JFP **23** (1): 38–144, 2013. © Cambridge University Press 2012
doi:10.1017/S0956796812000366 First published online 10 October 2012

Syntactic soundness proof of a type-and-capability system with hidden state

FRANÇOIS POTTIER
INRIA, BP 105, 78153 Le Chesnay Cedex, France
(*e-mail:* Francois.Pottier@inria.fr)

Abstract

This paper presents a formal definition and machine-checked soundness proof for a very expressive type-and-capability system, that is, a low-level type system that keeps precise track of ownership and side effects. The programming language has first-class functions and references. The type system's features include the following: universal, existential, and recursive types; subtyping; a distinction between affine and unrestricted data; support for strong updates; support for naming values and heap fragments via singleton and group regions; a distinction between ordinary values (which exist at runtime) and capabilities (which do not); support for dynamic reorganizations of the ownership hierarchy by disassembling and reassembling capabilities; and support for temporarily or permanently hiding a capability via frame and anti-frame rules. One contribution of the paper is the definition of the type-and-capability system itself. We present the system as modularly as possible. In particular, at the core of the system, the treatment of affinity, in the style of dual intuitionistic linear logic, is formulated in terms of an arbitrary *monotonic separation algebra*, a novel axiomatization of resources, ownership, and the manner in which they evolve with time. Only the peripheral layers of the system are aware that we are dealing with a specific monotonic separation algebra, whose resources are references and regions. This semi-abstract organization should facilitate further extensions of the system with new forms of resources. The other main contribution is a machine-checked proof of type soundness. The proof is carried out in the Wright and Felleisen's syntactic style. This offers an evidence that this relatively simple-minded proof technique can scale up to systems of this complexity, and constitutes a viable alternative to more sophisticated semantic proof techniques. We do not claim that the syntactic technique is superior: We simply illustrate how it is used and highlight its strengths and shortcomings.

1 Introduction

This paper presents a formal definition and machine-checked soundness proof for a core type-and-capability system, that is, a low-level type system that keeps precise track of side effects.

1.1 Why control side effects?

Many prominent academic and industrial programming languages, such as the members of the ML family, on the one hand, and Java and C#, on the other hand, have the following key features in common:

1. Thanks to automated garbage collection, dynamic memory management is implicit and efficient.
2. Side effects are permitted: They include the use of modifiable data structures, which is sometimes essential for efficiency and modularity; system calls, which are required in order to interact with the real world; and concurrency, which can help deal with input/output and exploit the processing power of multi-core and multi-processor machines.
3. A sound, polymorphic type system rules out many programming mistakes, and at the same time fosters the construction of independent components that can be easily and safely assembled.

One may argue that these features are among the main reasons why these programming languages are so widely adopted and studied. Yet, although the type systems of ML, Java, and C# are effective at describing the structure of memory and at imposing abstraction barriers between components, they do not attempt to control the use of side effects in a precise manner.

As far as the treatment of mutable state is concerned, for instance, these type systems keep track of the memory locations that are mutable and immutable, and achieve soundness by requiring every mutable location to have a fixed type. However, they do not keep track of accesses to mutable memory: if f has type "function of A to B", they are able to tell that f requires an argument of type A and produces a result of type B, but are unable to tell which mutable memory locations (or, more abstractly, which mutable data structures) are potentially read and written by f. Similarly, if p has type "pointer to A", they are able to tell that writing an arbitrary value of type A at address p is safe, in the sense that the heap will not become corrupted; but they are unable to tell, at a more abstract level, which mutable data structures this memory location participates in. As a result, they cannot tell which data structure invariants might be broken by this write operation.

As far as the treatment of system calls and concurrency is concerned, analogous limitations exist. If d has type "file descriptor", for instance, these type systems are unable to tell whether d is presently in an open or closed state, so they cannot detect that d is never closed, or closed twice. If p has type "pointer to hash table", they are unable to tell whether this table is thread-local or shared, and, in the latter case, which lock protects it, so they cannot detect that two threads race to access the table. In this paper, we do not attempt to perform "typestate checking", which would be required in order to ensure that file descriptors are properly managed, and we do not support concurrency: we work strictly in a sequential setting. However, we note that setting up a type system that keeps track of "who" owns "which" objects seems to be a good approach to address these issues: the papers by Bierhoff and Aldrich (2007) and Fähndrich *et al.* (2006), for instance, are excellent illustrations of this claim.

1.2 A partial survey of prior work and ideas

The limitations described above are well known. It seems desirable to overcome these, although one must acknowledge that it is not quite known yet whether the benefits of accurately keeping track of side effects will outweigh the cost of working with significantly more complex type systems. A huge literature exists on this topic, which cannot be fully surveyed here. In the following, we review some of the most relevant works and explain how these influence the type system that we present in this paper.

Researchers in the object-oriented programming community study the breach of abstraction that arises out of aliasing and mutable state and dub it the representation exposure problem (Detlefs *et al.*, 1998). They propose a number of disciplines to forbid representation exposure. These include islands (Hogg, 1991), balloons (Almeida, 1997), ownership types (Clarke *et al.*, 1998; Boyapati *et al.*, 2002), and universes (Müller & Poetzsch-Heffter, 2001; Dietl & Peter, 2005). These disciplines are often simple but restrictive: for instance, none of them allows the owner of an object to change over time.

Type-and-effect systems keep track of the read and write effects of each expression, and annotate function types with effects. Early type-and-effect systems (Gifford *et al.*, 1992) establish a crude distinction between pure and effectful expressions. Later systems (Talpin & Jouvelot, 1994) partition memory into regions and allow effects to refer to regions, thus permitting a finer-grained analysis. Monads, as found in the programming language Haskell, can be considered type-and-effect systems in another guise. The IO monad (Peyton Jones & Wadler, 1993) establishes a distinction between pure and effectful expressions, while the ST monad (Launchbury & Jones, 1995) distinguishes multiple disjoint regions of memory.

Linearity plays an important role in the control of side effects. Walker (2005) surveys the rich literature that exists on this topic. Technically, a linear value is one that is used exactly once. More interestingly perhaps, a linear type system typically enforces the property that there exists exactly one pointer to a linear value: a linear type system controls aliasing. This means that the graph of all linear values forms a forest, or in other words, the linear values are organized in a hierarchy. One can informally speak of an "ownership hierarchy" and consider that a linear-type system controls ownership. Instead of linearity, it is often sufficient to impose affinity: an affine value is one that is used at most once.

As pointed out, for instance, by Ahmed *et al.* (2007), a deep reason why linearity plays an important role in controlling and reasoning about side effects is that linearity enables strong updates. A strong update consists in changing the type (or, at a more abstract level, the logical facts that are known to hold) of the contents of a mutable object. Strong updates are necessary for reflecting protocols on resources whose state changes over time: for instance, changing the type of a file descriptor d from "open" to "closed" is a strong update. More generally, strong updates are necessary for proving properties of imperative programs: for instance, the assignment axiom of separation logic (Reynolds, 2002), which takes the form $\{(e \hookrightarrow -)\}\ e := e'\ \{(e \hookrightarrow e')\}$, is a strong update. This axiom requires the exclusive ownership of the memory location e. In general, some form of linearity is required for strong updates to be sound. Indeed, changing the information associated with the file descriptor d or with the memory location e in the above examples makes sense only if this information is recorded in exactly one place. If one allowed it to be duplicated, disseminated, and eventually become recorded in uncontrollably many places, one would be unable to update this information in a consistent manner.

Thus, linearity is crucial. However, linearity need not take as restrictive a form as it does in the early linear type systems. The key requirement for strong updates to be possible is that there should be one place where the type of the contents of a mutable object is recorded. Does this imply that there should be one pointer to the object? Yes, if the type of the pointer to the object mentions the type of the contents of the object. No, if it does not. In Smith *et al.*'s (2000) alias types, pointer types do not mention the type of the object that they point to, and (as a result) pointers can be duplicated. For every mutable object, there exists a capability, where the current type of the object is recorded. Capabilities are linear, so there exists only one place where the type of a mutable object is recorded. Holding a pointer to a mutable object does not, by itself, allow the object to be read or written: one needs, in addition, the capability for this object. Indeed, if one is reading, the capability must be consulted to find out what type of data are read; and, if one is writing, the capability must be updated with the type of the new data that are written. Because the capability is required for reading and writing, it can be thought of as a token that must be presented for access to be granted, hence its name. Capabilities can be given first-class status: They can be passed into and returned out of functions, and can be nested within values or within other capabilities. Capabilities do not exist at runtime: they exist only at type-checking time.

In a type-and-capability system, pointers (or, more generally, values) need not be linear, so the heap need not form a forest. Capabilities are linear and (because a capability can be nested within another) form a forest, or an ownership hierarchy. Because capabilities have first-class status, ownership transfers are possible, and the ownership hierarchy may evolve over time.

Type-and-capability systems subsume traditional linear type systems. The notion of a linear value is either directly supported or encoded as a pair of a duplicable value and a linear capability for this value. The system presented in this paper supports both views and allows moving from one to the other.

Type-and-capability systems subsume type-and-effect systems and the state monad.[1] Indeed, a function whose type advertises an effect is encoded as a function that requires a capability as an argument and returns it. Furthermore, type-and-capability systems are able to assign types to functions that perform transfers of ownership. Two archetypical examples of such functions are memory allocation (which requires no capability, and produces one) and memory de-allocation (which requires a capability, and produces none). These functions perform transfers of ownership between the caller and the memory manager.

Capabilities originate in Crary *et al.*'s (1999) work, where a capability governs a whole region of memory, that is, a group of mutable objects. In the alias types system and its descendants (Smith *et al.*, 2000; Walker & Morrisett, 2000; Ahmed

[1] An encoding of the IO and ST monads requires heterogeneous regions, whereas we formalize only homogeneous regions in this paper. See Section 2 for a brief discussion of the difference and choice between these two forms.

et al., 2007), on the other hand, a capability governs a singleton region, that is, a single mutable object. DeLine & Fähndrich (2001) and Fähndrich & DeLine (2002) consider both group regions and singleton regions and propose mechanisms, known as adoption, focus and defocus, for moving from one to the other. The Sing# programming language (Fähndrich *et al.*, 2006) is perhaps the most advanced and realistic incarnation of these ideas.

Type-and-capability systems and separation logic (2002) are closely related. In their traditional forms, they seem to be incomparable, because the former concern rich typed λ-calculi, whereas the latter concerns a simple first-order imperative language, and because the former guarantees just memory safety, whereas the latter can prove the correctness of a program with respect to a logical specification. However, it is possible to extend type-and-capability systems by incorporating logical assertions within types (see, for instance, Pilkiewicz & Pottier, 2011) and to extend separation logic with support for higher-order functions and higher-order store (Birkedal *et al.*, 2006; Reus & Schwinghammer, 2006; Schwinghammer *et al.*, 2009), so there is ultimately no deep gap between these approaches. The Hoare Type Theory (Nanevski *et al.*, 2008) is particularly a rich marriage between the type theory and the separation logic.

One might think that type-and-capability systems are primarily concerned with linearity, while separation logic is concerned with separation. However, linearity and separation are two sides of the same coin. In a type-and-capability system, the fact that a capability for a region of memory is linear (non-duplicable) implies that a conjunction of two capabilities must involve distinct capabilities, that is, capabilities for separate regions. Conversely, in separation logic, the fact that conjunction is separating implies that the formula $(e \hookrightarrow e')$, which holds of a heap cell at address e, does not entail $(e \hookrightarrow e') * (e \hookrightarrow e')$. Thus, the formula $(e \hookrightarrow e')$ has linear (non-duplicable) behavior, and for this reason can be thought of as a claim of ownership of the memory cell at address e. Ishtiaq and O'Hearn (2001) show that separation logic can be considered as one interpretation of the sub-structural logic **BI**. They also note the connection between separation logic and alias types (Smith *et al.*, 2000).

Separation algebras, which have been discovered and studied in the setting of separation logic (Calcagno *et al.*, 2007; Dockins *et al.*, 2009), serve in the present paper, under the generalized form of *monotonic* separation algebras, as a basis for the definition of a type-and-capability system. This is another hint of the close relationship between type-and-capability systems and separation logic.

1.3 A type-and-capability system with hidden state

Charguéraud and Pottier (2008) present a type-and-capability system for a λ-calculus equipped with products, sums, and references. The system borrows most of its features from prior papers, with a number of minor technical improvements.[2]

[2] In particular, Charguéraud and Pottier (2008) introduce distinct types for region inhabitants and pointers, whereas earlier works conflate these notions, and they note that the value restriction is not required. We come back to these technical aspects in Section 2.

Charguéraud and Pottier (2008) point out that the type-and-capability system can be used to direct a functional translation: Every well-typed imperative program can be translated to an equivalent purely functional program. The translation turns a capability for a region into a finite map that represents the contents of this region, and turns a value that inhabits this region into a key that can be used to access the map. The translation accounts for advanced operations, such as DeLine & Fähndrich's (2001) and Fähndrich & DeLine's (2002) adoption, focus and defocus, in a way that produces reasonable purely functional code, and one might claim, intuitively "explains" these operations. The soundness of the type-and-capability system and the existence of the functional translation are intimately linked: the two are proved simultaneously.

In subsequent work (Pottier, 2008), we extend this type-and-capability system with support for *hidden state*. What is this?

As explained by O'Hearn *et al.* (2004), the primitive memory allocation and de-allocation operations perform transfers of ownership between the caller and an invisible memory manager. The de-allocation operation, for instance, requires a capability as an argument, and does not return it. This capability seems to disappear because it becomes part of the state of the memory manager, which is hidden. It would be nice, O'Hearn *et al.* argue, if there was nothing magic about the memory manager, so that programmers could implement their own memory-manager-like objects, offering allocation and de-allocation methods.

