Hiding local state in direct style: a higher-order anti-frame rule

François Pottier

June 17th, 2008

INRIA

In a nutshell

Several type systems and logics keep *precise track* of how mutable state is allocated, affected, de-allocated, or aliased.

The upside of this idea is that this can help prove properties of programs.

The downside is that this can *betray* the use of mutable state in functions whose behavior is pure...

In a nutshell

A function that allocates a fresh piece of state, works with it, and de-allocates it prior to returning (à la runST) *can* be given a pure specification in these type systems and logics.

A function whose state persists across invocations *cannot*.

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

Hidden state

Many "*objects*" (*or* "modules", "*components*", "functions") rely *on* a piece *of* *modifiable internal state*, which *persists across invocations*, yet they publish an informal specification that *does* not reveal the *existence of* such a state.

Hidden state

For instance, a *memory manager* might maintain a linked list of freed memory blocks.

Yet, clients *need not*, and *wish not*, know anything about it.

They need not *even* be told that the memory manager has a certain abstract invariant.

Telling them so would force them to publish the fact that they require and preserve this invariant. In short, *every* (direct or indirect) client of the memory manager would have to *declare* itself as such! That would not be *modular.*

Hiding versus abstraction

Hiding is not abstraction. Hiding pretends that there is no internal state, while abstraction acknowledges that there is one, but makes its type (and properties) abstract.

Both *protect the internal state* from interference by clients, and *protect clients* from changes in the representation of the internal state.

Hiding versus abstraction

Hiding offers the additional advantage that objects with internal state appear as ordinary objects, hence can be *untracked.* It is not necessary to ask how they are *aliased*, who *owns* them, or how they internal state is *threaded* through client computations.

Abstraction offers the additional advantage that clients can reason about state changes. The computational state, which has abstract type, can be declared to *represent* some logical state, at a concrete type. For instance, the internal state of a hash table can be declared to represent a mathematical finite map.

In practice, *both* hiding and abstraction are useful, albeit in different circumstances.

Hiding versus abstraction

Consider an object that produces a stream of the prime numbers.

If it is specified that each invocation returns the *next* prime number, then the internal state can only be *abstract.*

If it is only specified that each invocation returns *some* prime number, then the state can be *hidden.*

Hiding versus abstraction

Whether an object's internal state can be hidden depends not on the object's actual behavior, but only on its *specification.*

As specifications become *weaker,* opportunities for hiding state *increase!*

When specifications are just *types,* which describe the structure of data and the structure of the store, these opportunities are quite numerous.

Towards a formalization

How could the concept of hidden state be made precise in a formal framework for reasoning about programs?

In this talk, I attempt to provide an answer...

Towards a formalization

Which formal frameworks provide an appropriate setting in which to ask (and answer) this question?

Any system that keeps track of *aliasing* and *ownership* properties, and allows expressing *pre-* and *post-conditions* that describe the structure of the store, should do.

Pick one of: Hoare logic / separation logic / bunched typing / type systems with regions and capabilities / Hoare type theory / you-name-it...

Towards a formalization

In this talk, I use the vocabulary of a type system for an ML-like programming language [Charguéraud and Pottier, 2008].

It should be possible to transpose the main idea to another setting. (If you think I should do so, please do come and talk to me!)

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

The host type system

This type system is the setting in which I develop *a rule for hiding state* and prove *(syntactic) type soundness.*

The details of the type system are somewhat unimportant for this talk, so I will flash them by...

Regions

A *region* σ is a static name for a set of values.

The type $[\sigma]$ is the type of the values that inhabit the region σ.

In this talk, there are only *singleton regions*, so a region σ is a static name for a value, and $[\sigma]$ is a singleton type.

Capabilities

A *singleton capability* $\{\sigma : \theta\}$ is a static token that serves two roles.

First, it carries a *memory type* θ, which describes the structure and extent of the memory area to which the value σ gives access. Second, it represents *ownership* of this area.

For instance, $\{\sigma : \text{ref int}\}$ asserts that the value σ is the address of a reference cell, and asserts ownership of this cell.

Capabilities are *linear:* they are never duplicated.

