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

François Pottier

January 28th, 2008

INRIA

Contents

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

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

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

It is safe for them to consider that the allocation and deallocation functions have *no side effect*, other than the obvious effect of providing the client with, or depriving the client from, ownership of a unique memory block.

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 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* or who *owns* them.

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 represents a mathematical finite map.

In practice, *both* hiding and abstraction are useful.

Consider an object that produces the 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 just on the object's actual behavior, but also on its *specification.*

As specifications become *less precise,* opportunities for hiding state *increase!*

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

This talk attempts to provide an answer...

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

There are several. *Separation logic* is one. A *type system* with regions and capabilities is another.

In fact, the two are quite close: both keep track of *aliasing* and *ownership* properties. Both allow assigning *pre-* and *post-conditions* to code.

Hidden state has been previously studied in the setting of separation logic [O'Hearn et al., 2004, Birkedal et al., 2006].

In this talk, I use the vocabulary of a type system [Charguéraud and Pottier, 2007] for an ML-like programming language.

It should be possible to transpose the main idea to the setting of separation logic.

Contents

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. I just wish to convey a flavor of the system before we embark on a journey towards hidden state...

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

The type $[\rho]$ is the type of the values that inhabit the region ρ.

In this talk, there are only *singleton regions*, so a region ρ is a static name for a value, and $[\rho]$ is a singleton type. My paper with Arthur Charguéraud [2007] also has *group regions*, which become necessary when there is *aliasing*.

A *singleton capability* $\{\rho : \theta\}$ 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 ρ gives access. Second, it represents *ownership* of this area.

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

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

$$
\begin{array}{lcll}
C & ::= & \emptyset & \text{empty heap} \\
  & | & \{\rho : \theta\} & \text{singleton heap} \\
  & | & C_1 \wedge C_2 & \text{(separating) conjunction} \\
  & | & \exists \rho . C & \text{embedded region} \\
  & | & C_1 \otimes C_2 & \text{(explained later on)}
\end{array}
$$

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

Here is a summary of memory types:

θ	$::=$	$\bot \mid \mathsf{unit} \mid \theta_1 + \theta_2 \mid \theta_1 \times \theta_2$	data
	$\mid$	$\sigma_1 \rightarrow \sigma_2$	functions
	$\mid$	$[\rho]$	indirection via a region
	$\mid$	$\mathsf{ref}\ \theta$	reference cell
	$\mid$	$C \wedge \theta$	embedded capability
	$\mid$	$\exists\rho.\theta$	embedded region
	$\mid$	$\theta \otimes C$	(explained later on)

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

I assume that capabilities and types can be *recursive*. For instance, the unique solution to the following equation:

$$R = \{\rho : \mathsf{ref}\ ((R \wedge \sigma_1) \longrightarrow (R \wedge \sigma_2))\}$$

is a singleton capability R for a reference cell that contains a function that requires, and preserves, R.

Recursive capabilities are required in *many applications*, and, in this system, appear necessary for the *subject reduction* proof to go through.

Values receive *value types:*

$$
\begin{array}{lll}
\tau & ::= & \bot \mid \mathsf{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
 & \mid & \sigma_1 \rightarrow \sigma_2 \\
 & \mid & [\rho] \\
 & \mid & \tau \otimes C
\end{array}
$$

data

functions

indirection via a region

(explained later on)

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:

$$\Gamma \vdash v : \tau$$

Type environments Γ 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* take the form:

$$\Gamma ; C \vdash t : \sigma$$

The *capability C* serves a *pre-condition*, while the *computation type σ* serves as a *post-condition.* Judgements about terms are analogous to Hoare triples in separation logic.

Computation types are:

$$\sigma ::= \tau \mid C \wedge \sigma \mid \exists \rho . \sigma \mid \sigma \otimes C$$

References are *tracked*: allocation produces a singleton capability, which is later required for access. Read and write accesses are restricted to non-linear *value types*, because they duplicate or discard a value.

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

References to non-value types can be created and exploited via *focusing*:

$$
\text{(focus-ref)} \qquad \{\rho_1 : \text{ref}\ \theta\} \equiv \exists \rho_2.\{\rho_1 : \text{ref}\ [\rho_2]\}\{\rho_2 : \theta\}
$$

Contents

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:

$$\frac{\Gamma; C_2 \vdash t : \sigma}{\Gamma; (C_1 \wedge C_2) \vdash t : (C_1 \wedge \sigma)}$$

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

$$\sigma_1 \rightarrow \sigma_2 \quad \leq \quad (C \wedge \sigma_1) \rightarrow (C \wedge \sigma_2)$$

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.*

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].

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

$$C \wedge ((C \wedge \text{unit}) \longrightarrow (C \wedge \text{int}))$$

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.

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

Applying the frame rule to the provider would not help.

We must apply the frame rule to the *client*, assuming it is known.

Imagine that we also have a *client*, 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.

At first, the function application (client provider) seems ill-typed. The provider offers:

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

while the client requires:

$$\mathrm{unit} \longrightarrow \mathrm{int}$$

The former is *not a subtype* of the latter: in fact, according to the first-order frame rule, it is the other way around!