For this purpose, the system must give programmers the ability to hide the internal state of an object, that is, to pretend that the object has no internal state at all. This is different, and more radical, than to expose the existence of an internal state while abstracting away its nature.

The concepts of both hiding and abstraction appear in Hoare's influential paper (1972). Most of the paper is devoted to the study of a form of abstraction: a relation between concrete states and abstract states, defined by a pair of an abstraction function $\mathscr{A}$ and a concrete invariant $\mathscr{I}$. Nevertheless, the idea that "benevolent side-effects", concrete side effects that do not change the abstract state, can be "wholly invisible" to the client, amounts to a form of hidden state. (In particular, if the abstract state has type unit, every concrete side effect is benevolent.) O'Hearn *et al.* (2004) revisit this idea in the setting of separation logic and attempt to explain it in terms of the so-called hypothetical frame rule. Birkedal *et al.* (2006) present an even more general rule, known as the higher-order frame rule. These approaches do permit a form of hidden state, albeit, we argue (Pottier, 2008), in an awkward way: When an object with hidden internal state is built, it cannot be returned, but must instead be passed on to a continuation. In short, the frame rules allow a capability to become temporarily hidden, whereas what is needed is for a capability to become permanently hidden. The use of continuation-passing style allows encoding the latter in terms of the former, but is undesirable in practice.

The anti-frame rule (Pottier, 2008) is meant to solve this problem. It allows the methods of an object, or a group of closures, to share a hidden internal state. It does not impose an unnatural use of continuation-passing style. It is sound in the setting of an ML-like calculus, where functions can be passed as arguments to functions, returned out of functions, and stored in the heap. In short, the rule allows a capability I to be visible within a certain lexical scope, and invisible to the outside. This capability is made available whenever control enters this scope, and must be released whenever control exits this scope: it plays the same role as Hoare's invariant $\mathscr{I}$.

Pottier (2008) presents three applications of the anti-frame rule, including an encoding of weak references in terms of strong references (we explain the distinction in Section 2), an implementation of lazy thunks, and an implementation of Landin's fixed point combinator. Pilkiewicz and Pottier (2011) explain how the anti-frame rule allows implementing a "lock" abstraction that serves a similar purpose as the anti-frame rule itself, but is more flexible and provides a weaker guarantee. These locks offer exactly the same interface as the primitive locks found in several versions of concurrent separation logic (Gotsman *et al.*, 2007; O'Hearn, 2007; Hobor *et al.*, 2008; Buisse *et al.*, 2011). Pilkiewicz and Pottier (2011) use these locks to hide the mutable internal state of a hash-consing facility.

The purpose of the present paper is to offer a unified and streamlined presentation of Charguéraud and Pottier's (2008) type-and-capability system , including the anti-frame rule (Pottier, 2008), and to prove the system as sound.

Charguéraud (unpublished observations) has an informal proof of the soundness of the type-and-capability system, without the anti-frame rule, and of the existence of the functional translation. This proof, which is purely syntactic, is about 25 pages long, and probably lacks detail in several places. The introduction of the anti-frame rule makes it significantly more difficult to establish type soundness. Pottier's conference paper (2008) contains only a proof sketch, which boils down to explicitly proving one key case of one key lemma.[3] Thus, there is a need for a detailed proof of soundness.

Schwinghammer *et al.* (2010) present the first complete soundness proof of the anti-frame rule, in the setting of a separation logic. They build a semantic model of the logic using certain recursively defined worlds. The construction of the set of worlds is non-trivial, as it requires solving a non-standard recursive domain equation. This equation exhibits a certain built-in monotonicity requirement, whose presence is made necessary by the anti-frame rule. Birkedal *et al.* (2011) present a semantic model of Charguéraud and Pottier's (2008) type-and-capability system, extended with a higher-order frame rule, but without group regions and the anti-frame rule. Compared with the previously cited work, this paper abandons denotational semantics in favor of operational semantics and step-indexing; and solves a simpler recursive domain equation, because it does not account for the anti-frame rule. Schwinghammer *et al.* (2011) put everything together, so to speak. By reusing and simplifying Schwinghammer *et al.*'s (2010) main ideas, they are able to extend

[3] In hindsight, this is indeed where most of the interesting difficulty is concentrated. However, the conference paper assumes the equational theory of the $\otimes$ operator, as well as the existence of certain recursive types, without justification. The construction of type equality, it turns out, is non-trivial (Section 9). Furthermore, the conference paper omits to mention the fact that adopting the anti-frame rule requires reintroducing the value restriction (Section 2).

Birkedal *et al.*'s (2011) step-indexed model with support for the anti-frame rule.[4] The journal paper by Schwinghammer *et al.* (2012) sums up this line of work.

In the present paper, we present a type soundness proof for a streamlined version of Charguéraud and Pottier's (2008) system (Pottier, 2008), with group regions and the anti-frame rule. Several changes in the presentation of the system are motivated by the desire to simplify the formalization. The most important such change is an abstract treatment of affinity, based on dual intuitionistic linear logic (DILL) and an axiomatization of monotonic separation algebras.

The soundness proof is purely syntactic: It follows the classic scheme proposed by Wright and Felleisen (1994), which is based on the properties of subject reduction and progress. It was developed concurrently with (and independently of) the proofs cited above. It confirms (if needed) that the type system is sound and shows that the syntactic proof technique can scale up to a system of this complexity.

The soundness proof is machine-checked. It consists of approximately 20,000 lines of Coq scripts and is available online for browsing (Pottier, 2012a) and downloading (Pottier, 2012b). It relies on the CoLoR library (Blanqui & Koprowski, 2011) at two places: a lexicographic path ordering is used in the proof that type equality is transitive (Lemma 6.3), and a multiset ordering is used to argue that the reduction of values terminates (Lemma 13.4).

1.4 Organization of the paper

The paper is divided in three parts. Part 1 is an overview of the architecture of the type-and-capability system and its soundness proof. Part 2 contains the definition of the system, and Part 3 presents the proof of type soundness.

To the reader who would like to quickly obtain a rough idea of how the system works, we suggest reading just the first part (Sections 2 and 3). To the reader who would like to quickly have access to the definition of the system, we suggest glancing at the syntax and the equational theory of types (Figures 8 and 9 in Section 6) and moving directly to the presentation of the typing and subtyping rules (Sections 7 and 8).

PART ONE
Overview

In this part, we present an overview of the architecture of the type-and-capability system and of its soundness proof (Section 2). We give a detailed example that shows several of the features of the system working in concert (Section 3).

[4] In fact, they are able to support the generalized frame and anti-frame rules. These rules, proposed in an unpublished note by Pottier (2009a), allow reasoning about the well-bracketing of function calls and returns.

2 Architectural and technical overview

In this section, we first explain the architecture of the type-and-capability system, and discuss several design choices (Section 2.1). Then we review the architecture of the type soundness proof, and discuss a number of technical choices (Section 2.2).

2.1 Architecture of the type-and-capability system

The type-and-capability system is constructed by starting with the polymorphic λ-calculus and successively adding a relatively small number of features, namely, affinity, references, capabilities and regions, an explicit treatment of erasure, and the anti-frame rule. Unfortunately, once the scaffolding is removed, this gradual construction is no longer apparent, and the definition of the system as a large set of subtyping and typing rules can seem overwhelming. Thus, before attempting to formally define the system, we offer a record of its gradual construction, and take this opportunity to discuss a few design choices.

The core. At the core of the system is the polymorphic λ-calculus, also known as System F (Girard, 1972; Reynolds, 1974). We equip it with universal, existential, and recursive types: all three forms of quantification play essential roles in the system. Existential types are typically used to describe operations that create new regions. Recursive types serve, as usual, to encode algebraic data types; furthermore, in the presence of the anti-frame rule, recursive types are required for subject reduction to hold.

Anticipating that many operations that transform capabilities and regions are presented as subtyping axioms, we equip the system with subtyping. Mitchell (1988) extends the polymorphic λ-calculus with a natural notion of subtyping, under the name F_η. We follow this route. The elimination of a universal quantifier and the introduction of an existential quantifier become subtyping axioms. Since we have recursive types, we include rules for subtyping recursive types, in the style of Brandt and Henglein (1998).

For the record, the definition and soundness proof for F_η took less than one week and about 2,000 lines of Coq scripts. Adding recursive types, to obtain F_η^μ, took two more weeks and brought the line count to 4,000. These figures do not have much absolute significance, but can be compared to those associated with the complete type-and-capability system whose formalization required roughly six months and represents 20,000 lines of Coq scripts.

Affinity. The next step is to set up machinery to prevent the duplication of certain values.

At this point, a question arises: What should the system support: linear values, which are used exactly once, affine values, which are used at most once, or both? The distinction between linear and affine values boils down to a distinction between explicit and implicit de-allocation. While linear values must be explicitly discarded (the act of discarding a value counts as one use), affine values may be implicitly discarded (the value is then never used). Linearity is appropriate for resources that are not automatically managed by the runtime system, such as file descriptors: It is desirable that the programmer be warned if she forgets to close a file descriptor. Affinity is appropriate for resources that are automatically reclaimed by the runtime system, such as memory in a garbage-collected system. In a realistic programming language, it would be important to support both linear and affine resources. In the present paper, for the sake of simplicity, we follow Charguéraud and Pottier (2008) and support only affine resources. Our resources are references (which are automatically reclaimed) and regions (which do not exist at runtime), so this is good enough.

Thus, we introduce a distinction between affine values, which are used at most once, and unrestricted values, which can be used any number of times. We set up the type system so as to keep track of this distinction and enforce the rule that affine values cannot be duplicated. Thus, the type system now counts how many times each variable is used and ensures that affine variables are used at most once. This is done in the style of DILL (Barber, 1996). This style is used, for instance, by Ahmed *et al.* (2007). At least one alternative exists; we discuss it later in Section 2.2.

By itself, the machinery that counts uses of variables serves no purpose: it is a mechanism without an end. In the end, it does not matter how many times variables are used; what matters is that the permission to access certain resources is not duplicated. At this level, however, we have not yet introduced any concrete resources: references, regions, etc. are introduced later on. Instead, we axiomatize an abstract notion of resource (described in detail in Section 10) and introduce a resource in the typing judgement. Then, by proving that subject reduction holds, we abstractly demonstrate that resources are not duplicated. Again, by itself, this result is not directly meaningful. However, once we introduce references and strong updates, for instance, it will allow us to establish type soundness in a straightforward way.

Concrete resources. We are now ready to introduce concrete resources and exploit the machinery that we have just set up in order to prevent the duplication of resources. In this paper, we are interested in two particular kinds of concrete, non-duplicable resources: references, on the one hand, and capabilities for regions, on the other hand.

These two features are entirely independent of one another, and can be introduced in any order. This remark is in fact a contribution of Charguéraud and Pottier's work (2008). In the prior work, regions are usually considered as sets of memory locations. Charguéraud and Pottier point out that it is simpler and more general to view regions as sets of values, which may or may not be memory locations.

We discuss references first, followed with capabilities and regions.

References. We extend the calculus with memory locations and stores, and extend the type system with a type of references. At this point, a question arises. What should the system support: strong references, weak references, or both?

Strong references support strong updates, that is, type-varying updates: the type of their content may change with time. Thus, they cannot be duplicable: they must be affine. Charguéraud and Pottier's (2008) references are strong. Weak references, on the other hand, are duplicable, but do not allow strong updates. That is, the type of their content is fixed at allocation time, and can never change. ML's references are weak.

Strong and weak references are incomparable. Strong references enable precise and flow-sensitive reasoning, but are somewhat inconvenient, because access requires a permission. Weak references are more lightweight, because whoever has the address is allowed to read and write, but they impose flow-insensitive reasoning.

In a realistic programming language, it seems desirable to support both strong and weak references. Several researchers study their combination and the interactions between them. Alms (Tov & Pucella, 2011), an affine λ-calculus, offers strong references under the name "`aref`" and weak references under the name "`ref`" (in the implementation only). Tan *et al.* (2009) study a separation logic equipped with both forms of references. The logic allows a strong reference to be (irreversibly) turned into a weak one. This is useful: in particular, it allows initializing a weak reference in a non-atomic manner. Ahmed *et al.* (2007) present a linear λ-calculus equipped with both forms of references. More precisely, they distinguish pointers and capabilities, so there is just one type of pointers, but there are two flavors of capabilities, a strong one and a weak one. Memory allocation creates a strong capability, which can later be "frozen", that is, turned into a weak (duplicable) capability. A weak capability does not allow reading or writing. However, it can be "thawed", that is, turned into a strong capability again. This allows reading and writing, including strong updates. A thawed capability can be "refrozen". The system requires that thawing and refreezing be performed in a type-consistent manner: indeed, the type of a weak reference must remain fixed. Furthermore, when freezing or thawing a memory location, one must prove that this location is not already thawed. This is a "global" proof obligation, which we believe can be difficult to meet. One way of meeting it is to restrict oneself to thawing at most one memory location at any time: This is analogous to the manner in which we allow focusing on at most inhabitant of a group region (see below). Another way is to use dynamic checks: Pilkiewicz and Pottier's (2011) implementation of "lock" and "unlock" in terms of the anti-frame rule can be viewed as a version of "thaw" and "refreeze" where the check that the location is not already thawed (the lock is not already held) is performed at runtime.

Pottier (2008) notes that in the presence of the anti-frame rule weak references can be programmed up on top of strong references. (Although not demonstrated in the conference paper, Tan *et al.*'s (2009) rule for turning a strong reference into a weak one can also be programmed up.) Thus, in principle, there is no need for weak references to be considered primitive. In the present paper, we follow this approach, and do not formalize weak references.[5]

[5] One should note that the current encoding of weak references in terms of strong references and the anti-frame rule introduces an inefficiency. Indeed, a weak reference is encoded as a hidden strong reference whose access is mediated by a pair of "get" and "set" functions. In a realistic programming language implementation, one would not accept the penalty imposed by this encoding. Thus, one would still want weak references to be primitive, or one would need to look for a cleverer encoding. At present, it is not known to the author whether one exists.

