Internal Adequacy of Bookkeeping in Coq

Alberto Ciaffaglione
Università di Udine
Dipartimento di Matematica e Informatica
via delle Scienze, 206 - 33100 Udine, Italia
alberto.ciaffaglione@uniud.it

Ivan Scagnetto
Università di Udine
Dipartimento di Matematica e Informatica
via delle Scienze, 206 - 33100 Udine, Italia
ivan.scagnetto@uniud.it

ABSTRACT

We focus on a common problem encountered in encoding and formally reasoning about a wide range of formal systems, namely, the representation of a typing environment. In particular, we apply the bookkeeping technique to a well-known case study (i.e., System $F_{\omega}$’s type language), proving in Coq an internal correspondence with a more standard representation of the typing environment as a list of pairs.

In order to keep the signature readable and concise, we make use of higher-order abstract syntax (HOAS), which allows us to deal smoothly with the representation of the universal binder of System $F_{\omega}$ type language.

Categories and Subject Descriptors

F.3.1 [Specifying and Verifying and Reasoning about Programs]: Mechanical verification

General Terms

Theory, Verification

Keywords

Type Theory, Logical Frameworks, Higher-Order Abstract Syntax

1. INTRODUCTION

In the field of Computer-Aided Formal Reasoning, encoding in a sound way an object language and developing formally its metatheory are not the only issues. 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 already 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 Logical Framework [15], 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 $\lambda$-calculus, and the binders are represented by functional constants. In this way, the common notions of 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, capture-avoiding substitution, etc.). In the literature, there is a lot of work devoted to recover some degree of expressivity for HOAS-based encodings in several settings, namely, layered approaches [14], well-formedness (a.k.a. validity) predicates [11], nominal calculi [24], axiomatic theories [19], and so on.

We observe that the type systems beneath type theory-based logical frameworks are usually given in natural deduction-style; whence, 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.

In this paper, we adopt the HOAS methodology in the inductive setting of the Coq proof assistant, to focus on a common problem encountered in encoding and formally reasoning about a wide range of formal systems, namely, the representation of a typing environment. We take as object system the pure type language of System $F_{\omega}$ (already used as a test-bed for the famous POPLmark Challenge [1]), and in particular we address its algorithmic subtyping. In [8, 10], we presented a first encoding of this object system, to address the POPLmark Challenge 1A; however, we were not completely satisfied by the mechanization of the subtyping judgment, since carrying around a representation of the typing environment structured as a list was rather laborious. Actually, while developing System $F_{\omega}$’s metatheory, sev-
eral technical lemmas (corresponding to the 40% of the preliminary script) were devoted only to manipulate such list, distracting us from the main goals.

Therefore, in the present paper we rephrase System $F_{\lt}$.’s subtyping in natural deduction-style, by providing an alternative representation of the typing environment. Precisely, we decide to render the typing assumptions contained in the environment by means of an auxiliary “bookkeeping” judgment $[4, 18, 22, 5, 6, 7]$, which simply records the existence of such assumptions. Moreover, to be able to reason formally in HOAS about variables, binders, etc., we employ the Theory of Contexts (ToC), introduced in [19] as a set of axioms about basic properties of names/variables, and proved sound by means of a categorical model $[3]$.

The main result of our effort is the proof in Coq that the shallow encoding of System $F_{\lt}$’s subtyping introduced in this paper is adequate (i.e., both sound and complete) w.r.t. the one that we developed by using the explicit list-machinery for representing the type environment, to address the POPLmark Challenge 1A $[8, 10]$. Therefore, we stress that the present paper has not the goal of providing a further solution to the POPLmark Challenge itself. Instead, we think that our achievement is significant independently from the particular object language taken as a case study.

We consider the present work as a contribution to the ongoing deep vs. shallow debate raised by the seminal paper $[2]$, 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 $[17]$. 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, a shallow encoding 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 machinery of $\alpha$-conversion and capture-avoiding substitution).

Synopsis.

We present in Section 2 System $F_{\lt}$’s subtyping system in natural deduction-style, and in Section 3 its shallow encoding in Coq. In Section 4 we provide a brief excursus about the ToC, which we use in the formal development of Section 5 to prove that the shallow encoding is adequate w.r.t. its deep version $[8, 10]$. In Section 6 we illustrate the benefits of our approach through a sample derivation. We conclude with related work and directions for future work. The Coq script is available at the web appendix of this paper $[9]$.