Capabilities (summary)

On top of singleton capabilities, one builds *composite capabilities:*

C	$::=$	$\emptyset$	*empty heap*
	$\mid$	$\{\sigma : \theta\}$	*singleton heap*
	$\mid$	$C_1 * C_2$	*(separating) conjunction*
	$\mid$	$\exists \sigma . C$	*embedded region*
	$\mid$	$C_1 \otimes C_2$	*(explained later on)*

There is a clear analogy between capabilities and *separation logic assertions.*

Memory types

Here is a summary of memory types:

$$\begin{array}{rcll}
\theta & ::= & \bot \mid \mathsf{unit} \mid \theta_1 + \theta_2 \mid \theta_1 \times \theta_2 & \text{data} \\
& \mid & \chi_1 \rightarrow \chi_2 & \text{functions} \\
& \mid & [\sigma] & \text{indirection via a region} \\
& \mid & \mathsf{ref}\,\theta & \text{reference cell} \\
& \mid & \theta * C & \text{embedded capability} \\
& \mid & \exists \sigma.\theta & \text{embedded region} \\
& \mid & \theta \otimes C & \text{(explained later on)}
\end{array}$$

Memory types express ownership, so they are *linear.*

Value types (summary)

Values receive *value types:*

$$\begin{array}{rcll} \tau & ::= & \bot \mid \mathsf{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 & \textit{data} \\ & \mid & \chi_1 \rightarrow \chi_2 & \textit{functions} \\ & \mid & [\sigma] & \textit{indirection via a region} \\ & \mid & \tau \otimes C & \textit{(explained later on)} \end{array}$$

Values are *non-linear:* they can be discarded or duplicated at will.

Value types form a subset of memory types, deprived of references and embedded capabilities.

Judgements about values

Judgements about *values* take the form:

$$\Delta \vdash v : \tau$$

Duplicable type environments Δ associate value types with variables.

Values do not involve computation, which is why this judgement form does not involve any capabilities, either as input or as output.

Judgements about terms

Judgements about *terms* take the form:

$$\Gamma \Vdash t : \chi$$

The *type environment* Γ and the *computation type* χ respectively describe the initial and final shapes of the store. This is analogous to a Hoare triple in separation logic.

A computation type is a value type, plus regions and capabilities. A type environment associates computation types with variables, and contains capabilities.

$$\begin{array}{lcl} \chi & ::= & \tau \mid \chi * C \mid \exists \sigma.\chi \mid \chi \otimes C \\ \Gamma & ::= & \varnothing \mid \Gamma, x : \chi \mid \Gamma, C \mid \Gamma \otimes C \end{array}$$

Typing rules for references

References are *tracked:* allocation produces a singleton capability, which is later required for access.

$$\begin{array}{rcl}
\text{ref} & : & \tau \longrightarrow \exists\sigma.\{\sigma : \text{ref}\ \tau\}\,[\sigma] \\
\text{get} & : & \{\sigma : \text{ref}\ \tau\}\,[\sigma] \longrightarrow \{\sigma : \text{ref}\ \tau\}\,\tau \\
\text{set} & : & \{\sigma : \text{ref}\ \tau_1\}\,([\sigma] \times \tau_2) \longrightarrow \{\sigma : \text{ref}\ \tau_2\}\,\text{unit}
\end{array}$$

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

The first-order frame rule

The first-order *frame rule* states that, if a term behaves correctly in a certain store, then it also behaves correctly in a larger store, and does not affect the part of the store that it does not know about:

$$\frac{\Gamma \Vdash t : \chi}{\Gamma, C \Vdash t : \chi * C}$$

This rule can also take the form of a simple subtyping axiom:

$$\chi_1 \rightarrow \chi_2 \quad \leq \quad (C * \chi_1) \rightarrow (C * \chi_2)$$

The first-order frame rule

The frame rule makes a capability *unknown to a term*, while *known to its context.*

To hide a piece of local state is the exact dual: to make a capability *known to a term,* yet *unknown to its context.*

Hidden state via frame rules

In a programming language with higher-order functions, one could hope to be able to exploit the duality between terms and contexts.

By *viewing the context as a term,* a continuation, one could perhaps use a frame rule to hide a piece of local state.

This is the approach of Birkedal, Torp-Smith, and Yang [2006], who follow up on earlier work by O'Hearn, Yang, and Reynolds [2004].

Hidden state via frame rules

Imagine that we have a *provider*, a term of type:

$$((\mathsf{unit} * C) \longrightarrow (\mathsf{int} * C)) * C$$

The provider initially establishes C and returns a function that requires C and preserves it.

This could be the type of a stream of integers, with internal state.

Hidden state via frame rules

We now wish to *hide* C and pretend that the provider is an ordinary function, of type unit $\longrightarrow$ int.

