

Implicit and noncomputational arguments using monads

Pierre Letouzey¹ and Bas Spitters^{*2}

¹ Mathematisches Institut, LMU Universität München, Germany

² Institute for Computing and Information Sciences, Radboud University Nijmegen, the Netherlands

Abstract. We provide a monadic view on implicit and noncomputational arguments. This allows us to treat Berger’s non-computational quantifiers in the Coq-system. We use Tait’s normalization proof and the concatenation of vectors as case studies for the extraction of programs. With little effort one can eliminate noncomputational arguments from extracted programs. One thus obtains extracted code that is not only closer to the intended one, but also decreases both the running time and the memory usage dramatically. We also study the connection between Harrop formulas, lax modal logic and the Coq type theory.

Keywords: Program extraction, implicit arguments, monads, lax modal logic, Coq type theory.

1 Introduction

We use a monad, which we call `nc`, to deal with implicit and noncomputational arguments. One motivation for this research is a question already posed by Berger [9]: is it possible to use the `Prop/Set` distinction in the Coq-system [10] to treat the non-computational quantifiers which are present in the minlog system [28]? The Coq proof assistant is based on the Calculus of (co)inductive constructions and provides a mechanism to extract programs from constructive proofs [25, 16, 17]. To eliminate noncomputational content from such programs Coq has two sorts `Prop` and `Set`. Types in the former sort are considered non-computational and thus are not extracted. Types in the latter sort are considered computationally relevant and thus are extracted. Another approach, followed by Berger in [9], is to propose different quantifiers with or without computational content. The same idea is also present in the works of Hayashi [14] or Krivine and Parigot [15, 24]. We try here to embed these technics within Coq.

The main reason for eliminating implicit arguments from extracted programs is to improve the efficiency both in terms of speed and memory usage. At the same time, the readability of the code is also increased, which is important for understanding it and increasing, even more, the trust in the correctness of the program.

* This author was supported by the Netherlands Organization for Scientific Research (NWO).

The article is organized as follows. In section 2 we discuss implicit arguments in extracted programs by considering the example of concatenation of vectors. In section 3 we discuss how to implement the monad `nc` in Coq. In section 4 we discuss the limitation of this approach when one uses noncomputational quantifiers and how to get around these problems. This is then used in the extraction of a program from Tait’s normalization proof. Finally, in section 6 we view the sort `Prop` as the `nc`-stable types and we connect this to Harrop formulas and lax modal logic.

2 Implicit arguments

By implicit arguments we mean arguments that are used to assign types to terms, but that are not used in the computation. This interpretation is similar to, but different from, the implicit arguments in Coq. In Coq arguments are implicit if they can be inferred by the type checker. As an example we now consider vectors, that is, lists annotated with their length. In this case the length is an implicit argument. There are several ways to treat the implicit arguments: using a copy of the unary natural numbers `nat` in `Prop`, using the monad `nc`, or using a combination of both. In the first approach we can still use Coq’s conversion to do most of the work for us, but we need to duplicate parts of the library for natural numbers. In the second approach we can reuse all our old results, moreover, this approach also works for data types that are not inductive. The third approach is a middleway.

Fortunately, for `nat`, the unary natural numbers, both approaches are equivalent. However, proving this equivalence is slightly complicated by the fact that the automatically generated induction principles of Coq are too weak. To be explicit, when we define

```
Inductive natnc : Prop :=
  | Onc : natnc
  | Snc : natnc → natnc.
```

the generated induction principle is not dependent. We fix this by

```
Scheme natnc_ind' := Induction for natnc Sort Prop.
```

Now we define `nc` by

```
Inductive nc (A:Set) : Prop := nc_i : A → nc A.
```

It is not difficult to prove that the types `nc nat` and `natnc` are isomorphic, that is there are functions `f`, `g` such $(f (g n))=n$ and $(g (f n))=n$, for all `n`. Here `=` denotes the Leibniz equality. It should be noted that it is not possible to prove that `Onc` and `(Snc Onc)` are different, in fact, they are equal under the proof-irrelevant interpretation of the Coq universe.

We will later show that `nc` is (almost) a monad.

