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

François Pottier
INRIA

Abstract

Separation logic involves two dual forms of modularity: local reasoning makes part of the store invisible within *a static scope, whereas hiding local state makes part of the store invisible* outside *a static scope. In the recent literature, both idioms are explained in terms of a higher-order frame rule. I point out that this approach to hiding local state imposes continuation-passing style, which is impractical. Instead, I introduce a higher-order* anti-frame rule, *which permits hiding local state in direct style. I formalize this rule in the setting of a type system, equipped with linear capabilities, for an ML-like programming language, and prove type soundness via a syntactic argument. Several applications illustrate the expressive power of the new rule.*

1 Introduction

Setting the stage There are several approaches to reasoning about imperative programs with dynamic memory allocation. The literature offers type systems equipped with regions, singleton types, and linear capabilities [13, 5, 3]; separation logic, a Hoare logic extended with formulae that assert ownership of a fragment of the heap [12, 10, 9, 11, 2]; and Hoare Type Theory [8, 7], which marries an expressive dependent type system with aspects of separation logic.

The gap between the type systems and the program logics cited above is rather narrow. Regions and singleton types are just a way of assigning static names to objects, or to collections of objects, while avoiding some of the difficulties associated with dependency in the presence of effects. Capabilities are just lightweight separation logic assertions [10], which describe the shape of the store, but do not allow expressing arbitrary properties of values.

The ideas presented in this paper can be developed indifferently within a "type system" or within a "logic". I choose a recent type system [3], with which I am familiar. Although less expressive than a logic, it is nevertheless an appropriate setting in which to reason about hidden state.

On hidden state One often designs a piece of software so that its implementation is imperative and relies on an internal state, but its specification does not betray this fact. By this, I do not mean that the state appears under an abstract type in the specification, so that clients do not have access to its concrete representation. I mean that the very existence of an internal state is not revealed in the specification, so that clients have no knowledge whatsoever of it. A typical example is that of a memory manager [9, 2, 7]: no knowledge of the manager's internal free list should be necessary when reasoning about a client.

One might think that it is seldom possible to hide an internal state, on the basis that, in many situations, its existence is betrayed by the code's observable behavior. However, it is important to understand that it is not the code's actual behavior that matters, but only its specification. Specifications are partial descriptions of the behavior: the more partial they are, the more opportunities for hidden state arise. Consider, for instance, a prime number generator. In order to express the fact that each invocation returns the *next* prime number, a full specification must reveal the existence of an internal state. However, a partial specification, which merely states that each invocation produces *some* prime number, need not mention the generator's state, which can thus be hidden. In general, as specifications become less precise, opportunities for hidden state increase. In particular, when specifications are built out of types and capabilities, as in this paper, these opportunities are quite numerous.

Motivation Why is it important not to reveal the existence of an internal state? For one thing, revealing it pollutes client code with an invariant (that is, a capability). Making this invariant abstract [11, 7] helps, but does not eliminate the problem. Indeed, the inconvenience becomes particularly acute when an unbounded number of objects are allocated at runtime, each with its own internal state and invariant. Then, clients have to keep track of an unbounded collection of invariants, which can be difficult, and perhaps impossible.

Here is a technical description of this problem. A function that owns a piece of internal state, described by an abstract capability γ, has type:

$$\exists\gamma.(((\chi_1 * \gamma) \to (\chi_2 * \gamma)) * \gamma)$$

where γ does not occur free in χ_1 or χ_2. This is an existential package, containing a pair of a function that requires and returns an abstract capability γ, and of γ itself. Because capabilities are linear, the above type, which contains a capability as a component, must be linear too. In short, when internal state is abstract, but not hidden, *objects with internal state must be linear*. This means that one must carefully keep track of ownership and aliasing information about these objects. On the other hand, when internal state is hidden via the machinery presented in this paper, then the above function has type:

$$\chi_1 \to \chi_2$$

This type is simpler, but that is not the main point. The point is, this type is non-linear. In the presence of an appropriate treatment of hidden state, *objects with internal state are ordinary values*.

I have just argued that hiding an invariant is preferable to abstracting it. However, as I have pointed out earlier, hiding an invariant is possible only under a sufficiently weak specification, whereas abstracting it is always possible. As a result, both techniques are useful, and should co-exist.

The higher-order frame rule The frame rule of separation logic [12] states that, if a term t 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. In the present paper, as in [3], this is stated like this:

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

The capability C is not known within t, even though it is available to the enclosing context. Up to $\beta\eta$, this rule is equivalent to a simple subtyping axiom:

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

Does this rule allow hiding a piece of local state? No. This rule hides a capability within a static scope, while making it visible outside this scope. To hide a piece of local state is the exact dual: to make a capability available within a static scope, while hiding it outside this scope.

In order to address this need, O'Hearn *et al.* [9] introduce a generalized version of the frame rule, known as the hypothetical frame rule. Soon afterwards, Birkedal *et al.* [2] present an even more general version, known as the higher-order frame rule. This rule again takes the form of a simple subtyping axiom:

$$\textsc{Higher-order frame}\quad \chi \quad \leq \quad \chi \otimes C$$

Here, χ is an arbitrary type, possibly a higher-order function type. The type $\chi \otimes C$ (read: "χ in the presence of C", or "χ under C" for short) describes exactly the same behavior as χ, but additionally requires the capability C to be available at every interaction between the term and its context. Perhaps more concretely, $\chi \otimes C$ can be thought of as a modified version of χ, where an extra C is added on the left-hand and right-hand sides of every arrow. This is expressed by the following axiom (together with a family of structural axioms, see Figure 2):

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

The higher-order frame rule can also be presented in a form that operates on judgements:

$$\textsc{Higher-order frame}\quad \frac{\Gamma \Vdash t : \chi}{(\Gamma \otimes C), C \Vdash t : (\chi \otimes C) * C}$$

