# Generalizing the higher-order frame and anti-frame rules

François Pottier
INRIA
Francois.Pottier@inria.fr

## Abstract

This informal note presents generalized versions of the higher-order frame and anti-frame rules. The main insights reside in two successive generalizations of the "tensor" operator $\otimes$. In the first step, a form of "local invariant", which allows implicit reasoning about "well-bracketed state changes", is introduced. In the second step, a form of "local monotonicity" is added.

## 1. Introduction

Consider a type system with types and capabilities, in the style of Charguéraud and Pottier [3]. There is no need, here, to recall the rules of the system: suffice it to say that the typing judgement for terms takes the form:

$$\Gamma \vdash \chi$$

where $\Gamma$ is a typing environment and $\chi$ is a (computation) type. (The term itself is omitted here.)

***The higher-order frame rule*** In such a setting, Birkedal, Torp-Smith, and Yang's higher-order frame rule [2] takes the form of a subtyping axiom:

$$\chi \leq \chi \otimes I \quad \text{FRAME}$$

Here, $I$ is a capability, which at the same time asserts ownership and describes the type structure of a piece of state. (One could also think of $I$ as a separation logic assertion.)

***The $\otimes$ operator*** The "tensor" operator $(\cdot \otimes I)$ applies to every syntatic category: typing environments, types, capabilities, etc. Its definition is simple. At arrows, it acts as per the following law:

$$(\chi_1 \to \chi_2) \otimes I = (\chi_1 \otimes I) * I \to (\chi_2 \otimes I) * I$$

Here, $\chi * I$ is the (separating) conjunction of a computation type $\chi$ and a capability $I$; it is, itself, a computation type. At every constructor other than arrows, the tensor operator simply distributes into the sub-terms.

In short, the effect of $(\cdot \otimes I)$ is to change the interpretation of arrows throughout its argument. Every "pure" arrow $\cdot \to \cdot$ is replaced with an "effectful" arrow $\cdot * I \to \cdot * I$. Such an arrow describes a function that requires $I$ and returns it, that is, a function that has a side-effect in the area of memory controlled by $I$.

Thus, the higher-order frame rule states that it is sound to degrade every arrow into an arrow that has an additional effect $I$. This axiom is far from trivial, because it applies to higher-order types, that is, to interactions that involve callbacks. For instance, imagine the function *map* declares that it expects a client function $f$ that has no side effects, and purports that *map* $f$ itself has no side effects. In the presence of the higher-order frame rule, *map* can nevertheless be applied to a client function $f$ that *has* an effect over $I$, and *map* $f$ will then itself have an effect over $I$. That is,

$$(\mathit{int} \to \mathit{int}) \to \mathit{list\,int} \to \mathit{list\,int}$$

is a subtype of

$$(\mathit{int} * I \to \mathit{int} * I) \to \mathit{list\,int} * I \to \mathit{list\,int} * I$$

In short, the higher-order frame rule allows a piece of code (such as *map*) that was written (and type-checked) with no side effects in mind to be used in a context where there are side effects.

***The anti-frame rule*** The higher-order anti-frame rule [4] does exactly the converse: it allows a piece of code that was written with side effects to be used in a context that expects no side effects. Its formulation is:

$$\frac{\Gamma \otimes I \vdash (\chi \otimes I) * I}{\Gamma \vdash \chi} \quad \text{AF}$$

The rule has a slightly dissymmetric look, because of the $* I$ conjunct that appears in the right-hand side of the premise, but not in its left-hand side. There is, indeed, dissymmetry. The code must *allocate* and initialize its hidden state $I$, but this state can never be *de-allocated*, because there is no way to statically predict its lifetime.

The anti-frame rule rule allows hiding a piece of state within a certain lexical scope. Thus, it allows defining an "object" with internal mutable state that is accessible only via the object's "methods". Furthermore, the rule allows such an object to receive a non-linear type. (The computation type $\chi$ in the conclusion may happen to be a *non-linear* value type $\tau$ [3], whereas the type $(\chi \otimes I) * I$ is definitely *linear* if $I$ is a non-trivial capability.) Assigning the object a non-linear type means that the object's internal state is technically *untracked*. In other words, as far as the type system is concerned, such objects are considered ordinary values, and can be freely aliased.