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

$$(\text{unit} \longrightarrow \text{int}) \longrightarrow a \quad \leq \quad (C \wedge (C \wedge \text{unit} \longrightarrow C \wedge \text{int})) \longrightarrow (C \wedge a)$$

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

Here, the first-order frame rule would yield a weaker statement:

$$(\text{unit} \longrightarrow \text{int}) \longrightarrow a \quad \leq \quad (C \wedge (\text{unit} \longrightarrow \text{int})) \longrightarrow (C \wedge a)$$

The general form of the higher-order frame rule is:

$$\sigma \quad \leq \quad \sigma \otimes C$$

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

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

$$(\sigma_1 \rightarrow \sigma_2) \otimes C \;=\; (C \wedge (\sigma_1 \otimes C)) \rightarrow (C \wedge (\sigma_2 \otimes C))$$

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

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

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

Recall that the provider has type:

$$C \wedge ((C \wedge \text{unit}) \longrightarrow (C \wedge \text{int}))$$

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

In a modular setting, the client is unknown. One can then 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.*

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 applies not to the term, but to its context, without requiring an explicit switch to continuation-passing style.

Towards a higher-order anti-frame rule

An approximation of such a rule is:

$$C \wedge (\sigma \otimes C) \quad \leq \quad \sigma \qquad\qquad \text{(unsound)}$$

The left-hand side of the rule states that:

- Term must *guarantee* C when abandoning control to Context;
- Term may *assume* C when receiving control from Context;

In that case, it should be safe for Context to *not know* about C. The intended invariant is, C holds whenever Context has control.

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 that can be reached via the environment or the store.

As a result, *the type environment* and *the type of the store* too must come in two versions. Term's view is that *C* holds at every interaction, while Context's view does not even mention *C*.

A sound version of the rule is:

Anti-frame

$$\frac{\Gamma \otimes C_1 \,;\, C_2 \otimes C_1 \;\vdash\; t : C_1 \wedge (\sigma \otimes C_1)}{\Gamma \,;\, C_2 \;\vdash\; t : \sigma}$$

This is *dual* to the frame rule: the invariant C_1 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:* roughly speaking, a valid type derivation would remain valid if all hidden capabilities were revealed to the world.

A valid judgement remains valid after a previously hidden invariant R is revealed:

Lemma (Revelation)

$$\Gamma \vdash v : \tau \quad \textit{implies} \quad \Gamma \otimes R \vdash v : \tau \otimes R$$
$$\Gamma ; C \vdash t : \sigma \quad \textit{implies} \quad \Gamma \otimes R ; R \wedge (C \otimes R) \vdash t : R \wedge (\sigma \otimes R)$$

Here is the case of an application:

$$\frac{\Gamma \vdash v : \sigma_1 \to \sigma_2 \qquad \Gamma ; C \vdash t : \sigma_1}{\Gamma ; C \vdash (v\ t) : \sigma_2} \quad \text{becomes} \quad \frac{\Gamma \otimes R \vdash v : (\sigma_1 \to \sigma_2) \otimes R \qquad \Gamma ; R \wedge (C \otimes R) \vdash t : R \wedge (\sigma_1 \otimes R)}{\Gamma \otimes R ; R \wedge (C \otimes R) \vdash (v\ t) : R \wedge (\sigma_2 \otimes R)}$$

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

$$(\sigma_1 \to \sigma_2) \otimes R = (R \wedge (\sigma_1 \otimes R)) \to (R \wedge (\sigma_2 \otimes R))$$

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

$$\dfrac{\dfrac{\dfrac{\Delta}{\Gamma \otimes R ; C \otimes R \vdash t : R \wedge (\sigma \otimes R)}}{\Gamma ; C \vdash t : \sigma}\,\text{AF}}{\dfrac{\dots}{\Gamma' ; C' \vdash E[t] : \sigma'}}
\qquad
\dfrac{\dfrac{\dfrac{\Delta}{\Gamma \otimes R ; C \otimes R \vdash t : R \wedge (\sigma \otimes R)}}{\dots \otimes R}}{\dfrac{\Gamma' \otimes R ; R \wedge (C' \otimes R) \vdash E[t] : R \wedge (\sigma' \otimes R)}{\Gamma' ; C' \vdash E[t] : \sigma'}\,\text{AF}}$$

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

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

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 reasoning about *state changes.*

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.

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

But wouldn't that be redundant?

