an environment used to derive a subtyping relation preserves the validity of the latter (where, of course, we must rename all the occurrences of the old variable with the new one).

Lemma 9 (subGTP_rw). *If $\text{sub}((\Gamma, \langle X, U \rangle), S[X], T[X])$ and $X \notin \text{fv}(S[\cdot]) \cup \text{fv}(T[\cdot])$, then $\text{sub}((\Gamma, \langle Y, U \rangle), S[Y], T[Y])$ for all the variables Y such that $X \neq Y$ and $Y \notin \text{fv}(S[\cdot]) \cup \text{fv}(T[\cdot])$.*

The common gist of all the previous lemmas is that variables are like placeholders; in particular, the actual name of a variable (X, Y, \dots) is not important as long as its uniqueness is preserved in the current typing environment. If this holds, then the key properties of our encoding are not affected by swapping a given variable for a new one, provided the latter is suitably fresh. This is fundamental in HOAS-based encodings, in order to deal appropriately with binders and schematic judgments in the activity of proof development.

We have proved the mentioned renaming properties either by structural induction on the environment \mathbf{G} (`Gfresh_rw`, `okEnv_rw`), or by complete induction on the number of \mathbf{Tp} -constructors (`isinG_rw`, `Gclosed_rw`), or else by complete induction on the number of `subGTP`-constructors (`subGTP_rw`), *i.e.*, by complete induction on the number of the subtyping rules used in the derivation, by means of a suitable measure judgment, following the same pattern of `lntp` in Section 4.

Obviously, the reader may find in the `Coq` script other auxiliary lemmas, but they are either mere variants of those described or very trivial properties.

9.3. Deep vs. shallow

The ultimate metatheoretical result of the previous section, *i.e.*, the `Coq` internal correspondence between our shallow and deep encodings of subtyping, is in fact not completely satisfactory under a practical perspective. Actually, if one picked out two individual types and wanted to prove that the former is a subtype of the latter, it would be nice to carry out the proof using the shallow encoding (because of the simpler handling of the typing environment) and then “to validate” such a proof by “translating” it, internally in `Coq`, to its counterpart in the deep encoding, via the `imp2exp` lemma (to ensure the consistency of the bookkeeping assumptions). Unfortunately, this is not feasible, since the second premise of the lemma cannot be demonstrated in `Coq`:

```
forall X:Var, forall U:Tp, envBook X U <-> isinG X U G
```

It is apparent that we are not able to prove such a statement, due to the presence of the universal quantifications: having delegated to `Coq`’s metalanguage the handling of typing assumptions, we cannot enumerate them at the object level.

Moreover, the nature of the `unsat`’ axiom (see Section 4) could suggest the potential presence of an infinite set of `envBook`-assumptions in our shallow proof environment¹⁸, while `envTp`-objects in a deep environment are always finite lists. However, this is not the case from a practical point of view, because in any *concrete* derivation we can apply the `unsat`’ axiom only a finite number of times (in fact, derivations of `subTp` judgments are finite inductive objects).

¹⁸We recall that the axiom states that, given any two types T and U , we can always assume a fresh (*w.r.t.* T and U) variable X of type U in the current set of `envBook`-assumptions.

This is precisely the main reason that allows us to exploit our adequacy result as a *protocol* for verifying the soundness of a *concrete* formal development carried out via the shallow encoding, just by using the two premises of the `imp2exp` lemma (the first one is `(okEnv G)`) in a different way. Since the two premises actually formalize the equivalence between the set of bookkeeping assumptions in the shallow encoding and the explicit environment `G` in the deep one, it is sufficient to build “manually” the equivalent structured environment `G` via the set of the bookkeeping assumptions *actually* used in the shallow derivation at hand, and prove that `G` is well-formed (*i.e.*, `(okEnv G)` is derivable in `Coq`).