2.1 vectors

Consider the following definition of vectors, seen as lists annotated with their length:

```
Inductive listn (A:Set) : nat → Set :=
| niln : listn A 0
| consn : ∀n, A → listn A n → listn A (S n).
```

This definition allows us to define, for example, a safe tail operation whose type is $\forall A n, \text{listn } A (S n) \rightarrow \text{listn } A n$. This definition of tail does not need the use of exceptions. It is easy to define concatenation with the informative type $\forall A n m, \text{listn } A n \rightarrow \text{listn } A m \rightarrow \text{listn } A (n+m)$. This illustrates an advantage of dependently typed programming in preserving certain invariants in the type. However, now consider the following example.

```
Definition example : listn bool 4 :=
concat
  (list2listn (true::false::nil)) (list2listn (false::true::nil))
```

Here `list2listn` is the obvious map from lists to vectors. The obtained extracted Ocaml code still contains all the typing information.

```
let example =
concat (length (Cons (True, (Cons (False, Nil))))))
  (length (Cons (False, (Cons (True, Nil))))))
  (list2listn (Cons (True, (Cons (False, Nil))))))
  (list2listn (Cons (False, (Cons (True, Nil))))))
```

2.2 vectors with inductively generated noncomputational argument

In this approach we use the noncomputational copy of the natural numbers `natnc` instead of the ordinary natural numbers in the definition of `listn` above. The proofs are almost verbatim the same as the non-implicit ones. One only needs to replace `Set` by `Prop` occasionally.

In this case we obtain the following extracted code, which is the one one would expect.

```
let example =
concat (list2listn (Cons (True, (Cons (False, Nil))))))
  (list2listn (Cons (False, (Cons (True, Nil))))))
```

2.3 vectors using nc

Using `nc nat` instead of `natnc`, one obtains the same short extracted code as before, however the proofs need to be modified a little since we have to do some of conversions manually. Using a tactic described in the next section, writing these proofs is considerably simplified.

2.4 vectors, a hybrid approach

In this last approach we still use `nc`, but prove that the noncomputational inductive type `natnc` and the noncomputational versions of `+` and successor `S` are equivalent to the ones that are lifted using `nc`. Then we use a tactic to rewrite with this equivalence so that Coq will do the conversions for us.

A similar treatment can be made by the use of views [29][20].

```
Inductive Natind : (nc nat) → Prop :=
| Oncind : Natind (nc_i 0)
| Sncind:∀n:nc nat, Natind n → Natind (S' n).
```

Definition `natind` : $\forall n:nc\ nat, Natind\ n$.

Here `S'` is a noncomputational version of the successor. One still needs to redefine the inductively defined functions, like `+`, which is a rather straightforward process. Then once one arrives at a non-computational goal, one uses a case analysis, where only one constructor is possible, to 'open up' the variables of type `(nc nat)` to obtain a variable of type `nat` and makes the conversion algorithm do its work.

In conclusion, there we have a variety of possibilities all of which require some work, but all are rather straightforward to use.

3 Nc is a Monad

Monads allow to extend pure functional programs with impure features, like exceptions, side effects, I/O, etc., in a flexible way. They have become the standard way of adding these features in the Haskell functional programming language. Here we use a monadic extension to extend type theory with non-computational arguments. It is easy to check that `nc` is a monad. In fact, we have defined a general module for monads with the Kleisli presentation of a monad — that is, using the presentation of a monad using `Unit` and `Bind`. We followed the presentation of monads in Haskell described by Wadler in [30]. We recall, using the Coq syntax, the definition of a monad.

```
Module Type Monad.
Parameter M : Type → Type.
Parameter Unit : ∀A:Type, A → M A.
Parameter Bind : ∀A B:Type, M A → (A → M B) → M B.
Infix "bind" := Bind (at level 50, left associativity).

Hypothesis Bind_ext: ∀A B:Type, ∀f g:A→M B, ∀a:M A,
  f==g → (a bind f)=(a bind g).
Hypothesis Leftunit: ∀A B:Type, ∀a:A, ∀k:A→M B,
  (Unit a) bind k = k a.
Hypothesis Rightunit: ∀A:Type, ∀m:M A,
  m bind (Unit (A:=A)) = m.
Hypothesis Associative: ∀A B C:Type, ∀m, ∀k:A→M B, ∀h:B→M C,
```

```

  m bind (k bind h) = m bind k bind h.
End Monad.

```

Here `f==g` is a notation for the extensional equality: $\forall a, (f\ a)=(g\ a)$.

Using functors between modules we have also provided the alternative presentation of monads using `Unit`, `Map` and `Join`¹. We have implemented a general tactic to rewrite with the usual identities for monads. Of course, this tactic can also be used for any other monad, like the `option` and `list` monads. Designing this tactic was slightly more complicated than one would expect. First, `Coq` lacks η -conversion. Second, a priori, we could have tried to use the built-in `autorewrite` tactic. However, currently `autorewrite` does not support rewriting with equalities other than the Leibniz equality, or rewriting in the hypothesis. Fortunately, since the submission of this paper, the first problem has been addressed by Claudio Sacerdoti Coen in the development version of `Coq`. Building on this work a small patch by Lionel Mamane and the second author solves the second problem. The automatic instantiation of tactics seems to be an advantage of using modules over using dependent type records, a rival approach for modularity in type theory. Although we do not want to go into the general theory of `Coq`-modules, we will consider one example. To define `Map` for the monad `nc` one applies the functor `Monad_prop`, which defines the derived monad properties, to the module `ncmonad`, which defines the monad `nc`.

```

Module nc_prop := Monad_prop ncmonad.

