

# Hiding local state in direct style

François Pottier

September 18th, 2008



- *Why hide state?*
- *Setting the scene: a capability-based type system*
- *Towards hidden state: a bestiary of frame rules*
- *Applications: untracked references and thunks*
- *Conclusion*
- *Bibliography*

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.

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*.

- *Why hide state?*
- *Setting the scene: a capability-based type system*
- *Towards hidden state: a bestiary of frame rules*
- *Applications: untracked references and thunks*
- *Conclusion*
- *Bibliography*

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*  $\sigma$  is a static name for a value.

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

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

First, it carries a *memory type*  $\theta$ , which describes the structure and extent of the memory area to which the value  $\sigma$  gives access.

Second, it represents *ownership* of this area.

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

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

$$\text{ref} : \tau \rightarrow \exists \sigma. ([\sigma] * \{\sigma : \text{ref } \tau\})$$

$$\text{get} : [\sigma] * \{\sigma : \text{ref } \tau\} \rightarrow [\sigma] * \{\sigma : \text{ref } \tau\}$$

$$\text{set} : ([\sigma] \times \tau_2) * \{\sigma : \text{ref } \tau_1\} \rightarrow \text{unit} * \{\sigma : \text{ref } \tau_2\}$$

- *Why hide state?*
- *Setting the scene: a capability-based type system*
- *Towards hidden state: a bestiary of frame rules*
- *Applications: untracked references and thunks*
- *Conclusion*
- *Bibliography*

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.

It can take the form of a subtyping axiom:

$$\begin{array}{ccc} \chi_1 \rightarrow \chi_2 & \leq & (\chi_1 * C) \rightarrow (\chi_2 * C) \\ \text{(actual type of Term)} & & \text{(type assumed by Context)} \end{array}$$

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

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

$$\begin{array}{ccc} \chi & \leq & \chi \otimes C \\ \text{(actual type of Term)} & & \text{(type assumed by Context)} \end{array}$$

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

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

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

# The higher-order frame rule: examples

A first-order example:

$$\begin{aligned} \text{int} \rightarrow \text{int} &\leq (\text{int} \rightarrow \text{int}) \otimes C \\ &= \text{int} * C \rightarrow \text{int} * C \end{aligned}$$

A second-order example:

$$\begin{aligned} &((\text{int} \rightarrow \text{int}) \times \text{list int}) \rightarrow \text{list int} \\ &\leq (((\text{int} \rightarrow \text{int}) \times \text{list int}) \rightarrow \text{list int}) \otimes C \\ &= ((\text{int} * C \rightarrow \text{int} * C) \times \text{list int} * C) \rightarrow \text{list int} * C \end{aligned}$$

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

Think of *corruption* [Lebresne, 2008].

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

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

$$\begin{array}{ccc} \neg\neg((\chi \otimes C) * C) & \leq & \neg\neg\chi \\ \text{(actual type of Term)} & & \text{(type assumed by Context)} \end{array}$$

where  $\neg\neg\chi$  is  $(\chi \rightarrow O) \rightarrow O$ .

The derivation is as follows:

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

The higher-order frame rule 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:

$$\begin{array}{ccc} (\chi \otimes C) * C & \leq & \chi & \text{(unsound)} \\ \text{(actual type of Term)} & & \text{(type assumed by Context)} & \end{array}$$

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:

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

Type soundness is proved via subject reduction and progress.

- *Why hide state?*
- *Setting the scene: a capability-based type system*
- *Towards hidden state: a bestiary of frame rules*
- *Applications: untracked references and thunks*
- *Conclusion*
- *Bibliography*

# 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 =

(unit  $\rightarrow$  int)  $\times$  (int  $\rightarrow$  unit)

– a non-linear type!