At this point, the system corresponds roughly to Alms (Tov & Pucella, 2011) in terms of its feature set. Alms differs mainly in its treatment of affinity; we come back to this point later on in Section 2.2.

Capabilities and regions. As we have argued earlier, directly preventing the duplication of values results in a very rigid type discipline. It is much more flexible to permit the duplication of values. If the use of values is governed by affine permissions, or capabilities, this remains sound. This idea appears in Alias Types (Smith *et al.*, 2000; Walker & Morrisett, 2000) and is developed in a very elegant way by Ahmed *et al.* (2007).

Naturally, as soon as one decides to introduce capabilities, it becomes necessary to keep track of which capabilities govern which values. In other words, when considering an instruction that attempts to access a certain value, and attempts to justify this access by presenting a certain capability, the type-checker needs a way of assuring that this particular capability is indeed associated with this particular value.

One simple way to do this is to introduce names for values. These names are static: they appear as part of types. The type of a capability refers to such a name: it says, "I grant access to the value named r". The type of a value also refers to such a name: informally, it says, "I am the value named r". (This is known as a singleton type.) The type-checker can then verify that the value and the capability that are presented together do refer to a common name r. The name r, which represents a single value, is known as a singleton region.

In order to allow writing real code, it is necessary to permit aliasing, that is, to allow situations where one does not statically know with certainty which values are equal and which are distinct. In our setting, this can be done by letting a name r denote not just a single value but a set of values. In that case, r is known as a group region. Then, the type of a value says, informally, "I am a member of the region r", while the type of a capability says, "I grant access to every value in the region r". There is just one capability for an entire region: there cannot be one per inhabitant, since one does not statically know how many inhabitants are there.

In summary, capabilities and regions go hand-in-hand. Regions are names that serve to keep track of the connection between values and capabilities. Regions are names for sets of values.

A group region has an arbitrary number of inhabitants, including zero, one, or more, while a singleton region has exactly one. For each region, whether it is a singleton or a group, is indicated by the capability for this region. The capability also carries information about the inhabitants of the region, namely, their type.

The type carried by a capability for a singleton region is the type of the single inhabitant of the region. If we wish to somehow affect this inhabitant (say, the inhabitant is a reference, and we write a new value to it), then it is easy to update the capability to carry a new type. Thus, singleton regions support strong updates. On the other hand, the type carried by a capability for a group region is the common type of all inhabitants of the region. If we wish to somehow affect one inhabitant, we must do so in a way that does not modify its type, because this type is shared with all other inhabitants whose number is unknown. Thus, group regions support weak updates only.

In Alias Types (Smith *et al.*, 2000; Walker & Morrisett, 2000; Ahmed *et al.*, 2007), every region is a singleton region. Vault (DeLine & Fähndrich, 2001) has group regions. In Vault, a guarded type "$R : T$" denotes a value of type T that inhabits region R. Vault also has singleton regions, and introduces mechanisms that let singleton and group regions interact (Fähndrich & DeLine, 2002). Charguéraud and Pottier (2008) adopt these mechanisms, and, because they assume the presence of a garbage collector, are able to simplify them.

Singleton regions are created as a result of memory allocation: whenever a fresh object is allocated, a fresh singleton region is created as well, which this object inhabits. Group regions are created empty. Group regions are populated via adoption: at any point in time, a singleton region may be merged into a group region, causing the latter to grow.

At any time, an inhabitant of a group region can be placed into a fresh singleton region. This is known as focusing. This allows "working with" this value: because it is now viewed as an inhabitant of a singleton region, various kinds of strong updates can be performed. However, as long as this process is taking place, any access to the original group region must be forbidden. One returns to the original situation by defocusing, that is, by abandoning the singleton region.

Focus and defocus represent the only way of accessing an inhabitant of a group region. For instance, dereferencing a pointer that inhabits a group region requires focusing, dereferencing, and defocusing. Of course, in a surface language, one might wish to offer syntactic sugar for these steps, or to infer where focus and defocus must take place.

Remarks on capabilities and regions. As explained above, when one focuses on one inhabitant of a group region, this object is temporarily "carved out" of the region, and the region becomes disabled until this object is returned. Following Boyland and Retert (2005), it would be possible to allow carving of multiple objects out of a group region, provided one statically proves or dynamically checks that these objects are distinct. The degenerate case where only one object can be carved out represents a sweet spot, in the sense that no side condition is needed. However, it may well turn out to be too restrictive in situations where simultaneous access to two distinct elements of a group region is required.

Our group regions are homogeneous: All inhabitants of the region must have a common structure, which is carried by the capability, whereas the type of an inhabitant carries no information beside the name of the region. One could also wish to support heterogeneous regions, where every inhabitant can have a different structure. In that case, the type of an inhabitant carries both the name of the region and the structure of this inhabitant, while the capability for the region carries no information beside the name of the region. Following Talpin and Jouvelot (1994), regions are traditionally heterogeneous (Tofte & Talpin, 1997; Crary *et al.*, 1999; DeLine & Fähndrich, 2001; Swamy *et al.*, 2006). Charguéraud and Pottier (2008) choose homogeneous regions because, through their functional translation, regions are represented by finite maps. Homogeneous regions become simply-typed finite maps, whereas heterogeneous regions would become dependently-typed finite maps, where the type of the data depends upon the key. It is not clear to us whether, in practice, one can get away with only, say, homogeneous regions. Monnier (2008) introduces an interesting hybrid that subsumes homogeneous and heterogeneous regions.

Boyland (2010) describes nesting, a mechanism by which one permission can become (irreversibly) nested within another permission. A nested permission can be temporarily recovered, or “carved out”, via a focus/defocus mechanism. Boyland notes that nesting is an extension of adoption. Indeed, it seems that heterogeneous group regions, together with adoption, focus and defocus, can be encoded in terms of singleton regions and nesting: The idea is to nest the permissions for each of the inhabitants within the permission for one master object that stands for the group region. We do not attempt to formalize nesting in this paper, but note that it might be relatively straightforward to do so. Boyland points out that the set of nesting facts increases over time, and, for this reason, a nesting fact may be safely duplicated. The same is true, in our system, of region membership facts. Our abstract treatment of resources accounts, once and for all, for the idea that monotonic information is duplicable.

Here regions are static names for sets of values. Regions have no existence at runtime. This makes sense because we assume garbage collection. Assuming garbage collection, in turn, makes sense because, in our system, not all values are affine, so not all values can be manually de-allocated. In Crary *et al.*’s Calculus of Capabilities (1999), regions are also static names, and, in addition, regions are represented at runtime by region handles. Region handles are run-time data structures. A region handle must be presented when allocating an object in a region, and in return, allows de-allocating the region together with all of the objects that it contains. The situation in Vault (Fähndrich & DeLine, 2002) is analogous. In Cyclone (Swamy *et al.*, 2006), one distinguished region, known as the “heap”, is garbage-collected, while other regions are explicitly managed by the programmer via region handles.

At this point, the system corresponds roughly to Ahmed *et al.*’s (2007) presentation of Alias Types, extended with group regions.

Erasure. Capabilities need not exist at runtime. This is an important remark, as the operations that rearrange capabilities tend to be quite numerous and might represent a significant cost if they were not completely erased. Of course, this remark is not new. Ahmed *et al.* (2007) note: “[Capabilities] have no operational significance. This observation yields a number of opportunities for optimizing and erasing the manipulation of capabilities”. Similarly, Tov and Pucella (2011) write that “capabilities have no run-time significance”, but list “not representing capabilities at run-time” as a part of the future work.

In these approaches, it sounds as if erasing capabilities is an optimization. This is not fully satisfactory, as it is then up to the compiler to silently decide whether, and to what extent, the optimization is applied. Furthermore, the design of the system might render optimization impossible in some situations. For instance, in the two systems cited above, the distinction between capabilities (which should be erased) and ordinary values (which cannot be erased) is encoded in the types. Yet, these systems allow abstracting over a type variable. Thus, when a value has type α, it is impossible to tell whether this value represents a capability or an ordinary value. Thus, the optimization cannot be applied, unless one is willing to perform code specialization.

We believe that the system should be designed in such a way that there is an unambiguous distinction between what is erased and not erased at runtime. This distinction can be encoded in the kinds, as in Charguéraud and Pottier's system (2008), or by other means; we come back to this issue in Section 2.2. As a benefit of this distinction, the programmer is presented with a clear cost model: the system guarantees that all operations that manipulate capabilities are erased. Technically, this means that we are able to equip our untyped calculus (a version of Core ML with references) with a completely standard operational semantics. We do define also an instrumented operational semantics, where capabilities exist at runtime. We prove that erasure is a simulation, that is, the two semantics are equivalent. It is important to establish this equivalence, as the instrumented semantics is somewhat exotic. In order to account for the anti-frame rule, for instance, the instrumented semantics needs to somehow explain how a capability appears out of thin air when control enters the scope of the rule, and disappears when control leaves this scope. In our approach the instrumented semantics is purely part of the type soundness proof; it is not part of the model that is presented to the programmer.

In the present paper, the distinction between the values that exist at runtime ("ordinary values") and those that do not ("capabilities") is entirely independent of the distinction between affine and unrestricted values. In principle, all four combinations of choices make sense and are useful. Charguéraud and Pottier (2008) make only two of the four combinations available: they fix the convention that ordinary values are duplicable, while capabilities are affine. Affine ordinary values are not quite lost, as they can be encoded as pairs of a duplicable value and an affine capability (Walker & Morrisett, 2000; Fähndrich & DeLine, 2002; Ahmed *et al.*, 2007). Unrestricted capabilities are however unavailable. Pilkiewicz and Pottier (2011) reintroduce them and illustrate two of their uses: there, purely logical assertions, as well as the so-called "observations", are duplicable capabilities.

Hidden state. It is fair to say that the study of the anti-frame rule is the main motivation for writing the present paper. Many of the features that we have described so far are required, in one way or another, before we can begin to discuss the anti-frame rule. Recursive types, for instance, are required in the subject reduction proof for the anti-frame rule: they are used to construct commutative pairs (Lemma 6.12), which, in turn, are used in the revelation lemma (Lemma 12.4). A notion of capability and a guarantee that capabilities are erased at runtime are required if one wishes to guarantee that the anti-frame rule has no operational significance. Mutable state, although not technically required by the anti-frame rule, is involved in all useful applications of the rule known to date.

With this large mass of prerequisites in place, we are ready to study the anti-frame rule. Introducing the rule, *per se*, is easy: we add one typing rule, and do not modify any of the existing rules. Significant work is involved, however, in introducing the type constructor $\otimes$, which appears in the anti-frame rule, together with its equational theory (Section 6); in introducing a technical subtyping axiom, $\otimes$-exch_n, and proving that it does not compromise type soundness (Section 8.7); in extending the instrumented operational semantics with support for the anti-frame rule (Section 13); and in establishing type soundness (Section 15).

Charguéraud and Pottier (2008) point out that there is no adverse interaction between polymorphism and strong references. Even though their type-and-capability system does not impose a syntactic value restriction in the style of Wright (1995), it is sound. Once the anti-frame rule is introduced, this pleasant state of affairs breaks down: some restriction becomes necessary again. We unfortunately did not make this point in the conference paper (Pottier, 2008); it appears in an unpublished note (Pottier, 2009b). Of course, in retrospect, this is obvious, because there is a well-known adverse interaction between polymorphism and weak references, and the anti-frame rule allows encoding weak references. The Coq formalization supports two variants of the system: one may choose between having both the value restriction and the anti-frame rule, or neither of them. The choice is encoded as a Boolean parameter, with respect to which the entire formalization is parametric: that is, type soundness holds regardless of the value of this parameter.

We believe that it is a contribution of this paper to emphasize the fact that unsoundness does not arise out of the interaction between polymorphism and the act of allocating fresh mutable state; it arises out of the interaction between polymorphism and the act of hiding. This is further discussed in Section 15.

Further features. Some features are omitted, which could have been welcome. In particular, recursive functions and sums are absent. Charguéraud and Pottier (2008) include both of these features. Here recursion can be obtained via one of the standard fixed-point combinators (which are well-typed, since the system has recursive types) or via Landin's knot (Pottier, 2008). We do not expect a major difficulty to arise while adding these features. We do expect that, as is often the case in large mechanized proofs, several minor difficulties might arise and might require existing definitions to be altered. We have not yet attempted to find out.

2.2 *Architecture of the proof*

Proof technique. The main result that we establish is type soundness: "Well-typed programs do not go wrong". The proof technique is syntactic: following Wright and Felleisen (1994), we separately prove subject reduction and progress statements. These results are first established at the level of an ad hoc instrumented calculus, then ported back down to a "raw" calculus, whose syntax and semantics are standard.

In comparison with "semantic" approaches (Ahmed, 2004; Ahmed *et al.* 2010; Birkedal *et al.* 2010, 2011; Schwinghammer *et al.* 2011), the syntactic approach seems more simple-minded. It is purely about syntax and reduction, whereas the semantic approach involves more complex mathematics. This is not to say that the overall effort involved in a syntactic proof is less significant than that required by a semantic proof. Our wild guess is, in the setting of a one-time endeavor, these efforts are roughly comparable. As argued by Bell *et al.* (2008), the semantic approach may represent a better investment in the long run, because its mathematical foundations are independent of the programming language and type system, and because the Hoare logic that serves as an intermediate layer is independent of the type system. Also, the semantic approach scales up to a binary setting (that is, to logical relations), whereas the syntactic approach does not.

In spite of its plausible lack of reusability, we adopted the syntactic technique because it seemed more elementary and because we wanted to test whether it would scale up to a complex type system without losing its perceived simplicity. The answer is positive, with reservations.