2. System $F_{\lt}$. IN NATURAL DEDUCTION

The syntax of System $F_{\lt}$’s pure type language $[1]$ features the constant $Top$ (the supertype of any type), variables (taken from an infinite set $Var$ of distinct symbols), functions, and bounded quantification (i.e., universal types):

$$Type : S, T, U, V ::= X, Y, Z \quad \text{type variable}$$
$$Top \quad \text{maximal type}$$
$$S \rightarrow T \quad \text{function type}$$
$$\forall X:\lt S. T \quad \text{universal type}$$

Universal types, which are the individual characteristic of $F_{\lt}$, 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 $\forall X:\lt S$ carry subtyping constraints. In fact, the universal type $\forall X:\lt S. T$ has the effect of binding the occurrence of $X$ in $T$, but not in $S$.

The objective of the present section is twofold. First, we rephrase on paper System $F_{\lt}$’s algorithmic subtyping, by addressing its well-scoping discipline (left implicit, on purpose, in $[1]$). While carrying out such a presentation, we adopt, moreover, a natural deduction encoding approach, in order to pursue the goals pointed out in Section 1.

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 $\Gamma$, whereas sequent-style systems actually handle derivation assertions made of premises and conclusions, such as $\Gamma \vdash \Delta$. Therefore, a natural deduction formal proof of $\Delta \vdash \Gamma$ is represented by a tree whose root is labeled with $\Delta$ and whose set of leaves form the derivation context $\Delta$ (see Section 6 for an example derivation).

To address the scoping discipline (the other mechanisms implicit in $[1]$, i.e., $\alpha$-conversion and capture-avoiding substitution), will be delegated (tout court to Coq’s metalinguage, in the next section), we add a third predicate $closed \subseteq Type$, to state that the free variables appearing in a type $T$ (written $fv(T)$) must be recorded in the context $\Delta$:

$$closed(T) \equiv \forall Y \in Var. Y \notin fv(T) \Rightarrow \exists U \in Type. book(Y, U)$$

In the end, by using the two extra judgments $book$ and $closed$, we may define System $F_{\lt}$’s subtyping in natural deduction-style, which looks more concise than in $[8]$. 

**Definition 1** (Subtyping). Let be $S, S_1, S_2, T, T_1, T_2, U \subseteq Type$, $X \in Var$. Moreover, in the (all) rule below, let $fresh(X)$ stand for the two conditions $X \notin {fv(S_1) \cup fv(T_1) \cup fv(S_2) \cup fv(T_2)}$ and there does not exist any $V \in Type$ different from $T_1$ such that $book(X, V)$. Then the predicate $sub$ is defined by induction, as follows:

$$closed(S) \quad \text{(top)} \quad book(X, U) \quad sub(U, T) \quad \text{(sub)}$$

$$\frac{}{sub(S, Top)} \quad \frac{book(X, U) \quad sub(U, T)}{sub(X, T)} \quad \text{(tr-s)}$$
The (all) 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 can be discharged, i.e., cancelled, in the course of a formal proof (because it represents a local hypothesis).

We can prove formally that our system concerns only well-scope types: if \( sub(S,T) \) is derived, all the free variables in \( S \) and \( T \) have to be “booked” in the derivation context \( \Delta \).

**Lemma 1 (Scoping).** For all \( S,T \in \text{Type} \): \( sub(S,T) \Rightarrow closed(S) \land closed(T) \). □

### 3. THE ENCODING

Now we port in Coq the natural deduction-style subtyping system introduced in the previous section: on the one hand we have to specify the internal counterparts of the concepts defined so far, on the other hand to deal formally with \( \alpha \)-conversion and capture-avoiding substitution, which (as usual) are left implicit in presentations on paper.

#### 3.1 The syntax of types

We define \( \text{Var} \) as the open (i.e., non-inductive) type representing System F\(,_'s variables: variables like \( X,Y,... \) are encoded through metalanguage variables \( x,y,... \), the only terms that can inhabit \( \text{Var} \). Then, we introduce the inductive type \( \text{Tp} \) to represent types, with constructors for the maximal type, variables\(^2\), function and universal types:

**Parameter** \( \text{Var} : \text{Set} \).

**Inductive** \( \text{Tp} : \text{Set} := \mid \text{var} : \text{Var} \rightarrow \text{Tp} \mid \text{top} : \text{Tp} \)**

This encoding, via the open type \( \text{Var} \) for variables and the inductive type \( \text{Tp} \) for terms of the object system, is in fact a weak-HOAS one [11]. In particular, the constructor \( \text{fa} \), which is higher-order (as it takes as second argument a function from \( \text{Var} \) to \( \text{Tp} \)), allows us to represent correctly System F\(,'s binder \( \forall \), 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 \( X \) is explicitly denoted by \( X \)), the representation of \( \forall X:S.T \) is given by \( \text{fa} S \ (\text{fun} X: \text{Var} \Rightarrow T X) \). Hence, the variable \( X \) is bound by the metalanguage functional constructor \( \text{fun} \); as a result, \( \alpha \)-conversion and capture-avoiding substitution of variables for variables are automatically dealt with by the metalanguage of Coq.