We illustrate this insight through the example of Section 2.3, about the derivation of $sub_N(\forall X < Top.X, \forall X < Z.Y)$ from $book(Y, Top), book(Z, Y)$:

```
Lemma sampleShallow: forall Y Z:Var, envBook Y top -> envBook Z Y ->
  subTp (fa top (fun X:Var => X)) (fa Z (fun X:Var => Y)).
```

Now, it is sufficient to build the corresponding environment and prove that it is well-formed in the deep encoding (provided the involved variables are distinct):

```
Lemma envWF: forall Y Z:Var, Y <> Z -> okEnv ((Z,(var Y))::(Y,top)::nil).
```

Indeed, the careful user can act even faster, just inspecting the bookkeeping assumptions and verifying *informally*, on paper, that they correspond to a well-formed environment. The alternative is using *tout court* the deep encoding:

```
Lemma sampleDeep: forall Y Z:Var, Y <> Z ->
  subGTp ((Z,(var Y))::(Y,top)::nil)
    (fa top (fun X:Var => X)) (fa Z (fun X:Var => Y)).
```

The proof of this goal, compared to the one carried out for the shallow encoding, has the following drawbacks. Obviously, addressing formally the well-formedness issue (which may occur more than once per proof) cannot be skipped. Second, looking for a variable-type association requires to scan the list-like environment (an operation which has linear complexity), whereas in the shallow case one is allowed to pick out the right assumption directly, by means of trivial tactics (*i.e.*, `assumption` in `Coq`).

However, the simplicity of the shallow approach has a drawback; actually, the bookkeeping predicate can represent only *one* “global” typing environment at a time. This is not sufficient, *e.g.*, in the narrowing property (see Section 2.1), where one must deal first with an environment where $X <: Q$ and then with the same environment where $X <: P$. Obviously, $P \neq Q$ would lead to two distinct assumptions, namely, `(envBook X Q)` and `(envBook X P)`, which is inconsistent. In fact, one could resort to hypothetical judgments in order to limit the scope of such conflicting `envBook`-assumptions, but this would raise other issues which, for instance, prevent one to argue by induction on `subTp`-derivations. The ultimate reason of the mentioned problems is that the narrowing property is strictly tied to a manipulation of the type environment; hence, delegating the latter to the metalanguage does not allow to reason about it at the object level.

Further discussion. Despite the “practical” considerations addressed above, the most serious criticism which can be raised against our shallow encoding is that it does not seem to be *independent* from the deep one. So far indeed, the only apparent formal way to ensure the consistency of a subtyping derivation in the shallow approach is to relate the bookkeeping assumptions to a well-formed environment in the deep approach. However, this is not the case, since this fact can be viewed as the “formal” counterpart of the *adequacy* Theorem 1 we have introduced in Section 8.1. In fact, adequacy theorems are common practice, according to [1], when using *open judgments*¹⁹ and encodings in natural deduction-style, in order to relate in a sound and complete way object languages and their representations in the logical framework at hand. It is clear that, with Theorem 1, the consistency of derivations in the logical framework is strictly tied to a syntactic check on the proof term D ; namely, the only variables of type `(envBook ...)` occurring free in the latter must be among those representing the valid typing environment on paper. In other words, the user is not free to introduce at will new hypotheses without breaking the contract established by the adequacy statement. This approach is perfectly compliant with well-known encodings within Edinburgh’s LF (see, *e.g.*, the ones about first-order logic in [1]). The interested reader may also refer to [18] for a broader discussion about shallow encodings and the notion of adequacy.

On the other hand, trying to recover some form of “control” over a concept which has been delegated to the meta-level (namely, the typing environment) would mean to introduce in Coq other constructs, trying to impose a kind of discipline on bookkeeping assumptions:

1. Functionality (the same variable cannot be assigned two different types):

```
Definition funcBook (E:Var->Tp->Prop): Prop :=
  forall X:Var, forall S T:Tp, (E X S) -> (E X T) -> S=T.
```

2. Scoping discipline (see Section 2.3):

```
Definition scopeBook (E:Var->Tp->Prop): Prop :=
  forall X:Var, forall S:Tp, (E X S) ->
  forall Y:Var, (isin Y S) -> exists T:Tp, (E Y T).
```