Hidden state via the higher-order frame rule It has been argued [9, 2] that the higher-order frame rule permits hidden state. How does this work? Let us assume that we have built a term (the *provider*) of type $((\chi_1 * C) \to (\chi_2 * C)) * C$. For simplicity, assume that χ_1 and χ_2 are base types, so that $\chi_1 \otimes C$ is χ_1 and $\chi_2 \otimes C$ is χ_2. We now wish to hide C and pretend that *provider* has type $\chi_1 \to \chi_2$. The higher-order frame rule does not directly allow this. Instead, the idea is to explicitly parameterize the rest of the program (the *client*) with respect to *provider*. That is, we implement *client* as a term of type $(\chi_1 \to \chi_2) \to \chi$, where χ is some answer type. (Again, assume χ is a base type.) We now apply the higher-order frame rule to *client*, and find that it also has type $((\chi_1 \to \chi_2) \to \chi) \otimes C$, that is,

$$(((\chi_1 * C) \to (\chi_2 * C)) * C) \to (\chi * C)$$

so the application (*client provider*), which represents the entire program, is well-typed and has type $\chi * C$. By subtyping, it also has type χ. As a last step, if we are implementing *provider* as a separate module, without advance knowledge of its clients, then we must abstract *provider* over its future clients. That is, we package it as λ*client*.(*client provider*). This term has type $((\chi_1 \to \chi_2) \to \chi) \to \chi$, the double negation of the type that we wished for.

To a certain extent, the goal is met: the client has no knowledge of C, and is type-checked under the assumption that the provider has type $\chi_1 \to \chi_2$. However, this approach imposes continuation-passing style: when an object with hidden internal state is built, it cannot be returned, but must instead be passed on to a continuation. This is impractical.

A higher-order anti-frame rule Instead of applying a frame rule to an explicit continuation, it seems more natural to work in direct style and to apply an *anti-frame* rule, that is, a dual of the frame rule. An anti-frame rule should formalize the following idea: if a term owns a piece of state, initially described by a capability C, and if every interaction between the term and its context requires and preserves C, then it is sound for the context to have no knowledge of this internal state. In short, C is a hidden invariant.

A simple, but unsound, approximation of such a rule is:

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

On the left-hand side, the conjunct C represents the requirement that the invariant initially hold, while the conjunct $\chi \otimes C$ is supposed to encode the requirement that the invariant hold whenever control is transferred from term to context or from context to term. On the right-hand side, C disappears: the invariant becomes hidden, and supposedly can never be violated.

The reason why the above subtyping axiom is unsound is that the interaction between a term and its context is described not just by its type χ, but also by the type environment under which the term is type-checked. As a result, this environment, too, must receive different inside and outside views. A sound, higher-order anti-frame rule is:

$$\frac{\Gamma \otimes C \Vdash t : (\chi \otimes C) * C}{\Gamma \Vdash t : \chi}\ \text{ANTI-FRAME}$$

The outside type environment, Γ, becomes $\Gamma \otimes C$ on the inside. This means that, if the term t wishes to invoke a function supplied by the context, then the invariant C must hold upon invocation, and will hold upon return.

There are various ways for a function to be "supplied by the context". It can be passed as an argument, if, say, the environment contains a binding $x : \chi_1 \rightarrow \chi_2$. It can be stored in a reference cell, if, say, the environment contains a capability $\{\sigma : \text{ref } (\chi_1 \rightarrow \chi_2)\}$. Both cases are accounted for by the above rule: indeed, the effect of $\Gamma \otimes C$ is to turn *all* of the arrows within Γ into C-preserving arrows.

It is important to note that, in the presence of higher-order store, the law $C_1 \otimes C_2 = C_1$ is invalid. This equality holds only if no arrow (or type variable) occurs in C_1. As a consequence, the law $(\cdot \otimes C_1) \otimes C_2 = \cdot \otimes (C_1 * C_2)$, found in Birkedal *et al.*'s work [2], is also invalid. Instead, I have:

$$(\cdot \otimes C_1) \otimes C_2 = \cdot \otimes (C_1 \circ C_2)$$

where the composition $C_1 \circ C_2$ stands for $(C_1 \otimes C_2) * C_2$.

Applications The anti-frame rule has many applications. The classic example of a memory manager, which maintains a private free list [9, 2, 7], is expressible, provided objects have uniform size. This works even if an unbounded number of memory manager objects can be dynamically allocated, each with its own free list [11].

In this paper, I sketch three applications of the anti-frame rule, which have practical and theoretical interest.

$$\begin{array}{lll}
\textit{Values} & v ::= x \mid () \mid \text{inj}^i\ v \mid (v_1, v_2) \mid (\lambda x.t) \mid p \mid l \\
\textit{PrimOps.} & p ::= \text{case} \mid \text{proj}^i \mid \text{ref} \mid \text{get} \mid \text{set} \\
\textit{Terms} & t ::= v \mid (v\ t)
\end{array}$$

$$\begin{array}{lcl}
((\lambda x.t)\ v)\ /\ s & \longrightarrow & ([x \rightarrow v]\, t)\ /\ s \\
(\text{proj}^i\ (v_1, v_2))\ /\ s & \longrightarrow & v_i\ /\ s \\
(\text{case}\ ((\text{inj}^i\ v), v_1, v_2))\ /\ s & \longrightarrow & (v_i\ v)\ /\ s \\
(\text{ref}\ v)\ /\ s & \longrightarrow & l\ /\ s \uplus [l \mapsto v] \\
(\text{get}\ l)\ /\ s & \longrightarrow & s[l]\ /\ s \\
(\text{set}\ (l, v))\ /\ s & \longrightarrow & ()\ /\ s[l \mapsto v] \\
(v\ t)\ /\ s & \longrightarrow & (v\ t')\ /\ s' \\
 & & \text{if } t\ /\ s \longrightarrow t'\ /\ s'
\end{array}$$

Figure 1. Untyped syntax and semantics

The first application is an encoding of untracked references, that is, references that can be read and written without exhibiting any capability. The type system does not have primitive untracked references. This encoding shows that it can simulate them. This makes it an extension of the ML type system [15], a property that would not hold in the absence of the anti-frame rule. This application involves first-order functions, and is comparable in nature with the memory manager example.

The second application is an encoding of untracked lazy evaluation thunks. The third one is a generic fixed point combinator. These two applications involve higher-order interaction (at orders 2 and 3, respectively), and illustrate how the type system statically deals with several potential re-entrancy issues.

Summary The main contribution of this paper is a higher-order anti-frame rule, together with a syntactic proof of type soundness (sketched), and three non-trivial applications.