\(^2\)Notice that \( \text{var} \) is declared as a coercion operator, which avoids to type explicitly the constructor, where a variable should stand for a term of type \( \text{Tp} \).

To reason formally, we need occurrence/non-occurrence (the latter also second-order) of variables into types, which we represent via the inductive predicates \( \text{isin} \) and \( \text{notin} \):

**Inductive** \( \text{isin} \ (X: \text{Var}) : \text{Tp} \Rightarrow \text{Prop} := \mid \text{isin-var} : \text{isin} X X \mid \text{isin-arr} : \forall S : \text{Tp}, \text{isin} X S \land \text{isin} X T \Rightarrow \text{isin} X (arr S T) \mid \text{isin-fa} : \forall S : \text{Tp}, \forall U : \text{Var} \Rightarrow \text{Tp}, \text{isin} X S \land \text{isin} X U \Rightarrow \text{isin} X (U Y) \Rightarrow \text{isin} X (\text{fa} S U) \).**

**Inductive** \( \text{notin} \ (X: \text{Var}) : \text{Tp} \Rightarrow \text{Prop} := \mid \text{notin-top} : \text{notin} X \text{top} \mid \text{notin-var} : \forall Y : \text{Var}, \text{notin} X Y \Rightarrow \text{notin} X Y \mid \text{notin-arr} : \forall S : \text{Tp}, \text{notin} X S \Rightarrow \text{notin} X T \Rightarrow \text{notin} X (arr S T) \mid \text{notin-fa} : \forall S : \text{Tp}, \forall U : \text{Var} \Rightarrow \text{Tp}, \text{notin} X S \Rightarrow \text{notin} Y : \text{Var}, \text{notin} X Y \Rightarrow \text{notin} X (U Y) \Rightarrow \text{notin} X (\text{fa} S U) \).

**Definition** \( \text{notin-ho} := \text{fun} X : \text{Var} \Rightarrow \text{fun} U : \text{Var} \Rightarrow \text{fun} Y : \text{Var}, X \Rightarrow Y \Rightarrow \text{notin} X (U Y) \).**

The intuitive meaning of \( \text{isin} X T \) is that the variable \( X \) occurs free in \( T \in \text{Env}(T) \) in Section 2, while \( \text{notin} X T \) stands for the opposite concept, \( X \notin \text{Env}(T) \). The two definitions are syntax-driven, with just one introduction rule for each constructor of \( \text{Tp} \) (apart from the top case for \( \text{isin} \)).

#### 3.2 The subtyping relation

The **book** predicate, introduced in Section 2 as our bookkeeping representation of the typing environment, is realized in Coq via the declaration:

**Parameter** \( \text{envBook} : \text{Var} \Rightarrow \text{Tp} \Rightarrow \text{Prop} \).

Again (as for \( \text{Var} \) above), we define \( \text{envBook} \) as an open type. This allows us to “mimic” in Coq the assumptions we make on paper (when we say “let us assume the constraints \( X_1:<T_1,...,X_n:<T_n) \)” by introducing the following declarations, for suitable metavariables \( Xi: \text{Var} \) and \( Ti: \text{Tp} \):

**Parameter** \( \text{oi} : \text{envBook} X1 T1 \ldots \text{Parameter} \) \( \text{dn} : \text{envBook} Xn Tn \).

The next step is to exploit the above definitions to encode the closure of types, i.e., the predicate \( \text{closed} \) of Section 2:

**Definition** \( \text{closed} (T: \text{Tp}) : \text{Prop} := \forall S : \text{Var}, \text{isin} X T \Rightarrow \exists U : \text{Tp}, \text{envBook} X U \).**

The representation of the subtyping relation follows naturally its counterpart on paper, apart from the constructor of the universal type \( \text{sub-fa} \), which is accommodated via an hypothetical premise about a locally quantified variable, in the style of the weak-HOAS encoding approach:

**Inductive** \( \text{subTp} : \text{Tp} \Rightarrow \text{Tp} \Rightarrow \text{Prop} := \mid \text{sub-top} : \forall S : \text{Tp}, \text{closed} S \Rightarrow \text{subTp} S \text{ top} \mid \text{sub-var} : \forall S : \text{Var}, \forall U : \text{Tp}, \text{envBook} X U \Rightarrow \text{subTp} U X \mid \text{sub-trs} : \forall S : \text{Var}, \forall U : \text{Tp}, \text{envBook} X U \Rightarrow \text{subTp} U T \Rightarrow \text{subTp} X T \mid \text{sub-arr} : \forall S : \text{Var}, \forall U : \text{Tp}, \text{envBook} X U \Rightarrow \text{subTp} U T \Rightarrow \text{subTp} X T \mid \text{sub-fa} : \forall S : \text{Var}, \forall U : \text{Tp}, \text{envBook} X U \Rightarrow \text{subTp} U T \Rightarrow \text{subTp} (\text{fa} S U) \mid \text{notin-var} : \forall S : \text{Var}, \forall U : \text{Tp}, \text{isin} X (U Y) \Rightarrow \exists U : \text{Var}, \text{isin} X (U Y) \Rightarrow \text{isin} X (\text{fa} S U) \).

\[^2\]Notice that \( \text{var} \) is declared as a coercion operator, which avoids to type explicitly the constructor, where a variable should stand for a term of type \( \text{Tp} \).
Some subtle remarks about the constructor sub.fa, which formalizes the conditional (all) rule of Section 2, are in order.

First, a new variable $X$ is generated by the metalanguage, and such a name must not appear neither in the type $T_1$, to which it is associated in the derivation context $\Delta$ via the bookkeeping predicate envBook, nor in $S_1$ (remember that $X$ is not bound in $S$ within universal types such as $\forall X . S T$). Correspondingly, we must require that the new $X$ is really fresh w.r.t. the (names of the variables distributed in the) current derivation context $\Delta$. This is captured by the constraint that $X$ does not appear in some list of variables $L$ [20] (L is intended to be constructed, in the course of a formal proof, by inspecting the $\Delta$ at hand), which is formalized via the predicate notin_list (whose definition is trivial, see [9]). Finally, it is also necessary to ensure that the occurrences of $X$ are bound in $T$ within universal types such as $\forall X : S T$. This is grasped via the second-order types $S_2 T_2$. $\lambda y . \text{var}$, which in fact give rise to a family of rules that represent the derivation context. The formalization is actually completed through the second-order non-occurrence predicate notin_ho, and by instantiating the types $S_2$ and $T_2$ with the new name $X$.

4. **THE THEORY OF CONTEXTS**

Having chosen a weak-HOAS approach for the representation of System Fc’s binder $\forall$, we cannot rely on Coq’s support for inductive types to deal with variable-related mechanisms and properties (see, e.g., [20, 19]). Hence, we adopt and instantiate the Theory of Contexts (ToC [21, 3]): this is a type-theoretic axiomatization which has been proposed to give a metalogical account of the fundamental notions of variable and context\(^3\) 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 like, in the case, the Coq system.

In the following, the expression $M[\_\_]$ will denote a context, i.e., a term with holes, like, e.g., $M[\_\_] \equiv (\_ \rightarrow \_ \rightarrow \top)$ for a context with one kind of hole with two occurrences of the latter. Then, the context $M[\_\_]$ filled in by a variable $X$ will be denoted by $M[X] \equiv (X \rightarrow X) \rightarrow \top$ (this means that all the occurrences of the hole will be filled in by $X$). We can have of course more than one kind of hole (each kind with its set of occurrences) like, e.g., $N[\_\_]\equiv \forall Y . \forall T : (\_ \rightarrow *) \rightarrow (\_ \rightarrow *)$; in this case $N[X][Z]$ gives rise to the term $\forall Y . \forall T . (X \rightarrow Z) \rightarrow (X \rightarrow Y)$.

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 the ToC’s axioms, together with their instantiation in our encoding.

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\(^4\). The scenario where they are exploited follows the general pattern of renaming lemmas, which, in turn, require fresh-renaming properties. These kinds of properties cannot be derived in standard type theories using HOAS-based encodings, but need the use of $\beta$-expansion and extensionality.