Let us attempt to compare the syntactic and semantic techniques in a slightly more detailed manner. In either case, we examine where ingenuity is required and where non-modular definitions or proofs have to be constructed.

In the syntactic approach, one needs ingenuity to define the syntax and semantics of the instrumented calculus, as well as to come up with suitable typing rules for the constructs that play a role in the semantics but do not appear in source programs. In addition, proving that the instrumented semantics agrees with the standard semantics is more difficult than one might expect, due to the need to prove that computationally irrelevant computation steps cannot cause an infinite loop (Lemma 13.4). This proof is non-modular: It requires coming up with a termination criterion for a reduction relation whose definition involves a large number of rules. As a result, it has an unfortunate tendency to break when new rules are introduced. The proofs of subject reduction and progress for the instrumented semantics, on the other hand, are fairly mechanical and modular. They consist of many independent cases, each of which usually goes through in a relatively straightforward way. Furthermore, extending the system with new features causes new cases to appear but usually does not break any existing cases.

The semantic approach stems from a desire to interpret types as sets of values. This approach sounds natural, but quickly leads to a number of difficulties. Accounting for impredicative polymorphism and recursive types is challenging (MacQueen *et al.*, 1986; Abadi *et al.*, 1991; Vouillon & Melliès, 2004). Accounting for types whose meaning evolves over time is also difficult. The latter problem is usually solved by setting up a Kripke model, that is, by parameterizing the interpretation of types over a world, for some ordered set of worlds, where the ordering represents a "possible future" relation. A Kripke model can be used, for instance, to account for the facts that new regions can be allocated and that the population of an existing region can only grow. A Kripke model can also be used to account for weak references. The model must then reflect the facts that new references can be allocated and that the type of an existing reference cannot change. For this purpose, a world must contain a map from memory locations to some form of types. Here, one can choose to use semantic types (that is, functions of worlds to sets of values), in which case the domain of worlds must be recursively defined; or syntactic types, in which case the interpretation of types must be recursively defined. Either way, subtle recursive definitions are required, and, in order to formulate these definitions in a sound manner, one must usually introduce some form of step-indexing or approximation (Ahmed, 2004; Ahmed *et al.*, 2010; Birkedal *et al.*, 2010). Modeling the higher-order frame and anti-frame rules requires a similar array of techniques (Schwinghammer *et al.* 2010, 2011; Birkedal *et al.* 2011). In summary, one needs ingenuity to come up with an appropriate definition of worlds, as well as interpretations of types and typing judgements. Furthermore, this construction is non-modular, in that extending the system with new features may require significant changes to these definitions. Once this is done, however, checking that each typing rule is valid with respect to the semantic model is a relatively mechanical and modular process.

Some concepts are common to the two approaches. For instance, something that resembles a Kripke model is present in a syntactic proof of type soundness. In a proof of type soundness for weak references, a store typing (Harper, 1994) serves as a world: it is used to give meaning to the `ref` type constructor, that is, to type-check memory locations. The statement of the subject reduction lemma constrains how the store typing evolves: over time, new memory locations may be allocated, but the type of an existing location does not change. A monotonicity lemma states that the evolution of the store typing preserves typing judgements. In the present paper, store typings are subsumed by an abstract notion of resource (Section 10), which is used to give meaning to strong references and regions. The manner in which resources evolve over time is constrained by two ordering relations. An analogous monotonicity lemma (Lemma 15.4) is established.

An instrumented calculus. Our “raw” calculus (Section 4) has standard syntax and semantics: it is an untyped λ-calculus equipped with pairs, references, nothing more. The final statement that “well-typed programs do not go wrong” is relative to this calculus, so it is perfectly clear what is meant by “to go wrong”.

However, as mentioned earlier, we do not establish subject reduction and progress directly with respect to the semantics of the raw calculus. It would be extremely difficult to do so. Indeed, the type system has a large number of non-syntax-directed typing rules, including universal quantifier introduction, existential quantifier elimination, introduction and elimination of the “!” modality, type equality, subtyping, the anti-frame rule, and more. The subtyping relation is defined by several dozen axioms and rules. The raw calculus has very few forms of redexes, but there are myriad ways in which each of these forms could be well-typed. Thus, a direct attempt at a subject reduction proof would be hopeless.

We work around this problem by artificially making the type system completely syntax-directed.[6] We introduce an instrumented calculus (Section 5), which has

[6] This technique is commonly used, for instance, to establish type soundness for System *F*. One first proves subject reduction and progress for a version of System *F* where type abstractions and type applications are explicit and require extra reduction steps. One then proves that these extra reduction steps have no computational meaning and can be erased. Type soundness with respect to the “raw”

syntax to indicate where quantifiers are introduced and eliminated, where subtyping is exploited, where the anti-frame rule is used, and so on. In order to make subtyping completely explicit, we introduce a syntactic category of coercions, in the style of Brandt and Henglein (1998), which can be thought of as proof terms, or witnesses, for a subtyping fact. In the instrumented calculus, this information is a part of the syntax of programs so that when the structure of a program is known, the structure of its type derivation is also known. (For the sake of convenience, the type equality rule remains non-syntax-directed.)

Naturally, we must arrange for the new syntactic forms that we introduce not to block reduction. We equip the instrumented calculus with an operational semantics, where new reduction rules explain how these new syntactic forms behave at runtime.

The two calculi are connected by an erasure function, which maps instrumented terms down to raw terms. The two semantics are related by a simulation lemma (Lemma 15.18): the erasure function is a weak simulation. To establish this lemma, we must prove two facts. First, a reduction step in the instrumented calculus is mapped by the erasure function to either a reduction step in the raw calculus, or no step at all. Second, the instrumented semantics does not introduce artificial non-termination: An infinite sequence of reduction steps in the instrumented calculus cannot be mapped by the erasure function to zero reduction steps in the raw calculus. (Hence, it must be mapped to at least one step; thus, it must be mapped to an infinite number of steps.)

The simulation lemma allows transporting the subject reduction and progress statements, which are initially proved at the level of the instrumented calculus, down to the level of the raw calculus. Thus, neither the semantics of the instrumented calculus nor the simulation lemma are part of the final statement of type soundness. The syntax of the instrumented calculus and the definition of the erasure function are part of it, because a "well-typed program" in the raw calculus is, by definition, the image through the erasure function of some well-typed program in the instrumented calculus.

Terms versus values. We view values and terms as distinct syntactic categories and adopt an administrative normal form where many positions are required to hold values. (The components of a pair must be values etc.) This allows us to distinguish the reduction of values, which does not affect the store, and the reduction of terms, which does affect.

In a traditional call-by-value calculus, the syntactic category of values embodies two distinct concepts. On the one hand, values are irreducible, and an irreducible term that is not a value is considered an error. On the other hand, values can be substituted for variables. In our instrumented calculus, however, these concepts no longer coincide. Because there is a subtyping rule for values, the syntax of values includes the application of a coercion to a value. Because coercion applications can reduce, it follows that values can reduce. Thus, our values are not irreducible.

λ-calculus follows. Thus, the technique is standard, but has seldom been applied at as large a scale as is the case here.

Fortunately, value reduction remains a relatively simple and well-behaved relation. It does not involve the store: it reduces a value to a value. It has no computational content: its image through the erasure function is the identity (Lemma 13.3). It is terminating (Lemma 13.4). For these reasons, it remains acceptable to substitute values for variables, even though a value may be able to reduce.

We define a subset of canonical values (Section 13.1) and prove that every well-typed value eventually reduces to a canonical value. The proof exploits subject reduction for values (Lemma 15.9), progress for values (Lemma 15.10), and the fact that value reduction is terminating (Lemma 13.4).

In summary, in the raw calculus, we distinguish raw values and raw terms, while in the instrumented calculus, we distinguish canonical values, values, and terms. The erasure function maps values (whether canonical or not) to raw values and terms to raw terms.

Representing names and binding. The POPLmark challenge (Aydemir *et al.*, 2005) has drawn attention to the difficulties associated with the representation of names and binders within proof assistants. Various representations have been recently investigated, including nominal representations (Urban, 2008), locally nameless representations with cofinite quantification (Charguéraud, 2012), and canonical locally named representations (Pollack *et al.*, 2012).

In this work, we choose to stick with the oldest solution to this problem, namely, de Bruijn indices (de Bruijn, 1972), because it is well understood and easy to work with. We set up a series of generic definitions, lemmas, and tactics about de Bruijn indices. These help equip an inductive type of interest (say, types τ) with the operations of lifting and substitution, and establish their properties. Lifting, written $k\uparrow\tau$, produces a copy of τ where every index above k is incremented. It can be thought of as an end-of-scope marker: Informally speaking, the type $k\uparrow\tau$ can be used in a context where the variable k is in scope if τ is a type where k is not in scope. Substitution, written $[\tau'/k]\tau$, substitutes the type τ' for the variable k through the type τ. It can be thought of as a beginning-of-scope marker: the variable k is in scope within τ, but not within τ' or within the result $[\tau'/k]\tau$.

Throughout this paper, we use these notations. We apologize to the reader for using de Bruijn indices in a paper that is meant to be read by humans. However, there is a reason to do so: The definitions and lemmas that we present are mechanically extracted from the Coq formalization, and, for this reason, cannot be easily translated into informal nominal syntax. This decision does mean that certain statements will look a bit unusual. Most of the time, fortunately, the reader will be able to ignore the details of de Bruijn indices by pretending that "$k\uparrow\tau$" is "τ". (In informal nominal syntax, this would amount to ignoring a freshness side condition.) Also, remember that "0" represents the most recently bound variable. We will translate a few statements into an informal nominal syntax to help the reader become familiar with our notation.

In hindsight, the use of de Bruijn indices seems to have been a reasonable choice. Setting up a generic machinery took only a couple of days, and this machinery was instantiated without difficulty for each of the syntactic categories that involve names and binders, namely, types, raw values and terms, coercions, and instrumented values and terms. We do acknowledge though that on a few occasions we had difficulty finding out how to express a statement in the de Bruijn notation.

Encoding affinity. The type system must somehow distinguish between affine and unrestricted values. There are several ways of doing this.

One possibility is to make this distinction explicit in the syntax of types. There are again multiple ways of doing this. One approach is to view all type constructors as affine by default and use an explicit "!" modality to indicate where duplication is permitted. In this case, explicit typing rules must be set up to deal with the modality. One could perhaps adopt the four classic rules of intuitionistic linear logic, namely, weakening, dereliction, contraction, and promotion, but Barber (1996) argues convincingly in favor of DILL, a formulation where there is just one introduction rule and one elimination rule for "!". One could also decorate every type constructor with an explicit qualifier, in the style of Ahmed *et al.* (2005), but it seems more economical to have an explicit modality than to build it into every other type constructor. Thus, we follow Barber's approach (1996). Because we are interested in only two modes, namely, "affine" and "unrestricted", one modality is enough.

Another possibility would be to not make any distinction in the syntax of types, and instead to use the kind assignment judgement to distinguish between affine and unrestricted types. Charguéraud and Pottier (2008) adopt this approach. There, a type of kind VAL describes a duplicable value, while a type of kind MEM describes an affine value. The kind VAL is a sub-kind of MEM so that a duplicable value can be provided where an affine value is expected. Charguéraud and Pottier (2008) use just one kind CAP of affine capabilities, but Pilkiewicz and Pottier (2011) introduce a kind DCAP of duplicable capabilities together with the sub-kinding axiom DCAP $\leqslant$ CAP. This mechanism is studied in isolation by Mazurak *et al.* (2010) and Tov and Pucella (2011), who incorporate a form of kind polymorphism.

In a surface language, we believe that encoding affinity information at the level of kinds is preferable, because this allows types to become significantly less verbose, while requiring relatively few kind annotations. This was indeed Charguéraud and Pottier's (2008) motivation for adopting this style. In a kernel language, where verbosity is not an issue, the choice is not so clear-cut. We choose to encode this information at the level of types because this allows us to get away with no kinds at all. Mazurak *et al.* (2010) note that either style can be translated into the other, so in principle one can encode a surface language that represents affinity at the level of kinds into a kernel language that represents it at the level of types.

Encoding erasability. We have argued above in favor of an unambiguous distinction between the "physical" and "logical" layers, that is, between what exists at runtime and what is erased. Charguéraud and Pottier (2008) encode this distinction at the level of kinds: values, which have kind VAL or MEM, exist at runtime, while capabilities, which have kind CAP, are erased. Thanks to the absence of kind polymorphism, this means that there exists a simple, kind-driven erasure algorithm. In this paper, motivated by the (perhaps artificial) desire to have no kinds at all, we again pursue another approach. We explore the idea that the distinction between what exists at runtime and what is erased should be so simple as to be syntactic. That is, ideally, a simple examination of the syntax of an instrumented term should let us determine what parts exist at runtime and what are erased. In other words, the erasure function should be a function of a term alone: it should not require any auxiliary information in the form of types or kinds.

When applied to a *physical* term (that is, a term that is meant to exist at runtime), the erasure function should produce a raw term. When applied to a *logical* term (that is, a term that is meant to be completely erased at runtime), the erasure function should produce "nothing". However, for simplicity, we would like to be able to view the erasure function as a total function from terms down to raw terms. This leads us to incorporating "nothing", that is, a special constant $\bullet$, as part of the syntax of raw terms.

We expect the erasure of a valid physical term to be a raw term where $\bullet$ does not occur. We expect the erasure of a valid logical term to be $\bullet$. In order to enforce this, we restrict our attention to the "well-layered" terms of the instrumented calculus. In short, it does not make sense to write a logical term where a physical one is expected, or *vice-versa*. We define a well-layeredness judgement that makes this rule precise, and classifies the values and terms of the instrumented calculus as either physical (Phy) or logical (Log).

