Frame type theory

Cyril Cohen[1], Assia Mahboubi[2], and Xavier Montillet[2]

[1] Université Côte d'Azur, Inria, France
[2] Inria, LS2N, France firstname.lastname@inria.fr

Writing modular programs in proof assistants is notoriously difficult. A significant literature[1] and implementation effort is devoted to the topic, with approaches ranging from adding new constructions to the underlying logic, to adding features to the proof assistant. However, all current options (including records, sections and modules [4, 2]) are unsatisfactory in one way or another. In this work (in progress), we aim at reconciling the pros of several options using frames. Figure 1 summarizes the pros and cons of each option as implemented in the Coq proof assistant [1], and the desired properties of frames.

	Sections	Modules	Records	Frames
First-class objects	No	No	Yes	Yes
Named application / Named currying	No	Yes / No	Yes / No	Yes
Minimal discharge / Field commutation	Yes	No	No	Yes
Import / Flattening	No	Yes	No	Yes
Refinement / Definitional singleton	No	Yes	No	Yes
Subtyping	No	Yes	No	No

Figure 1: Summary of pros and cons of each option

Conjecture 1. *There exists a calculus of* dependently typed frames *that satisfies the properties of figure 1, is confluent, is strongly-normalizing, has decidable type checking, and is conservative over CiC (extended with a definitional singleton).*

This talk will describe and discuss the properties of our candidate calculus.

Frames The central idea is to consider records where some fields do not have a value yet. We will call these generalized records frames, and will say that a field is a definition (resp. abstraction) if it has (resp. does not have) a value. Frames can also be thought of as a reification of the contexts of CiC, as presented in the Coq manual. For example, the frame $\tau \stackrel{\text{def}}{=} \{x := 1, \lambda y, z := x + y\}$ has two definitions x and z and one abstraction y. Projections are only allowed for definition fields that are not preceded by an abstraction field (that they depend on). For example $\tau.x \equiv 1$ but neither $\tau.y$ nor $\tau.z$ are defined.

From Coq to frames Vernacular commands of Coq (like Definition) can be thought of as acting on frames contexts, i.e. frames with a hole. For example, if we write τ for frames and p for Coq programs, the rules in Figure 2[2] define some mock-up operational semantics for Coq programs (where the box represents the hole of the frame context, and the program being evaluated is placed inside the hole).

$$\begin{aligned}
&\{\tau, \boxed{\mathsf{Definition}\, x := t.\ p}\} && \triangleright && \{\tau, x := t, \boxed{p}\} \\
&\{\tau, \boxed{\mathsf{Variable}\, x.\ p}\} && \triangleright && \{\tau, \lambda x, \boxed{p}\} \\
&\{\tau, \boxed{\mathsf{Module}\, M.\ p}\} && \triangleright && \{\tau, M := \{\boxed{p}\}\} \\
&\{\tau_1, M := \{\tau_2, \boxed{\mathsf{EndModule}\, .\ p}\}\} && \triangleright && \{\tau_1, M := \{\tau_2\}, \boxed{p}\} \\
&\{\tau_1, M := \{\tau_2\}, \tau_3, \boxed{\mathsf{Import}\, M.\ p}\} && \triangleright && \{\tau_1, M := \{\tau_2\}, \tau_3, \tau_2, \boxed{p}\}
\end{aligned}$$

Figure 2: Mock-up operational semantics

[1] That we do not have the room to cite in due form.
[2] Where the last one is a special case for the sake of brevity.

First-class objects Just like records, frames should be first-class objects. This allows, for example, to define the n^{th} power of a monoid as its iterated product. It also allows to quantify over all real closed fields (with the carrier living in some fixed universe), for example to state that there is a quantifier elimination procedure.

Named application / Named currying / Partial instantiation Abstracting over several fields should be equivalent to abstracting over a record with those fields. In other words, it should be possible to give several arguments at once, with the names allowing to match abstractions and the corresponding arguments, and currying should be allowed:

$$\{\lambda x, \lambda y, z := x + y\}\,\{x := 0, y := 1\} \rhd^* \{x := 0, y := 1, z := 0 + 1\} \lhd^* (\{\lambda x, \lambda y, z := x + y\}\,\{x := 0\})\,\{y := 1\}$$

Minimal discharge / Field commutation Abstractions in frames should behave as abstractions in sections: If the value of some field x does not depend on the value of another field y (and does not depend either on fields that depend on y), then it should be possible to get the x projection of the frame without instantiating y. For example, $\{\lambda x, y := t, \dots\}.y \equiv t$ if x is not free in t. This allows to make each theorem proof depend only on the hypotheses it uses, without needing to make this set of hypotheses explicit.

More generally, fields that do not depend on each other should commute, including when one of the fields is a definition and the other one is an abstraction. For example, $\{\lambda x, y := t, \dots\} \equiv \{y := t, \lambda x, \dots\}$ if x is not free in t. Under these commutations, projections can be thought of as only being allowed for the first field (or more generally, fields that are the first field in a convertible frame).

Import / Flattening It should be possible to "import" a frame in another, just like for modules. This would allow to "reopen sections" by importing the corresponding frame. When compared to the use of records to represent sets equipped with some structure, this allows to "flatten" the structure without having to duplicate code (as when defining groups independently of monoids) or nest them (as when defining groups as a monoid with some extra structure). This "bundled vs unbundled" dilemma and related problems are described in [5, 3].

Refinement / Definitional singleton It should be possible to refine frames and their types. For example, the type of categories should be refinable to the type of monoids by forcing the type of objects to be unit, and the product of categories should be refinable to the product of monoids.

One could also define a ring as a monoid acting on an abelian group with the same carrier. This second example makes it clearer that we want the refined type to ensure that the two carriers are definitionally equal, and not just propositionally equal. In the context of the usual dependent types, this amounts to wanting a definitional singleton type $\text{Sing}(t)$ inhabited by terms convertible to t, and such that a hypothesis $x : \text{Sing}(t)$ in the context is understood as a definition $x := t$.

References

[1] The Coq Development Team. *The Coq Reference Manual, version 8.4*. Aug. 2012.

[2] Judicaël Courant. "MC_2 A module calculus for Pure Type Systems" (2007).

[3] François Garillot et al. "Packaging Mathematical Structures". 2009.

[4] Elie Soubiran. "Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq. (Modular development of theories and name-space management for the Coq proof assistant)". PhD thesis. École Polytechnique, Palaiseau, France, 2010.

[5] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory" (2011).