```

After importing `nc_prop` `Map` is available.

There is a minor issue regarding the type hierarchy. It seems most natural to define `nc` as a map from `Set` to `Prop`, unfortunately in this case we can not define `Join`, since it is not even well-typed. In order to characterize `Prop` using the bijection in section 6.2 we define `nc` to be a map from `Type` to `Prop`, in which case `Join` is definable. A related issue arose in the LEGO formalization of terms as monads [4].

When developing proofs in `Coq`, it is sometimes necessary to copy proofs that exist in `Type` and convert them to ‘equivalent’ proofs in `Prop`. In this case it is sometimes convenient to map an implication $A \rightarrow B$ in `Type` to an implication $(nc\ A) \rightarrow (nc\ B)$, which is in `Prop`, and use the equivalences of `nc sigT` and `exists` and of `nc {A}+{B}` and $A \vee B$. That is, we use mappings like²:

```

Definition sig_ex:  $\forall X:Type, \forall P:X \rightarrow Prop,$ 
  nc (sigT P)  $\rightarrow \exists n, P\ n.$ 

```

Here again we use the fact that `nc` applied to an inductive type is isomorphic to the copy of this inductive type in `Prop`. Using the rewrite tactic for setoid equality for the setoid $(Prop, \leftrightarrow)$ it is possible to automate this.

¹ See [30], or any other text on monads, for more information on the merits of these different presentations

² In this code the type of `n` can be inferred and thus may be omitted in the new `Coq` (v.8.0) syntax.

Of course, it is not possible in general to translate all statements in `Type` in such a straightforward way. For instance, consider the ‘axiom of choice’ which is provable at the `Type` level, but not at the `Prop` level. When trying to translate this statement to the `Prop`-level using `nc`, one finds that

$$\text{nc } (\forall x:X, P x) \rightarrow \forall x:X, (\text{nc } (P x))$$

is not provable³. In section 6 we will see how this is related to general considerations of lax modal logic.

4 Non-computational quantifiers

The monadic construction for implicit arguments we developed above works well for functions, as we showed in the list example, but unfortunately, it does not seem to be easily extendable to predicates and relations. Let $P:A\rightarrow\text{Prop}$ and suppose that we want to make `A` implicit. Since predicates are contravariant, we need a map $f:\text{nc } A \rightarrow A$ to obtain a predicate $P':\text{nc } A \rightarrow \text{Prop}$. In general, such a map does not exist. Fortunately, we can solve this problem if we allow ourselves to use an axiom.

```
Axiom ProofRelevance : ∀(A:Set)(a b:A),
  nc_i a = nc_i b → a = b.
```

By using the Propositions-as-Types interpretation, i.e. seeing `Prop` as a clone of `Set` it seems likely that this axiom is consistent: in this model `nc_i` would just correspond to the identity. For this identification of `Prop` and `Set` to be correct, we should use the impredicative version of `Set`. However, since our axiom can be safely added to the impredicative Coq, it can certainly be added safely to its predicative variant, which is a subsystem. In addition, being in `Prop`, the axiom has no computational content, it can be safely added when we are only interested in extracting programs. It should be noted that `ProofRelevance` is restricted to the sort `Set` instead of `Type`. This latter version would be much stronger (for instance consider `A` being `Prop` or `Set`), and its consistency quite uncertain. Finally, it should be noted that `ProofRelevance` clearly contradicts the axiom of Proof Irrelevance, $\forall A:\text{Prop}, \forall p q:A, p=q$ which is for instance a consequence of the excluded middle for the sort `Prop`.

For a type `A`, the universal quantifier without computational content is simply a usual universal quantifier, but over `nc A`:

```
Notation "'forallnc' x : A , p" := (∀x:(nc A),p).
```

Concerning the existential quantifier, we define a new inductive type. Its head is logical but the body is informative:

```
Inductive existsnc_t (A:Prop)(P:A→Type) : Type :=
  existsnc_I : ∀x:A, P x → existsnc_t A P.
```

³ In fact, when replacing `nc` with the squash operator in an extensional type theory with a such an operator this equivalent to the axiom of choice, see [5]

Then the actual existential is a use of this type:

Notation "'existsnc' $x : A$, p " := (existsnc_t (nc A) (fun x => p)).

