Nested Abstract Syntax in Coq

André Hirschowitz
CNRS, UNS

Marco Maggesi
Univesitá degli Studi di Firenze

Abstract. We illustrate *Nested Abstract Syntax* as a high-level alternative representation of langages with binding constructs, based on nested datatypes. Our running example is a partial solution in the Coq proof assistant to the POPLmark Challenge. The resulting formalization is very compact and does not require any extra library or special logical apparatus. Along the way, we propose an original, high-level perspective on environments.

Keywords: POPLmark, Abstract Syntax, Semantics, Nested Datatypes, Coq, System $F_{<:}$

1. Introduction

We present a partial solution to the POPLmark Challenge (Aydemir et al., 2005) in the Coq proof assistant. Our formalization addresses part (1A) of the Challenge, concerning the transitivity property of the subtyping relation of System $F_{<:}$. The specific feature of our solution is the use of nested datatypes to encode the syntax of System $F_{<:}$. Accordingly, we propose an original "higher-order" encoding of environments, as functions on the varying set of variables.

Nested datatypes, also called heterogeneous datatypes, have been popularized by the work of Bird and Meertens (Bird and Meertens, 1998). It is a general technique to enforce certain invariants through typing, which allows a high-level and perfectly natural representation of variable binding, as was experienced already ten years ago (Altenkirch and Reus, 1999; Bird and Paterson, 1999). We propose the name "Nested Abstract Syntax" for this (old) approach, which we see as an alternative in particular to Higher Order Abstract Syntax (Pfenning and Elliott, 1988; Lee, Crary and Harper, 2007) and to Weak Higher Order Abstract Syntax (Honsell, Miculan and Scagnetto, 2001).

Here are some advantages of using this technique:

– Typing is refined, taking into account the variation of the set of free variables.

- There is no α-equivalence, which makes life much simpler. It may be considered as a kind of de Bruijn encoding enriched with an enlightening typing discipline (Bird and Paterson, 1999).
- Substitution is perfectly understood in a high-level way, as the main ingredient of the monadic structure (see, e.g., (Altenkirch and Reus, 1999)).

Despite these advantages, nested datatypes seem to have been employed only sparsely in real formalizations of higher-order languages. Besides the first achievements mentioned above, we may quote our earlier (Hirschowitz and Maggesi, 2007) and the contributions of Matthes (Matthes, 2008a; Matthes, 2008b; Matthes, 2009). In particular, we are not aware of other implementations of System $F_{<:}$ based on nested datatypes.

From our point of view, the main conclusions of our experience are

- Nested Abstract Syntax is perfectly suited for formalizations of the present kind.
- Coq offers convenient support in order to program statements in this style with a satisfactory level of readability.
- On the other hand, Coq does not yet offer sufficient support for easy proving in this style, due to the systematic use of dependent-types.
- The formalization of a computer language as designed by the POPLmark Challenge has revealed to be an excellent benchmark for the overall maturity and expressiveness of our favorite theorem prover.

The complete source code of the formalization exposed in this paper can be freely downloaded from the web page of the second autor: `http://www.math.unifi.it/~maggesi/` .

2. The Option

The characteristic feature of the nested approach is best understood through the typing, within the untyped lambda-calculus, of the abstraction operator:

$$\texttt{abs} : \texttt{term}\ \hat{}V \to \texttt{term}\ V.$$

Here, given the type V (V for variables), we consider a new type ˆV which denotes V extended with a new distinguished element (the "fresh" variable). This is realized by the *option* datatype that is defined as follows.[1]

```
Inductive option (A:Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

The option type comes equipped with a companion operator `option_map` which gives its functor structure:

```
Definition option_map (A B:Type)
    (f:A->B) (x:option A) : option B :=
  match x with
  | Some a => Some (f a)
  | None => None
  end.
```

The terms `option` and `option_map` are heavely used in this work, thus we extend the Coq syntax with a special notation for them given by a prefixed hat.[2]

```
Notation "^ f" := (option_map f).
Notation "^ X" := (option X) : type_scope.
```

3. Nested Encoding of $F_{<:}$

Here we describe our encoding of $F_{<:}$-types.

3.1. Nested Syntax for $F_{<:}$

We recall that type expressions in $F_{<:}$ are constructed according to the following grammar