Yes. Type theorists will be happy to hear that, at least in principle, *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.

```
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 : (R ∧ unit) → (R ∧ int) =
    λ(). get r
  and uset : (R ∧ int) → (R ∧ unit) =
    λ(v : int). set (r, v)
  in (uget, uset)                                 — this pair has type uref ⊗ R
                                                  — to the outside, uref
```

```
def type uref a =                                  — parameterize over a
  (unit → a) × (a → unit)

let mkuref : ∀a.a → uref a =
λ(v : a).
  let ρ, (r : [ρ]) = ref v in                      — got { ρ: ref a }
  hide R = { ρ: ref a } ⊗ R outside of             — got { ρ: ref a } ⊗ R
  let uget : (R ∧ unit) → (R ∧ (a ⊗ R)) =          — that is, R
    λ(). get r                                     — also { ρ: ref (a ⊗ R) }
  and uset : (R ∧ (a ⊗ R)) → (R ∧ unit) =
    λ(v : a ⊗ R). set (r, v)
  in (uget, uset)                                  — type: (uref a) ⊗ 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. Yet, the type system guarantees that *each thunk is evaluated at most once.*

A thunk contains a hidden reference to an internal state with *three* possible colors (unevaluated, being evaluated, evaluated). Any attempt to ignore the dangers of *re-entrancy* and use only two colors would be ill-typed, by virtue of the anti-frame rule.

```
def type thunk α =
  unit ⟶ α

def type state γ α =                        — internal state:
  W (γ ∧ unit) + G unit + B α               — white/grey/black

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

```
let force : (R ∧ unit) → (R ∧ a ⊗ R) =
  λ().                          — state γ a = W (γ ∧ unit) + 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
```

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.

```
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
```

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].

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.

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

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

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

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

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

fun
$$\frac{(\Gamma,x:\tau);C\vdash t:\sigma\qquad \bar\rho\ \#\ \Gamma,\sigma}{\Gamma\vdash (\lambda x.\,t):(\exists\bar\rho.(C\wedge\tau))\rightarrow\sigma}$$

val
$$\frac{\Gamma \vdash v : \tau}{\Gamma ; C \vdash v : C \wedge \tau}$$

app
$$\frac{\Gamma \vdash v : \sigma_1 \to \sigma_2 \qquad \Gamma ; C \vdash t : \sigma_1}{\Gamma ; C \vdash (v\ t) : \sigma_2}$$

sub-left
$$\frac{\Gamma ; C_2 \vdash t : \sigma \qquad C_1 \leq C_2}{\Gamma ; C_1 \vdash t : \sigma}$$

sub-right
$$\frac{\Gamma ; C \vdash t : \sigma_1 \qquad \sigma_1 \leq \sigma_2}{\Gamma ; C \vdash t : \sigma_2}$$

$\exists\rho$-elim
$$\frac{\Gamma ; C \vdash t : \sigma \qquad \rho \# \Gamma, \sigma}{\Gamma ; (\exists \rho . C) \vdash t : \sigma}$$

frame
$$\frac{\Gamma ; C_2 \vdash t : \sigma}{\Gamma ; (C_1 \wedge C_2) \vdash t : (C_1 \wedge \sigma)}$$

anti-frame
$$\frac{\Gamma \otimes C_1 ; C_2 \otimes C_1 \vdash t : C_1 \wedge (\sigma \otimes C_1)}{\Gamma ; C_2 \vdash t : \sigma}$$

$$
\begin{aligned}
\text{func} &: \quad \tau \equiv \exists\rho.\{\rho : \tau\}\,[\rho] \\
\text{free} &: \quad C \leq \emptyset \\
\text{embed-rgn} &: \quad \{\rho_1 : \exists\rho_2.\theta\} \equiv \exists\rho_2.\{\rho_1 : \theta\} \\
\text{embed-cap} &: \quad \{\rho_1 : C \wedge \theta\} \equiv C \wedge \{\rho_1 : \theta\}
\end{aligned}
$$

$$\begin{aligned} \mathsf{proj}^1 &: \{\rho : \tau_1 \times \theta_2\}\,[\rho] \rightarrow \{\rho : \tau_1 \times \theta_2\}\,\tau_1 \\ \mathsf{focus\text{-}pair}^1 &: \{\rho : \theta_1 \times \theta_2\} \equiv \exists \rho_1 . \{\rho : [\rho_1] \times \theta_2\}\{\rho_1 : \theta_1\} \end{aligned}$$

$$
\begin{aligned}
\mathit{case} \quad &: \quad \{\rho : \theta_1 + \theta_2\}\,([\rho] \\
&\qquad\qquad \times ((\exists \rho_1.\{\rho : [\rho_1] + \bot\}\{\rho_1 : \theta_1\}\,[\rho_1]) \to \sigma) \\
&\qquad\qquad \times ((\exists \rho_2.\{\rho : \bot + [\rho_2]\}\{\rho_2 : \theta_2\}\,[\rho_2]) \to \sigma)) \to \sigma
\end{aligned}
$$

$$
\begin{aligned}
\text{sub-sum}^1 \quad &: \quad \{\rho : \theta_1 + \bot\} \le \{\rho : \theta_1 + \theta_2\} \\
\text{focus-sum}^1 \quad &: \quad \{\rho : \theta_1 + \bot\} \equiv \exists \rho_1.\{\rho : [\rho_1] + \bot\}\{\rho_1 : \theta_1\}
\end{aligned}
$$

(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. 2007.
Functional translation of a calculus of capabilities.
Submitted.

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.