Applying the frame rule to the provider would not help.

We must apply the frame rule to the *client.*

Hidden state via frame rules

Imagine the *client* is a term of type:

$$(\mathsf{unit} \longrightarrow \mathsf{int}) \longrightarrow a$$

This client is *explicitly abstracted* over the provider. The type a is some answer type.

The client *does not know* about the invariant C. It views the provider as an ordinary function, without side effects.

Hidden state via frame rules

The first-order frame rule alone does not help type-check the function application (client provider).

This is where Birkedal *et al.*'s *higher-order frame rule* [2006] comes into play. The rule guarantees:

$$(\mathsf{unit} \rightarrow \mathsf{int}) \rightarrow a \quad \leq \quad (C * (C * \mathsf{unit} \rightarrow C * \mathsf{int})) \rightarrow (C * a)$$

That is, if C holds initially and if the provider preserves C, then, the client will unwittingly preserve it as well.

Hidden state via frame rules

After applying the higher-order frame rule, the client has type:

$$(C * (C * \mathrm{unit} \longrightarrow C * \mathrm{int})) \longrightarrow (C * a)$$

Recall that the provider has type:

$$((\mathrm{unit} * C) \longrightarrow (\mathrm{int} * C)) * C$$

So the function application (client provider) is in fact *well-typed*, and has type $C * a$.

Hidden state via frame rules

In a modular setting, the client is unknown. One must abstract the provider over the client. If one admits the subtyping axiom $C \leq \emptyset$, then the value:

$$\lambda \text{client}.(\text{client provider})$$

has type:

$$((\text{unit} \longrightarrow \text{int}) \longrightarrow a) \longrightarrow a$$

This is the *double negation* of the desired type.

We succeeded, but were led to use *continuation-passing style.*

Hidden state via frame rules

Is this approach to hidden state realistic?

I claim *not:* continuation-passing style is not practical.

What is a *direct-style* analogue of the higher-order frame rule?

Towards a higher-order anti-frame rule

We need a (higher-order) *anti-frame* rule, that is, a rule that explains hidden local state without requiring a switch to continuation-passing style.

The higher-order frame rule

Let me first recall the higher-order frame rule.

Its general form is:

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

The type $\chi \otimes C$ ("χ under C") describes the same behavior as χ, and additionally requires C to be available at every interaction between the term and its context.

The higher-order frame rule

The operator $\cdot \otimes C$ makes C a new pre-condition and a new post-condition of *every* arrow within its left-hand argument:

$$(\chi_1 \rightarrow \chi_2) \otimes C \;=\; (C * (\chi_1 \otimes C)) \rightarrow (C * (\chi_2 \otimes C))$$

The operator $\cdot \otimes C$ commutes with products, sums, references, etc. It vanishes at base types.

Towards a higher-order anti-frame rule

A reasonable approximation of the anti-frame rule is:

$$(\chi \otimes C) * C \quad \leq \quad \chi \qquad\qquad \text{(unsound)}$$

The rule states that:

- Term must *guarantee* C when abandoning control to Context;
- then, C will hold whenever Context has control, even though Context *does not know* about C;
- thus, Term may *assume* C when receiving control from Context.

Towards a higher-order anti-frame rule

The candidate rule on the previous slide is sound only for *closed* terms that run in an *empty* store.

In general, interaction between Term and Context takes place also via the function values found in the environment or in the store.

As a result, *the type environment* and *the type of the store* too must have internal and external versions.

A higher-order anti-frame rule

A sound version of the rule is:

$$\text{Anti-frame}\quad \frac{\Gamma \otimes C \Vdash t : (\chi \otimes C) * C}{\Gamma \Vdash t : \chi}$$

This is *dual* to the frame rule: the invariant C is known inside, unknown outside.

Type soundness

The type system is proven sound via a standard syntactic argument, which involves *subject reduction* and *progress* theorems.

A key lemma is *Revelation:* a valid judgement remains valid after a previously hidden invariant R is revealed.

Lemma (Revelation)

$$\begin{array}{lll} \Gamma \vdash v : \tau & \textit{implies} & \Gamma \otimes R \vdash v : \tau \otimes R \\ \Gamma \Vdash t : \chi & \textit{implies} & (\Gamma \otimes R), R \Vdash t : (\chi \otimes R) * R \end{array}$$

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

Applications

If there is time, I would like to present three applications of the anti-frame rule:

- untracked references, in the style of ML;
- untracked lazy thunks;
- a generic fixed point combinator.