***Hidden state is invariant*** A characteristic feature of the higher-order frame and anti-frame rules is that the hidden state is described by an *invariant* $I$. In both rules, whenever control is exchanged between Term and Context, (that is, whenever control enters or leaves the lexical scope delimited by the rule,) $I$ holds. In the case of the anti-frame rule, this means that Term can *assume* $I$ whenever it receives control (either because it is called by Context, or because a call to Context returns), and Term must *guarantee* $I$ whenever it relinquishes control (either by returning to Context, or by invoking a callback provided by Context). Term is free to temporarily break $I$, but must restore it in due time.

The property that $I$ is an invariant is in fact hard-wired in the very definition of the "tensor" operator $\cdot \otimes I$. This operator adds a copy of $I$ as a pre-condition and a post-condition to every arrow, so $I$ must hold at every interaction, period.

In the following, I argue that this is sometimes a limitation. I present two modest generalizations of the "tensor" operator (and, accordingly, of the frame and anti-frame rules) that introduce some new expressive power.

```
1  let mk () =
2    let x = ref 0 in
3    let m f =
4      x := !x + 1;
5      let v1 = !x in
6      f();
7      let v2 = !x in
8      assert (v1 = v2);
9      x := !x - 1
10   in m
```

**Figure 1.** "callee-save register"

```
1  let mk () =
2    let σ, (x : [σ]) = ref 0 in
3    let cap I = { σ: ref int } in
4    got cap I;
5    hide I outside of
6    let m : (unit * I → unit * I) * I → unit * I =
7      got cap I;
8      x := !x + 1;          – permitted by I
9      let v1 = !x in
10     f();                  – permitted by I
11     let v2 = !x in
12     assert (v1 = v2);     – not statically proven safe!
13     x := !x - 1
14   in m                    – m has external type (unit → unit) → unit
```

**Figure 2.** "callee-save register", with AF

```
1  let mk () =
2    let σ, (x : [σ]) = ref 0 in
3    let cap I i = { σ: ref int i } in
4    pack cap ∃i. I i;
5    hide I outside of
6    let m : ∀i. (∀j. unit * I j → unit * I j) * I i → unit * I i =
7      got cap I i;
8      x := !x + 1;
9      got cap I (i + 1);
10     let v1 : int (i + 1) = !x in
11     f();                  – instantiated with i + 1 for j
12     got cap I (i + 1);
13     let v2 : int (i + 1) = !x in
14     assert (v1 = v2);     – statically proven safe
15     x := !x - 1;
16     got cap I i           – restores the initial invariant
17   in m                    – m has external type (unit → unit) → unit
```

**Figure 3.** "callee-save register", with GAF

## 2. Local invariants

### 2.1 A motivating example

Figure 1 presents an (untyped) piece of ML code, which, in object-oriented terminology, implements an object with internal state $x$ and a single method $m$. The toplevel function *mk* can be viewed as a constructor. It allocates and initializes an integer reference $x$, defines the method $m$, which has access to $x$, and returns $m$ to the client. The method $m$ accepts a callback $f$.

Why is this code interesting? The method $m$ modifies $x$ internally, by incrementing it; however, it restores its original value, by decrementing $x$, before returning. (For this reason, this example is dubbed "callee-save register".) As a result, a call to $m$ preserves the value of $x$. Since no function other than $m$ has access to $x$, one can informally argue that every function call preserves the value of $x$. This includes the call to $f$ within $m$, so it seems safe for $m$ to assert that $f$ preserves the value of $x$. This is expressed by the assertion $v_1 = v_2$ on line 8 of Figure 1.

Can we hide $x$ and typecheck this code using the anti-frame rule (AF)? Yes. The code, with a few annotations, appears in Figure 2. On line 2, $\sigma$ is the singleton region inhabited by $x$. On line 3, we define $I$ as an abbreviation for the capability $\{\sigma : \mathit{ref\ int}\}$, which represents the ownership of the region $\sigma$, and indicates that this region holds a reference to an integer. On line line 4, we assert that we hold the capability $I$. (This statement is optional; like a type annotation, it is a machine-checked comment.) Then (line 5), we apply the anti-frame rule, so that $I$ is hidden outside of the definition of the method $m$.

