Three comments on the anti-frame rule

François Pottier
INRIA
Francois.Pottier@inria.fr

This informal note presents three comments about the anti-frame rule, which respectively regard: its interaction with polymorphism; its interaction with the higher-order frame axiom; and a problematic lack of modularity.

Interaction between anti-frame and polymorphism

It is well-known that a careless combination of parametric polymorphism and weak references is unsound. (A *weak* reference is one that can be read and written without restrictions, as in ML.) The standard way to work around this problem is to rely on the *value restriction* [7] that is, to restrict the $\forall$-introduction rule to values (as opposed to arbitrary terms).

Charguéraud and Pottier [3] pointed out that, on the other hand, there is no adverse interaction between polymorphism and strong references. (A *strong* reference is one that can be read and written only by presenting a linear capability.) As a result, in a type-and-capability system equipped with strong references, the value restriction is not required.

The anti-frame rule was presented in the setting of Charguéraud and Pottier's type-and-capability system, which does not have the value restriction. I proved [4] that the combination of anti-frame and strong references allows encoding weak references. Therefore, the combination of Charguéraud and Pottier's system with the anti-frame rule is unsound. This important fact was unwittingly omitted in the anti-frame paper. In summary,

> *There is an adverse interaction between parametric polymorphism and the anti-frame rule. This interaction can be avoided by re-introducing the value restriction.*

Interaction between anti-frame and higher-order frame

In the introduction of the anti-frame paper [4], I considered the higher-order frame axiom:

$$\chi \leq \chi \otimes C$$

(The computation type χ can be thought of as a conjunction of a value type, which describes the value produced by a computation, and a capability, which describes the final state of the store. C is a capability. The effect of the operator $\otimes$, borrowed from Birkedal *et al.* [2], is to make C an input and output parameter of every arrow within χ.)

At the time of writing this paper, my desire was to include both the higher-order frame axiom and the anti-frame rule in a single system. However, I found that the soundness proof sketch broke when the higher-order frame axiom was introduced (in short, the Revelation lemma does not hold, because $(\cdot \otimes C_1) \otimes C_2$ is not $(\cdot \otimes C_2) \otimes C_1$.) For this reason, I withdrew this axiom, and wrote: *"It would be desirable to formalize a system where the higher-order frame rule and the higher-order anti-frame rule co-exist. I have not yet worked out the details of such a combination."*

More recently, a shadow of a doubt was cast in my mind by Schwinghammer, Birkedal, Reus, and Yang [6]. They published a separation logic where the higher-order frame axiom is unsound, due to a similar lack of a commutation property. (Their logic does not include the anti-frame rule.) (This does not imply that Charguéraud and Pottier's system [3], extended with a higher-order frame axiom, is unsound. On the contrary, it is quite likely that it is sound.)

This prompted me to more closely study the interaction between the anti-frame rule and the higher-order frame axiom. The result was negative:

> *Charguéraud and Pottier's type-and-capability system, extended with both the higher-order frame axiom and the anti-frame rule, is unsound.*

The counter-example is short and instructive. It relies on hidden, higher-order store, that is, on a hidden reference to a function. The encoding of weak references [4] takes the form of a function *mkref*, which allocates a fresh reference and returns a pair of a getter and a setter:

val *mkref* : $\forall\alpha.\ \alpha \to (\mathit{unit} \to \alpha) \times (\alpha \to \mathit{unit})$

In order to get into trouble, it suffices to instantiate the type variable α with a function type, say $\mathit{unit} \to \mathit{unit}$, and to apply the higher-order frame axiom *to the setter alone*:

let $(\mathit{get} : \mathit{unit} \to (\mathit{unit} \to \mathit{unit}))$, $(\mathit{set} : (\mathit{unit} \to \mathit{unit}) \to \mathit{unit}) =$
  mkref (**fun** $x \to x$) – *a dummy initial value*

let $\mathit{set} : (\mathit{unit} * C \to \mathit{unit} * C) * C \to \mathit{unit} * C =$
  set – *apply the higher-order frame axiom*

(Here, C is an arbitrary capability.) We can now "launder" a function that has a side effect on C, and pretend that it has none, just by writing it into the reference (using the second version of *set* above) and reading it back (using *get*) at a type that does not mention C:

let $\mathit{launder} : (\mathit{unit} * C \to \mathit{unit} * C) * C \to (\mathit{unit} \to \mathit{unit}) * C =$
  $\lambda f.(\mathit{set}\, f; \mathit{get}())$

(The dynamic semantics of *launder* is the identity.) To carry the counter-example up to its conclusion, one could allocate an integer reference c; define C to be the capability over c; define a function (say, *increment*) that requires c to exist, and has type $\mathit{unit} * C \to \mathit{unit} * C$; launder this function so that its type becomes $\mathit{unit} \to \mathit{unit}$; de-allocate c; and call *increment*—crash!

In summary, the higher-order frame axiom becomes unsound, in the presence of hidden higher-order store, because "some arrows escape it". Intuitively, for the axiom to be sound, *every* arrow in the type under consideration must be turned into a C-preserving arrow. In the counter-example, the capability over the reference allocated by *mkref* contains an arrow, but this capability is hidden, so it escapes the application of the higher-order frame axiom.