**let** mkuref : int  $\rightarrow$  uref =

$\lambda(v : \text{int}).$

**let**  $\sigma, (r : [\sigma]) = \text{ref } v$  **in**

**hide**  $R = \{ \sigma : \text{ref int} \}$  **outside of**

**let** uget : (unit  $*$   $R$ )  $\rightarrow$  (int  $*$   $R$ ) =

$\lambda(). \text{get } r$

**and** uset : (int  $*$   $R$ )  $\rightarrow$  (unit  $*$   $R$ ) =

$\lambda(v : \text{int}). \text{set } (r, v)$

**in** (uget, uset)

– got {  $\sigma : \text{ref int}$  }

– this pair has type uref  $\otimes$   $R$

– to the outside, uref

# An encoding of untracked generic references

**def type** uref  $a =$

$(\text{unit} \rightarrow a) \times (a \rightarrow \text{unit})$

– parameterize over  $a$

**let** mkuref :  $\forall a.a \rightarrow \text{uref } a =$

$\lambda(v : a).$

**let**  $\rho, (r : [\rho]) = \text{ref } v$  **in**

**hide**  $R = \{ \rho : \text{ref } a \} \otimes R$  **outside of**

**let** uget :  $(\text{unit} * R) \rightarrow ((a \otimes R) * R) =$

$\lambda(). \text{get } r$

**and** uset :  $((a \otimes R) * R) \rightarrow (\text{unit} * R) =$

$\lambda(v : a \otimes R). \text{set } (r, v)$

**in** (uget, uset)

– got  $\{ \rho : \text{ref } a \}$

– got  $\{ \rho : \text{ref } a \} \otimes R$

– that is,  $R$

– also  $\{ \rho : \text{ref } (a \otimes R) \}$

– type:  $(\text{uref } a) \otimes R$

– 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.

**def type** thunk  $a =$   
 $\text{unit} \rightarrow a$

**def type** state  $a =$  – internal state:  
 $W \text{ unit} + G \text{ unit} + B a$  – white/grey/black

**let** mkthunk :  $\forall a. (\text{unit} \rightarrow a) \rightarrow \text{thunk } a =$   
 $\lambda(f : \text{unit} \rightarrow a).$

**let**  $\rho, (r : [\rho]) = \text{ref } (W ())$  – got {  $\rho$ : ref (state  $a$ ) }

**hide**  $R = \{ \rho : \text{ref } (\text{state } a) \} \otimes R$  **outside of**

·  
·  
·

– got  $R$

–  $f : (\text{unit} \rightarrow a) \otimes R$

–  $f : (\text{unit} * R) \rightarrow ((a \otimes R) * R)$

```

let force : (unit * R) → ((a ⊗ R) * R) =
  λ().
    case get r of
      | W () →
        set (r, G ());
        let v : (a ⊗ R) = f() in
          set (r, B v);
          v
      | G () → fail
      | B (v : a ⊗ R) → v
in force

```

- state  $a = W \text{ unit} + G \text{ unit} + B a$
- got  $R = \{ \rho: \text{ref}(\text{state } a) \} \otimes R$
- got  $R$
- got  $R$
- got  $R$
- force: (think  $a$ )  $\otimes R$
- to the outside, think  $a$

## 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*.

**def type** thunk  $a =$   
 unit  $\rightarrow a$

**def type** state  $\gamma a =$   
 $W (\text{unit} * \gamma) + G \text{unit} + B a$      *– when white,  $\gamma$  is available*

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

**let** force : (unit \* R) → ((a ⊗ R) \* R) =

λ().

**case** get r **of**

| W () →

set (r, G ());

**let** v : (a ⊗ R) = f() **in**

set (r, B v);

v

| G () → **fail**

| B (v : a ⊗ R) → v

**in** force

– state γ a = W (unit \* γ) + G unit + B a

– got R = { ρ: ref (state γ a) } ⊗ R

– got { ρ: ref (W unit + G ⊥ + B ⊥) } \* (γ ⊗

– got R \* (γ ⊗ R)

– got R; (γ ⊗ R) was consumed by f

– got R

– without γ ⊗ R, invoking f is forbidden

– force: (thunk a) ⊗ R

– to the outside, thunk a

- Why hide state?
- Setting the scene: a capability-based type system
- Towards hidden state: a bestiary of frame rules
- Applications: untracked references and thunks
- Conclusion
- Bibliography

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.

# One direction for future work: monotonicity

The anti-frame rule crucially relies on *invariance*: the hidden state must remain fixed.



It would be worth investigating *monotonicity*: what if the hidden state *grows* with time, in some suitable sense?

This could help explain:




- Okasaki and Danielsson's amortized complexity analysis of thunks;
- why hash-consing admits a pure specification.

- Why hide state?
- Setting the scene: a capability-based type system
- Towards hidden state: a bestiary of frame rules
- Applications: untracked references and thunks
- Conclusion
- Bibliography

(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)*.
-  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 Verlag, 323–335.

## Bibliography]Bibliography

-  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. Springer Verlag.
-  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.
-  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.



Reynolds, J. C. 2002.

*Separation logic: A logic for shared mutable data structures.*

In *IEEE Symposium on Logic in Computer Science (LICS)*. 55–74.



Smith, F., Walker, D., and Morrisett, G. 2000.

*Alias types.*

In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 1782. Springer Verlag, 366–381.