The paper is laid out as follows. The programming language and its type system are presented (§2). A syntactic type soundness argument is sketched (§3). Then, several applications are described (§4). The paper ends with discussions of related and future work (§5, §6).

2 The type system

2.1 Basic machinery

I use a calculus equipped with sums, products, first-class functions, and first-class references. It is identical to that studied by Charguéraud and Pottier [3]. Its untyped syntax and semantics appear in Figure 1. There are syntactic categories for values (v), primitive operations (p), and terms (t). The small-step operational semantics ($\longrightarrow$) reduces configurations, that is, pairs of a term and a memory store (m).

The type system is closely based upon Charguéraud and Pottier's earlier system [3]. Present there, but omitted here,

$$
\begin{array}{lrcl}
\text{Capabilities} & C & ::= & C \otimes C \mid \emptyset \mid \{\sigma : \theta\} \mid C_1 * C_2 \mid \exists \sigma . C \\
\text{Value types} & \tau & ::= & \tau \otimes C \mid \bot \mid \text{unit} \mid \tau + \tau \mid \tau \times \tau \mid \chi \rightarrow \chi \mid [\sigma] \\
\text{Memory types} & \theta & ::= & \theta \otimes C \mid \bot \mid \text{unit} \mid \theta + \theta \mid \theta \times \theta \mid \chi \rightarrow \chi \mid [\sigma] \mid \text{ref } \theta \mid \theta * C \mid \exists \sigma . \theta \\
\text{Computation types} & \chi & ::= & \chi \otimes C \mid \tau \mid \chi * C \mid \exists \sigma . \chi \\
\text{Duplicable environments} & \Delta & ::= & \Delta \otimes C \mid \varnothing \mid \Delta, x : \tau \\
\text{Linear environments} & \Gamma & ::= & \Gamma \otimes C \mid \varnothing \mid \Gamma, x : \chi \mid \Gamma, C
\end{array}
$$

Capabilities

$$
\begin{array}{rcl}
\emptyset \otimes C & = & \emptyset \\
\{\sigma : \theta\} \otimes C & = & \{\sigma : \theta \otimes C\}
\end{array}
$$

Value and memory types

$$
\begin{array}{rcl}
\bot \otimes C & = & \bot \\
\text{unit} \otimes C & = & \text{unit} \\
(\theta_1 + \theta_2) \otimes C & = & (\theta_1 \otimes C) + (\theta_2 \otimes C) \\
(\theta_1 \times \theta_2) \otimes C & = & (\theta_1 \otimes C) \times (\theta_2 \otimes C) \\
(\chi_1 \rightarrow \chi_2) \otimes C & = & ((\chi_1 \otimes C) * C) \rightarrow ((\chi_2 \otimes C) * C) \\
[\sigma] \otimes C & = & [\sigma] \\
(\text{ref } \theta) \otimes C & = & \text{ref } (\theta \otimes C)
\end{array}
$$

(Duplicable and linear) environments

$$
\begin{array}{rcl}
\varnothing \otimes C & = & \varnothing \\
(\Gamma, x : \chi) \otimes C & = & (\Gamma \otimes C), x : (\chi \otimes C) \\
(\Gamma, C_1) \otimes C_2 & = & (\Gamma \otimes C_2), C_1 \otimes C_2
\end{array}
$$

Shared among several syntactic categories

$$
\begin{array}{rcl}
(\,\cdot\, * \,\cdot\,) \otimes C & = & (\,\cdot\, \otimes C) * (\,\cdot\, \otimes C) \\
(\exists \sigma . \,\cdot\,) \otimes C & = & \exists \sigma . (\,\cdot\, \otimes C) \\
(\,\cdot\, \otimes C_1) \otimes C_2 & = & \cdot\, \otimes ((C_1 \otimes C_2) * C_2) \\
 & = & \cdot\, \otimes (C_1 \circ C_2)
\end{array}
$$

Figure 2. Syntax and structural equivalence of capabilities and types

$$
\overset{\text{VAR}}{\frac{(x : \tau) \in \Delta}{\Delta \vdash x : \tau}}
\qquad
\overset{\text{UNIT}}{\frac{}{\Delta \vdash () : \text{unit}}}
\qquad
\overset{\text{INJ}}{\frac{\Delta \vdash v : \tau_i}{\Delta \vdash (\text{inj}^i\ v) : (\tau_1 + \tau_2)}}
\qquad
\overset{\text{PRIM}}{\frac{p : \tau}{\Delta \vdash p : \tau}}
\qquad
\overset{\text{PAIR}}{\frac{\Delta \vdash v_1 : \tau_1 \qquad \Delta \vdash v_2 : \tau_2}{\Delta \vdash (v_1, v_2) : (\tau_1 \times \tau_2)}}
$$

$$
\overset{\text{FUN}}{\frac{\Delta, x : \chi_1 \Vdash t : \chi_2}{\Delta \vdash (\lambda x . t) : \chi_1 \rightarrow \chi_2}}
\qquad
\overset{\text{VAL}}{\frac{\Delta \vdash v : \tau}{\Delta \Vdash v : \tau}}
\qquad
\overset{\text{APP}}{\frac{\Delta \vdash v : \chi_1 \rightarrow \chi_2 \quad \Delta, \Gamma \Vdash t : \chi_1}{\Delta, \Gamma \Vdash (v\ t) : \chi_2}}
\qquad
\overset{\text{SUB}}{\frac{\Gamma \Vdash t : \chi_1 \quad \chi_1 \leq \chi_2}{\Gamma \Vdash t : \chi_2}}
\qquad
\overset{*\text{-INTRO (FRAME)}}{\frac{\Gamma \Vdash t : \chi}{\Gamma, C \Vdash t : \chi * C}}
\qquad
\overset{*\text{-ELIM-1}}{\frac{\Gamma, (x : \chi_1), C \Vdash t : \chi_2}{\Gamma, x : (\chi_1 * C) \Vdash t : \chi_2}}
$$

$$
\overset{*\text{-ELIM-2}}{\frac{\Gamma, C_1, C_2 \Vdash t : \chi}{\Gamma, (C_1 * C_2) \Vdash t : \chi}}
\qquad
\overset{\exists\text{-ELIM-1}}{\frac{\Gamma, x : \chi_1 \Vdash t : \chi_2}{\Gamma, x : \exists \sigma . \chi_1 \Vdash t : \chi_2}}
\qquad
\overset{\exists\text{-ELIM-2}}{\frac{\Gamma, C \Vdash t : \chi}{\Gamma, \exists \sigma . C \Vdash t : \chi}}
\qquad
\overset{\textbf{ANTI-FRAME}}{\frac{\Gamma \otimes C \Vdash t : (\chi \otimes C) * C}{\Gamma \Vdash t : \chi}}
$$