We define $m$ at type $((\mathit{unit} \to \mathit{unit}) \to \mathit{unit}) \otimes I$, that is, $(\mathit{unit} * I \to \mathit{unit} * I) * I \to \mathit{unit} * I$. The two negative occurrences of $I$ in this type mean that $m$ can *assume* $I$ initially (as confirmed by the assertion on line 7), and can *assume* $I$ after $f$ returns. Symmetrically, the two positive occurrences of $I$ mean that $m$ must *guarantee* $I$ when $f$ is called, and must *guarantee* $I$ when $m$ returns.

The effect of the anti-frame rule is to remove an application of $\cdot \otimes I$ in the type of the value that is returned, so, to the outside, $m$ appears to have type $(\mathit{unit} \to \mathit{unit}) \to \mathit{unit}$. (This is the return type of *mk*.)

Can we statically prove that the assertion on line 12 of Figure 2 is correct? No. The invariant $I$ only guarantees that $x$ is a reference to *some* integer value. There is no way to express the fact that this value is preserved across (well-balanced) interactions between Term and Context.

In the following, we will use a singleton type of integers, of the form $\mathit{int}\ i$, where $i$ is a (type-level) integer index, to reason precisely about integer values. For the moment, though, this precision does not help. We could define $I$ as $\exists i.\{\sigma : \mathit{ref\ int}\ i\}$, but that would not help. We would have a guarantee that a call to $f$ preserves the *closed* capability $\exists i.\{\sigma : \mathit{ref\ int}\ i\}$, but no guarantee that the witness $i$ is preserved. In fact, $\exists i.\{\sigma : \mathit{ref\ int}\ i\}$ is equivalent to $\{\sigma : \mathit{ref}\ (\exists i.\mathit{int}\ i)\}$, which itself is interconvertible with $\{\sigma : \mathit{ref\ int}\}$, because $\exists i.\mathit{int}\ i$ is just *int*.

In summary, $I$ is a *global* invariant, one that is preserved across *arbitrary* sequences of calls and returns. What we would like to have, here, is a *local* invariant, one that is preserved across *well-balanced* such sequences.

Another way to put this is: $x$ measures the number of re-entrant calls to $m$, that is, the number of frames associated with $m$ on the (implicit) call stack. We would like to allow each level of the call stack to have its own invariant, possibly distinct with the invariant used at the next level.

### 2.2 A first generalization

There is a surprisingly simple solution to this issue. Instead of defining $I$ as $\exists i.\{\sigma : \mathit{ref\ int}\ i\}$, let us parameterize $I$ over $i$, as follows:

$$I\ i = \{\sigma : \mathit{ref\ int}\ i\}$$

$I$ now has kind $\mathbb{Z} \to \mathrm{CAP}$, where $\mathbb{Z}$ is the kind of integer indices, and CAP is the kind of capabilities. In other words, $I$ is no longer an invariant, but a $\mathbb{Z}$-indexed family of invariants. There remains to generalize the definition of $\otimes$ as follows:

$$(\chi_1 \to \chi_2) \otimes I = \forall i.\big((\chi_1 \otimes I) * I\ i \to (\chi_2 \otimes I) * I\ i\big)$$

The idea is simple. A *universal* quantification is used to share the index $i$ between the pre-condition and post-condition of every arrow, so every *well-balanced* interaction is now required to preserve the open invariant $I\ i$. Less obviously, perhaps, different instantiations of $i$ can be used at different levels of the stack. This will be illustrated below.

In the above example, $i$ has kind $\mathbb{Z}$. Of course, the definition of the generalized "tensor" operator is not specialized for this kind. In general, the index $i$ can have any kind $\kappa$ that exists in the ambient programming language. In particular, if tuple kinds are available, the invariant $I$ can effectively be parameterized over any number of indices. When it is parameterized over zero indices (that is, when $\kappa$ is the unit kind), we obtain the traditional "tensor" operator as a particular case.

It is nice to find that the statement of the frame rule is unchanged:

$$\frac{}{\chi \leq \chi \otimes I}\ \text{FRAME}$$

There is no visible difference. The (invisible) difference is that $I$ is now a $\kappa$-indexed family of invariants, for some kind $\kappa$.