```
T ::=                           types
     Top                          maximum type
     X                            type variable
     T → T                        type of functions
     ∀X<:T.T                      universal type
```

[1] The definitions of `option` and `option_map` that follows are taken from the Coq Standard Library. The option datatype is also known as *maybe* in other contexts, for instance in the Haskell community.

[2] Coq's extensible syntax uses a type driven scope mechanism which is able to disambiguate the two notations.

where in the universal type $\forall X <: S.T$ the variable X is bound in T.

We define an inductive type `ftype` V which represents the set of type expressions over the context V as follows

```
Inductive ftype (V:Type) : Type :=
  | Top : ftype V
  | TVar : V -> ftype V
  | Arr : ftype V -> ftype V -> ftype V
  | Uni : ftype V -> ftype ^V -> ftype V.
```

The occurrence of the option type $\hat{}V$ in the signature reveals the presence of a binding construction, more precisely that a new variable (the "freshest" one) is bound in the second argument of the constructor `Uni`. For better readability, we introduce the syntactic sugar s `-->` t and `all` s, t to denote `Arr` s t and `Uni` s t respectively.

Let us discuss how far we stand from the specification:

– Our `ftype` is stratified, hence, for instance, we have one term `Top` for each context V. We would be happy to read $\texttt{Top}_V$ instead of `Top` V, but this is not (yet?) possible in Coq.

– For a "variable" $v \in V$, `TVar` v is in `ftype` V while v itself is not. So here we depart slightly from the specification. From the theoretical point of view, this may be presented as a clarification. On the other hand, from the pretty-print point of view, it would have been fine to consider `TVar` as a coercion. Unfortunately, this could not be performed smoothly in Coq.

– In our syntax for `Uni`, the bound variable does not show up. As far as meta-theory is concerned, this may be considered a good thing.

3.2. Recursion and the like

A crucial advantage of our encoding is that Coq provides for free the recursion and induction principles which we need. Furthermore Coq provides us with tactics dedicated to inductive types, notably

– *injection* derives new equalities from old ones by applying the injectivity of the constructions of our inductive type.

– *discriminate* searches for an absurd assumption, claiming that two structurally different terms (like $\texttt{Top}_V$ and $\texttt{Top}_V \to \texttt{Top}_V$) are equal.

Note that the fact that Coq handles such nested inductive types is pretty recent (version 8.1). More general forms of inductive types, involving for instance constructions like the explicit substitution:

$$\texttt{expl\_subst} : \texttt{term}(\texttt{term}\ V) \to \texttt{term}\ V$$

cannot yet be implemented so smoothly in Coq, and are the object of active research (Matthes, 2008a; Matthes, 2008b; Matthes, 2009).

3.3. Renaming

The assignment $V \mapsto$ `ftype` V is endowed with a structure of functor through renaming. This is defined using the recursion principle (Fixpoint) as follows:

```
Fixpoint ftype_map V W (f:V->W) (t:ftype V) : ftype W :=
  match t with
  | Top => Top W
  | TVar X => TVar (f X)
  | S --> T => %f S --> %f T
  | all S, T => all %f S, %(^f) T
  end
where "% f" := (ftype_map f).
```

3.4. Functoriality

The functoriality of our renaming can be easily stated (and proven):

```
Lemma ftype_map_id : forall V (t:ftype V) (f:V->V),
  (forall x, f x=x) -> %f t=t.
```

```
Lemma ftype_map_comp : forall U t V (f:U->V) W (g:V->W),
  %g (%f t) = %(g o f) t.
```

3.5. Substitution

The renaming functor can be upgraded into a monad, just in the way it is done for the λ-calculus e.g. in (Altenkirch and Reus, 1999). In fact, this substitution and its main properties could be generated from the signature, just as is the recursion principle.

4. Stratified Environments for $F_{<:}$

4.1. Environments as Functions

As for type expressions, type environments are stratified over contexts. We simply encode them as maps $V \to$ `ftype` V.