There is no obvious way of working around this conflict. It would be nice if one could somehow permit applying the higher-order frame axiom to objects "that do not involve any hidden state". However, the very point of hidden state is that it is hidden! That is,

whether an object does or does not have hidden state does not, and should not, appear in its type.

An inherent lack of modularity of the anti-frame rule

The anti-frame rule takes the form:

$$\frac{\Gamma \otimes I \vdash t : \chi \otimes I * I}{\Gamma \vdash t : \chi}$$

(The invariant I is a capability, which is visible to the term t and is made invisible outside of it.)

The tensor operator $\otimes$ is applied to the entire judgement in the premise. This includes the type environment Γ, which describes the "outside world", that is, the set of values and functions that are available to the term t. As far as t is concerned, the outside world is described by $\Gamma \otimes I$.

For instance, if Γ offers a function f of type $int \to int$, then, when type-checking t, the type of f becomes $int * I \to int * I$. This is a more restrictive type than $int \to int$, because it requires establishing the invariant I (which may have been temporarily broken) before calling f.

This is required for soundness. One must guarantee that the hidden invariant holds before giving control to the outside world, because the outside world could make a re-entrant call into the inside world. That is, the function f might cause the control to enter the scope of t again—and, at this point, I will be expected to hold. This issue with "callbacks" is known in the object-oriented community [1].

Unfortunately, this feature of the anti-frame rule becomes problematic when the outside world *does not* in fact re-entrantly invoke the inside world. For instance, imagine that the hidden invariant I expresses the ownership of a doubly-linked list. Imagine that I, the implementor of the inside world, wish to use an existing library for manipulating doubly-linked lists. The type environment Γ presumably contains a description of the functions offered by this library (insertion, deletion, etc.). Each of these functions requires a capability over some doubly-linked list, performs a side effect on the list, and returns this capability. Now, in the modified type environment $\Gamma \otimes I$, each of these functions requires (and returns) *two* capabilities: a capability over some doubly-linked list, on the one hand, and the capability I, on the other hand. This means that I actually *cannot* invoke these library functions to operate on the hidden doubly-linked list: for this purpose, I would need *two* copies of I. However, I is a linear capability, and cannot be duplicated.

One could coin a name for this problem:

The anti-frame rule is paranoid.

(It is rightfully so, but that is still a problem.)

This problem was not apparent in the anti-frame paper [4], because the three applications that are sketched there are small enough to not require the use of any external library: they are programmed up only in terms of primitive operations.

The problem can be worked around, to some extent, by using dynamic checks to guard against re-entrancy, and by exploiting the presence of these dynamic checks to make the code well-typed again. In short, if the intended invariant is I, then the actual invariant, say I', should be:

$$\exists b.(\{\sigma : ref\, bool\, b\} * (\neg b \Rightarrow I))$$

Here, σ is a singleton region, which is inhabited by a mutable Boolean cell. The capability $\{\sigma : ref\, bool\, b\}$ represents the ownership of this cell; the index b represents the contents of the cell. This invariant states that, if the cell holds the value *false*, then I holds.

The code is modified as follows:

1. the Boolean flag is initialized to *false*;
2. prior to calling an external function, one does not re-establish I; instead, one sets the flag to *true*, which has the effect of establishing I', as required by the anti-frame rule;
3. when one is called from the outside, the anti-frame rule allows assuming that I' holds; one then consults the flag: if it is *false*, then I holds, and all is well; if it is *true*, then one must signal a runtime failure;
4. when one returns control to the outside, one establishes I, then sets the flag to *false*, so as to establish I'.

(Some, but not all, of this defensive machinery can be eliminated using the generalized anti-frame rule [5]. Indeed, the generalized version of the rule allows expressing the property that the value of the flag does not change over a balanced sequence of calls and returns.)

The dynamic check technique is interesting, but has a runtime cost, and weakens the static guarantees that can be made about the code, so it is not satisfactory.

Of course, one could avoid the problem altogether, simply by not hiding the invariant I. Every function that (directly or indirectly) might call our code would then have its type "tainted" with I. This, however, would entail a loss in flexibility and modularity: because I is a linear capability, it would have to be explicitly threaded through the clients of our code.

One might wish to somehow detect that certain library functions are safe, because "they cannot possibly lead to a re-entrant call", and relax the anti-frame rule in that case. However, again, it is not clear how to track which functions are safe without contradicting the very principle of hidden state.

A solution to this problem seems required for the anti-frame rule to become useful in a production-scale programming language. Such a solution would perhaps eliminate, at the same time, the incompatibility with the higher-order frame axiom. At present, I know of no such solution.

It would be desirable to take a closer look at what has been done in the object-oriented community, so as to determine what form the problem takes there and how it is solved.

Acknowledgements

A discussion with Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang prompted me to look more closely into the interaction between the anti-frame rule and the higher-order frame axiom. Alexandre Pilkiewicz pointed out the inherent lack of modularity of the anti-frame rule.

References

[1] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 387–411. Springer, July 2005.

[2] Lars Birkedal, Noah Torp-Smith, and Hongseok Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science*, 2(5), November 2006.

[3] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

[4] François Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340, June 2008.

[5] François Pottier. Generalizing the higher-order frame and anti-frame rules. Unpublished, July 2009.

[6] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. In

Computer Science Logic, September 2009.

[7] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.