A module system based on opacity

Bruno Barras

June 21, 2016

1 Introduction

Deciding efficiently the conversion of λ-terms is arguably one of the most significant bottle-neck in the implementation of proof-checker of dependent type theories.

Thanks to the confluence property (for systems which enjoy it), this amounts to finding a common reduct to the input terms. Moreover, when strong normalization holds, the simplest algorithm consists in comparing the normal forms of the input terms.

However, this naive algorithm does not scale to large theories for various reasons, among which:

- Producing the normal form may require many reduction steps. This is sometimes unavoidable (e.g. as in proving $2^{10} = 1024$), but in many occasions, fewer reduction steps are actually needed (e.g. as in proving $(1+1)^{10} = 2^{(5+5)}$).
- As the number of definitions using previous definitions grows, the size of terms by just expanding definitions can grow very quickly.[1]

In this note we shall focus on the first issue. The general strategy is to try to perform as few constant expansions as possible, in the hope of finding a common reduct before having to go into expensive reductions.

Finding a strategy that is close to the optimal in all situations is very hard, so many implementations allow the user to guide the system. One idea is to let the user play with the opacity of constants, by instructing the system to temporarily consider some defined constants as uninterpreted. In the above example, making the exponential opaque is often useful, unless we actually want to compute with it.

This mechanism is most of the time seen as an implementation feature. The reason is that, besides the fact that it does not endanger the consistency of the system (as it only allows to derive less facts), it is also not well-behaved when used without care: deductions steps that require the expansion of a constant

[1] The iterated definition $x_n := (x_{n-1}, x_{n-1})$ makes the normal form of x_n grow exponentially with n although no actual computation is involved.

will not be valid anymore when that constant is made opaque, and so replacing a theorem proved when the constant was transparent by its proof-term may be ill-typed. In Type Theory, the symptoma is that subject-reduction property is broken.

What we propose here is to introduce a better principled way of dealing with transparency and opacity, that preserves the important meta-theoretic properties of the system. We support the claim that this may provide an alternative to the feature module systems (as implemented in SML, Ocaml) that assigns abstract interfaces to implementations. Moreover, we claim that this is closer to mathematical practice.

To illustrate this, consider arithmetics. Natural numbers and its operations can be defined, but after proving that it is an instance of relevant algebraic structures, most of subsequent results are built on those grounds. However, occasionally, we may make later definitions that rely on the actual representation of the natural numbers. The big picture about this is that we can distinguish two *abstraction levels* about arithmetics: the computational level, where we can use the definition of all operations, and the algebraic level, where operations are seen as uninterpreted symbols, with algebraic properties to reason about them. Then, each definition or lemma shall be tagged according to the level it belongs to. Obviously, we need to adapt the typing discipline to ensure that the extra knowledge of the computational level do not leak into the algebraic level. This approach contrasts with SML-like module systems that permanently forbids relying on the implementation details, once an implementation has been sealed with an abstract signature.

Note that the ideas in this note should apply to many type systems. We have in mind type systems like MLTT or PTS, but this is certainly not limitative. For the purpose of making the presentation more concrete, the primary type system will be the Calculus Constructions with judgments $\Sigma, \Gamma \vdash$ for context formation, $\Sigma, \Gamma \vdash M : T$ for typing and $\Sigma, \Gamma \vdash M = M'$ for (untyped) convertibility in a global signature Σ and local context Γ, as in the traditional PTS style. Concrete theory examples will use Coq notations.

2 An example

Consider the following theory, relying on a type `nat` of natural numbers, the number 0, the equality relation $=$, and `eq_refl` the proof of reflexivity of equality:

```
Definition x : nat := 0.
Definition x_def : x = 0 := eq_refl 0.
Definition x_def_def : x_def = eq_refl 0 := eq_refl (eq_refl 0).
```

Note that the definition of `x_def` has type $0 = 0$, which is the same type as $x = 0$ only if we allow the expansion of `x`. Then, the type of `x_def_def` also requires the definition of `x`, but not that of `x_def`. Finally, the definition

of `x_def_def` has the expected type when `x_def` can be unfolded (but the expansion of `x` is not *directly* needed).