Let us now express properties using these Prop-encoded objects. If we have a predicate P depending on a type A , and an object a being in $\text{nc } A$, we cannot write directly $P a$. Instead, we propose a new construction `letnc a' := a in P a'` that relates a with an object in A . Here is the definition of this construction:

Notation "'letnc' $x := y$ 'in' p " := ($\forall x, \text{nc } x = y \rightarrow p$).

The name `letnc` is motivated by the similarity with the usual notation `let $x := y$ in p` for (`y bind (fun $x \Rightarrow p$)`). It's worthwhile to point out that the following analogue of the Leftunit rule holds.

$\forall A : \text{Set}, \forall P : A \rightarrow \text{Prop}, \forall y,$
 $(\text{letnc } x := \text{nc } y \text{ in } (P x)) \leftrightarrow (P y).$

It seems to be difficult to state any sensible such variants for the `Rightunit` and `Associative` rules.

5 A formalization of Tait's normalization proof

The original motivation for the use of non-computational quantifiers in Coq was a case study on formalizing a normalization proof à la Tait proposed by Berger [9]. In this work, Berger showed that a normalization proof for the simply-typed λ -calculus can lead by extraction to the well-known "Normalization By Evaluation" algorithm. But at that time, the proof was only made on paper, and the extraction done "manually". In a recent effort, Schwichtenberg, Berger, Berghofer and the first author have formalized this proof both in Minlog and Coq and Isabelle⁴, and then performed automatic extractions. A more complete description of these parallel formalizations can be found in [8]. Here, we will focus on a point only shortly mentioned in [8]: using non-computational quantifiers in Coq is crucial for the final extracted program to have the right shape.

5.1 Strong computability

In this normalization proof, the key point is to use a "Strongly Computable" (SC) predicate, defined by induction over simple types:

- In the case of a base type (called here `Iota`), being strongly computable amounts to being strongly normalizable.
- An object `r` of an arrow type is strongly computable whenever for all strongly computable argument `s`, the application (`App r s`) is still strongly computable.

⁴ In <http://www.lri.fr/~letouzey/download/tait.tgz> can be found the source files of the Coq formalization. Some details of the present description are simplified, for instance contexts and typing judgments are omitted when non-essential.

This SC predicate can be defined in a straightforward way in Coq:

```

Fixpoint SC ( $\rho$ :type)(r:nc term) {struct  $\rho$ } : Type :=
  match  $\rho$  with
  | Iota  $\Rightarrow$  SN Iota r
  | Arrow  $\rho$   $\sigma \Rightarrow \forall s$ :nc term, SC  $\rho$  s  $\rightarrow$  SC  $\sigma$  (App' r s)
  end.

```

Note that the previous definition has been adapted to use `nc` for its second argument, the term `r`, and for the internal quantification over the term `s`. In a consistent manner, `App'` is the application `App` over terms, which we extend to a noncomputational variant in the usual monadic way.

This adaptation to `nc` can be justified in the following way. On the computational level, if we forget the dependencies, `SC` can be seen as a union between a recursive arrow type `SC \rightarrow SC` and a base type `SN`. The role of the `SC` arguments are hence quite different: the type `ρ` is used to determine the current side of the union, whereas the term `r` just provides some annotations that precise the meaning of the predicate. Finally, the first argument is crucially used when computing with objects of type `SC`, whereas the second argument will only be used in proof parts. As a consequence, using `nc` for the second argument leads to cleaner and shorter extracted program, as noted by Berger [9].

5.2 Adapting proofs to the nc version

The main part of the normalization proof consists of three lemmas and the final result. Thanks to the Coq module system, this core of the normalization proof forms an autonomous part of 300 lines. It is well separated from the rest of the formalization that consists mainly in results about λ -terms: 3 000 lines dedicated to substitution, typing, $\beta\eta$ -conversion, etc. The glue between the auxiliary and the core parts is an interface providing some predicates like `SN` and a few requirements for these predicates. The core proofs relies only on these requirements⁵, that are designed to have no computational content – they were Harrop formulas in the original paper [9], and are `Prop` statements in Coq. Hence all the computational content of the normalization proof is concentrated in the three lemmas and final theorem.

In a first step, we have built proofs for these three lemmas in the regular Coq way, without using any `nc`. For instance, the simplest of the three lemmas has the following statement, where `H` is the predicate denoting one step of head β -reduction:

Lemma two : $\forall(\rho$:type)(r r₀:term), SC ρ r₀ \rightarrow H ρ r r₀ \rightarrow SC ρ r.

⁵ In addition, this small core is also quite independent from the chosen representation for λ -terms. We were familiar with de Bruijn indices, and hence used them, but this core can be easily adapted to other encodings (HOAS, nominal, ...). For instance, Berger used a named approach for his Minlog development presented in [8].