Figure 3. Type-checking values and terms

References

$$
\begin{array}{lcl}
\text{ref} & : & \tau \rightarrow \exists \sigma . [\sigma] * \{\sigma : \text{ref } \tau\} \\
\text{get} & : & [\sigma] * \{\sigma : \text{ref } \tau\} \rightarrow \tau * \{\sigma : \text{ref } \tau\} \\
\text{set} & : & ([\sigma] \times \tau_2) * \{\sigma : \text{ref } \tau_1\} \rightarrow \text{unit } * \{\sigma : \text{ref } \tau_2\} \\
\text{FOCUS-REF} & : & \{\sigma_1 : \text{ref } \theta\} \equiv \exists \sigma_2 . \{\sigma_1 : \text{ref } [\sigma_2]\} * \{\sigma_2 : \theta\}
\end{array}
$$

Pairs

$$
\begin{array}{lcl}
\text{proj}^1 & : & [\sigma] * \{\sigma : \tau_1 \times \theta_2\} \rightarrow \tau_1 * \{\sigma : \tau_1 \times \theta_2\} \\
\text{FOCUS-PAIR}^1 & : & \{\sigma : \theta_1 \times \theta_2\} \equiv \exists \sigma_1 . \{\sigma : [\sigma_1] \times \theta_2\} * \{\sigma_1 : \theta_1\}
\end{array}
$$

Sums

$$
\begin{array}{lcl}
\text{case} & : & \big(((\exists \sigma_1 . ([\sigma_1] * \{\sigma : [\sigma_1] + \bot\} * \{\sigma_1 : \theta_1\} * C)) \rightarrow \chi) \\
 & & \times ((\exists \sigma_2 . ([\sigma_2] * \{\sigma : \bot + [\sigma_2]\} * \{\sigma_2 : \theta_2\} * C)) \rightarrow \chi) \\
 & & \times [\sigma] \big) * \{\sigma : \theta_1 + \theta_2\} * C \rightarrow \chi \\
\text{FOCUS-SUM}^1 & : & \{\sigma : \theta_1 + \bot\} \equiv \exists \sigma_1 . \{\sigma : [\sigma_1] + \bot\} * \{\sigma_1 : \theta_1\}
\end{array}
$$

Miscellaneous

$$
\begin{array}{rcl}
C & \leq & \emptyset \\
\bot & \leq & \cdot \\
\tau & \leq & \exists \sigma . [\sigma] * \{\sigma : \tau\} \\
[\sigma] * \{\sigma : \tau\} & \leq & \tau * \{\sigma : \tau\} \\
\{\sigma_1 : \exists \sigma_2 . \theta\} & \equiv & \exists \sigma_2 . \{\sigma_1 : \theta\} \\
\{\sigma_1 : \theta * C\} & \equiv & \{\sigma_1 : \theta\} * C \\
C_1 * C_2 & \equiv & C_2 * C_1 \\
\cdot\, * (C_1 * C_2) & \equiv & (\,\cdot\, * C_1) * C_2 \\
\cdot\, * \emptyset & \equiv & \cdot \\
\exists \sigma_1 . \exists \sigma_2 . \,\cdot & \equiv & \exists \sigma_2 . \exists \sigma_1 . \,\cdot \\
\cdot\, * (\exists \sigma . C) & \equiv & \exists \sigma . (\,\cdot\, * C) \\
[\sigma \rightarrow \sigma']\ \cdot & \leq & \exists \sigma . \,\cdot
\end{array}
$$

Figure 4. Primitive operations and subtyping

are group regions, adoption, and focus. These mechanisms are orthogonal to the issue of hiding state.

A *singleton region* σ is a static name for a value. The type $[\sigma]$ is the type of the single value that inhabits region σ. It is a singleton type. A *singleton capability* $\{\sigma : \theta\}$ is a static token that serves two roles. First, it represents ownership of the value denoted by σ. Second, it carries the type of this value, namely θ. For instance, a singleton capability of the form $\{\sigma : \mathrm{ref}\ \theta\}$ asserts that the value denoted by σ is the address of a reference cell, and, simultaneously, represents an exclusive right to read and write this cell. Compound capabilities are built via conjunction and existential quantification over regions (Figure 2). These constructs correspond to singleton heaps, separating conjunction, and existential quantification over values in separation logic [12].

The system involves non-linear *value types* (τ), linear *computation types* (χ), and linear *memory types* (θ). *Duplicable type environments* (Δ) bind variables to value types, while *linear type environments* (Γ) bind variables to computation types, and also contain capabilities. Typing judgements about values and terms respectively take the form:

$$\Delta \vdash v : \tau \qquad \text{and} \qquad \Gamma \Vdash t : \chi$$

The environment Γ describes the shape of the store before the term t is run, while the type χ describes its shape after t is run. They can be thought of as a precondition and a postcondition: judgements about terms in this type system are analogous to Hoare triples in separation logic. Memory types appear within capabilities: in a capability $\{\sigma : \theta\}$, the memory type θ describes the extent of the piece of memory that this capability controls.

The typing judgements for values and terms are defined in Figure 3. The types of the primitive operations and the subtyping rules appear in Figure 4. The typing rules are identical to those found in [3], with the addition of the anti-frame rule (ANTI-FRAME).

2.2 New machinery

I introduce the type constructor $\cdot \otimes C$, borrowed from Birkedal *et al.* [2], which intuitively "adds a copy of C to the left-hand and right-hand sides of every arrow within its left-hand argument." I assume that capabilities and types are equated modulo the equational theory in Figure 2.

