Hidden state and the anti-frame rule

François Pottier

December 2nd, 2009

Contents

Precise systems for mutable state

This work assumes the following two basic ingredients:

- a programming language in the style of ML, with first-class, higher-order *functions* and *references;*
- a type system, or a program logic, that keeps track of *ownership* and *disjointness* information about the mutable regions of memory.

Examples include Alias Types [Smith et al., 2000] and Separation Logic [Reynolds, 2002].

Keeping precise track of mutable data structures:

- allows their type (and properties) to evolve over time;
- enables safe memory de-allocation;
- helps prove properties of programs.

Unfortunately, in these systems, any *code* that *reads* or writes a piece of mutable state must *publish* that fact in its interface.

A programming idiom: hidden, persistent state

It is common software engineering practice to design "objects" (or "modules", "components", "functions") that:

- rely on a piece of *mutable internal state*,
- which *persists across invocations*,
- yet publish an (informal) specification that *does not* reveal the very *existence* of such a state.

Example: the memory manager

For instance [O'Hearn et al., 2004], a *memory manager* might maintain a linked list of freed memory blocks.

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

It is sound for them to believe that the memory manager's methods have *no side effect*, other than the obvious effect of providing them with, or depriving them from, ownership of a unique memory block.

A distinct idiom: abstraction

Hiding must not be confused with *abstraction*, a different idiom, whereby:

- one acknowledges the existence of a mutable state,
- whose type (and properties) are accessible to clients only under an abstract name.

Abstraction has received recent attention: see, e.g., Parkinson and Bierman [2005, 2008] or Nanevski et al. [2007].

The memory manager, with abstract state

If the memory manager publishes an abstract invariant I, then every direct or indirect client must declare that it requires and preserves I.

Furthermore, all clients must cooperate and exchange the token I between them.

Exposing the existence of the memory manager's internal state leads to a loss of *modularity*.

Contents

A *region-* and *capability-*based type system
[Charguéraud and Pottier, 2008] forms my starting point.

To this system, I will add a single typing rule, which enables *hiding*.

A *singleton region* σ is a static name for a value.

The *singleton type* $[\sigma]$ is the type of the value that inhabits σ.

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, $\{\sigma : \mathsf{ref}\,\mathsf{int}\}$ asserts that the value σ is the address of an integer reference cell, and asserts ownership of this cell.

Typing rules for references

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

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

Contents

The first-order frame axiom

The first-order *frame axiom* states that, if a term behaves correctly in a certain store, then it also behaves correctly in a larger store.

It can take the form of a subtyping axiom:

$$\underset{\text{(actual type of Term)}}{\chi_1 \to \chi_2} \quad \leq \quad \underset{\text{(type assumed by Context)}}{(\chi_1 * C) \to (\chi_2 * C)}$$

This makes a capability *unknown to the term*, while it is *known to its context*. We need the opposite!

The higher-order frame axiom

Building on work by O'Hearn et al. [2004], Birkedal et al. [2006] define a *higher-order frame axiom:*

$$\underset{\text{(actual type of Term)}}{\chi} \quad \leq \quad \underset{\text{(type assumed by Context)}}{\chi \otimes C}$$

The operator $\cdot \otimes C$ makes C a pre- and post-condition of *every* arrow:

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

It commutes with products, sums, refs, and vanishes at base types.

The higher-order frame axiom: examples

A first-order example:

$$\begin{array}{rcl} \mathsf{int} \longrightarrow \mathsf{int} & \leq & (\mathsf{int} \longrightarrow \mathsf{int}) \otimes C \\ & = & \mathsf{int} * C \longrightarrow \mathsf{int} * C \end{array}$$

A second-order example:

$$\begin{array}{rcl} & & ((\mathsf{int} \longrightarrow \mathsf{int}) \times \mathsf{list\ int}) \longrightarrow \mathsf{list\ int} \\ \leq & & (((\mathsf{int} \longrightarrow \mathsf{int}) \times \mathsf{list\ int}) \longrightarrow \mathsf{list\ int}) \otimes C \\ = & & ((\mathsf{int} * C \longrightarrow \mathsf{int} * C) \times \mathsf{list\ int} * C) \longrightarrow \mathsf{list\ int} * C \end{array}$$

If applied to an effectful function, "map" becomes effectful as well.

Think of *corruption* [Lebresne, 2008].

What does the higher-order frame axiom have to do with hiding?