Untracked references

In this type system, references are *tracked:* a reference cannot be read or written unless an appropriate capability is presented. This is heavy — capabilities are *linear* — but allows *strong update.*

In ML, references are *untracked:* no capability is required to read or write a cell, and references can be aliased. This is lightweight, but the type of a reference must remain *fixed* forever.

Untracked references

Tracked and untracked references have different qualities, so it seems pragmatically desirable for a programming language to offer both.

The good news is, in theory, *untracked references can be encoded* in terms of tracked references and the anti-frame rule.

Untracked references

The following two slides present the encoding.

For simplicity, the first slide shows integer references. The second slide presents the general case of references to an arbitrary value type a.

Untracked integer references

```
def type uref =                                  — a non-linear type!
  (unit → int) × (int → unit)

let mkuref : int → uref =
λ(v : int).
  let ρ, (r : [ρ]) = ref v in                     — got { ρ: ref int }
  hide R = { ρ: ref int } outside of
  let uget : (unit * R) → (int * R) =
    λ(). get r
  and uset : (int * R) → (unit * R) =
    λ(v : int). set (r, v)
  in (uget, uset)                                 — this pair has type uref ⊗ R
                                                  — to the outside, uref
```

Generic untracked references

```
def type uref $a$ =                                        — parameterize over $a$
  (unit $\longrightarrow$ $a$) $\times$ ($a$ $\longrightarrow$ unit)

let mkuref : $\forall a.a \longrightarrow$ uref $a$ =
$\lambda$(v : $a$).
  let $\rho$, (r : [$\rho$]) = ref v in                    — got { $\rho$: ref $a$ }
  hide $R$ = { $\rho$: ref $a$ } $\otimes$ $R$ outside of  — got { $\rho$: ref $a$ } $\otimes$ $R$
  let uget : (unit * $R$) $\longrightarrow$ (($a \otimes R$) * $R$) =   — that is, $R$
    $\lambda$(). get r                                     — also { $\rho$: ref ($a \otimes R$) }
  and uset : (($a \otimes R$) * $R$) $\longrightarrow$ (unit * $R$) =
    $\lambda$(v : $a \otimes R$). set (r, v)
  in (uget, uset)                                          — type: (uref $a$) $\otimes$ $R$
                                                           — to the outside, uref $a$
```

Lazy thunks

I now define *lazy thunks*, which are built once and can be forced any number of times.

Thunks are untracked and can be freely aliased.

A thunk contains a hidden reference to an internal state with three possible colors (unevaluated, being evaluated, evaluated).

Let's have a look at a slightly simplified version, first...

Lazy thunks — part 1

```
def type thunk $a$ =
  unit $\longrightarrow$ $a$

def type state $a$ =                      — internal state:
  W unit + G unit + B $a$                 — white/grey/black

let mkthunk : $\forall a.$(unit $\longrightarrow$ $a$) $\longrightarrow$ thunk $a$ =
  $\lambda$(f : unit $\longrightarrow$ $a$).
    let $\rho$, (r : [$\rho$]) = ref (W ()) in   — got { $\rho$: ref (state $a$) }
    hide $R$ = { $\rho$: ref (state $a$) } $\otimes$ $R$ outside of
      .                                   — got $R$
      .                                   — f: (unit $\longrightarrow$ $a$) $\otimes$ $R$
      .                                   — f: (unit $*$ $R$) $\longrightarrow$ (($a$ $\otimes$ $R$) $*$ $R$)
```

Lazy thunks — part 2

```
let force : (unit * R) → ((a ⊗ R) * R) =
  λ().                                  — state a = W unit + G unit + B a
    case get r of                       — got R = { ρ: ref (state a) } ⊗ R
    | W () →
      set (r, G ());                    — got R
      let v : (a ⊗ R) = f() in          — got R
      set (r, B v);                     — got R
      v
    | G () → fail
    | B (v : a ⊗ R) → v
  in force                              — force: (thunk a) ⊗ R
                                        — to the outside, thunk a
```

Lazy thunks — with a one-shot guarantee

The code on the previous slides could be broken in several ways, *e.g.* by failing to distinguish white and grey colors, or by failing to color the thunk grey before invoking f, *while remaining well-typed.*

We would like the type system to catch these errors, and to guarantee that *the client function f is invoked at most once.*

This can be done by making f a *one-shot function* — a function that requires a capability, but does not return it — and providing “mkthunk” with *a single cartridge*.