In a second time, we have adapted this 300-line core to `nc`. We have already seen above the `nc` version of `SC`. Concerning the lemma `two`, we had to change the reference to the `H` predicate, that still expects non-`nc` arguments. For that, we use the `letnc` construct seen earlier:

```
Lemma two : ∀(ρ:type)(r r₀:nc term), SC ρ r₀ →
  (letnc r':=r in letnc r₀':=r₀ in H ρ r' r₀') → SC ρ r.
```

Concerning the adaptation of proofs to `nc`, some changes have to be made. Since our `nc` can be seen as a wrapping, some encoding and decoding has to be performed. Fortunately, we actually managed to lessen the burden by designing a custom tactic called `nc`. In front of a `Prop` subgoal, we are free to use `nc` objects at will, and this `nc` tactic translates all `nc` objects to their non-`nc` counterparts, leaving a `nc`-free subgoal. This process sometimes involves the use of the `ProofRelevance` axiom seen earlier.

In addition to placing some `nc` tactics, the only other changes in proofs concern the use of `nc`-translated functions like `app'` instead of the regular `append` function `app` on lists. First, these translated functions cannot be reduced by `Coq` as simply as their original versions, we need to perform a `nc` decoding in order for the reduction to occur. Secondly, the library lemmas such as the `append` associativity are not compatible with the `nc` version, leaving us with the need for another decoding or for a translated lemma. Fortunately, the number of such problems during the `nc` adaptation has been rather low. Over the 300 lines of the proofs, we changed or added only 57 lines, of which only 10 are not straightforward instances of our `nc` tactic.

5.3 Comparison of the extracted codes

After this translation to `nc`, we can now compare the extracted code obtained with the original and modified settings. And without much surprise, the refined version is similar to the first one, but shorter. As an example, we continue to study the shortest lemma `two`. In the original extracted code for `two`, we have framed the code that will later correspond to `nc` and disappears in the refined version:

```
(** val two : type0 → term → term → SC → SC **)

let rec two t r r' h =
  match t with
  | Iota → Obj.magic (fun k _ → Obj.magic h k _)
  | Arrow (t1, t2) →
    Obj.magic (fun s x →
      two t2 (App (r, s)) (App (r', s)) (Obj.magic h s x))
```

The `nc`-refined version looks as follows:

```
(** val two : type0 → SC → SC **)
```

```

let rec two t h =
  match t with
  | Iota → Obj.magic (fun k _ → Obj.magic h k _)
  | Arrow (t1, t2) →
    Obj.magic (fun _ x → two t2 (Obj.magic h _ x))

```

In this second version, the extracted code is now really similar to Berger’s manually extracted code, but not exactly identical:

- First, one may notice the presence of `Obj.magic` functions in the code. From the execution point of view, we can disregard these functions: they only influence the typing, while their value is the identity function. In particular, when extracting to the (untyped) Scheme language, these functions are not there any more. We will comment the need for such functions for Ocaml in the next section.
- Secondly, the code contains abstractions over the anonymous variable `_` and applications to a constant `_`. These are the remnants of some logical abstractions and arguments that have been detected and collapsed during extraction, but not removed. In order to support safely the whole Coq system, the new Coq extraction may indeed have to leave such remnants instead of removing them completely. The justification of this fact and the precise implementation of the constant `_` can be found in [17]. Otherwise, in a first approximation, one may see `_` as arbitrary code, for instance the unit value `()`. In fact, lots of these remnants generated by extraction are simplified during an optimization phase. For instance, `(fun _ → t) _` becomes `t`. But in the above code, such a local optimization is not enough: simplifying the logical remnants would mean changing the semantics of the second argument of `two`. This may be done, but is not supported yet by the current extraction. By doing it, we would meet Berger’s version for `two`:

```

let rec two t h =
  match t with
  | Iota → fun k → h k
  | Arrow (t1, t2) → fun x → two t2 (h x)

```

- In fact, for this particular example of lemma `two`, there is a last refinement proposed by Berger. By looking closely at this function, we can see recursively that every call `(two t h)` is in fact extensionally equivalent to the second argument `h`. This function `two` is in fact the identity! However, taking advantage of this fact is currently out of the reach of Coq’s extraction mechanism, which normally produces extracted terms that are faithful to the structure of the original term. Normally, the only difference between the original and extracted code is the pruning of logical parts, and a term built by induction like `two` will give a recursive function. The extraction includes a few optimizations that are able to modify locally the structure of the extracted code, for instance if all branches of a `match` are equal, but these

optimizations cannot help here. In `minlog`, the identity function and lemma `two` can be related via the internal realizability and some axioms, but this approach cannot be followed in `Coq`.