Because the anti-frame rule is expressed not only in terms of $\cdot \otimes I$, but also in terms of $\cdot * I$, its statement changes slightly. Here is the *generalized anti-frame* rule (GAF):

$$\frac{\Gamma \otimes I \vdash (\chi \otimes I) * \exists i.I\ i}{\Gamma \vdash \chi}\ \text{GAF}$$

The conjunct $\exists i.I\ i$ requires *some* invariant in the family $I$ to hold initially. Then, as in the frame rule, the "tensor" $\cdot \otimes I$ is used to express that, if some member $I\ i$ of the family holds at a call (from Term to Context or from Context to Term), then *the same* member holds when this call returns.

**Conjecture 1** *The above generalized higher-order frame and anti-frame rules are sound.* ◇

### 2.3 Back to the example

Let us now come back to the "callee-save register" example of Figures 1 and 2. Using the generalized anti-frame rule, this code can be type-checked in such a way that the internal state is hidden and the assertion is statically proven valid. This is shown in Figure 3. On line 3, $I$ is now defined as a family of invariants, as announced above. The reference $x$ is initialized to 0, so, initially we hold the capability $I\ 0$, which, by introducing an existential quantifier, becomes $\exists i.I\ i$ (line 4). On line 5, the generalized anti-frame rule is applied to the family $I$. As before, the method $m$ is internally defined at type:

$$((\mathit{unit} \to \mathit{unit}) \to \mathit{unit}) \otimes I$$

This time, however, this type means:

$$\forall i.(\forall j.\mathit{unit} * I\ j \to \mathit{unit} * I\ j) * I\ i \to \mathit{unit} * I\ i$$

The "tensor" operator has altered the meaning of the two arrows in the type $(\mathit{unit} \to \mathit{unit}) \to \mathit{unit}$, and has introduced a *distinct* universal quantifier at each arrow.

The universal quantification over $i$ and the two occurrences of $I\ i$ mean that the method $m$ can initially assume $I\ i$, for some unknown $i$, and must eventually guarantee $I\ i$, for the *same* index $i$. The universal quantification over $j$ and the two occurrences of $I\ j$ mean that $m$ must guarantee $I\ j$ when $f$ is called, *for some $j$ whose choice is up to $m$*, and $m$ can assume $I\ j$, for the *same* index $j$, after $f$ returns.

When the execution of $m$ begins, we have $I\ i$ (line 7). After incrementing $x$, this capability is changed to $I\ (i+1)$; this is a strong update (line 9). Thus, the value $v_1$ of $x$ at this time has type $\mathit{int}\ (i+1)$ (line 10). The call to $f$ is well-typed, provided the universally quantified index $j$ in the type of $f$ is instantiated to $i+1$ (line 11). This is how different "levels in the stack" can have different local invariants, as long as they all are members of a single family. The call to $f$ preserves the capability $I\ (i+1)$ (line 12), so the value $v_2$ of $x$ after the call has type $\mathit{int}\ (i+1)$ (line 13). Thus, $v_1$ and $v_2$ have the same (singleton) type. This is enough for the type-checker to statically prove that $v_1$ and $v_2$ are equal, and to prove that the runtime check on line 14 must succeed.

```
1  let mk () =
2    let x = ref 0 in
3    let locked = ref false in
4    let m f =
5      if not !locked then begin
6        locked := true;
7        let v1 = !x in
8        f();
9        let v2 = !x in
10       assert (v1 = v2);
11       x := !x + I;
12       locked := false
13     end
14   in
15   m
```

**Figure 4.** "callback with lock"

### 2.4 A subtler example

Figure 4 presents an example known in the literature under the name of "callback with lock" [1, 5].

As in the previous example (Figure 1), a private integer reference $x$ is accessible via a public method $m$. Contrary to the previous example, the value of $x$ is not preserved by $m$: sometimes, $m$ increments $x$ (line 11). As a result, it is not sound to assume that $f$ preserves $x$. Indeed, $f$ could make a re-entrant call to $m$.

To guard against this situation, the code uses a dynamic check. A Boolean flag, *locked*, is set before $f$ is invoked, and cleared afterwards. If $m$ is invoked while this flag is set, it does not modify $x$. (One could wish for $m$ to fail, by aborting the program or raising an exception, in that case.) It is now sound to assume that the the value of $x$ is preserved across the call to $f$. This is expressed by the assertion on line 10.