Thus, if we wish to make `x_def` opaque, then `x_def_def` must also be made opaque, otherwise replacing `x_def_def` with its definition may produce ill-typed terms.

Furthermore, making `x` opaque implies making `x_def` opaque, and hence also `x_def_def`. But this is not all, because the *type* of `x_def_def` would be ill-typed. In this situation, we forbid to use that constant.

We relate this to module systems by considering *opacity configurations* (or its dual: transparency configurations), sets of constants we want to make opaque. Given a configuration, the signature can be turned into an abstract signature by eliding the definition of opaque constants (and removing constants which type cannot be expressed). When this abstract signature is well-typed, we say that the configuration is *consistent*.

There are 4 possible opacity configurations, leading to 4 possible signatures:

- When all constants are transparent, then this correspond to the signature that is as detailed as the implementation.
- When only `x_def_def` is opaque, we get the signature

```
Definition x : nat := 0.
Definition x_def : x = 0 := eq_refl 0.
Parameter x_def_def : x_def = eq_refl 0.
```

- When both `x_def` and `x_def_def` are opaque, we get the signature

```
Definition x : nat := 0.
Parameter x_def : x = 0.
Parameter x_def_def : x_def = eq_refl 0.
```

- When all constants are opaque, the signature hides `x_def_def`:

```
Parameter x : nat.
Parameter x_def : x = 0.
```

3 A type system with transparency annotations

As the example above suggests, we want to devise a system where we can trace, for each global declaration, which parts of the global signature are needed to ensure its soundness.

We introduce the notion of transparency configuration. Given a signature Σ, a configuration τ is a representation of a context which is Σ where some constants have been made opaque, by erasing their definition, and some other constants have been erased. This abstract view of Σ will be noted $\Sigma_{|\tau}$.

A configuration τ is characterized by two sets $\mathrm{Exp}(\tau)$ and $\mathrm{Ref}(\tau)$, subsets of $\textsc{Dom}(\Sigma)$, the domain of Σ. The latter represents the constants that remain

visible, and the former represents the constants that can be expanded. An invariant of configurations is that $\mathrm{Exp}(\tau) \subseteq \mathrm{Ref}(\tau)$. We do not formally exclude that uninterpreted constants belong to $\mathrm{Exp}(\tau)$, but this should not matter.

The order of configurations is defined by the inclusion of the respective sets:

$$\tau \subseteq \tau' := \mathrm{Exp}(\tau) \subseteq \mathrm{Exp}(\tau') \wedge \mathrm{Ref}(\tau) \subseteq \mathrm{Ref}(\tau')$$

The minimal configuration $\emptyset$ maps the signature to the empty sigmature, and the configuration formed of the set of all constants of the signature is maximal. Note that τ is the maximal configuration of Σ_τ.

Configurations can be extended by a constant declaring it as either opaque or transparent:

$$\mathrm{Exp}(\mathrm{Opq}(c,\tau)) = \mathrm{Exp}(\tau) \qquad \mathrm{Ref}(\mathrm{Opq}(c,\tau)) = \{c\} \cup \mathrm{Ref}(\tau)$$

$$\mathrm{Exp}(\mathrm{Transp}(c,\tau)) = \{c\} \cup \mathrm{Exp}(\tau) \qquad \mathrm{Exp}(\mathrm{Transp}(c,\tau)) = \{c\} \cup \mathrm{Exp}(\tau)$$

3.1 Syntax

The syntax of terms is exactly the same as those of the initial system. The global declarations will be annotated with transparency configurations. The original syntax of declarations, which is

$$\Delta ::= c : T \mid c : T := M$$

where c is a constant name and M, T terms, becomes

$$\Delta' ::= c :_\tau T \mid c :_\tau T :=_{\tau'} M$$

where τ and τ' are transparency configurations. τ ensures that T is a well-formed type when constants of τ are allowed to be expanded. Similarly, τ' is a requirement that entails that M has type T.

This intended meaning can be captured by defining a predicate on configurations called consistency. A configuration τ is consistent, written $\textsc{Cons}(\tau)$, if it is closed under the following inductive rules:

$$\frac{c \in \mathrm{Ref}(\tau) \qquad \Sigma(c) = (c :_{\tau'} T \ldots)}{\tau' \subseteq \tau}$$

$$\frac{c \in \mathrm{Exp}(\tau) \qquad \Sigma(c) = (c :_{\tau'} T :=_{\tau''} M)}{\tau'' \subseteq \tau}$$

In the first rule, we can see that whenever a constant c needs to be refered to, then the condition required by its type must be satisfied. Similarly, whenerver a constant needs to be expanded, the constraints on its body must be satisfied.