I introduce recursive capabilities and recursive types, for two reasons. First, they are needed to describe situations where functions access the store and the store contains functions. This is noted by Thielecke [14], and illustrated by the examples in §4. Second, perhaps more surprisingly, recursive capabilities seem necessary for the proof of Revelation (Lemma 3.1) to go through.

In order to keep things lightweight, I do not introduce any notation (such as μ binders) for recursive capabilities and types. I simply do not regard the definition of capabilities and types in Figure 2 as inductive. That is, I forbid structural induction over capabilities or types. In order to construct recursive capabilities or types, I assume that systems of contractive recursive equations have unique solutions. I assume that the operator $\otimes$ is contractive in its right-hand side. More work is admittedly needed in order to clarify why these assumptions are consistent.

3 Type soundness

I now give some elements of the syntactic type soundness proof. I omit the actual statements of subject reduction and progress: they would be similar to those found in [3].

Let the meta-variable R stand for a capability. The Revelation lemma states that each of the judgements that appear in the definition of the type system is preserved when R is revealed. In other words, the higher-order frame rule is admissible. It is interesting to find that this property plays a key role in explaining ANTI-FRAME.

Lemma 3.1 (Revelation) *The following hold:*

$$\begin{array}{lll} C_1 \le C_2 & \textit{implies} & C_1 \otimes R \le C_2 \otimes R \\ \chi_1 \le \chi_2 & \textit{implies} & \chi_1 \otimes R \le \chi_2 \otimes R \\ \Gamma \vdash v : \tau & \textit{implies} & \Gamma \otimes R \vdash v : \tau \otimes R \\ \Gamma \Vdash t : \chi & \textit{implies} & (\Gamma \otimes R), R \Vdash t : (\chi \otimes R) * R \end{array} \quad \diamond$$

Proof. By (mutual) structural induction. I present two of the more interesting cases in the proof of the last assertion above.

∘ *Case* APP. We begin with:

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

Apply an induction hypothesis to each premise, and build a new instance of APP:

$$\frac{\begin{array}{c}\Delta \otimes R \vdash v : (\chi_1 \to \chi_2) \otimes R \\ (\Delta \otimes R), (\Gamma \otimes R), R \Vdash t : (\chi_1 \otimes R) * R\end{array}}{(\Delta \otimes R), (\Gamma \otimes R), R \Vdash (v\ t) : (\chi_2 \otimes R) * R}$$

This is a valid instance of APP, thanks to the equality:

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

∘ *Case* ANTI-FRAME. We begin with:

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

Let R' and C' be the (unique) two capabilities that satisfy the following (contractive) recursive equations:

$$\begin{array}{lll} C' & = & C \otimes R' \\ R' & = & R \otimes C' \end{array}$$

These capabilities are chosen so that:

$$\begin{array}{rcll} C \circ R' & = & (C \otimes R') * R' & \text{(by def. of } \circ) \\ & = & C' * R' & \text{(by def. of } C') \\ & = & C' * (R \otimes C') & \text{(by def. of } R') \\ & = & R \circ C' & \text{(by def. of } \circ) \end{array}$$

In other words, we have a commutation property:

$$(\cdot \otimes C) \otimes R' = (\cdot \otimes R) \otimes C'$$

Now, apply the induction hypothesis to ANTI-FRAME's premise, revealing R':

$$(\Gamma \otimes C) \otimes R', R' \Vdash t : ((\chi \otimes C) \otimes R') * (C \otimes R') * R'$$

By the commutation property, this can also be written:

$$(\Gamma \otimes R) \otimes C', R' \Vdash t : ((\chi \otimes R) \otimes C') * (C \otimes R') * R'$$

By definition of C' and R', this is:

$$((\Gamma \otimes R), R) \otimes C' \Vdash t : (((\chi \otimes R) * R) \otimes C') * C'$$

This is in the format of a premise of ANTI-FRAME. Build a new instance of ANTI-FRAME, hiding C', to conclude:

$$(\Gamma \otimes R), R \Vdash t : (\chi \otimes R) * R$$

This is the goal. We have shown that the sequence of hiding C and revealing R is equivalent to the sequence of revealing R' and hiding C'. When two originally independent hidden invariants R and C meet, they become R' and C', whose definitions are, in general, mutually recursive. □

Here is a sketch of how the revelation lemma serves to justify ANTI-FRAME. In the subject reduction proof, instances of ANTI-FRAME become problematic when they reach an evaluation position, that is, when considering a type derivation for a closed term $E[t]$, where E is an evaluation context ($E ::= [\cdot] \mid v\ E$), and the sub-derivation for the term t has an instance of ANTI-FRAME at its root. Let R stand for the capability hidden by this instance of ANTI-FRAME. Then, the term t "knows about" R, while the context E doesn't. Applying Lemma 3.1 to the type derivation for $E[\cdot]$ yields a new type derivation for $E[\cdot]$, where, this time, E "knows about" R. This in turn yields a new type derivation for $E[t]$, where ANTI-FRAME is used no longer at the root of t, but at the root of the entire derivation.

In short, ANTI-FRAME extrudes up through evaluation contexts. This is sufficient to get it "out of the way" in the subject reduction proof. Intuitively, this means that hidden local invariants are revealed, and become global invariants, at runtime.

In order to allow ANTI-FRAME to float up, revelation is applied to the *context*. By contrast, if revelation was used to justify a higher-order frame rule, it would be applied to the *term*. This idea is evidently present in Birkedal *et al.*'s work [2]. The merit of the anti-frame rule is to allow applying revelation to the context without being forced to reify the context as a term, that is, without having to switch to continuation-passing style.

4 Applications

In what follows, the syntax "hide $R = C$ outside of t", where R is a capability variable and C is a capability that can have free occurrences of R, has the double effect of (i) binding R to the unique solution of the recursive equation $R = C$, and (ii) applying the anti-frame rule, so that R becomes hidden outside of the term t.

According to the anti-frame rule, it is necessary for R to be available only at exit of the "hide" construct, that is, at the *end* of the execution of t. In the first two applications that follow, R is in fact available already upon entry into the "hide" construct. In the third application, this is not the case: the full flexibility of the rule is exploited.