(In some previous papers [1, 5], a *get* method is added so as to make the value of $x$ observable and allow stating a meta-level assertion about the successive values of $x$. Here, this is not required, because the assertion is part of the code. Of course, we could add a *get* method if desired.)

The anti-frame rule (AF) allows hiding the following invariant:

$$\{x : \mathit{ref\ int}\} * \{\mathit{locked} : \mathit{ref\ bool}\}$$

(where, by abuse of language, I use the same name for a program variable and for the singleton region that it inhabits). This is sufficient to hide the object's internal state, but does not allow proving $v_1$ and $v_2$ equal.

Does the generalized anti-frame rule (GAF) offer a solution to this problem? In a first attempt, one might attempt hiding the following invariant family:

$$I\ (i,b) = \{x : \mathit{ref\ int}\ i\} * \{\mathit{locked} : \mathit{ref\ bool}\ b\}$$

($i$ and $b$ have respective kinds $\mathbb{Z}$ and $\mathbb{B}$, so $I$ is a $\mathbb{Z} \times \mathbb{B}$-indexed family of invariants. I assume $\mathit{int}\ \cdot$ and $\mathit{bool}\ \cdot$ are the singleton types of integers and Booleans.) This attempt fails, because this invariant is too strong: it requires the values of $x$ and *locked* to be preserved by $m$, whereas only the latter is, in general, preserved. The value of $x$ is preserved only when *locked* is set. This is expressed via a slightly subtler invariant:

$$\begin{aligned} I\ (i,b) = \exists j.(\{x : \mathit{ref\ int}\ j\} * (b = \mathit{true} \Rightarrow i = j)) \\ *\{\mathit{locked} : \mathit{ref\ bool}\ b\} \end{aligned}$$

```
1  let mk () =
2    let x = ref 1 in
3    let m f =
4      x := 0;
5      f();
6      x := 1;
7      f();
8      assert (!x = 1)
9    in
10   m
```

**Figure 5.** "well-bracketed monotonic state change"

(I assume that a logical proposition, such as $(b = \mathit{true} \Rightarrow i = j)$, is viewed as a non-linear capability. If $P$ and $Q$ are propositions, I assume that $(P) * (Q)$ is equivalent to $(P \wedge Q)$. I assume that, if $P$ can be shown to entail $Q$, then $(P)$ can be weakened to $(Q)$. With these assumptions, the existing machinery for transporting capabilities subsumes Hoare logic.)

The first line in the definition of $I$ above states, that upon entry or exit of $m$, $x$ must have some value $j$, such that, if *locked* is set, then $j$ is $i$. The existentially quantified index $j$ is not necessarily the same upon entry and upon exit of $m$. However, the index $i$, which is universally quantified by the "tensor" operator, must be the same upon entry and exit of $m$.

When the flag *locked* is cleared, $b$ must be *false*, so there is no requirement that the value of $x$ be preserved, because different instantiations of $j$ upon entry and upon exit can be used. In fact, since the type $\exists j.\mathit{int}\ \ j$ is equivalent to the type *int*, one could simplify $I\ (i, \mathit{false})$ as follows:

$$I\ (i, \mathit{false}) = \{x : \mathit{ref\ int}\} * \{\mathit{locked} : \mathit{ref\ bool\ false}\}$$

On the other hand, when the flag *locked* is set, $b$ must be *true*, so $j$ is forced to be $i$, and the flexibility afforded by the existential quantification over $j$ vanishes:

$$I\ (i, \mathit{true}) = \{x : \mathit{ref\ int}\ i\} * \{\mathit{locked} : \mathit{ref\ bool\ true}\}$$

This forces the value of $x$ to be preserved by $m$.

It is easy to check that an application of GAF, with the above definition of $I$, is well-typed. (In fact, checking that $m$ preserves $I$ does not even require exploiting the hypothesis that $f$ preserves $I$.) The hypothesis that $f$ preserves $I$ is exploited in order to statically prove that the assertion on line 10 must succeed. If the value of $x$ before the call to $f$ is represented by the index $i$ (so that $v_1$ has type $\mathit{int}\ \ i$), then $I\ \ (i, \mathit{true})$ holds prior to the call. By hypothesis, $f$ preserves this particular instance of the invariant family, so $I\ \ (i, \mathit{true})$ still holds after the call. This means that the index $i$ still represents the value of $x$ after the call. As a result, $v_2$ has type $\mathit{int}\ i$ as well, and the assertion on line 10 must succeed.