Lazy thunks — part 1

```
def type thunk a =
  unit → a

def type state γ a =
  W (unit * γ) + G unit + B a        — when white, γ is available

let mkthunk : ∀γa.(((unit * γ) → a) * γ) → thunk a =
  λ(f : (unit * γ) → a).             — got γ
    let ρ, (r : [ρ]) = ref (W ()) in — got { ρ: ref (state γ a) }
    hide R = { ρ: ref (state γ a) } ⊗ R outside of
      .                              — got R
      .                              — f: ((unit * γ) → a) ⊗ R
      .                              — f: (unit * R * (γ ⊗ R)) → ((a ⊗ R) * R)
```

Lazy thunks — part 2

```
let force : (unit ∗ R) ⟶ ((a ⊗ R) ∗ R) =
  λ().                                  — state γ a = W (unit ∗ γ) + G unit + B a
    case get r of                       — got R = { ρ: ref (state γ a) } ⊗ R
    | W () ⟶                            — got { ρ: ref (W unit + G ⊥ + B ⊥) } ∗ (γ ⊗
      set (r, G ());                    — got R ∗ (γ ⊗ R)
      let v : (a ⊗ R) = f() in          — got R; (γ ⊗ R) was consumed by f
      set (r, B v);                     — got R
      v
    | G () ⟶ fail                       — without γ ⊗ R, invoking f is forbidden
    | B (v : a ⊗ R) ⟶ v
  in force                              — force: (thunk a) ⊗ R
                                        — to the outside, thunk a
```

A fixed point combinator

The fixed point combinator *ties a knot in the store* in the style of Landin.

It is perhaps not very surprising, but illustrates:

- a use of the anti-frame rule at order 3;
- a *delayed* initialization, via a strong update;
- a hidden invariant that *does not* hold upon entry, but *does* hold upon *exit*, of the **hide** construct.

A fixed point combinator

```
let fix : ∀a₁a₂.((a₁ → a₂) → (a₁ → a₂)) → a₁ → a₂ =
λ(f : (a₁ → a₂) → (a₁ → a₂)).
  let ρ, (r : [ρ]) = ref () in          – got { ρ: ref unit }
  hide R = { ρ: ref (a₁ → a₂) } ⊗ R outside of
  .                                     – haven't got R yet!
  let g : (a₁ → a₂) ⊗ R =               – g invokes !r
    λ(x : a₁ ⊗ R). get r x              – within g, got R
  in let h : (a₁ → a₂) ⊗ R =            – h invokes f, routing recursive calls to g
    λ(x : a₁ ⊗ R). f g x                – f: ((a₁ → a₂) → (a₁ → a₂)) ⊗ R
  in set (r, h);                        – a strong update establishes R
  h                                     – got R now, as required by anti-frame
                                        – h: (a₁ → a₂) ⊗ R
                                        – to the outside, a₁ → a₂
```

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

Conclusion

In summary, a couple of key ideas are:

- a practical rule for hiding state must be in *direct style;*
- it is safe for a piece of hidden state to be *untracked,* as long as *its invariant holds at every interaction* between Term and Context.

There are more details in the paper [Pottier, 2008].

Future work

Here are a few directions for future research:

- formally *relate frame and anti-frame* via a CPS transform;
- extend the *functional interpretation* developed with Charguéraud in the absence of anti-frame.

Appendix: typing rules for values

var
$$\frac{(x : \tau) \in \Delta}{\Delta \vdash x : \tau}$$

unit
$$\frac{}{\Delta \vdash () : \mathsf{unit}}$$

inj
$$\frac{\Delta \vdash v : \tau_i}{\Delta \vdash (\mathsf{inj}^i\ v) : (\tau_1 + \tau_2)}$$

prim
$$\frac{p : \tau}{\Delta \vdash p : \tau}$$

pair
$$\frac{\Delta \vdash v_1 : \tau_1 \qquad \Delta \vdash v_2 : \tau_2}{\Delta \vdash (v_1, v_2) : (\tau_1 \times \tau_2)}$$

fun
$$\frac{\Delta, x : \chi_1 \Vdash t : \chi_2}{\Delta \vdash (\lambda x.t) : \chi_1 \longrightarrow \chi_2}$$