Ultimately, their combined effect 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:

$$\text{Lemma isin_mono: for all } T : \forall \rightarrow \top, \forall X : \forall \rightarrow \forall \rightarrow \top, X \rightarrow Y \rightarrow \text{isin } X (T Y) \rightarrow (\text{forall } Z : \forall, X \rightarrow Z \rightarrow \text{isin } X (T Z)).$$

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

$$\begin{align*}
\text{Freshness/Unsaturation Given any term } M, \text{ there exists a variable } X \text{ which does not occur free in it (i.e., there are infinite variables). We need this property for syntactical terms of type } T_1:
\end{align*}$$

$$\begin{align*}
\text{Axiom unsat: for all } T : T_1, \exists X : \forall, \text{ notin } X T.
\end{align*}$$

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 [5]:

$$\begin{align*}
\text{Axiom unsat': for all } T : T_1, \exists X : \forall, \text{ notin } X T / \bigcup \text{ notin } U T / \text{ envBook } U.
\end{align*}$$

where the fresh variable is required to be properly typed in the current environment.

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 renaming lemmas, which, in turn, require fresh-renaming properties. These kinds of properties cannot be derived in standard type theories using HOAS-based encodings, but need the use of $\beta$-expansion and extensionality.

Ultimately, their combined effect 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:

$$\text{Lemma isin_mono: for all } T : \forall \rightarrow \top, \forall X : \forall \rightarrow \forall \rightarrow \top, X \rightarrow Y \rightarrow \text{isin } X (T Y) \rightarrow (\text{forall } Z : \forall, X \rightarrow Z \rightarrow \text{isin } X (T Z)).$$

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

\(^3\)Contexts are “terms with holes”, where the holes can be filled in by variables.

\(^4\)This 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.

\(^5\)In presence of binders, such a property is not derivable.

\(^6\)Their consistency has been proved in [3], starting from an idea of M. Hofmann [16].
Lemma pre_isin_mono: forall n:nat, forall U:Tp, lntp n U -> forall V:Var, forall T:Var->Tp, notin bo V T -> U(=T V) -> forall X:Var, X<>Y -> isin X (T Y) -> forall Z:Var, X<>Z -> isin X (T Z).

where lntp counts the number of constructors of a type:

Inductive lntp: Tp -> nat -> Prop :=
| lntp_top : lntp top 1
| lntp_var : forall X:Var, lntp X 1
| lntp_array : forall T:Tp, forall n1 n2:nat, lntp T n1 -> lntp T' n2 -> lntp (arr T T') (S (plus n1 n2))
| lntp_tuple : 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':Var->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 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 such an hypothesis, we get the case (among others) where the equalities U=(T V) and U=top hold. Then, by considering the context U'=(fun X:Tp => top), we can state that (T V)=top=(U' V), whence we infer (T V)=(fun X:Tp => top) V). Finally, we “lift” such structural information to higher-order terms, via the extensionality axiom: namely, we deduce T=(fun X:Tp => top), i.e., we get the structural information we need about T.

5. INTERNAL ADEQUACY

The “non-standard” encoding of typing environments proposed in Section 3 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). Hence, we need a way to impose a kind of discipline. 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 3 and the corresponding deep version introduced in [8, 10]. As far as we know, it is the first time that such a result is proved for the bookkeeping technique in a weak-HOAS setting within a proof assistant.

To present the internal adequacy, we have first to recall our deep variant of the encoding; the crucial difference is the representation of the typing environment as a list of pairs:

For the sake of completeness, we notice that we must also define envTp: Set := (list (Var * Tp)).

Consequently, we formalized the freshness of a variable X∈dom(Γ) (Gfresh), the presence of a constraint (X,T)∈Γ (isinG), and the closure of a type closed(Γ,Γ) (Gclosed) w.r.t. the typing environment itself. Then, the inductive formulation of its well-formedness could be stated:

Inductive okEnv: envTp -> Prop :=
| okVoid : okEnv nil

Given this rendering of the typing environment, also the representation of the subtyping judgment (corresponding to Γ ⊑Γ+ S⊂T of [1]), that we rename here as subTp' to avoid confusion with the shallow counterpart subTp of Section 3, becomes dependent on the typing environment itself:

Inductive subTp' : envTp -> Tp -> Tp -> Prop :=
| subTp'_top : forall G:envTp, forall S:Tp, okEnv G -> Gclosed S G -> subTp' G S top
| subTp'_var : forall G:envTp, forall X:Var, forall U:Tp, okEnv G -> isinG X U G -> subTp' G X X
| subTp'_array : forall G:envTp, forall S1 S2 T1 T2:Tp, subTp' G T1 S1 -> subTp' G S2 T2 -> subTp' G (arr S1 S2) (arr T1 T2)
| subTp'_tuple : forall G:envTp, forall S1 T1:Tp, forall S2 T2:Var->Tp, subTp' G T1 S1 -> (forall X:Var, okEnv (cons (X,Ti) G) -> subTp' (cons (X,Ti) G) (S2 X) (T2 X)) -> subTp' G (fa S1 S2) (fa T1 T2).