In summary, the generalized anti-frame rule (GAF) can be used to state that, *under certain conditions*, certain properties (such as the value of certain variables) are preserved by every (well-balanced) interaction between Term and Context.

In the example of Figure 4, $x$ evolves in a *globally monotonic* manner: its value increases with time. This property is not established or exploited here. My work with Pilkiewicz (alluded to in §4) would allow doing so.

# 3. Local monotonicity

## 3.1 A second generalization

The generalized anti-frame rule of §2 uses a universally quantified index to enforce the preservation of some property of the hidden state by each function call across the "border". It is not difficult to imagine one step further: one could wish to impose a *local monotonicity* property, that is, to require each and every function call across the border to guarantee that the hidden state *evolves* in a certain way.

Again, let $I$ be a $\kappa$-indexed family of invariants, for some kind $\kappa$. In addition, let $R$ be a pre-order over $\kappa$, that is, a predicate of kind $\kappa \to \kappa \to \mathrm{PROP}$ that is provably reflexive and transitive. (I am glossing over the details of how these proofs are carried out.) For instance, if $\kappa$ is the kind $\mathbb{Z}$ of integer indices, $R$ could be the total order $\leq$ over the integers. I view a proposition, of kind PROP, as a non-linear capability, so that the existing machinery that deals with capabilities subsumes Hoare logic. (A function precondition is expressed by a proposition, viewed as a capability, in the position of a function argument; a function postcondition is expressed by a proposition in the position of a result.)

Now, generalize further the definition of $\otimes$, as follows:

$$\begin{array}{ll} & (\chi_1 \to \chi_2) \otimes I/R \\ = & \forall i.\big((\chi_1 \otimes I/R) * I\ i \to \exists j.((\chi_2 \otimes I/R) * I\ j * R\ i\ j)\big) \end{array}$$

Tensor is now a ternary operator, because it is parameterized over $I$ and $R$ instead of just $I$. I use the notation $\cdot \otimes I/R$. Note that, if $R$ is equality, this definition degenerates to the generalized tensor of §2.

The statements of the frame and anti-frame rules is unchanged with respect to §2, save the additional parameterization over $R$:

$$\frac{}{\chi \leq \chi \otimes I/R}\ \mathrm{FRAME} \qquad \frac{\Gamma \otimes I/R \vdash (\chi \otimes I/R) * \exists i.I\ i}{\Gamma \vdash \chi}\ \mathrm{GAF}$$

The "tensor" $\cdot \otimes I/R$ is used to express that, if some member $I\ i$ of the family holds at a call across the border (from Term to Context or from Context to Term), then *some stronger* member $I\ \ j$ holds when this call returns. The pre-state $i$ and the post-state $j$ are no longer required to be equal, but only related by $R$.

The generalized anti-frame rule above intuitively states that Term can *assume* that any call by Term to Context improves the state, and Term must *guarantee* that any call by Context to Term improves the state.

This is only a *local* monotonicity property, not a global one: there is no requirement that the invariant always gets stronger with respect to a global timeline. This is illustrated by the example in §3.2.

**Conjecture 2** *The above generalized higher-order frame and anti-frame rules are sound.* ◇

## 3.2 An example

The example of Figure 5 has appeared in the recent literature [1, 5], under the nickname "well-bracketed state change". I would like to change that nickname to "well-bracketed *monotonic* state change", so as to distinguish it from the examples considered earlier in this note (Figures 1 and 4), which *are* about well-bracketed state changes, but only involve preservation, not monotonicity.