Appendix: typing rules for terms

val
$$\frac{\Delta \vdash v : \tau}{\Delta \Vdash v : \tau}$$

app
$$\frac{\Delta \vdash v : \chi_1 \to \chi_2 \qquad \Delta, \Gamma \Vdash t : \chi_1}{\Delta, \Gamma \Vdash (v\ t) : \chi_2}$$

sub
$$\frac{\Gamma \Vdash t : \chi_1 \qquad \chi_1 \leq \chi_2}{\Gamma \Vdash t : \chi_2}$$

∗-intro (frame)
$$\frac{\Gamma \Vdash t : \chi}{\Gamma, C \Vdash t : \chi * C}$$

∗-elim-1
$$\frac{\Gamma, (x : \chi_1), C \Vdash t : \chi_2}{\Gamma, x : (\chi_1 * C) \Vdash t : \chi_2}$$

∗-elim-2
$$\frac{\Gamma, C_1, C_2 \Vdash t : \chi}{\Gamma, (C_1 * C_2) \Vdash t : \chi}$$

∃-elim-1
$$\frac{\Gamma, x : \chi_1 \Vdash t : \chi_2}{\Gamma, x : \exists \sigma . \chi_1 \Vdash t : \chi_2}$$

∃-elim-2
$$\frac{\Gamma, C \Vdash t : \chi}{\Gamma, \exists \sigma . C \Vdash t : \chi}$$

anti-frame
$$\frac{\Gamma \otimes C \Vdash t : (\chi \otimes C) * C}{\Gamma \Vdash t : \chi}$$

Revelation: excerpt of proof

Here is the case of an application:

$$\frac{\Gamma \vdash v : \chi_1 \longrightarrow \chi_2 \qquad \Gamma; C \Vdash t : \chi_1}{\Gamma; C \Vdash (v\ t) : \chi_2} \qquad \text{becomes} \qquad \frac{\Gamma \otimes R \vdash v : (\chi_1 \longrightarrow \chi_2) \otimes R \qquad \Gamma \otimes R; R * (C \otimes R) \Vdash t : R * (\chi_1 \otimes R)}{\Gamma \otimes R; R * (C \otimes R) \Vdash (v\ t) : R * (\chi_2 \otimes R)}$$

This is still a valid application, thanks to the equality:

$$(\chi_1 \longrightarrow \chi_2) \otimes R = (R * (\chi_1 \otimes R)) \longrightarrow (R * (\chi_2 \otimes R))$$

How Revelation is used

The gist of the subject reduction proof is that *anti-frame extrudes up through evaluation contexts:*

$$
\dfrac{\dfrac{\dfrac{\Delta}{\Gamma \otimes R ; C \otimes R \Vdash t : R * (\chi \otimes R)}}{\Gamma ; C \Vdash t : \chi}\,\text{AF}}{\dfrac{\cdots}{\Gamma' ; C' \Vdash E[t] : \chi'}}
\qquad
\dfrac{\dfrac{\dfrac{\Delta}{\Gamma \otimes R ; C \otimes R \Vdash t : R * (\chi \otimes R)}}{\cdots \otimes R}}{\dfrac{\Gamma' \otimes R ; R * (C' \otimes R) \Vdash E[t] : R * (\chi' \otimes R)}{\Gamma' ; C' \Vdash E[t] : \chi'}\,\text{AF}}
$$

The proof is immediate: *apply Revelation to* (the type derivation for) *the evaluation context $E[\cdot]$.*

How Revelation is used

This proof technique *backs up the intuition* that an application of the anti-frame rule amounts to an application of the higher-order frame rule to the evaluation context.

Note: I am quite confident that the type system is sound, but am not done writing the proof yet.

Contents

- Introduction
- Basics of the type system
- A higher-order anti-frame rule
- Applications
- Conclusion
- Bibliography

Bibliography I

(Most titles are clickable links to online versions.)

Birkedal, L., Torp-Smith, N., and Yang, H. 2006.
Semantics of separation-logic typing and higher-order frame rules for Algol-like languages.
Logical Methods in Computer Science 2, 5 (Nov.).

Charguéraud, A. and Pottier, F. 2008.
Functional translation of a calculus of capabilities.
In *ACM International Conference on Functional Programming (ICFP)*.
To appear.

O'Hearn, P., Yang, H., and Reynolds, J. C. 2004.
Separation and information hiding.
In *ACM Symposium on Principles of Programming Languages (POPL)*.
268–280.

Bibliography]Bibliography

Pottier, F. 2008.
Hiding local state in direct style: a higher-order anti-frame rule.
Submitted.