4.1 Encoding untracked references

The type system that I have used as a setting [3], as well as previous capability-based type systems [13, 5], share the following two features. First, references are *tracked*: a reference cannot be read or written unless an appropriate capability is presented. Second, a function closure cannot capture a capability. These facts imply that any function that reads or writes a piece of state requires a capability as an argument, and returns another capability. This is heavy, but offers an advantage: these capabilities specify precisely how an invocation of the function alters the state.

By comparison, in the ML type system [15], references are untracked. Because there is no way of deallocating a reference, every pointer remains valid forever. Because there is no strong update operation, the type of the contents can be carried in the type of the pointer. In short, "ref τ" is the type of a pointer to a cell that holds a value of type τ. No capability is required to read or write such a cell. This is lightweight, but comes at a price: when a reference is untracked, it is impossible to reason about the evolution of its state over time.

In practice, it seems desirable to be able to mix tracked and untracked references within a single program. Fortunately, thanks to the anti-frame rule, it is possible to define untracked references in terms of the primitive, tracked references. Of course, in a practical system, one would provide both flavors of references under a primitive form. The encoding presented here shows that untracked references are sound, and, in theory, redundant.

The encoding is shown in Figure 5. In the pseudo-code, I use polymorphism, which is not formalized in this paper

```
**def type** uref $\alpha$ =                                   – an untracked reference is represented as a pair of a getter and a setter
  (unit $\rightarrow \alpha$) $\times$ ($\alpha \rightarrow$ unit)                  – note that this is a value type, hence unrestricted (non-linear)

**let** mkuref : $\forall\alpha.\alpha \rightarrow$ uref $\alpha$ =                     – here is the function that creates a fresh untracked reference
$\lambda$(v : $\alpha$).
  **let** $\sigma$, (r : [$\sigma$]) = ref v **in**                         – allocate a tracked reference r within a fresh region $\sigma$; we hold { $\sigma$: ref $\alpha$ }
  **hide** $R$ = { $\sigma$: ref $\alpha$ } $\otimes R$ **outside of**              – $R$ is the internal invariant; we hold { $\sigma$: ref $\alpha$ } $\otimes R$, that is, $R$
  ·                                                      – r now has type [$\sigma$] $\otimes R$, that is, still just [$\sigma$]
  ·                                                      – note: $R$ is also { $\sigma$: ref ($\alpha \otimes R$) }
  **let** uget : (unit $* R$) $\rightarrow$ (($\alpha \otimes R$) $* R$) =                 – define a getter function
    $\lambda$(). get r                                         – within uget, we hold $R$, so r can be read
  **and** uset : (($\alpha \otimes R$) $* R$) $\rightarrow$ (unit $* R$) =                 – define a setter function
    $\lambda$(v : $\alpha \otimes R$). set (r, v)                             – within uset, we hold $R$, so r can be written
  **in** (uget, uset)                                        – this pair has type (uref $\alpha$) $\otimes R$, so, outside anti-frame, it has type (uref $\alpha$)
```

Figure 5. Encoding untracked references

```
**def type** thunk $\alpha$ =                                            – a thunk is represented as a force function
  unit $\rightarrow \alpha$                                                  – note that this is a value type, hence unrestricted (non-linear)

**def type** state $\gamma$ $\alpha$ =                                          – the internal state of a thunk: white, grey, or black
  W (unit $* \gamma$) + G unit + B $\alpha$                                   – when white, the capability $\gamma$ is stored in the thunk

**let** mkthunk : $\forall\gamma\alpha$.(((unit $* \gamma$) $\rightarrow \alpha$) $* \gamma$) $\rightarrow$ thunk $\alpha$ =     – here is the function that creates a fresh thunk
  $\lambda$(f : (unit $* \gamma$) $\rightarrow \alpha$).                                   – within mkthunk, we hold $\gamma$
    **let** $\sigma$, (r : [$\sigma$]) = ref (W ()) **in**                          – allocate a tracked reference r within a fresh region $\sigma$
    ·                                                            – we hold $\gamma *$ { $\sigma$: ref (W unit + G $\perp$ + B $\perp$) }
    ·                                                            – by subtyping, this implies { $\sigma$: ref (W (unit $* \gamma$) + G unit + B $\alpha$) }
    ·                                                            – that is, { $\sigma$: ref (state $\gamma$ $\alpha$) }
    **hide** $R$ = { $\sigma$: ref (state $\gamma$ $\alpha$) } $\otimes R$ **outside of**          – we now hold { $\sigma$: ref (state $\gamma$ $\alpha$) } $\otimes R$, that is, $R$
    ·                                                            – note: $R$ is also { $\sigma$: ref (state ($\gamma \otimes R$) ($\alpha \otimes R$)) }
    ·                                                            – f now has type ((unit $* \gamma$) $\rightarrow \alpha$) $\otimes R$
    ·                                                            – that is, (unit $*$ ($\gamma \otimes R$) $* R$) $\rightarrow$ (($\alpha \otimes R$) $* R$)
    **let** force : (unit $* R$) $\rightarrow$ (($\alpha \otimes R$) $* R$) =                 – define a function that forces the thunk
    $\lambda$().
      **case** get r **of**                                             – we hold $R$, so r can be read
      | W () $\rightarrow$                                              – we hold { $\sigma$: ref (W unit + G $\perp$ + B $\perp$) } $*$ ($\gamma \otimes R$)
        set (r, G ());                                         – color the thunk grey
        ·                                                      – we hold { $\sigma$: ref (W $\perp$ + G unit + B $\perp$) } $*$ ($\gamma \otimes R$)
        ·                                                      – by subtyping, this implies $R *$ ($\gamma \otimes R$),
        **let** v : ($\alpha \otimes R$) = f() **in**                            – so it is permitted to invoke f
        ·                                                      – we still hold $R$, but have consumed $\gamma \otimes R$
        set (r, B v);                                          – store the result and color the thunk black
        ·                                                      – we hold { $\sigma$: ref (W $\perp$ + G $\perp$ + B ($\alpha \otimes R$)) }
        ·                                                      – by subtyping, this implies $R$
        v                                                      – return the result, which has type $\alpha \otimes R$
      | G () $\rightarrow$                                              – this thunk is being evaluated
        **fail**                                                   – invoking f would be illegal, since we do not hold $\gamma \otimes R$
      | B (v : $\alpha \otimes R$) $\rightarrow$                                    – this thunk has been evaluated before
        v                                                      – return the previously stored result, without affecting $R$
    **in** force                                                   – force has type (thunk $\alpha$) $\otimes R$
                                                                 – so, outside anti-frame, it has type (thunk $\alpha$)
```