The well-layeredness judgement involves an environment, which maps variables to layers. It is entirely independent of (and significantly simpler than) the typing judgement. We set up the instrumented semantics so that well-layeredness is preserved by reduction (Lemma 14.2). We exploit well-layeredness as a hypothesis in the proof of the simulation lemma (Lemma 15.18), which states that the instrumented semantics and the raw semantics agree. In these lemmas, no well-typedness hypothesis is required. This can be considered a positive aspect insofar as it simplifies our architecture.

Unfortunately, well-layeredness and well-typedness cannot be completely independent. A difficulty arises in the treatment of pairs and unit. What is the erasure of a pair? Because we would like to allow each component of a pair to be physical or logical, we really have four varieties of pairs. The erasure function must be able to distinguish between them. The erasure of a pair could be a pair (if both components are physical), the erasure of just one component (if this component is physical and the other component is logical), or nothing at all (if both components are logical; this variety of pair corresponds to the separating conjunction of separation logic). For this reason, in the instrumented calculus, we annotate each component of a pair with a layer ι, one of Phy and Log. Similarly, the pair elimination construct, $\mathsf{letpair}$, is annotated with two layers, and its image through the erasure function depends upon these annotations. The unit constant is also annotated with a layer: $()_{\mathsf{Phy}}$ represents the unit value, while $()_{\mathsf{Log}}$ represents the unit capability and corresponds to Charguéraud and Pottier's $\varnothing$ (2008) and the empty heap assertion **emp** of separation logic.

Annotating both pair introduction and pair elimination with layers introduces the possibility for a mismatch. Even within a well-layered term, pair introduction and pair elimination constructs with mismatched layers may come into contact. Should such a term reduce? No: that would break both the fact that reduction preserves well-layeredness (Lemma 14.2) and the fact that erasure is a simulation (Lemma 15.18). Thus, such a term must be stuck. Because we must prove that a well-typed term cannot be stuck, we must arrange for this term to be ill-typed, and, for this purpose, we must incorporate some layer information as part of the syntax of types. We decide to annotate the pair type constructor with one layer per component, so a pair type has the form ${}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}$, where ι_1 and ι_2 are layers and τ_1 and τ_2 are types. Thus, the type system must keep track of the distinction between the four varieties of pairs. This seems natural. Charguéraud and Pottier's system (2008) also keeps track of this distinction at the level of kinds. Analogously, we annotate the unit type with a layer.

In summary, the distinction between well-layeredness and well-typedness mostly works. However, the fact that the preservation of well-layeredness (Lemma 14.2) and simulation (Lemma 15.18) must be established without a well-typedness hypothesis can be considered a weakness. We did encounter cases where it seemed that such a hypothesis was necessary, and, because this hypothesis was not available, we resorted to formulating the instrumented semantics in such a way that these cases would not arise.[7] In hindsight, using kinds to distinguish between physical and logical entities, in the style of Charguéraud and Pottier (2008), might have been more elegant and robust.

Notation. In an attempt to eliminate as many typographical errors as possible, the definitions and lemmas presented in this paper are extracted from the Coq source and typeset by an ad hoc tool. In the interest of readability, the notation used by the tool is sometimes overloaded, hence ambiguous. Furthermore, in a few cases, certain "uninteresting" hypotheses, such as the well-formedness of all types, are omitted. (The reader is explicitly warned.) Naturally, the Coq formalization (Pottier, 2012a, 2012b) contains no such abuse and serves as a reference. In the electronic version of this paper, the ♡ symbol that marks the end of every definition and lemma is also a hyperlink to the online Coq version of this definition or lemma.

3 Illustration: an encoding of weak references with affine content

Before delving into the definition of the type-and-capability system, let us illustrate its main features by means of an example. The conference paper (Pottier, 2008) presents an encoding of weak references in terms of strong ones. These weak

[7] This can often be done by making certain reductions subject to extra side conditions. Thus, we have fewer cases to deal with in Lemmas 14.2 and 15.18, which assume that a certain reduction takes place, and more cases to deal with in the proof of progress, where we must show that well-typed terms do not get stuck (Lemma 15.17), and where a well-typedness hypothesis is available. In general, in the syntactic proof technique, adjusting the operational semantics allows one to move proof obligations between the subject reduction and progress lemmas.

```
λinitv.                        – this is mkwref
  let r = new initv in
  λnewv.                       – this is swap
    let newv = newv in
    let oldv = read r in
    write (r, newv);
    oldv
```

Fig. 1. Encoding weak references with affine content: raw version.

references support separate read and write operations. Because reading a reference creates a copy of its content, they are restricted to *duplicable content*, that is, in the terminology of the present paper, to content of type $!\,\tau$. In this section, we present a slightly different encoding: we show how to encode weak references with *affine content* (in other words, with arbitrary content) in terms of strong references and the anti-frame rule. These weak references provide a single operation, namely, "swap". This operation writes a new value to the reference and returns its previous value. Even though we implement "swap" in terms of the primitive read and write operations, the system is sufficiently expressive to recognize that this operation does not involve duplication.

3.1 The raw code

The untyped code, a term of the raw calculus (Section 4), is short (Figure 1). For readability, we have used names instead of de Bruijn indices. A weak reference with affine content is encoded as an "object" that offers a single "method", namely, "swap". In other words, a weak reference *is* a first-class function which, when invoked, swaps a new value for the current one. In Figure 1, the main function (which we refer to as *mkwref*, for "make a weak reference") creates a fresh reference r and returns a function (which we refer to as *swap*) that has access to r.[8]

One can see, informally, that the only way for the user to access r is via *swap*. As a result, if the initial value *initv* has type τ and if *swap* presents itself to the user as a function that expects an argument of type τ, then the property that the reference r exists and contains a value of type τ is true forever: it is an invariant. Furthermore, the user need not be aware of this invariant: all she needs to know is that *swap* expects an argument of type τ and produces a result of type τ. The type derivation that we are about to present is a formal justification of this argument.

Some of the material that follows (Sections 3.2 and 3.3) cannot be fully understood until the instrumented calculus and the type system have been presented. Nevertheless, we hope that, upon first reading, the reader can still gather a rough understanding of what is going on, and that this will help to clarify how the various bits and pieces fit together.

[8] The seemingly redundant binding "let $newv = newv$ in ..." will be explained shortly (Section 3.2).

```
 1 $\Lambda\,!\,\lambda initv.$                                         – this is $mkwref$
 2   let $xrc$ = new $initv$ in
 3   unpack $rc$ = $xrc$ in
 4   let $rc$ = $rc$ in hide
 5   letpair$_{\mathsf{Phy,Log}}$ $(r, c)$ = $rc$ in
 6   let! $r$ = at-bang $r$ in
 7   $_{\mathsf{Phy}}($
 8     $!\,\lambda newvc.$                                              – this is $swap$
 9       let $xnewvcc$ = (focus ...) $newvc$ in
10       unpack $newvcc$ = $xnewvcc$ in
11       letpair$_{\mathsf{Phy,Log}}$ $(newvc_1, c_2)$ = $newvcc$ in
12       letpair$_{\mathsf{Phy,Log}}$ $(newv, c_1)$ = $newvc_1$ in
13       let $oldvc_1$ = read $_{\mathsf{Phy}}(r, \{$ref at-bang$\}\ c_1)_{\mathsf{Log}}$ in
14       letpair$_{\mathsf{Phy,Log}}$ $(oldv, c_1)$ = $oldvc_1$ in
15       let $uc_1$ = write $_{\mathsf{Phy}}(_{\mathsf{Phy}}(r, newv)_{\mathsf{Phy}}, c_1)_{\mathsf{Log}}$ in
16       letpair$_{\mathsf{Phy,Log}}$ $(u, c_1)$ = $uc_1$ in
17       $_{\mathsf{Phy}}($
18         (defocus path-root) $_{\mathsf{Phy}}($dereliction $oldv, c_2)_{\mathsf{Log}}$
19         $c_1$
20       $)_{\mathsf{Log}}$,
21     $c$
22   $)_{\mathsf{Log}}$
```

Fig. 2. Encoding weak references with affine content: instrumented version.

3.2 The instrumented code

In order to prove that the raw term of Figure 1 is well-typed, we must first construct a term of the instrumented calculus (Section 5) whose erasure is the raw term of interest. This instrumented term appears in Figure 2. Again, we use names instead of de Bruijn indices, and keep doing so until the end of this section.

This term may at first seem quite daunting. In fact, quite a lot of noise is caused by the mundane operations of constructing and deconstructing pairs of a value and a capability, which are required in order to explicitly "thread" capabilities through the code. The more interesting aspects of this code are the anti-frame rule, represented by the let/hide construct; the "focus" and "defocus" operations, which allow reading a reference with affine content; and the introduction and elimination of the "!" modality, which allow proving that certain values are duplicable and using these values multiple times.

This instrumented term is well-layered, which means that there is no confusion between values that exist and runtime and values that do not (Section 14). Its erasure (Section 5.2) is the raw term that we are interested in.[9]

The instrumented version of *mkwref* has type $\forall\alpha.!\,(\alpha \to (!\,(\alpha \to \alpha)))$. Here is a formal version of this claim, whose proof has been machine-checked.

[9] The "let" definition on line 9 of Figure 2 is the reason why, after erasure, there remains a redundant definition, of the form "let $newv = newv$ in ... ", in the code of Figure 1. In order to eliminate this redundancy, it seems that one would need to make "focus" a coercion, as opposed to a primitive operation.

Lemma 3.1 *Provided the anti-frame rule is enabled, the term mkwref is well-typed:*

$$\frac{\textit{anti-frame enabled}}{\textit{void}, \textit{nil}, \textit{nil} \vdash \textit{mkwref} \ : \forall(!\,(0 \to (!\,(0 \to 0))))} \qquad \heartsuit$$

What does this type mean?

First, the function *mkwref* is polymorphic in α, the type of the content of the reference. The type variable α ranges over all types, which is why we speak of weak references with "affine content", as opposed to the weak references with duplicable content of the conference paper (Pottier, 2008).

Second, the function *mkwref* is duplicable. This means that one can create as many weak references as one wishes. It also means that *mkwref* can have multiple independent users, because invoking *mkwref* does not require presenting a unique capability, its users need not cooperate or even be aware of each other's existence.

Last, the value produced by a call to *mkwref* is itself duplicable, which is why we speak of "weak" references. This value is a weak reference, represented by a "swap" function. The fact that this value is duplicable means that one can use a weak reference as many times as one wishes. It also means that a weak reference can have multiple independent users: invoking *swap* does not require presenting a unique token. In other words, weak references can be freely aliased. This is in contrast with traditional affine (or linear) type systems (Walker, 2005), where a container of affine data must itself be affine.

3.3 The type derivation

Let us briefly explain the most important aspects of the instrumented term and the type derivation. The reader who would like more details is referred to the Coq formalization (Pottier, 2012a, 2012b), which makes it possible to interactively step through the type derivation.

On line 1 of Figure 2, the type variable α and the variable *initv*, which represents the initial value of the reference, are introduced. Our terms do not refer to types, which is why there is no reference to α in Figure 2.

On line 2, a fresh reference is allocated. This is a strong reference: its use is governed by a capability. Technically, the primitive operation new produces an existential package. On line 3, this package is opened, which introduces a fresh type variable ρ. Inside the package, we find a pair *rc* of a memory location *r* and a capability *c*. The value *r* has type $[\rho]$, which means that it inhabits the region ρ. The capability *c* has type $\{\mathsf{S}\,\rho : \mathsf{ref}\ \alpha\}$, which means that this capability represents the ownership of the region ρ and guarantees that ρ is a singleton region whose inhabitant is a reference to a value of type α.

On line 4, the anti-frame rule is applied, so that the existence of *r* is known to the code that follows the hide keyword and unknown to the outside world. The general form of the let/hide construct, in nominal notation, is "let $x = v$ in hide t. (In the de Bruijn notation, it will be just "let v in hide t".) Its meaning is the same as that of an ordinary let construct, but it is type-checked in a special way: an invariant is visible to the term *t* and invisible to the outside world.

One key piece of information that must be supplied at this point in the type derivation is the invariant. The invariant is a type θ. It is the type of a capability to which the code inside the hide construct has access and to which the outside world is oblivious. Again, because terms do not refer to types, the invariant does not appear in Figure 2.

What is θ? To a first approximation, it is $\{\mathsf{S}\,\rho : \mathsf{ref}\ \alpha\}$. This means that the reference r exists forever, its content is always a value of type α, and the code that follows the hide keyword has read and write access to r, as long as it preserves this invariant.

This tentative invariant would be appropriate if, instead of working with an abstract type α, we decided to fix a type τ of "inert" data, that is, a type τ that is built out of base types, products, and sums. Here, however, α can be instantiated with an arbitrary type τ, including one that involves functions.

Imagine, for instance, that α stands for the function type $\mathsf{unit} \to \mathsf{unit}$. In that case, $\{\mathsf{S}\,\rho : \mathsf{ref}\ (\mathsf{unit} \to \mathsf{unit})\}$, which means that the reference r contains a procedure, is not strong enough an invariant. The invariant must be "self-stable", so to speak, it must additionally guarantee that *the procedure that is stored in r preserves the invariant*. This is a recursive sentence. We translate it in formal terms as follows: the invariant must satisfy the recursive equation $\theta \equiv \{\mathsf{S}\,\rho : \mathsf{ref}\ ((\mathsf{unit} * \theta) \to (\mathsf{unit} * \theta))\}$. Thus, the invariant must be a recursive type.

The idea remains the same in the presence of more complex function types. For instance, if α stands for $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$, then the invariant must satisfy the equation $\theta \equiv \{\mathsf{S}\,\rho : \mathsf{ref}\ ((((\mathsf{unit} * \theta) \to (\mathsf{unit} * \theta)) * \theta) \to (\mathsf{unit} * \theta))\}$.