This example is quite tricky: it seems difficult, at first sight, to understand what is going on, let alone to prove that the assertion on line 8 of Figure 5 cannot fail. Yet, the safety of this assertion is an immediate consequence of the generalized anti-frame rule, with:

$$\begin{array}{r} I\ i = \{x : \mathit{ref\ int}\ i\} * (i \in \{0, 1\}) \\ R = (\leq) \end{array}$$

Indeed, in order to prove that $m$ admits the external type $(\mathit{unit} \to \mathit{unit}) \to \mathit{unit}$, we must prove that it admits the internal type $(\mathit{unit} \to \mathit{unit}) \to \mathit{unit} \otimes I/R$. By definition, this type is:

$$\forall i_1.(\forall j_1.\mathit{unit} * I\ j_1 \to \exists j_2.\mathit{unit} * I\ j_2 * R\ j_1\ j_2) * I\ i_1 \to \exists i_2.\mathit{unit} * I\ i_2 * R\ i_1\ i_2$$

This is rather difficult to read. A translation in natural language would be:

> Admit that, if $x$ initially has value $j_1 \in \{0,1\}$, then a call to $f$ leaves $x$ with some new value $j_2 \in \{0,1\}$ such that $j_1 \leq j_2$.
>
> Prove that, if $x$ initially has value $i_1 \in \{0,1\}$, then a call to $m$ leaves $x$ with some new value $i_2 \in \{0,1\}$ such that $i_1 \leq i_2$.

The proof is obvious. After the assignment on line 6, $x$ holds the value 1 – that is, the capability $I\ 1$ is available. The hypothesis about $f$ implies that $x$ still holds 1 after the call to $f$ on line 7. (More precisely, we have $\exists j_2.(I\ j_2 * 1 \leq j_2)$, which provably implies $I\ 1$.) As a result, the assertion on line 8 is valid. Furthermore, since the final state of $x$ is 1, it is provably greater than or equal to its initial state $i_1$, which by hypothesis is a member of $\{0,1\}$.

In principle, a sufficiently advanced combination of a type-checker and a proof checker should be able to verify this proof.

It seems to me that that the natural-language version of this proof is about as short as it can get: $m$ "is monotonic" (read: $m$ has a monotonic effect on the hidden state) because $f$ can safely be assumed to be monotonic as well, and this assumption about $f$ is (recursively) justified by the fact that $f$ can affect the state only via $m$, which is monotonic.

This is only a *local* (or "well-bracketed") monotonicity property: although there is a guarantee that a call to $m$ or to $f$ causes $x$ to grow, there is no guarantee that $x$ grows over time. In fact, this is false, since the value of $x$ alternates between 0 and 1.

This example seems contrived. I would be interested in finding a real-world example of a local monotonicity property that is not also a global monotonicity property.

## 4. Conclusion

I am writing this draft, informal note in order to solicit some feedback from the community. At the moment, no type soundness proofs have been carried out. There are probably pieces of related work that I am not aware of. Please forgive these shortcomings.

I believe that these two generalized versions of the tensor operator are interesting, and could be useful in practice. The examples of Figures 1 and 4, for instance, correspond to realistic programming patterns.

Birkedal *et al.* [2] construct a model of the tensor operator and of the higher-order frame rule. In this model, every arrow is understood as implicitly polymorphic over all potential state extensions. It would be interesting to find out whether this model can be updated to support the generalized tensor operators presented in this note.

Alexandre Pilkiewicz and I are presently preparing a capability-based account of monotonicity. This account is orthogonal to most of the other features of the host programming language (such as mutable state, the frame rule, and the anti-frame rule), but interacts well with them. It is plausible that the second generalized anti-frame rule presented in this note could be *derived* from the first generalized anti-frame rule, combined with our general treatment of monotonicity.

Can the generalized anti-frame rules be derived from the basic anti-frame rule? It seems not, since the latter enforces an invariant property *of the state* alone, while the former enforce properties *of the state and of the (implicit) evaluation context in combination*. Can the generalized anti-frame rules be derived from the basic anti-frame rule, *up to a program transformation,* such as continuation-passing style transform, which makes the evaluation context explicit? Maybe – I have no idea.

At this time, it is not clear to me how far the tensor operator, and the frame and anti-frame rules, can (and should) be generalized. Are there further generalizations, waiting to be discovered? Could one prove that the current rules are complete in some sense? These are open questions.

## References


[1] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 340–353, January 2009.

[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] Eijiro Sumii. A complete characterization of observational equivalence in polymorphic lambda-calculus with general references. In *Computer Science Logic*, September 2009.