The closure condition is preserved by union and intersection, so consistent configurations are closed by these two operations.

The above inductive rules define a way to complete any configuration into a consistent one. Furthermore, if we assume that all of the configurations of the signature are consistent, a direct definition can be given:

$$\begin{array}{c} \mathit{Compl}(\tau) := \bigcup\{\mathrm{Transp}(c,\tau'') \mid c \in \mathrm{Exp}(\tau) \wedge \Sigma(c) = (c :_{\tau'} T :=_{\tau''} M)\} \cup \\ \bigcup\{\mathrm{Opq}(c,\tau') \mid c \in \mathrm{Ref}(\tau) \wedge \Sigma(c) = (c :_{\tau'} T \ldots)\} \end{array}$$

3.2 Judgments

For each judgment of the initial type system, we have a new judgment carrying a consistent configuration that reflects the transparency needed to derive the judgment. Thus, these new judgments have the form

$$\Sigma, \Gamma \vdash_\tau \qquad \Sigma, \Gamma \vdash_\tau M : T \qquad \Sigma, \Gamma \vdash_\tau M = M'$$

The extension of global contexts follows these inference rules:

$$(\Sigma\text{-Ax})\ \frac{\Sigma, [\,] \vdash_\tau T : s \qquad c \notin \textsc{Dom}(\Sigma)}{(\Sigma;\, c :_\tau T),\, [\,] \vdash_\emptyset}$$

$$(\Sigma\text{-Def})\ \frac{\Sigma, [\,] \vdash_\tau T : s \qquad \Sigma, [\,] \vdash_{\tau'} M : T \qquad c \notin \textsc{Dom}(\Sigma) \qquad \tau \subseteq \tau'}{(\Sigma;\, c :_\tau T :=_{\tau'} M),\, [\,] \vdash_\emptyset}$$

The typing premises express straightforwardly the meaning of annotations given in the previous section. We draw the attention on the premise $\tau \subseteq \tau'$, which ensures that expanding c makes a stronger requirement than refering to c. Beware that although $\Sigma, \Gamma \vdash_{\tau'} M : T$ implies $\Sigma, \Gamma \vdash_{\tau'} T : s$ for some sort s, we cannot relate τ to a subset of τ' in which T would be well-typed. See section 3.4.

The context formation judgment is also used to derive that a configuration is consistent with the following rules:

$$(\textsc{Opaque})\ \frac{\Sigma, [\,] \vdash_{\tau''} \qquad (c :_\tau T \ldots) \in \Sigma \qquad \tau \subseteq \tau''}{\Sigma,\, [\,] \vdash_{\mathrm{Opq}(c,\tau'')}}$$

$$(\textsc{Transp})\ \frac{\Sigma, [\,] \vdash_{\tau''} \qquad (c :_\tau T :=_{\tau'} M) \in \Sigma \qquad \tau' \subseteq \tau''}{\Sigma,\, [\,] \vdash_{\mathrm{Transp}(c,\tau'')}}$$

The maximal configuration is consistent.

All the rules of the initial system are promoted to the new system by just passing the configuration to the sub-derivations, with the notable exception of the rules related to constants:

$$(\textsc{Eq}\text{-}\delta)\ \frac{\Sigma, \Gamma \vdash_{\tau''} \qquad (c :_\tau T :=_{\tau'} M) \in \Sigma \qquad c \in \mathrm{Exp}(\tau'')}{\Sigma, \Gamma \vdash_{\tau''} c = M}$$

$$(\textsc{Cst})\ \frac{\Sigma, \Gamma \vdash_{\tau''} \qquad (c :_\tau T \ldots) \in \Sigma \qquad c \in \mathrm{Ref}(\tau'')}{\Sigma, \Gamma \vdash_{\tau''} c : T}$$

3.3 Metatheory

First of all, it is obvious that our system is sound w.r.t.. the original one, since the inference rules are those of the initial systems (modulo erasure of configurations) with stronger premises.