```
Definition env V := V->ftype V.
```

This is slightly too simple, and we will introduce in the next section an additional well-formedness condition `RWF`. On the other hand, thanks to this encoding, the judgement `TVar` x <: $t \in \Gamma$, which means that the variable x is bound to the value t in the environment Γ, is readily translated into the equation $\Gamma\ x = t$, and thus need no programming.

4.2. Adding a Binding

The operation of adding a new binding in the environment is denoted $\Gamma \,\&\, t$ and is implemented by the following definition.[3]

```
Definition consenv V (G:env V) (t:ftype V) : env ^V :=
  fun x =>
  match x with
  | None => %(@Some V) t
  | Some x =>  %(@Some V) (G x)
  end.
Notation "G & t" := (consenv G t)
  (at level 50, left associativity).
```

This can be pictorially represented by the following commutative diagram:

$$\begin{tikzcd}
\texttt{ftype}\ V \arrow[r] & \texttt{ftype}\ \hat{}V \\
V \arrow[u, "\Gamma"] \arrow[r, "\texttt{@Some}\ V"'] & \hat{}V \arrow[u, "\Gamma\&t"']
\end{tikzcd}$$

where the environment Γ is extended to $\Gamma \,\&\, t$ by adding a binding for a “fresh” type variable. Note that this commutative diagram has a lifting property: it can be completed with a map $h : \hat{}V \to$ `ftype` V.

[3] The prefix “`@`” disables the implicit arguments mechanism.

4.3. Extending Environments

The concatenation of environments is a crucial operation in the proof of the transitivity. The typing of this operation is slightly subtle: what is the nature of Δ in the concatenation $\Sigma = \Gamma, \Delta$?

Although this question can, of course, be answered in a satisfactory way, our choice has been to avoid this problem. We take advantage of the fact that the knowledge of Γ and Δ is equivalent to the knowledge of Γ and Σ, and formalize concatenation through *extensions*.

Given a commutative diagram

$$\begin{tikzcd}
\texttt{ftype}\,V \arrow[r, "\%f"] & \texttt{ftype}\,W \\
V \arrow[u, "\Gamma"] \arrow[r, "f"'] & W \arrow[u, "\Delta"']
\end{tikzcd}$$

we say that Δ is a *raw* extension of Γ along f.

We reserve the word *extension* for *well-formed* extensions. Indeed, for extensions as for environments, we need a well-formedness predicate. As a matter of fact, in the next section, we will define well-formedness of environments in terms of well-formedness of extensions.

The environment expression

$$\Sigma \quad = \quad \Gamma,\ X <: Q,\ \Delta$$

is formalized as a sequence of two extensions

$$\begin{tikzcd}
\texttt{ftype}\ V \arrow[r] & \texttt{ftype}\ \hat{}V \arrow[r] & \texttt{ftype}\ W \\
V \arrow[u, "\Gamma"] \arrow[r] & \hat{}V \arrow[u, "\Gamma \,\&\, Q"] \arrow[r] & W. \arrow[u, "\Sigma"']
\end{tikzcd}$$

5. Well-Formedness

Our very simple notion of environment leaves room for exotic terms. Indeed, in well-formed environments, each variable has to be bound to a type which only depends upon “earlier” variables. For instance, we have to rule out, for a non empty type V, the exotic environment $\texttt{TVar} : V \to \texttt{ftype}\ V$. We found it convenient to first introduce a relative well-formedness predicate for extensions of environments. Our notion of well-formedness will allow us to merge the permutation lemma and the weakening lemma (see section 7).

5.1. Relative Well-Formedness

We introduce a ternary predicate for relative well-formedness (`RWF`)

$$\Sigma \texttt{ extends } \Gamma \texttt{ along } f$$

defined inductively through two introduction rules, corresponding to the familiar nil+cons scheme.

The nil rule roughly says that the "empty" extension is well-formed[4]:

```
RWF_refl : forall V (G G':env V) (f:V->V),
  G==G' -> (forall x, f x = x) -> G' extends G along f
```

The cons rule says that the composition of two sequential extensions, as pictured in the following diagram, is well-formed as soon as the left hand square is well-formed and the right hand square is commutative:

$$\begin{tikzcd}
\texttt{ftype}\,V \arrow[r, "\%\,f_1"] & \texttt{ftype}\,W \arrow[r, "\%\,f_2"] & \texttt{ftype}\,X \\
V \arrow[u, "\Gamma_1"] \arrow[r, "f_1"'] & W \arrow[u, "\Gamma_2"] \arrow[r, "f_2"'] & X \arrow[u, "\Gamma_3"'] \arrow[ul, "h"]
\end{tikzcd}$$

This commutativity of the right hand side of the diagram expresses that variables in X are bound to terms in `ftype` W. Note that, thanks to the nil rule, the right hand square is a well-formed extension too, as we will state formally below.

Altogether, our inductive declaration reads as follows:

```
Inductive RWF : forall W (D:env W) V (G:env V) (f:W->V),
    Prop :=
  | RWF_refl : forall V (G G':env V) (f:V->V),
     G==G' -> (forall x, f x = x) -> G' extends G along f
  | RWF_append : forall V1 (G1:env V1) V2 (f1:V1->V2) V3
      (G3:env V3) (f2:V2->V3) (f:V1->V3) (E:V3->ftype V2),
     (forall x, G3 x = %f2 (E x)) ->
     (forall x, f2 (f1 x) = f x) ->
     E o f2 extends G1 along f1 -> G3 extends G1 along f
where "D 'extends' G 'along' f" := (RWF G D f).
```

Note that we made no injectivity assumption on the function f in the judgment Σ `extends` Γ `along` f. In fact, there are basically two classes of extensions that we are interested in.

[4] The double equality == stands for the extensional equality of functions.

- The first case is when the reindexing map f is a proper injection. This is the case where the term *extension* of environments is most appropriate.
- The second case is when f is a bijection. Here we may think of f as a permutation of the bindings of the environment, and the judgement

  $$\Delta \texttt{ extends } \Gamma \texttt{ along } f$$

  may be interpreted as *f is an allowed permutation transforming Γ into Δ*. Note that the notion of permutation alluded to in the paper version should be a change of order on the fixed set of variables, rather than a permutation of this set in the usual sense.

5.2. Reasoning with Well-Formed Extensions

The inductive declaration of the RWF predicate generates the desired induction principle which allows to reason smoothly about it.

For instance, we prove easily that a well-formed extension is actually an extension:

```
Lemma RWF_commute :
 forall W (D:env W) V (G:env V) (f:W->V),
 (G extends D along f) -> forall y, G (f y) = %f (D y).
```

Similarly we prove that well-formedness is extensional:

```
Lemma RWF_extens : forall W (D D':env W) (HD : D==D')
  V (G G':env V) (HG : G==G') (f f':W->V) (Hf : f==f'),
  G extends D along f -> G' extends D' along f'.
```

We also prove that the composition of two well-formed extensions is well-formed again:

```
Lemma RWF_trans : forall V1 (G1:env V1) V2 (G2:env V2)
    (f1:V1->V2) V3 (G3:env V3) (f2:V2->V3) (f:V1->V3),
  G2 extends G1 along f1 -> G3 extends G2 along f2 ->
  (forall x, f2 (f1 x) = f x) -> G3 extends G1 along f.
```

Finally we check that the `consenv` construction yields well-formed extensions:

```
Lemma RWF_consenv : forall V (G:env V) (t:ftype V),
  G&t extends G along @Some V.
```

The proof is a direct application of the cons rule (`RWF_append`) In the required diagram

$$\begin{tikzcd}
\texttt{ftype } V \arrow[r] & \texttt{ftype } \texttt{\^{}}V \\
V \arrow[u, "\Gamma"] \arrow[r, "\texttt{@Some } V"'] & \texttt{\^{}}V \arrow[u, "\Gamma\&t"'] \arrow[ul, "h"]
\end{tikzcd}$$

h is given by

$$h\ (\texttt{Some } v) = \Gamma\ v \text{ for all } v \in V \qquad h\ \texttt{None} = \texttt{\%(@Some } V)\ t$$

5.3. Absolute Well-Formedness

Finally, we recover the absolute notion of well-formedness as follows. An environment Γ is well-formed, noted $\texttt{WF}\,\Gamma$, if it is well-formed with respect to the empty environment (the obvious environment over the empty context). We denote by `empty` the empty inductive type and by `empty_inc (V:Type), empty -> V` the associated initial map.

```
Definition WF V (G:env V) :=
  G extends @empty_inc _ along @empty_inc V.
```

As immediate corollaries of lemmas `RWF_trans` and `RWF_consenv` we obtain the following two basic properties of `WF`:

```
Lemma WF_trans : forall W (D:env W) V (G:env V) (f:W->V),
  WF D -> G extends D along f -> WF G.