Figure 6. Encoding lazy evaluation thunks

```
let fix : $\forall\alpha_1\alpha_2.((\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)) \to \alpha_1 \to \alpha_2$ =
λ(f : $(\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)$).
  let $\sigma$, (r : [$\sigma$]) = ref () in                – we hold { $\sigma$: ref unit }
  hide $R$ = { $\sigma$: ref $(\alpha_1 \to \alpha_2)$ } $\otimes R$ outside of    – note that we do not hold $R$ yet, because r has been initialized to unit
  .                                            – f now has type $((\alpha_1 \to \alpha_2) \to (\alpha_1 \to \alpha_2)) \otimes R$
  let g : $(\alpha_1 \to \alpha_2) \otimes R$ =             – g requires $R$, and invokes the function held in the reference r
    λ(x : $\alpha_1 \otimes R$). get r x                – within g, we hold $R$, so r can be read
  in let h : $(\alpha_1 \to \alpha_2) \otimes R$ =          – h requires $R$, and invokes f, routing recursive calls to g
    λ(x : $\alpha_1 \otimes R$). f g x
  in set (r, h);                               – a strong update establishes { $\sigma$: ref $((\alpha_1 \to \alpha_2) \otimes R)$ }, that is, $R$
  h                                            – we hold $R$ at this point, as required by anti-frame
                                               – h has type $(\alpha_1 \to \alpha_2) \otimes R$, so, outside anti-frame, it has type $\alpha_1 \to \alpha_2$
```

Figure 7. A fixed point combinator

(see [3]). The variable α ranges over value types. An untracked reference is represented as a pair of functions that allow reading and writing a hidden tracked reference. The fact that these functions own and access a piece of state is not apparent in their types: indeed, "uref α" is a non-linear value type. As a result, an untracked reference can be aliased and used without restriction, just like in ML.

4.2 Encoding lazy thunks

Pure functional programming languages rely on lazy evaluation, whose efficient implementation involves the in-place update of a suspension, or *thunk*. There is apparently a paradox: the reason why functional programs are easy to reason about is that they are free of side effects, yet their execution does involve a side effect. Why is this sound?

The anti-frame rule provides an answer to this question: thanks to it, thunks can be defined as a library. Again, thunks are untracked: the type of a thunk is a non-linear value type, so thunks can be freely aliased, and can be forced any number of times without presenting any capability. The internal state of a thunk is hidden to its clients.

The encoding is shown in Figure 5. A thunk is created by "mkthunk", which expects a pair of a client function "f", of type $(\text{unit} * \gamma) \to \alpha$, and a capability γ. (The variable γ ranges over capabilities; this application exploits type and capability polymorphism.) A thunk is represented as a "force" function, of type unit $\to \alpha$, which holds a hidden reference to the thunk's internal state.

The capability γ is required, but not returned, by "f": it is consumed. Note that γ is supplied just once, when the thunk is built; it is not supplied again when the thunk is forced. Thus, the type system statically guarantees that, even though the thunk is non-linear and may be forced several times, the client function "f" is invoked at most once.

A thunk can be in one of three states. White (W) means that the thunk is not yet evaluated: the capability γ is still available, stored within the reference cell. Grey (G) means that the thunk is being evaluated: the capability γ is gone, but the result of the computation is not yet available. Black (B) means that the thunk has been evaluated: its result is stored within the reference cell.

It is interesting to examine how the type system deals with the re-entrancy issue. Internally, the function "force" requires the capability R and returns it, which means that it requires the thunk to be in a consistent state, and preserves this fact. To the client, however, "force" appears to have type unit $\to \alpha$, which does not mention R: the client does not know about the hidden reference. As a result, it is important to ensure that, whenever the client has control, the invariant R holds. This means, in particular, that one must not invoke the client function "f" at a time when R does not hold: in the unlucky situation where invoking "f" leads to forcing the current thunk again (a "black hole"), that would be unsound. The type system enforces this by changing the type of "f", which externally is $(\text{unit} * \gamma) \to \alpha$, to $((\text{unit} * \gamma) \to \alpha) \otimes R$, which means that "f" requires R (and preserves it).

This design rules out a few subtle potential errors. For instance, omitting the instruction "set (r, G ())", which colors the thunk grey, makes the code ill-typed. Indeed, invoking "f" requires both R and $\gamma \otimes R$, but, as long as the thunk is colored white, it is impossible to hold both of these capabilities simultaneously. Furthermore, it is impossible to color the thunk black before invoking "f", because the value "v" is not yet available. As a result, a programmer who naively thought that the colors black and white would suffice is forced by the type system to introduce a third color, namely grey, and to perform a dynamic check against black holes.

4.3 A fixed point combinator

Figure 7 presents a generic fixed point combinator, based on Landin's classic trick of "tying a knot in the store". The internal invariant is:

$$R \quad = \quad \{\sigma : \text{ref}\ (\alpha_1 \to \alpha_2)\} \otimes R$$

This invariant states that the reference “r” contains a function that requires, and preserves, the invariant. This is a recursive statement, which is why R must be a recursive capability.

The invariant R does not yet hold at the point where the anti-frame rule is applied. It is established only later by the instruction “set (r, h)”, which changes the type of the contents “r” from unit to $(\alpha_1 \to \alpha_2) \otimes R$. This code cannot be type-checked in ML. In ML, instead, one would initialize “r” with a dummy function that fails if invoked. Here, the benefit of avoiding such a dummy initialization is marginal: we do get a static guarantee that the code cannot fail, but no guarantee that it cannot loop. I present this variant of the code because it illustrates a more interesting use of the anti-frame rule.

The definitions of the auxiliary functions “g” and “h” exhibit η-redexes. Contracting either of these redexes would be unsound—because then “r” would be read, or “f” would be invoked, before the invariant R is established—and would make the code ill-typed.