On the other hand, annotating signatures and judgments with maximal consistent configurations allows to lift all derivations of the initial system to our system.

It remains to establish the main equivalence property: given a signature Σ and a consistent configuration τ, then $\Sigma, \Gamma \vdash_\tau M : T$ iff $\Sigma_{|\tau}, \Gamma \vdash M : T$, and similarly for other judgments. The forward implication can be seen as a form of strengthening result.

The first step is to show that if $\Sigma, [\,] \vdash_\tau$ holds, then $\Sigma_{|\tau}$ is well-typed. This amounts to showing that (1) judgments are annotated by consistent configurations, and (2) consistent configurations correctly describe the parts of the signature actually used to derive the global declarations.

3.4 Properties of consistent configurations

Let us now forcus on the closure properties of the consistent configurations of a given judgment.

The annotation of judgments with configurations is monotonic:

$$\frac{\Sigma, \Gamma \vdash_\tau M : T \qquad \tau \subseteq \tau'}{\Sigma, \Gamma \vdash_{\tau'} M : T}$$

This implies that if a judgment holds for two configurations, then it holds for their union.

We can expect that some conversion relation is such that the consistent configurations of any derivable judgment are closed by intersection. An important consequence would be the existence of a unique minimal consistent configuration for each judgment of the initial type system, and the possibility to infer the configuration annotations.[2] Another consequence is that having $\vdash_\tau T : s$ and $\vdash_{\tau'} M : T$ would imply $\vdash_{\tau \cap \tau'} T : s$, so the rule ($\Sigma$-DEF) would not need the condition $\tau \subseteq \tau'$ by replacing τ with $\tau \cap \tau'$.

We may conjecture that the conversion of the Calculus of Inductive Constructions, the $\beta\delta\iota$-conversion enjoys this property. We may also conjecture that this happens for conversion relations generated by a sequential reduction relation.

However, this is not the case for all conversion relations in the general case. Assuming we have an infix constant + with reduction rules

$$0 + x \to x \qquad x + 0 \to x$$

and 2 constants $(z_1 := 0)$ and $(z_2 := 0)$, then $\vdash_\tau z_1 + z_2 = z_2 + z_1$ holds for $\tau = \{z_1\}$ and $\tau = \{z_2\}$, but not $\tau = \emptyset$.

[2]Nonetheless, we see the ability for the user to prescribe a set of opaque constants as a feature, see the discussion in introduction.

4 Alternative presentation

In this presentation, we annotate judgments with minimal information: the set of constants that have been expanded in conversions.

$$\frac{(c :_\tau T :=_{\tau'} M) \in \Sigma \qquad c \in \tau''}{\Sigma, \Gamma \vdash^a_{\tau''} c = M} \qquad \frac{\Sigma, \Gamma \vdash^a_{\tau'} \qquad (c :_\tau T \ldots) \in \Sigma}{\Sigma, \Gamma \vdash^a_{\tau'} c : T}$$

We can see that the typing rule for constants is the same as in the initial system. So, only the conversion rule has been restricted.

The correspondance with the first presentation is that a judgment $\Sigma, \Gamma \vdash^a_\tau M : T$ corresponds to $\Sigma, \Gamma \vdash_{Compl(\tau, \mathrm{Cst}(\Gamma, M, T))} M : T$ where *Compl* is the completion of two sets of constants (the first one is Exp()) into a consistent configuration and $\mathrm{Cst}(T)$ is the set of constants appearing in T.

Therefore, the usage of a constant which reference is forbidden in the first presentation can be detected by inspecting whether a constant is in the completion but not in the intended configuration.

5 Further extensions

5.1 Local inductive types

In type theories with inductive definitions, we may want signatures to hide some of the constructors and possibly the elimination rules.

We can imagine three level of abstractions:

- The fully transparent level: eliminators and ι-reduction rules are allowed,
- A non-computational but fully inductive level: eliminators are allowed but ι-reduction is not allowed,
- An abstract level: eliminators and thus ι-reduction are not allowed, and some (or all) constructors can be hidden.

This would for a form of local inductive definitions, although probably not of the same kind as those implemented in Coq by Bertot.