Comparing the definition of subTp' with subTp (Section 3), 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 (Coq) the treatment of the object language typing environment. For instance, in the subFa rule we need to enforce explicitly the freshness conditions about the quantified variable X, by means of the notin bo notin list predicate. 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., (X,<:T)∈Γ, all we can do is to introduce a constant d of a suitable type (envBook X T). The latter can then be used and possibly discharged at some point during the proof development process, according to the usual rules of the Coq system. However, such 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 indeed 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.

instantiate the extensionality axiom (namely, envTp_ext in [9]) for terms of type envTp. 8This predicate holds if each free variable occurring in T belongs to the domain of Γ. 9As already noticed in Section 3, the list L may take into account any variable in the current typing environment, not only those directly involved in the judgments at hand.
Thus, in order to avoid the derivation of absurdities, we must define a formal mechanism to validate or discard subtyping derivations carried out in our shallow encoding. As already declared, we decide to do that by establishing an internal correspondence in Coq between the two representations. Such an adequacy amounts to the following lemmas:


In `exp2imp`, two recursive functions are used to “translate”, respectively, the typing environment $G$, involved in the hypothetical derivation (subTp $G$ $S$ $T$), into a list of env-Book-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 explicit list form, the generation of the equivalent bookkeeping assumptions, to deduce (subTp $G$ $S$ $T$) (their Coq definitions are straightforward, see [9]).

Dually, in lemma `imp2exp`, we may prove subTp $G$ $S$ $T$ 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:


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

### 5.1 Completeness

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

1. $Γ ⊢ S <\tau T$ implies that $Γ$ is well-formed:
   
   forall $G : S : Tp$, subTp $G$ $S$ $T$ -> okEnv $G$.

2. The “closedness” (Gclosed) of a type w.r.t. 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 $G$ $S$ ->
   

3. If $(X, U) ∈ Γ$ 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 $X$ $U$ $G$ ->
   

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

### 5.2 Soundness

On the other hand, deriving in Coq the proof of lemma `imp2exp` is definitely more challenging. Actually, passing from a derivation of subTp $S$ $T$ and all the related envBook-assumptions to a derivation of subTp $G$ $S$ $T$, 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:

**Lemma Gfresh_rw:** forall $G : envTp$, forall $G : Var : envTp$, forall $X : Var$, notinEnv $G$ $ho$ $G$ -> $G$ $(G’$ $Z$) ->
   
   forall $X : Var$, $X<2$ -> Gfresh $G$ $X$ ->
   
   forall $Y : Var$, $X<Y$ -> Gfresh $G$ $(G’$ $Y’$).

Essentially, `Gfresh_rw` states that if $Γ=Γ’[Z]$ (with $Z$ fresh in $Γ’[Z]$), $X≠Z$, and $X ∉ dom(Γ)$, then for all $Y$ (such that $X≠Y$) $X ∉ dom(Γ’[Y])$ holds; i.e., the non-occurrence of $X$ in $dom(Γ)$ is preserved by renamings of different variables.

**Lemma isinG_rw:** forall $G : envTp$, forall $U : Tp$, forall $G : Var : envTp$, forall $X : Var$, notinEnv $G$ $ho$ $X$ $G$' $G$ ->
   
   forall $X : Var$, $X<Y$ -> isinG $X$ $U$ $G$' ->
   
   exists $U’ : Tp$, isinG $U’$ $U$ $(G’$ $Y’$).

**Lemma isinG_rw** allows to rename variables with fresh ones, preserving the occurrence conditions in typing environments: if $X ∉ dom(Γ’[X]), Γ=Γ’(X)$ and $X<Z’ ∈ Γ$ (for a suitable $U$), then for each fresh variable $Y$ ($X≠Y$ and $Y ∉ dom(Γ’[X])$) we have that there exists a suitable $U’$ such that $Y<Z’ ∈ Γ’[Y]$.

**Lemma Gclosed_rw:** forall $T : Tp$, forall $T : Var : Tp$, forall $G : envTp$, forall $G : Var : envTp$, forall $X : Var$, notinEnv $G$ $ho$ $X$ $G$’ $G$ ->
   
   (forall $G : G’$ $G’$ $X$ $G$’ $G’$ ->
   
   G’ $(G’$ $X’$) ->
   
   Gclosed $(G’$ $T’$ $X’$) ->
   
   forall $Y : Var$, $X<Y$ -> notinHo $Y$ $T’$ ->
   
   notinEnv $Y$ $G’$ ->
   
   Gclosed $(G’$ $(Γ’$ $Y’)$ $(G’$ $Y’)$).