In order to support polymorphism, we must deal with the general case where α is a type variable and can later be instantiated with an arbitrary type. This leads us to introducing a new type operator, the "tensor" $\otimes$ (Section 6). For an arbitrary type τ, the type $\tau \otimes \theta$ can be thought of as a copy of τ where every function carries an additional argument and result of type θ. Thus, in the general case, the invariant must be a type θ that satisfies the equation $\theta \equiv \{\mathsf{S}\,\rho : \mathsf{ref}\ (\alpha \otimes \theta)\}$. Such a type exists: it is $\mu\beta.\{\mathsf{S}\,\rho : \mathsf{ref}\ (\alpha \otimes \beta)\}$. Thus, recursive types are required not only in the type soundness proof but also in practical applications of the anti-frame rule! This invariant implies in particular that, inside the hide construct, the reference r can be read and written at type $\alpha \otimes \theta$.

The anti-frame rule introduces a "change of vocabulary": whatever has type $\tau \otimes \theta$ inside the hide construct has type τ outside. This change is effective both "upon entry" and "upon exit", so to speak. If prior to entering the hide construct some variable x has type τ_1, then, within this construct, x appears to have type $\tau_1 \otimes \theta$. Symmetrically, if the value returned by the hide construct has type $\tau_2 \otimes \theta$, then, to the outside world, this value appears to have type τ_2.

For this reason, on line 4, the variable rc is redefined, with the same value, but with a new type. Outside "hide", its type was $[\rho] * \{\mathsf{S}\,\rho : \mathsf{ref}\ \alpha\}$. Inside "hide", its type is $([\rho] * \{\mathsf{S}\,\rho : \mathsf{ref}\ \alpha\}) \otimes \theta$. By the distribution laws that govern the "tensor" operator and by the equation that defines θ, this type is equal to $[\rho] * \theta$. Thus, on line 5, where we decompose the pair rc, we find that r has type $[\rho]$, while the capability c has type θ.

On line 6, we prepare for the variable r to be used by the function *swap*, where r occurs free. We want *swap* to be a duplicable function: this is indicated by the "!" introduction construct that precedes "λ" on line 8. This means that r must be "unrestricted", in the sense of DILL (Barber, 1996), when we type-check *swap*. At this point, however, r is not "unrestricted": it is "affine", which means that it can be used at most once. In order to rectify this, we proceed in two steps, both of which take place on line 6. First, we apply the coercion at-bang, which changes the type of r from $[\rho]$ to $![\rho]$. The intuitive idea is: because the fact that r inhabits the region ρ is true forever (it cannot be revoked), it is safe to mark this information as duplicable. Second, we use the let! construct, which we borrow from DILL, to eliminate the "!" modality and make r an "unrestricted" variable of type $[\rho]$. The general rule is that a variable bound by let! is unrestricted (it can be used for an arbitrary number of times), while a variable bound by any other construct is affine (it can be used at most once).

On lines 7–22, we build a pair of the *swap* function and of the capability c. This capability serves as a witness that the invariant initially holds: its presence here is required by the anti-frame rule.

Let us now examine the code of *swap*. On line 8, its argument, *newvc*, is introduced. It is a pair of a new value, which has type $\alpha \otimes \theta$, and a capability, which has type θ. The presence of this capability means that when *swap* is invoked the invariant holds.

On lines 9–12, we take a preparatory step whose purpose is to allow the reference r to be read. The primitive operation read is restricted to references whose content is duplicable. For this reason, it cannot directly be applied to r. In order to be able to read r, we must first "focus" on its content, that is, introduce a fresh singleton region σ, whose inhabitant is the value stored in r. This allows us to argue that this value has type $[\sigma]$, a duplicable type, and to read r. Reading r in this manner duplicates the value stored in r, but does not duplicate the capability that governs this value.

The focus operation on line 9 takes two arguments: a path, which indicates where to focus, and a value, *newvc*. For the sake of readability, the path has been elided in Figure 2, but will be shown here. Recall that *newvc* has type $(\alpha \otimes \theta) * \theta$, that is, $(\alpha \otimes \theta) * \{\mathsf{S}\,\rho : \mathsf{ref}\ (\alpha \otimes \theta)\}$. In other words, *newvc* is "a pair whose right-hand component is a capability for a singleton region whose inhabitant is a reference whose content is a value of type $\alpha \otimes \theta$". We wish to introduce a name, say σ, for the content of this reference. Thus, we must focus on the path path-right Phy Log (path-singleton (path-ref path-root)). This path indicates that, beginning at the root of the value *newvc*, we wish to descend into the right-hand component of a pair, then into a capability for a singleton region, and finally into a reference.

The focus operation, followed with the de-structuring operations on lines 10–12, produces the following entities: a fresh type variable σ; a value *newv* of type $\alpha \otimes \theta$; a capability c_1 of type $\{\mathsf{S}\,\rho : \mathsf{ref}\ [\sigma]\}$, which represents the ownership of the reference r, but not of its content, which inhabits the region σ; and a capability c_2 of type $\{\mathsf{S}\,\sigma : \alpha \otimes \theta\}$, which represents the ownership of the region σ, hence the ownership of the value currently held in r.

On lines 13–14, we read r. The capability c_1 serves as a proof that this is legal. We obtain a pair of a value *oldv*, which has type $![\sigma]$, and a capability c_1, whose type is unchanged.

On lines 15–16, we write r. The capability c_1 serves as a proof that this is legal. We obtain a pair of a unit value and a capability c_1, whose type is updated: this is a strong update. The type of c_1 is now $\{\mathsf{S}\,\rho : \mathsf{ref}\ (\alpha \otimes \theta)\}$, that is, θ. We have just re-established the invariant, which was broken earlier by the "focus" operation.

On lines 17–20, we are done. We return a pair of the value *oldv*, which is the "real" result of the function *swap*, and of the capability c_1, which proves that we have re-established the invariant. In order to make *oldv* a suitable result, however, a couple of steps are required. First, because we are about to use *oldv* only once, we apply the coercion dereliction, which converts the type of *oldv* from $![\sigma]$ to just $[\sigma]$. Second, by a "defocusing" operation, we bring together the value *oldv*, which has type $[\sigma]$, and the capability c_2, which has type $\{\mathsf{S}\,\sigma : \alpha \otimes \theta\}$. This operation produces a value of type $\alpha \otimes \theta$. Thus, we forget that *oldv* inhabits σ and keep only the information that *oldv* has type $\alpha \otimes \theta$.

What have we achieved so far? Considering our hypothesis about *swap*'s argument, *newvc*, and considering our analysis of *swap*'s result, we have proved that *swap* has type $!(((\alpha \otimes \theta) * \theta) \to ((\alpha \otimes \theta) * \theta))$. By the equational theory of tensor, this type is equal to $(!(\alpha \to \alpha)) \otimes \theta$.

Thus, the pair of *swap* and c, which extends from line 7 to line 22, has type $((!(\alpha \to \alpha)) \otimes \theta) * \theta$, a type which we also write $(!(\alpha \to \alpha)) \circ \theta$. This type means that (i) *swap* preserves the invariant, and (ii) the invariant holds initially.

This is where the "change of vocabulary" imposed by the anti-frame rule kicks in again. If the code that follows the hide keyword produces a value of type $(!(\alpha \to \alpha)) \circ \theta$, then the entire let/hide construct is deemed to have type $!(\alpha \to \alpha)$. In other words, as far as the outside world is concerned, this term behaves like a duplicable function of α to α.

The final step is to conclude that *mkwref*, which extends from lines 1 to 22, has type $\forall\alpha.!(\alpha \to (!(\alpha \to \alpha)))$, as claimed earlier.

3.4 Comments

We have shown how, thanks to the anti-frame rule, one can implement weak references with affine content in terms of strong references with affine content. Thus, the anti-frame rule offers a way of implementing *a duplicable container with affine content*, something that is forbidden by traditional affine or linear type systems (Walker, 2005).

Certain more recent systems do include duplicable references with non-duplicable content, with the restriction that these references cannot be read: they support only the "swap" and "write" operations (Ahmed *et al.*, 2005; Swamy *et al.*, 2006; Tov & Pucella, 2011). Our approach offers a reconstruction of these duplicable references with affine content. It is interesting to note that Ahmed *et al.* (2005) wrote: "Storing a unique object in a shared reference 'hides' the unique object in some way". The anti-frame rule turns this intuition into an explicit mechanism.

Our references with affine content offer a single “swap” operation. If one would like to have separate “get” and “set” operations, two possibilities come to mind.

One approach is to require the content to have a duplicable type, of the form $!\,\alpha$. This approach is illustrated in the conference paper (Pottier, 2008).

Another approach is to check *at runtime* that the content of a reference is never read twice. For this purpose, one builds on top of the weak references with affine content defined above. One creates a weak reference whose content has type *option* α. Then, on top of the “swap” function, which has type $!\,(\mathit{option}\,\alpha \to \mathit{option}\,\alpha)$, one defines a “set” function of type $!\,(\alpha \to \mathsf{unit})$ and a “get” function of type $!\,(\mathsf{unit} \to \mathit{option}\,\alpha)$. The “get” function can fail, that is, it can return the value *None*. Furthermore, this function has a hidden side effect: it writes the value *None* into the reference. Thus, if “get” is invoked twice, without an intervening call to “set”, then the second invocation must fail. This is *a dynamic mechanism for enforcing affinity*. Such mechanisms are useful because they can serve to link code that is type-checked within a type-and-capability system (which keeps track of ownership and distinguishes affine and duplicable data) with code that is type-checked within an ordinary type system (where everything is considered duplicable).

Tov and Pucella (2010) draw attention to this idea. They propose a primitive mechanism whereby an affine value is encapsulated within a wrapper that maintains one bit of hidden mutable state and presents itself as a duplicable value. In their more recent work on Alms (Tov & Pucella, 2011), these primitive wrappers are no longer necessary; instead, duplicable references with affine content, equipped with a “swap” operation, are considered a primitive construct, so the above approach, based on references of type *option* α, can be used.

In an unpublished note (Pottier, 2009b), we have pointed out that the anti-frame rule is “paranoid”, that is, very restrictive. Because it requires the invariant to hold whenever control enters or exits the area where the invariant is visible, it effectively prevents the code within this area from exploiting pre-existing libraries in order to manage its hidden state. Fortunately, there is a way around this problem: The anti-frame rule can be used to implement a “lock” abstraction that offers much greater flexibility than the direct use of the anti-frame rule (Pilkiewicz & Pottier, 2011). This comes at the cost of a dynamic check, and a potential runtime failure, should the code attempt to obtain a single lock twice. In short, locks are another instance of a dynamic mechanism for enforcing affinity.

The anti-frame rule is sound only in a sequential setting. To see this, consider the above encoding of weak references with affine content. It is clear that our implementation of “swap” in terms of reading and writing is not atomic. Hence, in a concurrent setting, this implementation would be incorrect: It would be possible for two concurrent invocations of “swap” to read the same value, thus violating affinity. There is no obvious way to “fix” the anti-frame rule for use in a concurrent setting. Instead, primitive dynamically allocated locks in the style of concurrent separation logic (Gotsman *et al.*, 2007; O’Hearn, 2007; Hobor *et al.*, 2008; Buisse *et al.*, 2011) can be considered a replacement for it. They are in fact easier to understand than the anti-frame rule because they do not involve a “tensor” and a

$$\begin{array}{rcl} V & ::= & x \mid \lambda T \mid (V,V) \mid () \mid l \mid \bullet \\ T & ::= & V \mid V\ T \mid \mathsf{letpair}\ V\ \mathsf{in}\ T \mid \mathsf{new}\ V \mid \mathsf{read}\ V \mid \mathsf{write}\ V \end{array}$$

Fig. 3. Syntax of the raw calculus.

"change of vocabulary". However, in their present form, these concurrent separation logics do not guarantee the absence of deadlocks.

PART TWO Definition

In this part, we define the type-and-capability system. We describe the syntax and semantics of the untyped ("raw") calculus (Section 4). We present the syntax of a non-standard ("instrumented") calculus (Section 5). The terms of this calculus carry type-checking information that facilitates the type soundness proof. The two calculi are related via an erasure function. We introduce types (Section 6) and define the typing (Section 7) and subtyping (Section 8) judgements.

4 The raw calculus

The raw calculus is a standard call-by-value λ-calculus equipped with pairs and references. The syntax of raw values and raw terms appears in Figure 3.

Raw values include variables x, where x is a de Bruijn index; functions λT, where one variable is bound in T; pairs (V, V); the unit value $()$; memory locations l; and a special value $\bullet$, pronounced "erased". A memory location is a natural integer. The value $\bullet$ plays a role in the definition of the erasure function, which connects the instrumented calculus with the raw calculus (Section 5.2). There it represents an "erasure error". It never appears in the erasure of a well-layered program (Section 14).

Raw terms include values V; function application $V\ T$; pair elimination $\mathsf{letpair}\ V\ \mathsf{in}\ T$, where two variables are bound in T; and three primitive operations for allocating, reading, and writing references, $\mathsf{new}\ V$, $\mathsf{read}\ V$, and $\mathsf{write}\ V$. For the sake of uniformity, all primitive operations expect exactly one argument; in particular, in $\mathsf{write}\ V$, the value V is expected to be a pair of a memory location and a value. We sometimes write $\mathsf{let}\ T_1\ \mathsf{in}\ T_2$ as syntactic sugar for $(\lambda T_2)\ T_1$.