The higher-order frame axiom allows deriving the following law:

$$\underbrace{\neg\neg((\chi \otimes C) * C)}_{\text{(actual type of Term)}} \quad \leq \quad \underbrace{\neg\neg\chi}_{\text{(type assumed by Context)}}$$

where $\neg\neg\chi$ is $(\chi \to 0) \to 0$.

The derivation is as follows:

$$
\begin{array}{lll}
 & (((\chi \otimes C) * C) \longrightarrow O) \longrightarrow O & \\
= & (((\chi \otimes C) * C) \longrightarrow (O \otimes C)) \longrightarrow O & \text{def. of } \otimes \\
\leq & (((\chi \otimes C) * C) \longrightarrow (O \otimes C) * C) \longrightarrow O & \text{drop a capability} \\
= & ((\chi \longrightarrow O) \otimes C) \longrightarrow O & \text{def. of } \otimes \\
\leq & (\chi \longrightarrow O) \longrightarrow O & \text{higher-order frame}
\end{array}
$$

The higher-order frame axiom is applied not to the effectful code, but to its *continuation*, which unwittingly becomes effectful as well.

This enables a *limited form of hiding*, with *closed scope*.

A naïve higher-order anti-frame rule

To enable *open-scope hiding*, it seems natural to drop the double negation:

$$\underset{\text{(actual type of Term)}}{(\chi \otimes C) * C} \quad \leq \quad \underset{\text{(type assumed by Context)}}{\chi} \qquad \text{(unsound)}$$

The intuitive idea is,

- Term must *guarantee* C when abandoning control to Context;
- (thus, C holds whenever Context has control;)
- Term may *assume* C when receiving control from Context.

A sound higher-order anti-frame rule

The previous rule does not account for interactions between Term and Context via functions found in the environment or in the store.

A sound rule is:

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

Type soundness is proved via subject reduction and progress.

It can also be proved via a semantic model [Schwinghammer et al., 2009]. This roughly amounts to explaining the anti-frame rule in terms of recursive types and polymorphism.

Contents

Tracked versus untracked references

In this type system, references are *tracked:* access requires a capability. This is heavy, but permits de-allocation and type-varying updates.

In ML, references are *untracked:* no capabilities are required. This is lightweight, but a reference must remain allocated, and its type must remain fixed, forever.

It seems pragmatically desirable for a programming language to offer both flavors.

An encoding of untracked integer references

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

let mkuref : int ⟶ uref =
λ(v : int).
  let σ, (r : [σ]) = ref v in                     — got { σ: ref int }
  hide R = { σ: 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
```

An encoding of untracked generic references

```
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 : (unit * R) ⟶ ((a ⊗ R) * R) =           — that is, R
    λ(). get r                                      — also { ρ: ref (a ⊗ R) }
  and uset : ((a ⊗ R) * R) ⟶ (unit * R) =
    λ(v : a ⊗ R). set (r, v)
  in (uget, uset)                                   — type: (uref a) ⊗ R
                                                    — to the outside, uref a
```

Purely functional languages exploit *thunks*, which are built once and can be forced any number of times.

In ML, a thunk can be implemented as a reference to an internal state with three possible colors (unevaluated, being evaluated, evaluated).

The anti-frame rule allows explaining why this reference can be hidden, and why (as a consequence) it is sound for thunks to be untracked.

Thunks, simplified — part 1

```
def type thunk a =
  unit ⟶ a

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

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

Thunks, simplified — 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
```

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*.

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)
```