Fresh renamings also preserve the closed property w.r.t. a given environment: if `closed(T, Γ)` holds in the deep encoding, then $closed(T, Γ)$ holds in the shallow encoding, where $Y∉ dom(Γ’[X])$. $T=dom(Γ’[X])$, then `closed(T[Γ’][Y’], Γ’[Y’])`, where $Y$ is any variable such that $X$ and $Y ≠ dom(Γ’[X])$ and $Y ∉ dom(T[Γ’][X])$.

**Lemma okEnv_rw:** forall $G : envTp$, forall $G : Var : envTp$, forall $X : Var$, notinEnv $G$ $ho$ $X$ $G$’ $G$ ->
   
   (forall $G : G’$ $G’$ $X$ $G’$ $G’$ ->
   
   okEnv $G$’ $G’$).

**okEnv_rw** ensures that a well-formed environment $Γ$ is still well-formed if we rename some variables in it: if $Γ$ is well-formed, $Y ∉ dom(Γ’[X])$ and $Γ=Γ’[X]$, then $Γ’[Y]$ is well-formed for all variables $Y$ such that $X$ and $Y ≠ dom(Γ’[X])$.

**Lemma notinHo:** forall $G : envTp$, forall $G : Var : envTp$, forall $X : Var$, notinHo $Y$ $G$’ $G$ ->
   
   forall $X : Var$, $X<Y$ -> notinHo $Y$ $G’$ $G’$ ->
   
   okEnv $G$’ $G’$.

Finally, `notinHo` states that we can rename the last variable in the domain of a typing environment used to derive a subtyping relation, preserving the validity of the latter (where, of course, we must rename all the occurrences of the old variable with the new one). More formally, if
Γ, X ⊈ U ⊩ S[X] ⊈ T[X] and X ∉fv(S[·]) ∪ fv(T[·]), then
Γ, Y, X ⊈ U ⊩ S[Y] ⊈ T[Y] holds for all the variables Y such
that X ≠ Y and Y ∉fv(S[·]) ∪ fv(T[·]).

We have proved the mentioned renaming properties either by
structural induction on the environment G (Γfresh_rv, okEnv_rv), or by complete induction on the number of Tp-
constructors (Γisid_rv, Γclosed_rv), or by complete induction
on the number of subTp'-constructors (subTp'-rv), 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 pattern of lntp in Section 4).

Obviously, the reader may find in the Coq script [9] other
auxiliary lemmas, but they are essentially either mere vari-
ants of those described or very trivial properties.

6. A SAMPLE DERIVATION

The ultimate metatheoretical result of the previous section,
i.e., the internal correspondence between our shallow and
deep [8, 10] encodings of subtyping, is in fact not com-
pletely 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 metalinguage the handling of typing as-
sumptions, we cannot enumerate them at the object level.

Nevertheless, we can still exploit the internal correspon-
dence as a protocol for verifying the soundness of a 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 book-
keeping assumptions in the shallow encoding and the explicit
environment G in the deep one, it is sufficient to build "man-
ually" the equivalent structured environment G via the set
of the bookkeeping assumptions used, and prove that G is
well-formed (i.e., okEnv G) is derivable in Coq.

We illustrate this insight with the following example:

<table>
<thead>
<tr>
<th>book(Z, Y) \ (def.)</th>
<th>book(Y, Top) \ (var)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ (book(Z, Y))(\uparrow)</td>
<td>\ (sub(Y, Y))(\uparrow)</td>
</tr>
<tr>
<td>\ (top) \ (book(X, Z))(\uparrow)</td>
<td>\ (sub(Y, Z))(\uparrow)</td>
</tr>
<tr>
<td>\ (subTp, Top) \ (top)</td>
<td>\ (book(Z, Y))(\uparrow)</td>
</tr>
<tr>
<td>\ (subTp, X) \ (top)</td>
<td>\ (sub(X, Y))(\uparrow)</td>
</tr>
<tr>
<td>\ (top) \ (book(X, Z))(\uparrow)</td>
<td>\ (al(\ell))(\uparrow)</td>
</tr>
<tr>
<td>\ (top) \ (book(X, Z))(\uparrow)</td>
<td>\ (top) \ (book(X, Z))(\uparrow)</td>
</tr>
</tbody>
</table>

This derivation, which is displayed in natural deduction-
style\textsuperscript{10} using the notation of Section 2 (notice the \(\text{all}\) rule), provides in fact the proof of the following statement:

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)).