We have finally made some tests for measuring the efficiency of the obtained codes. The benchmark chosen was the computation of exponential by applying a church numeral to another one, hence generating several β -reductions. As a rough estimate, the speed gain induced by the `nc` version is at least 40%, whereas the needed memory is reduced by at least 20%. But these figures may vary significantly according to the particular computation asked. For instance, computing 2^{20} takes only 20.2s and 147Mo instead of 28.3s and 187Mo on a 1.8Ghz Pentium 4, which is close to the mentioned 40% and 20% ratios. But computing 30^4 shows a much wider gap: 11.2s and 80Mo instead of 19.3s and 249Mo, hence 70% and 3x ratios! Smaller computations can even show 2x speed-up, but in a general way speed-up tends to decrease slightly when computations become larger. Finally, we have also compared with a third version, the ideal one described in the previous paragraph, obtained by manual modification of the `nc` version. Compared with purely extracted `nc` version, the speed-up is around 50%, with a a new 20% memory decrease. Using `nc` and the current extraction of `Coq` hence does not generate perfect code, but it clearly improves the situation compared with normal extraction.

5.4 Typing issues

In this paragraph unrelated with `nc` we describe briefly the role of `Obj.magic` in the extracted code. As we have already seen, the dependent predicate `SC` may either be the `SN` type or an arrow type. Hence it cannot be translated directly to a normal ML type. Our solution is to approximate it using an arbitrary type. Indeed, on the execution level, it is not an error to consider objects of “type” `SC` that can be either functions or not, since the `Coq` original typing prevent for example non-function to be applied. The extraction automatically detects the typing conflicts at the ML level and insert unsafe type coercions `Obj.magic` to force `Ocaml` compiler to accept this untyped but correct function. A more detailed description of this mechanism can be found in [17]. Another solution would be to use an encoding via a ML sum type:

```
type SC = SC_Iota of SN | SC_Arrow of SC → SC
```

But then every use of `SC` objects should be adapted with constructors and de-constructors of this type. Indeed, this solution leads to ML-typable code, but this is at the cost of additional operations that are useless from the operational point of view. In addition, there are similar but more complex situations in `Coq` where such a typable encoding is not available. Hence the extraction sticks to the untyped version, using the `Obj.magic` trick to please the `Ocaml` compiler.

6 Prop and the monad `nc`

In this section we use the observation that `nc` is a monad to highlight a similarity between the Coq type theory and two different type theoretical interpretations of constructive mathematics. Two common interpretations are the propositions as types interpretation and the proposition as squash-types⁶ interpretation. The latter one is used for instance in topos theory, see [5]. In both of these interpretations the propositions are the types stable under a monad, or modal operator. In the former case it is the identity monad, in the latter the squash-monad. Such monadic structure can also be used for the j -reinterpretations, of which the double negation translation and the A-translations are examples, we elaborate on this in the conclusions. Here we show that the Coq type theory has such a monadic structure.

6.1 Lax predicate logic

A nice way of understanding monads is through the connection with lax modal logic, see [23]. There is a precise Curry-Howard correspondence between intuitionistic propositional logic with a modal operator, Moggi's computational lambda calculus and CL-models. In fact, there is a three-way correspondence between term calculus, categorical models and logic [7].

Definition 1. *A CL-model is a CCC with finite co-products and a strong monad.*

Theorem 1. [Moggi] *CL-models provide a sound and complete interpretation of the computational lambda calculus.*

There is a similar connection [13] between lax natural deduction proofs and $\lambda_c^{\Sigma\Pi}$, a variant of the computational lambda calculus where one adds first-order sums and products. Note that in $\lambda_c^{\Sigma\Pi}$ the domain of quantification is untyped.

We will now discuss lax logic a bit further. We shall denote the lax modal operator \circ . We have the following axioms:

1. $U : a \rightarrow \circ a$;
2. $J : \circ \circ a \rightarrow \circ a$;
3. $M : (a \rightarrow b) \rightarrow (\circ a \rightarrow \circ b)$;
4. $St : a \wedge \circ b \rightarrow \circ(a \wedge b)$.

The first three (unit, join and map) say that \circ is a monad, the last one that it is strong. As a direct consequence of 1, we have $\top \rightarrow \circ\top$, that is $\circ\top$ is valid.

Under quite general hypotheses on logic supporting the lax-modality, one can show that monads are strong monads [22]. Indeed, in our type theoretic context this is the case:

⁶ A squash operator is an operator, denoted $[]$, on types which identifies all its objects. That is, $\mathbf{a}=\mathbf{b}$, whenever $\mathbf{a}, \mathbf{b} : [\mathbf{A}]$. Such an operator exists in extensional type theories with quotient types, but not in the Coq type theory.

```

Definition Str:= fun (A B : Type) (X : A ** T B) =>
  let b' := sndT X in Unit (pairT (fstT X) b').

