Hiding local state in direct style

François Pottier

September 18th, 2008

INRIA

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].

Strengths

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.

A weakness

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.

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

The host type system

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, $\{\sigma : \text{ref int}\}$ asserts that the value σ 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.

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

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.

It can take the form of a subtyping axiom:

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

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:*

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

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

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

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

The higher-order frame rule: examples

A first-order example:

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

A *second-order* example:

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

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

Think of *corruption* [Lebresne, 2008].

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

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

$$\neg\neg((\chi \otimes C) * C) \qquad \leq \qquad \neg\neg\chi$$

(actual type of Term) (type assumed by Context)

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

The derivation is as follows:

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

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:

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

(actual type of Term) (type assumed by Context)

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:

Anti-frame

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

Type soundness is proved via subject reduction and progress.

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.

```
def type thunk α =
  unit → α

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

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

```
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*.

```
def type thunk α =
  unit → α

def type state γ α =
  W (unit ∗ γ) + G unit + B α          – when white, γ is available

let mkthunk : ∀γα.(((unit ∗ γ) → α) ∗ γ) → thunk α =
  λ(f : (unit ∗ γ) → 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: ((unit ∗ γ) → α) ⊗ R
      .                                – f: (unit ∗ R ∗ (γ ⊗ R)) → ((α ⊗ R) ∗ R)
```

```
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
```

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.

(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.