\textsuperscript{10}As usual, local hypotheses are indexed with the rules
they are discharged by.

Therefore, it is sufficient to build the corresponding envi-
ronment and prove that it is well-formed in the deep encod-
ing (provided the involved variables are distinct):

Lemma envVF: 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 \(\text{tout court}\) the deep encoding:

Lemma sampleDeep: forall Y Z:Var, Y ⊈ Z ->
subTp' ((Z,(\var Y)):\:(Y,top):\:nil)
\((\exists a \text{ top } (\text{fun } X:\text{Var} => X))\ (\exists a Z (\text{fun } X:\text{Var} => Y))\).

The proof of this goal, compared to the one carried out
for the shallow encoding, has the following drawbacks. Obvi-
ously, 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 lin-
ear complexity), whereas in the shallow case one is allowed
to pick out the right assumption directly, in constant time.

Concluding, we can say that the deep encoding is better
suited for developing the metatheory of an object system
(in [8, 10] we have actually addressed System F_c's \text{Challenge}
1A [1]), while the shallow encoding is more handy for
animating and testing, i.e., to address the implementation
perspective, which becomes the goal once the formal prop-
erties of the object system have been guaranteed.

7. RELATED WORK AND CONCLUSIONS

Fresh-renaminng lemmas, like those in Section 5, are rather
common in languages with binders. In [13], the authors re-
mark that this kind of properties can be rephrased in terms
of a permutation action \(\pi\) on atoms (variables) which allows
them to formalize the notion of \textit{equivariance}. An equivariant
property essentially should be sensitive only to distinctions
between variable names \(i.e.\) it should not depend on the
particular names themselves. It is a property of sentences of
the form \(\forall x.\phi(x)\), \(i.e.\) \(\forall x.\phi(x)\Leftrightarrow \phi(x' - x)\). This
observation leads to the introduction of a special quantifier \(\forall\): the
intuitive meaning of \(\forall a.p\) is that "\(p\) holds for some/any
fresh name \(a\)" (indeed, \(\forall\) resembles both \(\forall\) and \(\exists\)):

\[\Gamma, a \# \bar{b} \vdash p\]
\[\Gamma \vdash \forall a.p\]
\[\Gamma \vdash q\]

where \(\bar{b}\) is the "support" of \(p\) (the set of "free names"), \(#\) an
atomic predicate stating the \textit{freshness} of atoms w.r.t. terms.

As already mentioned, such kinds of properties occur fre-
fently when there is the need to encode and formally rea-
son about the metatheory of an object language featuring
variables and binders. In the standard setting of the Coq
system, a weak-HOAS encoding approach with the Theory
of Contexts (ToC) allows one to render the \(\forall\) quantifier as
follows (here \(\bar{b}\) is the set of free names of \(p\)):

\[\forall a.p[a] \equiv \forall a.\forall \text{Var}. a \notinfv(v[p[\bar{b}])] \Rightarrow p[a] \equiv a \# \bar{b} \equiv a \notinfv(v[p[\bar{b}]])\]

and the rules stating the double (universal/existential) na-
ture of \(\forall\) can then be easily derived (using the ToC) in the
form of suitable fresh-renaming lemmas (see [23] for the
details). This is essentially the gist of our formal development
described in Section 5.
By combining a weak-HOAS encoding with the inductive features of Coq, we are forced to keep \texttt{Var} as an open (i.e., non-inductive) type to rule out exotic terms\(^{11}\) 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 terms; whence, ToC’s axioms allow to regain at object level the capability of reasoning about the syntactic structure of higher-order terms. In [12], instead, exotic terms are ruled out by means of a validity judgment, which holds only for legal (i.e., non-exotic) terms. Moreover, such 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 [12].

In [5, 6, 7], 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 functional and imperative Abadi and Cardelli’s \(\varsigma\)-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 implementation purposes (e.g., to carry out derivations with ground terms, see Section 6).

However, we remark that in both the present contribution and the previous one about System F-\(\text{e}\)’s type language [10], we have not pursued the optimization of the proof script. We leave this possibility as future work, together with the goal of providing some kind of automatization for the different phases of our methodology, so that it could be benefited by non-expert users too. In this direction it could be fruitful to explore the possibility of porting the encoding to environments supporting HOAS, e.g., the \texttt{Abella} system.

8. REFERENCES


[11] Terms which do not correspond to any entity of the object language [12], thus violating the adequacy of the encoding.