Lemma WF_consenv : forall V (G:env V) (t:ftype V),
  WF G -> WF (G & t).
```

6. Subtyping

6.1. The Subtyping Predicate

We are now ready to define the subtyping judgement $\Gamma \vdash s <: t$ of $F_{<:}$. This is denoted G `|~` s `<<` t in the machine syntax and encoded through an inductive predicate as follows:

```
Inductive sub : forall V (G:env V) (s t:ftype V), Prop :=
  | SA_Top : forall V (G:env V) (s:ftype V),
      WF G -> G |-- s << Top _
  | SA_Refl_TVar : forall V (G:env V) (x:V),
      WF G -> G |-- TVar x << TVar x
  | SA_Trans_TVar : forall V (G:env V) (x:V) t,
      G |-- G x << t -> G |-- TVar x << t
  | SA_Arrow : forall V (G:env V) (s1 s2 t1 t2:ftype V),
      G |-- t1 << s1 -> G |-- s2 << t2 ->
      G |-- s1 --> s2 << t1 --> t2
  | SA_All : forall V (G:env V) (t1 s1:ftype V)
        (s2 t2:ftype ^V),
      G |-- t1 << s1 -> G & t1 |-- s2 << t2 ->
      G |-- all s1, s2 << all t1, t2
where "G |-- s << t" := (sub G s t).
```

This definition is pretty close to the paper specification, with one constructor for each inference rule. Let us review the differences:

– In the `Trans_TVar` construction, we have only one premise, thanks to the functional nature of our environments.

– We have two occurrences of the WF predicate. We could easily have been closer to the paper version by defining a type for well-formed environments. We have preferred to make apparent that the well-formedness assumption is useful only in the rules without any premise (namely the first two ones).

6.2. Reflexivity and Adequacy

We can state the reflexivity statement as follows

```
Lemma sub_refl : forall V (G:env V) t,
  WF G -> G |-- t << t.
```

and its proof is straightforward.

Our adequacy statement for the subtyping, which says that subtyping entails well-formedness reads as follows:

```
Lemma sub_WF : forall V (G:env V) s t,
  G |-- s << t -> WF G.
```

and its proof is also straightforward.

7. Merging Permutation and Weakening

7.1. The Paper Version

The $F_{<:}$-theory contains two lemmas respectively for weakening and permutation. The permutation lemma reads as follows

Lemma 1. (Permutation) If Δ is a well-formed permutation of Γ then $\Gamma \vdash S <: T$ implies $\Delta \vdash S <: T$.

with a rather long and low-level definition of what is a well-formed permutation.

On the other hand, the weakening lemma reads

Lemma 2. (Weakening) If $\Gamma \vdash S <: T$ and $dom(\Gamma) \cap dom(\Delta) = \emptyset$, then $\Gamma, \Delta \vdash S <: T$.

This formulation involves the concatenation operation on environments. As already mentioned, the typing of Δ in the concatenation Γ, Δ is not completely evident. For the present weakening lemma, a cautious interpretation would be to understand Δ simply as an environment, just as Γ. But this would not cover the intended meaning of the lemma, namely the case where values in Δ are allowed to involve variables in Γ. For this case, a subtler typing of Δ is in order.

7.2. A Merging Interpretation

Our solution avoids this problem: when faced with $\Sigma := \Gamma, \Delta$, we type Σ (instead of Δ), and just say that Σ is a well-formed extension of Γ. This approach offers a new perspective where our two lemmas can be merged. The unified formulation is even more general than the conjunction of the two lemmas, since non injective renamings are allowed:

```
Lemma sub_weakening : forall W (D:env W) (s t:ftype W),
  D |-- s << t ->
  forall V (G:env V) (f:W->V),
  G extends D along f -> G |-- %f s << %f t.
```

The case where f is a bijection accounts for the permutation lemma, while the case where f is a (proper) injection accounts for the weakening lemma. The proof of our `sub_weakening` lemma is straightforward.

8. Transitivity and Narrowing

We now turn to the main results of part (1A) of the Challenge, that is, the proof of the transitivity property for the subtyping relation. In this proof, the transitivity property is coupled with the so-called narrowing property.

8.1. Statements

We begin with the statement of the transitivity lemma which should not give any surprise:

```
Lemma transitivity : forall V (G:env V) s q t,
  G |-- s << q -> G |-- q << t -> G |-- s << t.
```

The statement of the narrowing lemma is slightly more problematic. Let us first recall the paper version:

Lemma 3. (Narrowing) If $\Gamma, X <: Q, \Delta \vdash M <: N$ and $\Gamma \vdash P <: Q$ then $\Gamma, X <: P, \Delta \vdash M <: N$.

The difficulty comes from the concatenated environments. As explained in section 4.3 we solve this problem by rephrasing the statement in terms of well-formed extensions of environments. We introduce two environments `D` and `D'` over the context `W`, which are meant to correspond to $\Gamma, X <: P, \Delta$ and $\Gamma, X <: Q, \Delta$ respectively.

So we have an environment `G` over `V` and two extensions `D` and `D'` of `G&p` and `G&q` respectively, along the same map `f:^V->W`.

Our next task is to express that D and D' come from the same Δ, in other words that they agree outside f ($\hat{}V$). As a matter of fact, we define agreement outside a value instead of outside a map. Here is our definition:

```
Definition agrees_outside V (D E:env V) x : Prop :=
  forall y, y = x \/ D y = E y.
Notation "D 'agrees' 'with' E 'outside' x" :=
  (agrees_outside D E x) (at level 70).
```

Now we can state our version of the narrowing lemma.

```
Lemma narrowing :
  forall V (G:env V) W (D D':env W) (f:^V->W) p q m n,
  D agrees with D' outside f None ->
  D extends G&p along f -> D' extends G&q along f ->
  G |-- p << q -> D' |-- m << n -> D |-- m << n.
```

8.2. Proofs

Following a scheme found in the formalization by Chargueraud[5] we divide the common inductive proof of transitivity and narrowing in two sub-lemmas, namely, `transitivity_lemma` and `narrowing_lemma`. The results in which we are actually interested, i.e., `transitivity` and `narrowing`, are easy consequences of these lemmas.

In view of the `transitivity_lemma` we introduce a variant of the obvious property "q is transitive". The latter reads:

```
Definition transitive : forall V (q:ftype V), Prop :=
  forall (G:env V) s t,
  G |-- s << q -> G |-- q << t -> G |-- s << t.
```

Our variant is the apparently stronger "q is universally transitive":

```
Definition univ_trans : forall V (q:ftype V), Prop :=
  forall W (f:V->W) G s t,
  G |-- s << %f q -> G |-- %f q << t -> G |-- s << t.
```

Now we can state the `transitivity_lemma`:

```
Lemma transitivity_lemma : forall V (q:ftype V),
  univ_trans q.
```

In a similar way, we have to somehow recast our narrowing lemma:

```
Lemma narrowing_lemma :
  forall V (q:ftype V) (trs:trans_prop q) W (D':env W) m n
  (Hq : D' |-- m << n) (G:env V) (D:env W) (f:^V->W)
  (Hf : D agrees with D' outside f None) p
  (RWFp : D extends G&p along f)
  (RWFq : D' extends G&q along f),
  G |-- p << q -> D |-- m << n.
```

Note that the assumptions are arranged in a way better suited for our planned induction. Note also that the assumptions are given names: this is for easier proofs.

Our proof of this lemma is the most involved one in our code, with around forty tactics. Our proof of our `transitivity_lemma` is much simpler, with around twenty tactics. Of course, it invokes our `narrowing_lemma`.

[5] http://www.chargueraud.org/arthur/research/2006/poplmark/

9. Conclusions

This work aimed at positioning Nested Abstract Syntax (NAS) with respect to the main approaches to the binding representation problem. Here are our thoughts in this matter.

– With respect to first-order approaches, notably de Bruijn nameless encoding, we hope to have demonstrated that NAS avoids α-equivalence and related problems, like de Bruijn encoding, while allowing statements with a high-level of readability.

– With respect to higher-order approaches like HOAS (Pfenning and Elliott, 1988), WHOAS (Honsell, Miculan and Scagnetto, 2001), Nominal Logic (Gabbay and Pitts, 2002), we insist on the fact that NAS need no specific logical apparatus, and offers a high-level point of view, perfectly compatible with our everyday logic.

Since at least we are convinced that NAS is a perfect solution to the binding representation problem, the next question is whether it is ready "for the masses". Here is our answer:

– We have to admit that NAS requires a large amount of dependent typing. This should not be a bad point. Unfortunately, it occurs that dependent types easily bring Coq beyond its current capabilities. This does not appear in our formalization just because each time we were stuck, we have been able to find a better way to avoid the problem. Nevertheless, in our opinion, this is the main obstacle which will prevent the "masses" to use NAS.

– The second obstacle is probably the syntax allowed by Coq, which, although fairly sophisticated, is still far from what is allowed for instance by TEX.

– Finally, the "masses" would certainly appreciate a system generating more material, in particular the monadic and parallel substitutions associated with a given signature and their main properties. It is not yet clear to us how will develop our new approach to environments through functions and extensions, but we anticipate that some material could be generated also concerning these environments.

Let us end with a few more remarks:

– Concerning conciseness, we have elaborated a small test in order to compare the various existing solution to Part 1A of the POPLmark

Challenge. We estimate the total amount of the information contained in our code by measuring the size of the compressed sources. Thus we tried to compare the gzipped size of our code with some other available formalizations written in Coq. The results obtained are summarized in the following table.[6]

Contribution	**Part 1A**	**Extras**
B. Aydemir et al.	2.5 Kb	16.4 Kb
A. Chlipala	2.3 Kb	2.6 Kb
A. Hirschowitz & M. Maggesi	4.2 Kb	0 Kb
X. Leroy	5.0 Kb	1.0 Kb
J. Vouillon	4.6 Kb	0 Kb

Some formalizations are built on top of some kind of general purpose library, in which case we tried to evaluate the size of the library (reported in the column *Extras*) separately from the size of the code specific for the solution of the Challenge.

– When formalizing a theory, possibly pushed by the limitations of the system in use, one is often tempted to explore new definitions and proofs. This is part of a general process: we anticipate that computer proofs will more and more influence paper proofs. Although the formulation of the POPLmark Challenge did not leave much room for such exploration, we did not fully resist to the temptation. In particular, we have coined a new approach to environments through functions and extensions, allowing a nice merge of the permutation and weakening lemma. We hope that this approach will be fruitful again in the future.

– A word is in order to explain why we solved only part 1A of the POPLmark Challenge. The answer is just a matter of priority, we did not start yet to try the next part.

References

Altenkirch, T. and B. Reus: 1999, 'Monadic Presentations of Lambda Terms Using Generalized Inductive Types'. In: J. Flum and M. Rodríguez-Artalejo (eds.): *CSL*, Vol. 1683 of *Lecture Notes in Computer Science*. pp. 453–468.