The following step would be to render `subTp` a parametric judgment that takes a bookkeeping predicate as input and features the well-formedness of such predicate as premise of each introduction rule:

```
Definition goodBook (E:Var->Tp->Prop): Prop :=
  (funcBook E) /\ (scopeBook E).
```

```
Inductive subTp' (E:Var->Tp->Prop): Tp->Tp->Prop := ...
| subTp'_var: forall X:Var, forall U:Tp,
  (goodBook E) -> (E X U) -> (subTp' E X X) ...
```

¹⁹As already remarked in Section 8.1, our `envBook` predicate is an open judgment.

However, this approach would be too cumbersome for an interactive and user-friendly tool (especially from the point of view of a novice user), ultimately forcing one to implement the bookkeeping predicate as a list-like structure of assumptions to prove effectively the well-formedness premise. In other words, we would fall back into a “deep” rendering of the typing environment.

Concluding, we can say that the deep encoding is better suited for developing the metatheory of an object system (in this paper we have actually addressed the first task of the POPLmark Challenge), while the shallow encoding is more handy for animating and testing, *i.e.*, for addressing the interactive perspective of experimenting with the language, an activity which may become the goal once the formal properties of the object system have been guaranteed.

10. Related work and conclusion

In [13, 14, 15], the first author and his coworkers experimented with the application of the bookkeeping technique, combined with weak HOAS and the ToC, to formalize in **Coq** the type soundness of the functional and imperative Abadi and Cardelli’s ζ -calculus. The present work can be seen as an advancement, *w.r.t.* those contributions, in the following respects. First, we have formally justified the bookkeeping technique *internally* in **Coq**, by proving its adequacy *w.r.t.* the more traditional, *i.e.*, deep, representation approach. Moreover, by carrying out such an effort, we have implicitly pushed the shallow approach to its limits, pointing out that it is better suited for experimentation purposes (*e.g.*, to carry out derivations with ground terms, see Section 9.3). We leave for future work the goal of providing some automatization for the different phases of our methodology, so that it might be benefited by non-expert users too. In this direction, it could be fruitful to explore the possibility of porting the encoding to alternative environments supporting HOAS, *e.g.*, the **Abella** system.

Higher-Order Abstract Syntax. The gist of the weak HOAS approach to encode languages with binders is to reconcile the advantage of delegating to the metalanguage the representation of the binders themselves, the treatment of (free and bound) variables, the related machinery of α -conversion and *freshness* of variables with the (co)inductive features of type theories, like **Coq**, that support *traditional* functional programming. Obviously, like in all compromises, beside the points in favour (readability, elegance and conciseness of the encoding), there are some drawbacks.

First of all, we are forced to keep **Var** as an open (*i.e.*, non-inductive) type to rule out *exotic terms* and at the same time retain the induction and recursion principles automatically provided by the system. However, such principles are not extended to higher-order (*i.e.*, functional) terms; whence, ToC’s axioms allow to regain at object level the capability of reasoning about the syntactic structure of such terms. In [29], instead, exotic terms are ruled out by means of a *validity judgment*, which holds only for legal (*i.e.*, non-exotic) ones. Moreover, such a validity judgment allows the authors to generate an inductive principle for higher-order terms. It is interesting to notice that an analogous higher-order

inductive principle can be generated also from the axioms of the ToC. Indeed, a form of extensionality is also taken as a fundamental property in [29].

The second major drawback of our approach is the need to *reify* at the object level a notion which is very common in languages with binders: namely, many statements are essentially sensitive only to distinctions between variable names (*i.e.* they do not depend on the particular names themselves). This fact amounts to the notion of *equivariance*²⁰ [28]; fresh-renamings lemmas, like those in Section 9.2, provide precisely with examples of equivariant properties: actually, the need to derive them in **Coq** represents the price to pay for having delegated the handling of the (freshness of) bound variables to the metalanguage.