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
```

Contents

Analogy with concurrent separation logic

In concurrent separation logic [O'Hearn, 2007, Hobor et al., 2008], the invariant associated with a lock is made available when the lock is acquired, and must be re-established when the lock is released. Multiple threads can concurrently attempt to acquire a lock.

In short, *locks provide hidden state.*

The anti-frame rule can be viewed as a special case, which does not require any runtime check, but is sound only in a sequential setting.

The higher-order frame axiom and the anti-frame rule can be *generalized* to support not just a fixed invariant, but a family of invariants [Pottier, 2009a].

This allows implicit reasoning about the fact that calls and returns are well-matched.

The type-and-capability system that I have used as a starting point can be extended:

- *either* with the higher-order frame axiom,
- *or* the anti-frame rule,

but *not both*—their combination is unsound [Pottier, 2009b].

Paranoia

The anti-frame rule is *paranoid.*

The invariant must hold whenever a call to the outside world is made. This *forbids the use of a pre-existing library* to manipulate the hidden state, and leads to a lack of modularity [Pottier, 2009b].

Whether and how this can be fixed is an open issue.

(There is a work-around that involves a dynamic check—a lock.)

From a strong reference to a weak one, and back

We have used anti-frame to turn a strong reference into a weak one, forever—there was no deallocation.

However, *one can provide a deallocation method* that returns a strong reference, provided all three methods (read, write, free) can *fail.*

The hidden invariant includes a Boolean flag:

$$\{\sigma_1 : \mathsf{ref}\,(\mathsf{unit} + (\mathsf{unit} * \{\sigma_2 : \mathsf{ref}\,\mathsf{int}\}))\}$$

Similarly, one could equip thunks with both *force* and *cancel* methods, with a dynamic check to ensure that one most one of these methods is invoked.

Using the same idiom, one could allow a linear object to masquerade as a non-linear one, with a dynamic check to ensure that the object is used at most once.

This is *dynamic linearity enforcement.*

One could include a method that drops the mask and returns the original linear object, provided it has not been used yet.

On type-based complexity analysis

Tarjan's approach to amortized complexity analysis using *time credits* can be made formal by viewing a credit as a (linear) capability.

One modifies the rule that type-checks function applications so that *every function call consumes one credit.*

The result is a system that can check amortized asymptotic complexity claims.

Here is one way to play with *hidden credits*.

Imagine a pot that hides an integer reference and a number of credits, with an invariant that ties them together:

$$\exists k.\big(\{\sigma : \mathsf{ref}\ (\mathsf{int}\ k)\} * k\$\big)$$

The pot offers a method that requires one credit and increments the counter, and a method that fails if the counter is zero and otherwise decrements the counter and produces one credit.

This allows *dynamic complexity checking*. It is just a plain old counter, but the fact that the counter is used in a correct way is statically checked.

Okasaki [1999] uses *debits* to analyze the amortized complexity of algorithms that rely on lazy evaluation (thunks).

Danielsson [2008] formalizes this idea in a type system.

Pilkiewicz and I [2009] explain debits in terms of credits. The encoding uses anti-frame to explain that *a thunk hides not only a reference, but also a number of time credits.*

A separate notion of *monotonicity* is used to express the fact that the number of credits that remain to be paid before the thunk can be forced decreases with time.

Contents

In summary, a couple of key ideas are:

- a practical rule for hiding state must have *open scope;*
- it is safe for a piece of state to be hidden, as long as *its invariant holds at every interaction* between Term and Context;
- the rule seems to have interesting applications.

Contents

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)*. 213–224.

Danielsson, N. A. 2008.
Lightweight semiformal time complexity analysis for purely functional data structures.
In *ACM Symposium on Principles of Programming Languages (POPL)*.

Bibliography]Bibliography


Hobor, A., Appel, A. W., and Zappa Nardelli, F. 2008.
Oracle semantics for concurrent separation logic.
In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 4960. Springer, 353–367.


Lebresne, S. 2008.
A system F with call-by-name exceptions.
In *International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 5126. Springer, 323–335.

[][


Nanevski, A., Ahmed, A., Morrisett, G., and Birkedal, L. 2007.
Abstract predicates and mutable ADTs in Hoare type theory.
In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 4421. Springer, 189–204.


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.


O'Hearn, P. W. 2007.
Resources, concurrency and local reasoning.
Theoretical Computer Science 375, 1–3 (May), 271–307.

[][


Okasaki, C. 1999.
Purely Functional Data Structures.
Cambridge University Press.


Parkinson, M. and Bierman, G. 2005.
Separation logic and abstraction.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 247–258.


Parkinson, M. and Bierman, G. 2008.
Separation logic, abstraction and inheritance.
In *ACM Symposium on Principles of Programming Languages (POPL)*. 75–86.

[][


Pilkiewicz, A. and Pottier, F. 2009.
The essence of monotonic state.
Submitted.


Pottier, F. 2009a.
Generalizing the higher-order frame and anti-frame rules.
Unpublished.


Pottier, F. 2009b.
Three comments on the anti-frame rule.
Unpublished.


Reynolds, J. C. 2002.
Separation logic: A logic for shared mutable data structures.
In *IEEE Symposium on Logic in Computer Science (LICS)*. 55–74.

[][


Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., and Reus, B. 2009.
A semantic foundation for hidden state.
Submitted.


Smith, F., Walker, D., and Morrisett, G. 2000.
Alias types.
In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 1782. Springer, 366–381.