We let M range over partial functions of memory locations to raw values, that is, total functions of memory locations to optional raw values. We write $\bot$ for Coq's "`None`" and write V for Coq's "`Some` V". Thus, the image $M\,l$ of a memory location l through a function M is either $\bot$ or a value V. We let both l and ℓ range over memory locations. The meta-variable ℓ usually denotes the allocation limit, that is, the first available memory location. A *raw store* S is a pair of a function M, which represents the contents of the store, and an allocation limit ℓ. We write $M\ \mathsf{below}\ \ell$ for such a pair.

$$S / (\lambda T)\ V \longrightarrow S / [V/0]T \qquad S / \mathsf{letpair}\,(V_1, V_2)\,\mathsf{in}\ T \longrightarrow S / [V_1/0][0\uparrow V_2/0]T$$

$$\frac{S_1 / T_1 \longrightarrow S_2 / T_2}{S_1 / V\ T_1 \longrightarrow S_2 / V\ T_2} \qquad \frac{M_1[\ell \mapsto V] = M_2}{M_1\ \mathsf{below}\,\ell / \mathsf{new}\ V \longrightarrow M_2\ \mathsf{below}\,\ell + 1 / \ell}$$

$$\frac{M\,l = V}{M\ \mathsf{below}\,\ell / \mathsf{read}\ l \longrightarrow M\ \mathsf{below}\,\ell / V} \qquad \frac{M_1\,l = V_1 \qquad M_1[l \mapsto V_2] = M_2}{M_1\ \mathsf{below}\,\ell / \mathsf{write}\,(l, V_2) \longrightarrow M_2\ \mathsf{below}\,\ell / ()}$$

Fig. 4. Operational semantics of the raw calculus.

The operational semantics of the raw calculus takes the form of a single reduction relation: the proposition $S_1 / T_1 \longrightarrow S_2 / T_2$ holds when the raw configuration S_1 / T_1 reduces in one step to the raw configuration S_2 / T_2. (We use the word *configuration* for a pair of a store and a closed term.) The inductive definition of this predicate appears in Figure 4. The first two rules respectively reduce a β-redex and a letpair-redex. The third rule allows reduction under a context: in this calculus, the only evaluation context is the right-hand side of an application. The reduction rule for new V extends the store with a binding of ℓ to V, where ℓ is the current allocation limit, and increments the allocation limit. The reduction rule for read l fetches the value V that is currently stored at address l. The rule is applicable only if this address currently holds a value, that is, $M\,l \neq \bot$. The reduction rule for write (l, V_2) updates the store with a binding of l to V_2. Again, the rule is applicable only if a previous binding of l to V_1 exists, that is, $M\,l \neq \bot$.

5 The instrumented calculus

The instrumented calculus extends the raw calculus (Section 4) with explicit constructs for coercions, capabilities, quantifier introduction, etc. Its operational semantics describes how these new constructs behave: in particular, the application of a coercion to a value reduces to a new value. None of these constructs is intended to exist at runtime: an erasure function, which maps terms to raw terms, formalizes this idea.

In this section, we briefly present the syntax of the instrumented calculus and define the erasure function. The meaning of each construct is explained in greater depth as we present the typing and subtyping rules (Sections 7 and 8). The semantics of the instrumented calculus is discussed only after the type system is defined (Section 13).

5.1 Syntax

The syntax of the instrumented calculus, presented in Figure 5, involves several syntactic categories. *Layers* ι are used to distinguish between physical entities (which exist at runtime) and logical entities (which do not). *Coercions* c can be thought of as proof terms for subtyping judgements. *Focus paths* π appear in the focus and defocus operations. *Primitive operations* p include the standard operations for allocating, reading, and writing references, as well as operations that have no

ι	$::=$	$\mathsf{Phy} \mid \mathsf{Log}$	(layers)
c	$::=$	$\mathsf{id} \mid c;c \mid$	(reflexivity, transitivity)
		$c \rightarrow c \mid {}_{\iota}(c \times c)_{\iota} \mid \forall c \mid \exists c \mid\ !\,c \mid \mathsf{ref}\ c \mid \{c\} \mid \{c\backslash\} \mid$	(congruence)
		$\forall\mathsf{I} \mid \forall\mathsf{E} \mid \exists\mathsf{I} \mid \exists\mathsf{E} \mid$	(quantifier intro/elim)
		$\mathsf{distrib} \mid \exists\mathsf{LI} \mid \forall\text{-}\mathsf{pair} \mid \forall\text{-}\mathsf{bang} \mid \forall\text{-}\mathsf{ref} \mid \forall\text{-}\mathsf{regioncap} \mid$	(quantifier movement)
		$\forall\text{-}\mathsf{regioncappunched} \mid \mathsf{pair\text{-}exists\text{-}left} \mid$	
		$\mathsf{pair\text{-}exists\text{-}right} \mid \mathsf{bang\text{-}exists} \mid \mathsf{ref\text{-}exists} \mid$	
		$\mathsf{cap\text{-}exists} \mid$	
		$\mathsf{dereliction} \mid \mathsf{bang\text{-}idempotent} \mid \mathsf{pair\text{-}bang} \mid$	(affinity)
		$\mathsf{bang\text{-}pair} \mid \mathsf{unit\text{-}bang} \mid \mathsf{bang\text{-}ref} \mid \mathsf{bang\text{-}regioncap} \mid$	
		$\mathsf{bang\text{-}regioncappunched} \mid \mathsf{at\text{-}bang} \mid$	
		$\mathsf{singleton\text{-}to\text{-}group} \mid \mathsf{defocus}\,\pi \mid \mathsf{defocus\text{-}group} \mid$	(regions)
		$\mathsf{star\text{-}comm} \mid \mathsf{star\text{-}assoc} \mid \mathsf{star\text{-}ref} \mid \mathsf{ref\text{-}star} \mid$	(movement of stars)
		$\mathsf{star\text{-}singleton} \mid \mathsf{singleton\text{-}star} \mid \otimes\text{-}\mathsf{exch}_x \mid$	
		$x \mid \mu c$	(recursive coercions)
π	$::=$	$\mathsf{path\text{-}root} \mid \mathsf{path\text{-}left}\,\iota\,\iota\,\pi \mid \mathsf{path\text{-}right}\,\iota\,\iota\,\pi \mid \mathsf{path\text{-}ref}\,\pi \mid$	(focus paths)
		$\mathsf{path\text{-}singleton}\,\pi$	
p	$::=$	$\mathsf{defocus\text{-}dup}\,\pi \mid \mathsf{focus}\,\pi \mid \mathsf{newgroup} \mid \mathsf{adopt} \mid$	(primitive operations)
		$\mathsf{focusgroup} \mid \mathsf{new} \mid \mathsf{read} \mid \mathsf{write}$	
v	$::=$	$x \mid \lambda t \mid {}_{\iota}(v,v)_{\iota} \mid ()_{\iota} \mid \Lambda v \mid \mathsf{pack}\ v \mid c\ v \mid\ !\,v \mid l\%v \mid \{\vec{v}\} \mid$	(values)
		$\{?::\vec{v}\} \mid [v]$	
$\vec{v}$	$::=$	$\epsilon \mid v :: \vec{v}$	(lists of values)
t	$::=$	$v \mid v\ t \mid {}_{\mathsf{Phy}}(t,v)_{\mathsf{Log}} \mid \Lambda t \mid c\ t \mid \mathsf{unpack}\,v\,\mathsf{in}\,t \mid \mathsf{let!}\,v\,\mathsf{in}\,t \mid$	(terms)
		$\mathsf{letpair}_{\iota,\iota}\,v\,\mathsf{in}\,t \mid p\ v \mid \mathsf{let}\,v\,\mathsf{in\ hide}\,t$	

Fig. 5. Syntax of the instrumented calculus.

runtime effect. *Values* v, w and *terms* t have more elaborate syntax than their raw counterparts.

A pair value ${}_{\iota_1}(v_1, v_2)_{\iota_2}$ is annotated with two layers ι_1 and ι_2, one per component. This information guides the erasure process (Section 5.2). Similarly, a unit value $()_\iota$ is annotated with a layer. The value $()_{\mathsf{Phy}}$ is the usual unit value, whereas the value $()_{\mathsf{Log}}$ represents a trivial capability and corresponds to $\emptyset$ in Charguéraud and Pottier's paper (2008) and to **emp** in separation logic.

The constructs Λv, $\mathsf{pack}\ v$, and $!\,v$ mark the presence of an introduction rule for the type constructors $\forall$, $\exists$, and $!$ respectively. The construct $c\ v$ represents an application of the coercion c to the value v.

Remark 5.1 The syntax of the instrumented calculus is type-free. No coercion, value, or term refers to a type. In particular, although the placement of type abstractions and type applications is explicit, it is not said which type variables or types are involved. Thus, the construct Λv does not introduce a type variable. It is in fact not a binder. The coercion $\forall\mathsf{E}$ does not indicate which type is used to instantiate the universal quantifier. In most presentations of System F, terms do refer to types. This helps the machine perform type-checking, and allows the programmer to impose abstraction barriers. However, for the purpose of establishing type soundness, this information is not necessary. Getting rid of it simplifies the syntax and semantics of the instrumented calculus. ◇

A value of the form $l\%v$ represents a memory location l. It carries the value v that is currently contained in the store at location l. This design is unusual, and may come as a surprise. How is it sound, and how is it useful?

How is it sound? By decorating the pointer l with the value v that it points to, we seem to run the risk that someone writes a new value to l and v becomes stale. We are in fact safe because references are affine: a value of the form $l\%v$ has a type of the form $\mathsf{ref}\ \tau$, which is an affine type. Roughly speaking, the type system ensures that, for each memory location l, there is at most one value of the form $l\%v$ in existence, and v is precisely the value that is currently stored at l. This is expressed by the first typing rule in Figure 23.

How is it useful? Why decorate l with v? One technical reason is that this helps define a semantics for the coercion $\mathsf{ref}\ c$, which asserts that the type constructor ref is covariant. As we will see, the coercion application $(\mathsf{ref}\ c)\ (l\%v)$ reduces to $l\%(c\ v)$. In short, we are able to "push the coercion down into the store", without actually consulting the store. This allows us to formalize the semantics of coercions in terms of reduction of values, as opposed to reduction of pairs of a store and a value.

The value $\{\vec{v}\}$ represents a capability over a (singleton or group) region, whose inhabitants are exactly the values in the list $\vec{v}$. The name of the region is not mentioned. Again, the reason why it is possible for the capability to carry an exact list of the inhabitants is that the capability is affine.

The value $\{? :: \vec{v}\}$ represents a capability over a (group) region in which a hole has been punched (via the primitive operation $\mathsf{focusgroup}$), and whose remaining inhabitants are the values in the list $\vec{v}$.

A value of the form $[v]$ represents the value v, marked as a region inhabitant. That is, the constructor $[\cdot]$ serves as a hint to the type system that this value should receive a type of the form $[\rho]$, which means "inhabitant of region ρ", regardless of the structure of the value v.

Let us now move on to terms.

The term ${}_{\mathsf{Phy}}(t, v)_{\mathsf{Log}}$ represents a pair of a (physical) term t and a (logical) value v. Reduction is permitted inside the term t. As we will see (Section 7.2), this construct corresponds to the frame rule of separation logic: The value v represents a capability that is set aside during the computation t.

The construct Λt marks the introduction of a universal quantifier. The presence of this construct in the syntax of terms (as opposed to only in the syntax of values) means that we do not hardwire the value restriction in the syntax of the calculus. Indeed, we offer a choice between having both the anti-frame rule and the value restriction, or neither of them. The construct Λt is well-typed only in the latter case.

$$\begin{array}{rclcrcl}
\lfloor x \rfloor & = & x & \quad & \lfloor l\%v \rfloor & = & l \\
\lfloor \lambda t \rfloor & = & \lambda \lfloor t \rfloor & & \lfloor \Lambda v \rfloor & = & \lfloor v \rfloor \\
\lfloor {}_{\mathsf{Phy}}(w_1, w_2)_{\mathsf{Phy}} \rfloor & = & (\lfloor w_1 \rfloor, \lfloor w_2 \rfloor) & & \lfloor \mathsf{pack}\ v \rfloor & = & \lfloor v \rfloor \\
\lfloor {}_{\mathsf{Log}}(w_1, w_2)_{\mathsf{Phy}} \rfloor & = & \lfloor w_2 \rfloor & & \lfloor c\ v \rfloor & = & \lfloor v \rfloor \\
\lfloor {}_{\mathsf{Phy}}(w_1, w_2)_{\mathsf{Log}} \rfloor & = & \lfloor w_1 \rfloor & & \lfloor !\, v \rfloor & = & \lfloor v \rfloor \\
\lfloor {}_{\mathsf{Log}}(w_1, w_2)_{\mathsf{Log}} \rfloor & = & \bullet & & \lfloor [v] \rfloor & = & \lfloor v \rfloor \\
\lfloor ()_{\mathsf{Phy}} \rfloor & = & () & & \lfloor \{\vec{v}\} \rfloor & = & \bullet \\
\lfloor ()_{\mathsf{Log}} \rfloor & = & \bullet & & \lfloor \{? :: \vec{v}\} \rfloor & = & \bullet
\end{array}$$

Fig. 6. Erasure: values.

The constructs $\mathsf{unpack}\, v\, \mathsf{in}\, t$ and $\mathsf{let!}\, v\, \mathsf{in}\, t$ serve as elimination forms for the type constructors $\exists$ and $!$ respectively. Both bind a (term) variable in the term t. The universal quantifier $\forall$ does not require an elimination construct in the syntax of terms: it is eliminated via the coercion $\forall\mathsf{E}$.

The construct $\mathsf{letpair}_{\iota_1,\iota_2}\, v\, \mathsf{in}\, t$ deconstructs a pair. It binds two variables in the term t. Similar to the pair constructor, it is annotated with two layers ι_1 and ι_2, which guide the erasure function.

The construct $p\ v$ is the application of a primitive operation p to a value v. By convention, all primitive operations have arity one.