The use of Higher-Order Abstract Syntax has been recently revitalized thanks to a new encoding paradigm, *i.e.*, *Parametric HOAS (PHOAS)*, introduced in [6] by “merging” weak HOAS with another HOAS technique [30] that resorts to first-class polymorphism to reason about functional data structures. PHOAS is essentially weak HOAS where the global type parameter representing variables is replaced with a parameter bound locally. By adopting the PHOAS approach, we could introduce an encoding of System $F_{<}$ ’s types as $\mathbf{PTp} = \forall \mathcal{V}. * . \mathbf{tp}(\mathcal{V})$, where $\mathbf{tp}(\mathcal{V})$ is an inductive family type defined by abstracting the \mathbf{Tp} of Section 3.1 *w.r.t.* the global parameter **Var**, and the quantified variable \mathcal{V} represents type variables which can be instantiated with different values throughout a development (this form of parametricity allows one to rule out exotic terms).

On the one hand, the ability to choose ad-hoc variable types for different contexts gives PHOAS some additional power both in functional programming and in proving; on the other hand, according to the author of [6], PHOAS relies on axioms “more complicated and language-specific” than those of the ToC.

The POPLmark Challenge. A major source of comparison is supplied by the contributions submitted to the POPLmark Challenge web page [31], which collects, at the time of writing, thirteen solutions to the first task, included ours.

Berghofer’s work in **Isabelle/HOL**, and two ones in **Coq** by Charguéraud and Vouillon are based on the *pure* de Bruijn representation. The *locally nameless* encoding, an approach that keeps de Bruijn indices to represent bound variables and adopts (first-order) names to manage free ones, was first experimented in **Coq** by Leroy, then refined by Chlipala, Charguéraud, and ported to **Matita** by Ricciotti. The opposite choice of *named variables* is made by Stump, who represents in **Coq** bound variables via names and free variables via de Bruijn indices. An high-level encoding technique, introduced in **Coq** by Hirschowitz and Maggesi, exploits *nested abstract syntax* to provide a categorical perspective. We discuss now the approaches most related to the present contribution; note that another reference is provided by the POPLmark Special Issue [32].

The *full HOAS* formalization carried out by Gacek in **Abella** introduces a canonical representation of System $F_{<}$ ’s types (notice, in particular, the signature of the universal constructor “ \forall ”, named **all**):

²⁰More precisely, equivariance is a property of sentences of the form $\forall \vec{x}. \phi(\vec{x})$, *i.e.* $\forall \pi, \vec{x}. (\phi(\vec{x}) \Leftrightarrow \phi(\pi \cdot \vec{x}))$, where π is a permutation action.

`ty type. top ty. arrow ty -> ty -> ty. all ty -> (ty->ty) -> ty.`

Since variables are encoded by metavariables of type `ty`, the extra specification logic judgment `bound:ty->ty->o` has to be defined to cope with the environment assumptions, and a (simplified) environment well-formedness predicate `ctx:olist->prop` is introduced to reason about subtyping. To make structural induction on System $F_{<}$ ’s types feasible, a predicate `wfty:ty->prop` is added.

Another full HOAS encoding, performed at Carnegie Mellon University in the *Twelf* system, uses the same signature for the syntax of System $F_{<}$ ’s types:

`tp: type. ... forall: tp -> (tp->tp) -> tp.`

Again, the environment assumptions require the introduction of a distinguished judgment `assm:tp->tp->type`, but, differently from the above approach in *Abella*, there is no explicit environment; therefore, a judgment `var:tp->type` is defined to “mark” the types which play the role of variables.

Summing up, variables are represented by *Abella*’s and *Twelf*’s metavariables belonging to the types `ty` and `tp`, which are introduced to encode the syntax of System $F_{<}$ ’s types. Differently, we adopt a weak HOAS approach, by choosing a separate, non-inductive type `Var` for representing variables:

Parameter `Var: Set. Inductive Tp: Set := ... | fa: Tp -> (Var->Tp) -> Tp.`

In this way, we keep the advantage of delegating α -conversion and substitution of variables for variables to the metalanguage, while retaining *Coq*’s built-in induction principle for `Tp`. Of course, in *Abella* and *Twelf* one has the extra possibility of delegating to the metalanguage the substitution of *terms* (those inhabiting `ty` and `tp`) for variables, while we should write an ad-hoc predicate. However, this kind of substitution is not required to deal with subtyping.