```

and we have defined a functor producing a strong monad from a given monad.

We copy some results from Bell [6] to our context. They are readily checked in Coq.

Lemma 1. *For any (strong) lax modality \circ :*

1. $(a \rightarrow \circ b) \rightarrow (\circ a \rightarrow \circ b)$; and conversely (which is trivial).
2. $\circ(a \wedge b) \rightarrow (\circ a \wedge \circ b)$;
3. $(\circ a \wedge \circ b) \rightarrow \circ(a \wedge b)$;
4. $\circ(a \rightarrow b) \rightarrow (\circ a \rightarrow \circ b)$;
5. $\circ(a \rightarrow \circ b) \rightarrow (a \rightarrow \circ b)$;
6. $\exists x. \circ a \rightarrow \circ \exists x. a$;
7. $\circ \forall x. a \rightarrow \forall x. \circ a$.

Note that it is impossible to prove in general that $\circ \perp \rightarrow \perp$. (Take $\circ A := \top$ for a counterexample.) When this holds the modality is called a *strict* lax modality.

Although the bi-implications $(a \rightarrow \circ b) \leftrightarrow (\circ a \rightarrow \circ b)$ and $(\circ a \wedge \circ b) \leftrightarrow \circ(a \wedge b)$ hold, they are not isomorphisms in the computational type theory; for instance consider the option monad $A \mapsto A \vee \top$.

Finally, one can prove that the computational lambda calculus with the following reductions is strongly normalizing [7]:

$$\begin{array}{ll}
 \circ.\text{ass} & \text{let } z \leftarrow (\text{let } y \leftarrow p \text{ in } q) \text{ in } r = \text{let } y \leftarrow p \text{ in } (\text{let } z \leftarrow q \text{ in } r) \\
 \circ.\beta & \text{let } z \leftarrow \text{val } y \text{ in } p = p[z := y] \\
 \circ.\eta & \text{let } z \leftarrow y \text{ in } \text{val } z = y
 \end{array}$$

In other words, reading `unit` for `val` and `bind` for the `let` construction, we see that the tactic in section 3 which rewrites with `Associative`, `Rightunit` and `Leftunit` is strongly normalizing.

6.2 Prop as nc-stable types

Finally, we now come to the connection of lax logic with the Coq type theory. The sort `Prop` is closed under products — that is, $\forall b:B, A$ has type `Prop`, whenever `A:Prop` and `B:Type`. Consequently, `Prop` is also closed under non-dependent products, `A→B` and binary products `A∧B`. It is not difficult to define:

Definition `prop_stable` : $\forall A:\text{Prop}, \text{nc } A \rightarrow A$.

Definition `prop_stable2` : $\forall A:\text{Type}, A \rightarrow \text{nc } A$.

and prove that those maps are inverses of each other. That is, `Prop` is the universe of types stable under the monad `nc`. Remark that to define the composition of the maps above we used that `Prop` is a subtype of `Type`. It is not a subtype of `Set`. This also explains our choice in section 3 to define `nc` as a map from `Type` to `Prop`, and not only from `Set` to `Prop`.

There is a clear similarity between `Prop` and the class of Harrop formulas and the following Theorem, see Lemma 1.

Definition 2. A formula A is called \circ -stable if $\circ A \leftrightarrow A$, or, equivalently, $\circ A \rightarrow A$. The class of Harrop formulas is defined inductively as follows: atomic formulas are Harrop, when A and B are Harrop, then so are $A \wedge B$, $\forall x A$ and $C \rightarrow A$.

Theorem 2. Suppose that the atomic formulas are stable under \circ , then all Harrop formulas are \circ -stable.

This theorem suggests that we think of **Prop** as the sort of types stable under a given lax modal operator, it should however be noted that the empty and singleton elimination in the Coq logic do not hold in a generic lax logic. The folklore interpretation of **Prop** as either the class of proof-irrelevant types or as the class of double-negated formulas can both be derived from this: the former is the squash operator $\circ A := [A]$ in extensional type theory, which we will discuss later, the latter is the operator $\circ A := \neg\neg A$.

7 Conclusions

We have presented a lightweight treatment of implicit arguments for program extraction using the present Coq type theory. With the use of an axiom we showed how one can implement Berger’s non-computational quantifiers in Coq. Finally, we also provided a new view on the Coq type theory by making a connection with lax modal logic.