[6] These statistics have been obtained by removing comments and code which does not appear to be necessary for addressing part 1A and by compressing the files with `gzip` with the option `--best`. The whole process has been carried on mostly by hand, which means that it can be affected by errors.

Aydemir, B., A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic: 2005, 'Mechanized metatheory for the masses: The POPLmark Challenge'. In: *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*.

Bird, R. S. and L. G. L. T. Meertens: 1998, 'Nested Datatypes'. In: *MPC '98: Proceedings of the Mathematics of Program Construction*. London, UK, pp. 52–67.

Bird, R. S. and R. Paterson: 1999, 'de Bruijn notation as a nested datatype'. *J. Funct. Program.* **9**(1), 77–91.

Gabbay, M. and A. M. Pitts: 2002, 'A New Approach to Abstract Syntax with Variable Binding'. *Formal Asp. Comput.* **13**(3-5), 341–363.

Hirschowitz, A. and M. Maggesi: 2007, 'Modules over Monads and Linearity'. In: D. Leivant and R. J. G. B. de Queiroz (eds.): *WoLLIC*, Vol. 4576 of *Lecture Notes in Computer Science*. pp. 218–237.

Honsell, F., M. Miculan, and I. Scagnetto: 2001, 'An Axiomatic Approach to Metareasoning on Nominal Algebras in HOAS'. In: F. Orejas, P. G. Spirakis, and J. van Leeuwen (eds.): *ICALP*, Vol. 2076 of *Lecture Notes in Computer Science*. pp. 963–978.

Lee, D. K., K. Crary, and R. Harper: 2007, 'Towards a mechanized metatheory of standard ML'. In: M. Hofmann and M. Felleisen (eds.): *POPL*. pp. 173–184.

Matthes, R.: 2008a, 'Nested Datatypes with Generalized Mendler Iteration: Map Fusion and the Example of the Representation of Untyped Lambda Calculus with Explicit Flattening'. In: P. Audebaud and C. Paulin-Mohring (eds.): *International Conference on Mathematics of Program Construction (MPC), Marseille, 15/07/2008-18/07/2008*, Vol. 5133 of *LNCS*. http://www.springerlink.com/, pp. 220–242.

Matthes, R.: 2008b, 'Recursion on Nested Datatypes in Dependent Type Theory'. In: A. Beckmann, C. Dimitracopoulos, and B. Lwe (eds.): *Computability in Europe Logic and Theory of Algorithms (CiE), University of Athens, 15/06/2008-20/06/2008*, Vol. 5028 of *LNCS*. http://www.springerlink.com/, pp. 431–446. (Confrencier invit).

Matthes, R.: 2009, 'An induction principle for nested datatypes in intensional type theory'. *Journal of Functional Programming* **19**(3&4), 439–468.

Pfenning, F. and C. Elliott: 1988, 'Higher-Order Abstract Syntax'. In: *PLDI*. pp. 199–208.

References

Altenkirch, T. and B. Reus: 1999, 'Monadic Presentations of Lambda Terms Using Generalized Inductive Types'. In: J. Flum and M. Rodríguez-Artalejo (eds.): *CSL*, Vol. 1683 of *Lecture Notes in Computer Science*. pp. 453–468.

Aydemir, B., A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic: 2005, 'Mechanized metatheory for the masses: The POPLmark Challenge'. In: *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*.

Bird, R. S. and L. G. L. T. Meertens: 1998, 'Nested Datatypes'. In: *MPC '98: Proceedings of the Mathematics of Program Construction*. London, UK, pp. 52–67.

Bird, R. S. and R. Paterson: 1999, 'de Bruijn notation as a nested datatype'. *J. Funct. Program.* **9**(1), 77–91.

Gabbay, M. and A. M. Pitts: 2002, 'A New Approach to Abstract Syntax with Variable Binding'. *Formal Asp. Comput.* **13**(3-5), 341–363.

Hirschowitz, A. and M. Maggesi: 2007, 'Modules over Monads and Linearity'. In: D. Leivant and R. J. G. B. de Queiroz (eds.): *WoLLIC*, Vol. 4576 of *Lecture Notes in Computer Science*. pp. 218–237.

Honsell, F., M. Miculan, and I. Scagnetto: 2001, 'An Axiomatic Approach to Metareasoning on Nominal Algebras in HOAS'. In: F. Orejas, P. G. Spirakis, and J. van Leeuwen (eds.): *ICALP*, Vol. 2076 of *Lecture Notes in Computer Science*. pp. 963–978.

Lee, D. K., K. Crary, and R. Harper: 2007, 'Towards a mechanized metatheory of standard ML'. In: M. Hofmann and M. Felleisen (eds.): *POPL*. pp. 173–184.

Matthes, R.: 2008a, 'Nested Datatypes with Generalized Mendler Iteration: Map Fusion and the Example of the Representation of Untyped Lambda Calculus with Explicit Flattening'. In: P. Audebaud and C. Paulin-Mohring (eds.): *International Conference on Mathematics of Program Construction (MPC), Marseille, 15/07/2008-18/07/2008*, Vol. 5133 of *LNCS*. http://www.springerlink.com/, pp. 220–242.

Matthes, R.: 2008b, 'Recursion on Nested Datatypes in Dependent Type Theory'. In: A. Beckmann, C. Dimitracopoulos, and B. Lwe (eds.): *Computability in Europe Logic and Theory of Algorithms (CiE), University of Athens, 15/06/2008-20/06/2008*, Vol. 5028 of *LNCS*. http://www.springerlink.com/, pp. 431–446. (Confrencier invit).

Matthes, R.: 2009, 'An induction principle for nested datatypes in intensional type theory'. *Journal of Functional Programming* **19**(3&4), 439–468.

Pfenning, F. and C. Elliott: 1988, 'Higher-Order Abstract Syntax'. In: *PLDI*. pp. 199–208.