Also the solution proposed by Urban and coworkers in *Isabelle/HOL*, based on the *Nominal (Logic)* datatype package, is quite related to our approach:

```

atom – decl tyvrs
nominal – datatype ty =
  | Tvar tyvrs | Top | Arrow ty ty (– → – [100, 100] 100)
  | Forall << tyvrs >> ty ty

```

In this signature of System $F_{<}$ ’s types, the variables are represented by atoms, therefore the “ \forall ” binder is encoded via the abstraction operator `<< ... >>`; this allows one to prove that α -equivalent types are equal. Then, a measure on the size of types and the notion of capture-avoiding substitution are defined.

We remark that the intrinsic concepts of *finite support* and *freshness* play in Nominal Logic (and the related proof assistant Nominal Isabelle) a role which is similar to that of occurrence (`isin`) and non-occurrence (`notin`) predicates and of the unsaturation axiom, which are bundled with our axioms of the Theory of Contexts. Indeed, variables in our approach correspond to atoms in Nominal Logic; moreover, both theories rely on the intuition that terms are finite objects; hence, a single term cannot contain all the possible variables/atoms (which are infinite), *i.e.*, terms have a finite support. Hence, the unsaturation axiom is the propositional counterpart of the *fresh* operator in Nominal Logic. Actually, this

is not fortuitous, since in [33] the relation between the intuitionistic Nominal Logic and the Theory of Contexts is clearly explained by means of a translation of terms, formulas and judgments of the former into terms and propositions of the CC^{Ind} (the type theory beneath *Coq*), via a weak HOAS encoding. It turns out that the (translation of the) axioms and rules of the intuitionistic Nominal Logic are derivable in CC^{Ind} extended with the Theory of Contexts.

References

- [1] R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *J. of the ACM* 40 (1) (1993) 143–184.
- [2] A. D. Gordon, T. Melham, Five axioms of alpha-conversion, in: *TPHOL*, vol. 1125 of *LNCS*, Springer, 173–190, 1996.
- [3] J. Despeyroux, A. Felty, A. Hirschowitz, Higher-order abstract syntax in *Coq*, in: *TLCA*, vol. 902 of *LNCS*, Springer, 124–138, 1995.
- [4] A. M. Pitts, Nominal Logic, A First Order Theory of Names and Binding, *Information and Computation* 186 (2003) 165–193.
- [5] F. Honsell, M. Miculan, I. Scagnetto, An Axiomatic Approach to Metareasoning on Nominal Algebras in HOAS, in: *ICALP*, 2076 of *LNCS*, Springer, 963–978, 2001.
- [6] A. Chlipala, Parametric Higher-order Abstract Syntax for Mechanized Semantics, in: *ICFP*, ACM, 143–156, 2008.
- [7] B. Pientka, J. Dunfield, Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description), in: *Automated Reasoning*, vol. 6173 of *LNCS*, Springer, 15–21, 2010.
- [8] F. Honsell, M. Miculan, I. Scagnetto, π -calculus in (Co)Inductive Type Theories, *Theoretical Computer Science* 253 (2) (2001) 239–285.
- [9] The Coq Development Team, The Coq Proof Assistant, version 8.4, INRIA, 2012.
- [10] B. E. Aydemir, et al., Mechanized Metatheory for the Masses: The PoplMark Challenge, in: *TPHOLs*, vol. 3603 of *LNCS*, Springer, 50–65, 2005.
- [11] R. Burstall, F. Honsell, Operational semantics in a natural deduction setting, *Logical Frameworks* (1990) 185–214.
- [12] F. Honsell, M. Miculan, A natural deduction approach to dynamic logic, in: *TYPES*, vol. 1158 of *LNCS*, Springer, 165–182, 1996.
- [13] A. Ciaffaglione, L. Liquori, M. Miculan, Reasoning on an imperative object-based calculus in Higher Order Abstract Syntax, in: *MERLIN*, 1–10, 2003.
- [14] A. Ciaffaglione, L. Liquori, M. Miculan, Imperative object-based calculi in Co-inductive Type Theories, in: *LPAR*, vol. 2850 of *LNAI*, Springer, 59–77, 2003.
- [15] A. Ciaffaglione, L. Liquori, M. Miculan, Reasoning about object-based calculi in (Co)inductive type theory and the Theory of Contexts, *J. Autom. Reasoning* 39 (1) (2007) 1–47.
- [16] A. Bucalo, F. Honsell, M. Miculan, I. Scagnetto, M. Hofmann, Consistency of the Theory of Contexts, *J. Funct. Program.* 16 (3) (2006) 327–372.