5 Related work

“Information hiding” was studied by O’Hearn *et al.* [9] and by Birkedal *et al.* [2] in the setting of separation logic. I continue this study in the setting of a capability-based type system for an ML-like programming language, and propose “a form of alias types with information hiding” [9]. I reuse Birkedal *et al.*’s operator $\otimes$, but argue that their higher-order frame rule does not offer a satisfactory mechanism for hiding state, because it imposes continuation-passing style.

O’Hearn *et al.* [9] do not clearly distinguish hiding and abstraction, whereas I argue that these are distinct mechanisms. The former hides a capability, so that clients are not aware of the existence of an internal state, and are not submitted to a linearity restriction. The latter reveals an abstract capability, so that clients are aware of the existence of an internal state, whose concrete representation is not disclosed. In that case, clients can be made aware, via abstract predicates [11], that the internal state “represents”, or “implements”, some abstract data structure, and evolves over time. O’Hearn *et al.*’s memory manager [9, Table 2] is a typical example of hiding, where the existence of a free list is unknown to clients. On the other hand, their queue module [9, Table 3], where “an abstract variable is exposed”, is a typical example of abstraction. It can be explained purely in terms of abstract types, capabilities, and predicates, without recourse to a hiding mechanism such as the anti-frame rule.

Nanevski *et al.* [7] study the memory manager example using abstraction, but no hiding. As a result, an abstract capability, I, is revealed, and client code is polluted. Similarly, Parkinson and Bierman [11] study memory managers as first-class objects (“connection pools”) in terms of abstraction.

To reiterate my stance, both hiding and abstraction are useful. Hiding is preferable in situations where no information whatsoever about the internal state is exposed, while abstraction must be used in situations where some information, such as the fact that the internal state “represents” some abstract data structure, is exposed.

O’Hearn *et al.* [9] and Birkedal *et al.* [2] point out that the generalized frame rule is unsound when used together with “imprecise predicates” and Hoare’s conjunction rule. I view Hoare’s conjunction rule as analogous to the distributivity rule for intersection types, which is unsound in the presence of effects [4], so it seems sensible to simply not introduce such a rule.

Concurrent separation logic [10] allows declaring a (statically scoped) lock, which guards a capability. Within the scope of the lock, the capability is hidden, but is made visible again within the critical regions that the lock controls. The proof rule for critical regions is analogous to the anti-frame rule insofar as the hidden capability appears in the premise, but not in the conclusion, of the rule. It is worth noting that these two approaches have different ways of achieving soundness in the face of re-entrancy. O’Hearn’s proof rule for critical regions is sound only because the semantics of critical regions dictates that a thread cannot acquire a single lock twice, or it will deadlock—a runtime failure. The anti-frame rule, on the other hand, does not rely on a dynamic check: by systematically applying the $\otimes$ operator to all components of the typing judgement, it statically prevents re-entrancy. Of course, this is no panacea. For one thing, in order to satisfy the anti-frame rule, the programmer is sometimes forced to explicitly introduce a dynamic check, as illustrated, for instance, in the encoding of lazy thunks (§4.2). For another thing, the anti-frame rule is sound only in a sequential setting.

Berdine and O’Hearn [1] present a system of bunched types in which it is possible not only to hide a piece of state, but also to deallocate it. As an example, they present an integer counter object, whose state is hidden and shared by three functions: “lookup”, “increment”, and “destroy”. The type system guarantees that none of these functions can be called after “destroy” has been invoked. Berdine and O’Hearn impose continuation-passing style, and write: “it is nontrivial to define a direct-style language that supports disposal”. As if to confirm this claim, the anti-frame rule, which is in direct style, does not support disposal. This is illustrated, for instance, in the encoding of untracked references (§4.1). The types assigned to “get” and “set” do not allow controlling when these functions are invoked, so it would be unsound, and it is impossible, to define a "destroy" function.

Lebresne [6] extends System F with exceptions and with *corruption,* a version of the higher-order frame rule. It is unclear whether an anti-frame rule would make sense in his setting. The rule presented here relies on the persistence of state, whereas exceptions do not seem to be persistent in any sense.

6 Future work

It would be desirable to formalize a system where the higher-order frame rule and the higher-order anti-frame rule co-exist. I have not yet worked out the details of such a combination.

I have argued that the anti-frame rule is a direct-style analogue of the higher-order frame rule [2]. It would be interesting to formalize this claim. One could wish to define a continuation-passing transform that maps every instance of the anti-frame rule to an instance of the higher-order frame rule, applied to the current continuation.

I have provided a purely syntactic argument for the soundness of the anti-frame rule. It would be desirable to offer a deeper explanation of it, possibly via an interpretation in a denotational model, as commonly done in separation logic [1, 2]. I understand Birkedal *et al.*'s model of the higher-order frame rule as an encoding that introduces a universal quantifier at every arrow. Perhaps, analogously, anti-frame can be understood via an encoding that exploits both universal and existential quantification.

Acknowledgements Thanks to Arthur Charguéraud for developing the subject reduction proof [3] which this work extends, and for proof-reading this paper. Thanks to Didier Rémy for suggesting that it should be possible to encode untracked references. Thanks to Paul-André Melliès and Nicolas Tabareau for suggesting the $\circ$ notation.

References

[1] J. Berdine and P. W. O'Hearn. Strong update, disposal, and encapsulation in bunched typing. In *Mathematical Foundations of Programming Semantics*, volume 158 of *Electronic Notes in Theoretical Computer Science*, pages 81–98. Elsevier Science, May 2006.

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

[3] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. Submitted, Apr. 2008.

[4] R. Davies and F. Pfenning. Intersection types and computational effects. In *ACM International Conference on Functional Programming (ICFP)*, pages 198–208, Sept. 2000.

[5] M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, June 2002.

[6] S. Lebresne. A system F with exceptions. Manuscript, Feb. 2008.

[7] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer Verlag, Mar. 2007.

[8] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ACM International Conference on Functional Programming (ICFP)*, pages 62–73, Sept. 2006.

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

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

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

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

[13] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer Verlag, Mar. 2000.

[14] H. Thielecke. Frame rules from answer types for code pointers. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 309–319, Jan. 2006.

[15] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.