As a substitute for non-computational quantifiers the use of **nc** is likely to be a short-term solution to provide more flexible extractions in Coq. In addition to the induced changes in proofs, this solution must indeed be used with caution, due to the need for a proof-relevance axiom. A more satisfactory solution would be to directly integrate the **nc** distinction at the extraction level like in **minlog**. In Coq, this **nc** criterion could certainly be used as a complement of the current **Prop/Set** criterion. Then some additional verifications should be added, either in the Coq kernel or inside the extraction mechanism, to ensure that a **nc**-tagged object is not used in a constructive place.

The relation between lax modal logic and proof irrelevance seems to have appeared first, although not yet entirely explicit, in [26][27]. This work was a starting point for [5] which studies squash types in an extensional setting. Squash types [19] are also called bracket types [5], or mono types [21]. Awodey and Bauer note that in their extensional setting $[]$ is a monad.

Since the squash type is closely related to **nc**, it is not impossible that our treatment of ‘implicit’ arguments also works in the context of the **Prl** proof assistants. See [3] for a treatment of squash-types in **NuPrl**. The present theory of non-computational quantifiers clearly will not work in that context, because $[]$ is proof-irrelevant, thus contradicting our axiom of proof-relevance.

Many of the translations in proof theory may be seen as lax modal operators. Examples are: the double negation translation, the **A**-translation, etc, see [1] for an overview. Aczel and Gambino [2] describe a logic-enriched type theory in

which one can accommodate the so-called j -reinterpretations. It would be interesting to see to what extent it is possible to use the present view of **Propas** the sort of types which are stable under a lax-modality to give a more type-theoretical account of such reinterpretations. It would also be interesting to explore the connections with minimal type theory [18] which like logic-enriched type theory separates propositions from types, but has a more type theoretic flavour.

Finally, given the success in section 5 it would be interesting to see how the current technique can be used in the extraction of programs from other large proof developments [12][11]. This technique may possibly also be used to improve the new Coq evaluator, which currently always computes in **Prop** terms.

References

1. Peter Aczel. The Russell-Prawitz Modality. Technical Report UMCS-00-12-1, Department of Computer Science, University of Manchester, 20 December 2000.
2. Peter Aczel and Nicola Gambino. Collection principles in dependent type theory. *Lecture Notes in Computer Science*, 2277:1–??, 2002.
3. Aleksey Nogin. *Theory and implementation of an efficient tactic-based logical framework*. PhD thesis, Cornell University, August 2002.
4. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.
5. Steve Awodey and Andrej Bauer. Propositions as [Types]. Technical report, Institut Mittag-Leffler, 2001.
6. J. L. Bell. *Toposes and local set theories*, volume 14 of *Oxford Logic Guides*. The Clarendon Press Oxford University Press, New York, 1988. An introduction, Oxford Science Publications.
7. P.N. Benton, G.M. Bierman, and V.C.V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8:177–193, 1998.
8. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 2005. To appear.
9. Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 91–106. Springer-Verlag, 1993.
10. The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. INRIA-Rocquencourt, 2004.
11. L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. In *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus’2005*, 2005. To appear.
12. L. Cruz-Filipe and B. Spitters. Program extraction from large proof developments. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *LNCS*, pages 205–220. Springer-Verlag, 2003.
13. M. V. H. Fairtlough, M. Mendler, and M. Walton. First-order lax logic as a framework for constraint logic programming. Technical Report MIPS-9714, Department of Mathematics and Computer Science, University of Passau, 1997.
14. Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109(1-2):174–210, 1994.

15. Jean-Louis Krivine and Michel Parigot. Programming with proofs. *Journal of Information Processing and Cybernetics EIK*, 26(3):149–167, 1990.
16. Pierre Letouzey. A new extraction for Coq. In *Proceedings of the TYPES Conference 2002, to appear*, LNCS. Springer-Verlag, 2003.
17. Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris Sud, July 2004.
18. M. E. Maietti - G. Sambin. Toward a minimalist foundation for constructive mathematics. In P. Schuster L. Crosilla, editor, *From Sets and Types to Topology and Analysis: Towards Practicable Foundations for Constructive Mathematics*. Oxford UP.
19. Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, Edinburgh, 1995.
20. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
21. M.E. Maietti. *The type theory of Categorical Universes*. PhD thesis, Padova, 1998.
22. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, 1989.
23. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
24. Michel Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–336, 1992.
25. C. Paulin-Mohring. Extracting F_ω 's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, 1989. ACM.
26. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.
27. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications (IMLA'99)*, Trento, Italy, July 1999.
28. H. Schwichtenberg. Minimal logic for computable functionals. Technical report, Mathematisches Institut der Universität München, 2002.
29. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313, 1987.
30. Philip Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages*. ACM Press, Albuquerque, January 1992.