- [17] R. J. Boulton, A. D. Gordon, M. J. Gordon, J. Harrison, J. Herbert, J. Van Tassel, Experience with Embedding Hardware Description Languages in HOL, in: TPCD, vol. 10 of *IFIP Transactions*, 129–156, 1992.
- [18] F. Honsell, 25 years of formal proof cultures: Some problems, some philosophy, bright future, in: LFMTTP, ACM, 37–42, 2013.
- [19] A. Ciaffaglione, I. Scagnetto, A weak HOAS approach to the POPLmark Challenge, in: LSFA, EPTCS 113, 109–124, 2012.
- [20] A. Ciaffaglione, I. Scagnetto, Internal Adequacy of Bookkeeping in Coq, in: LFMTTP, ACM, 81–88, 2014.
- [21] A. Ciaffaglione, I. Scagnetto, The Web Appendix of this paper, <http://www.dimi.uniud.it/ciaffagl/POPLmark/index.html>, 2014.
- [22] B. C. Pierce, Types and programming languages, MIT Press, 2002.
- [23] F. Honsell, M. Miculan, I. Scagnetto, The Theory of Contexts for First Order and Higher Order Abstract Syntax, ENTCS 62, 116–135, 2001.
- [24] M. Hofmann, Semantical Analysis of Higher-Order Abstract Syntax, in: LICS, IEEE, 204–213, 1999.
- [25] A. Chlipala, Certified Programming with Dependent Types, Available at <http://adam.chlipala.net/cpdt/html/toc.html>, 2013.
- [26] A. Avron, F. Honsell, I. Mason, R. Pollack, Using Typed Lambda Calculus to Implement Formal Systems on a Machine, Journal of Automated Reasoning 9 (3) (1992) 309–354.
- [27] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich, Engineering Formal Metatheory, in: POPL, ACM, 3–15, 2008.
- [28] M. J. Gabbay, A. M. Pitts, A New Approach to Abstract Syntax with Variable Binding, Formal Aspects of Computing 13, 341–363, 2001.
- [29] J. Despeyroux, A. Hirschowitz, Higher-Order Abstract Syntax with induction in Coq, in: LPAR '94, vol. 822 of *LNCS*, 159–173, 1994.
- [30] G. Washburn, S. Weirich, Boxes Go Bananas: Encoding Higher-order Abstract Syntax with Parametric Polymorphism, in: ICFP, ACM, 249–262, 2003.
- [31] B. E. Aydemir, et al., The POPLmark Challenge Web Page, Available at <http://www.seas.upenn.edu/~plclub/poplmark/>, 2014.
- [32] B. C. Pierce, S. Weirich, Special Issue on the POPLMark Challenge, J. Autom. Reasoning 49 (3), 301–302, 2012.
- [33] M. Miculan, I. Scagnetto, F. Honsell, Translating specifications from nominal logic to CIC with the theory of contexts, in: MERLIN, 41–49, 2005.
- [34] C. Röckl, D. Hirschhoff, S. Berghofer, Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the pi-Calculus and Mechanizing the Theory of Contexts, in: FOSSACS, vol. 2030 of *LNCS*, Springer, 364–378, 2001.