The term $\mathsf{let}\, v\, \mathsf{in\ hide}\, t$ represents an application of the anti-frame rule. In order to simplify the operational semantics of the instrumented calculus, we adopt the convention that exactly one variable, represented by the de Bruijn index 0, is in scope in the term t. The value v is substituted for this variable before t is executed. Thus, to a first approximation, the semantics of $\mathsf{let}\, v\, \mathsf{in\ hide}\, t$ is just that of a normal "let" definition. More details are provided later on (Section 13).

Remark 5.2 In a higher-level language, it would be more natural to offer a construct "hide t", where t is allowed to have an arbitrary number of free variables $\vec{x}$. Such a construct is used in the conference paper (Pottier, 2008), where it is further annotated with the invariant that is being hidden. In theory, one can define "hide" in terms of "let/hide" by instantiating v with a tuple $(\vec{x})$ of the free variables of t. That is, in informal nominal syntax, "hide t" is sugar for "let $x = (\vec{x})$ in hide let $(\vec{x}) = x$ in t", where the variable x is fresh. The conference paper uses the syntax "hide $I = \tau$ outside of t", which indicates that the hidden invariant is the recursive capability $\mu I.\tau$. In the present paper, terms do not refer to types, so the corresponding construct is just "hide t". ◇

Let m range over partial functions of memory locations to values. A *store* s is a pair of a function m and an allocation limit ℓ. We write $m\, \mathsf{below}\, \ell$ for such a pair.

5.2 *Erasure*

The erasure function, defined in Figures 6 and 7, relates the instrumented calculus and the raw calculus. We write $\lfloor v \rfloor$ (resp. $\lfloor t \rfloor$) for the erasure of a value v (resp. of a term t). Erasure is not type-directed, thanks to the explicit layer annotations carried by some constructs (unit, pair construction, pair deconstruction), no type information is required.

$$
\begin{array}{rcl@{\qquad}rcl}
\lfloor v\ t \rfloor &=& \lfloor v \rfloor\ \lfloor t \rfloor & \lfloor (\mathsf{defocus\text{-}dup}\,\pi)\ v \rfloor &=& \lfloor v \rfloor \\
\lfloor {}_{\mathsf{Phy}}(t_1, v_2)_{\mathsf{Log}} \rfloor &=& \lfloor t_1 \rfloor & \lfloor (\mathsf{focus}\,\pi)\ v \rfloor &=& \lfloor v \rfloor \\
\lfloor \Lambda t \rfloor &=& \lfloor t \rfloor & \lfloor \mathsf{newgroup}\ v \rfloor &=& \lfloor v \rfloor \\
\lfloor c\ t \rfloor &=& \lfloor t \rfloor & \lfloor \mathsf{adopt}\ v \rfloor &=& \lfloor v \rfloor \\
\lfloor \mathsf{let!}\,v\,\mathsf{in}\,t \rfloor &=& [\lfloor v \rfloor/0]\lfloor t \rfloor & \lfloor \mathsf{focusgroup}\ v \rfloor &=& \lfloor v \rfloor \\
\lfloor \mathsf{unpack}\,v\,\mathsf{in}\,t \rfloor &=& [\lfloor v \rfloor/0]\lfloor t \rfloor & \lfloor \mathsf{new}\ v \rfloor &=& \mathsf{new}\ \lfloor v \rfloor \\
\lfloor \mathsf{let}\,v\,\mathsf{in\ hide}\,t \rfloor &=& [\lfloor v \rfloor/0]\lfloor t \rfloor & \lfloor \mathsf{read}\ v \rfloor &=& \mathsf{read}\ \lfloor v \rfloor \\
\lfloor \mathsf{letpair}_{\mathsf{Phy},\mathsf{Phy}}\ v\,\mathsf{in}\,t \rfloor &=& \mathsf{letpair}\ \lfloor v \rfloor\ \mathsf{in}\ \lfloor t \rfloor & \lfloor \mathsf{write}\ v \rfloor &=& \mathsf{write}\ \lfloor v \rfloor \\
\lfloor \mathsf{letpair}_{\mathsf{Phy},\mathsf{Log}}\ v\,\mathsf{in}\,t \rfloor &=& [\lfloor v \rfloor/0][\bullet/0]\lfloor t \rfloor & && \\
\lfloor \mathsf{letpair}_{\mathsf{Log},\mathsf{Phy}}\ v\,\mathsf{in}\,t \rfloor &=& [\bullet/0][0\!\uparrow\!\lfloor v \rfloor/0]\lfloor t \rfloor & && \\
\lfloor \mathsf{letpair}_{\mathsf{Log},\mathsf{Log}}\ v\,\mathsf{in}\,t \rfloor &=& [\bullet/0][\bullet/0]\lfloor t \rfloor & &&
\end{array}
$$

Fig. 7. Erasure: terms.

The erasure function can be considered a part of the definition of the type-and-capability system. Indeed, a raw term is considered well-typed if and only if it is the erasure of some well-typed instrumented term. A raw term is in general the erasure of many instrumented terms, which correspond to candidate type derivations.

The definition of the erasure of values (Figure 6) is straightforward. A pair of two physical components is a "real" pair: its erasure is a pair. The erasure of a "mixed" pair of one physical component and one logical component is the erasure of the physical component. A pair of two logical components is itself a purely logical value: its erasure is the special raw value $\bullet$. Similarly, the empty capability $()_{\mathsf{Log}}$, as well as capabilities for regions $\{\vec{v}\}$ and $\{? :: \vec{v}\}$, are mapped to $\bullet$.

The definition of the erasure of terms (Figure 7) is also relatively simple.

Several constructs that exist only in the instrumented calculus and involve binding are erased via a substitution. For instance, the erasure of $\mathsf{unpack}\,v\,\mathsf{in}\,t$ is defined as $[\lfloor v \rfloor/0]\lfloor t \rfloor$, rather than $\mathsf{let}\ \lfloor v \rfloor\ \mathsf{in}\ \lfloor t \rfloor$. (In informal nominal syntax, one would say: the erasure of $\mathsf{unpack}\,x = v\,\mathsf{in}\,t$ is defined as $[\lfloor v \rfloor/x]\lfloor t \rfloor$, rather than $\mathsf{let}\,x = \lfloor v \rfloor\ \mathsf{in}\ \lfloor t \rfloor$.) This makes it possible for these constructs to exist in the instrumented syntax and to be completely invisible in the raw syntax. For instance (in informal nominal syntax), the erasure of $\mathsf{unpack}\,x = x\,\mathsf{in}\,t$ is just the erasure of t. No spurious β-redex in the raw term betrays the fact that an unpack construct is present in the instrumented term. In summary, the syntax of a raw term T does not suggest where unpack constructs are needed: these can be inserted anywhere in an attempt to build a well-typed term of the instrumented calculus whose erasure is T and thus prove that T is well-typed.

The erasure of a $\mathsf{letpair}$ construct is a $\mathsf{letpair}$ construct only in the case where both components are physical. In the other three cases, the $\mathsf{letpair}$ construct is erased via a substitution; any variables that were bound by $\mathsf{letpair}$ to logical values are replaced with $\bullet$.

The primitive operations that have no runtime effect, such as $\mathsf{focus}\,\pi$, disappear. The primitive operations that do have a runtime effect, to wit, the three primitive operations on references, are translated to the constructs of the raw calculus that manipulate references.

q ::=		
	$\mid\ \forall$	universal quantifier
	$\mid\ \exists$	existential quantifier
κ ::=		
	$\mid\ \mathsf{S}$	singleton region
	$\mid\ \mathsf{G}$	group region
τ ::=		
	$\mid\ x$	type variable
	$\mid\ \tau \rightarrow \tau$	function type
	$\mid\ {}_{\iota}(\tau \times \tau)_{\iota}$	pair type
	$\mid\ \mathsf{unit}_{\iota}$	unit type
	$\mid\ q\tau$	universal or existential type
	$\mid\ \mu\tau$	recursive type
	$\mid\ !\,\tau$	duplicable type
	$\mid\ \mathsf{ref}\ \tau$	reference type
	$\mid\ r$	region identifier
	$\mid\ [\tau]$	region inhabitant type
	$\mid\ \{\kappa\,\tau : \tau\}$	region capability
	$\mid\ \{\tau : \tau \setminus \tau\}$	punched region capability
	$\mid\ \tau \otimes \tau$	tensor type

Fig. 8. Types.

The erasure function is extended to stores in the obvious way. We write $\lfloor s \rfloor$ for the erasure of a store s.

6 Types

6.1 Syntax

The syntax of types appears in Figure 8. We let τ and θ range over types. We let ρ and σ also range over types, with the informal idea that ρ and σ denote regions, although there is no formal such requirement.

Let us briefly review the meaning of each type construct. x is a type variable, represented as a de Bruijn index. $\tau_1 \rightarrow \tau_2$ is the type of functions that map an argument of type τ_1 to a result of type τ_2. ${}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}$ is a type of pairs whose left-hand component has type τ_1 and right-hand component has type τ_2. The layers ι_1 and ι_2 indicate which components of the pair exist at runtime. unit_{ι} is the type of a unit value. Again, the layer annotation indicates whether this value exists at runtime. $\forall\tau$ and $\exists\tau$ are universal and existential types, respectively, while $\mu\tau$ is a recursive type. These three forms bind a type variable in τ. The type $!\,\tau$ describes values (or terms) that have type τ and do not own (or consume) any affine resources. The type $\mathsf{ref}\ \tau$ is the type of references that (currently) hold a value of type τ. The type r is a region identifier. Region identifiers do not appear in source programs, where regions are represented by type variables. Similar to memory locations, region identifiers are dynamically allocated. The type $[\rho]$ is the type of the values that inhabit the region ρ. The type $\{\kappa\,\rho : \tau\}$ represents a capability over the region ρ. The kind of the region – singleton or group – is indicated by κ, while the common structure of the region inhabitants is given by τ. The type $\{\rho : \tau \setminus \sigma\}$ represents a capability over the group region ρ. The inhabitants of the region have common structure τ. A hole has been punched out of this region: one value has been taken out, and is now an inhabitant of the singleton region σ. Last, the type $\tau \otimes \theta$ can be thought of as a copy of τ where every function carries an additional argument and result of type θ. This additional argument and result is in the logical layer: it does not exist at runtime. In other words, a value of type $\tau \otimes \theta$ can be roughly described as a value of type τ that, in addition, requires and returns a capability θ at every interaction with its environment. This "tensor" construct appears in the statement of the higher-order frame and anti-frame rules (Birkedal *et al.*, 2006; Pottier, 2008).

It is useful to introduce a small amount of extra notation. Charguéraud and Pottier (2008) use $\times$ to denote a pair of two ordinary values, which exist at runtime. They use $*$ to denote a pair of an ordinary value and a capability, or a pair of two capabilities. By analogy, we write $\tau_1 \times \tau_2$ for ${}_{\mathsf{Phy}}(\tau_1 \times \tau_2)_{\mathsf{Phy}}$, $\tau_1 * \tau_2$ for ${}_{\mathsf{Phy}}(\tau_1 \times \tau_2)_{\mathsf{Log}}$, and also $\tau_1 * \tau_2$ for ${}_{\mathsf{Log}}(\tau_1 \times \tau_2)_{\mathsf{Log}}$. Hopefully, no serious ambiguity arises from this abuse of notation. Of course, the machine-checked proof uses unambiguous notation.

Following earlier papers (Pottier, 2008; Schwinghammer *et al.*, 2010), we write $\tau \circ \theta$ for $(\tau \otimes \theta) * \theta$, that is, for ${}_{\mathsf{Phy}}((\tau \otimes \theta) \times \theta)_{\mathsf{Log}}$.

We let $\vec{\tau}$ and $\vec{\theta}$ denote lists of types. We write $\tau \otimes \vec{\tau}$ for the iterated application of the tensor operator $\otimes$ and $\tau \circ \vec{\tau}$ for the iterated application of the composition operator $\circ$.

6.2 Type equality: axiomatization

The type system has a non-trivial notion of type equality. This relation must satisfy a number of properties, which we now list.[10] The actual construction of this relation can be considered a purely technical issue and is deferred to Section 9.

Lemma 6.1 *Lifting is compatible with type equality.*

$$\frac{\tau \equiv \theta}{k\uparrow\tau \equiv k\uparrow\theta} \qquad \heartsuit$$

Lemma 6.2 *Substitution is compatible with type equality.*

$$\frac{\tau_1 \equiv \tau_2 \qquad \theta_1 \equiv \theta_2}{[\theta_1/k]\tau_1 \equiv [\theta_2/k]\tau_2} \qquad \heartsuit$$

Lemma 6.3 *Type equality is reflexive, symmetric, and transitive.* ♡

Lemma 6.4 *Type equality is a congruence.* ♡

Lemma 6.5 *Type equality satisfies the equational theory in Figure 9.* ♡

[10] For better readability, lemmas are presented as inference rules.

$$\begin{aligned}
\mu\tau &\equiv [\mu\tau/0]\tau \\
(\tau_1 \to \tau_2) \otimes \theta &\equiv (\tau_1 \circ \theta) \to (\tau_2 \circ \theta) \\
({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}) \otimes \theta &\equiv {}_{\iota_1}((\tau_1 \otimes \theta) \times (\tau_2 \otimes \theta))_{\iota_2} \\
(\mathsf{unit}_\iota) \otimes \theta &\equiv \mathsf{unit}_\iota \\
(q\tau) \otimes \theta &\equiv q(\tau \otimes (0\uparrow\theta)) \\
(!\,\tau) \otimes \theta &\equiv !\,(\tau \otimes \theta) \\
(\mathsf{ref}\ \tau) \otimes \theta &\equiv \mathsf{ref}\ (\tau \otimes \theta) \\
[\sigma] \otimes \theta &\equiv [\sigma] \\
\{\kappa\,\sigma : \tau\} \otimes \theta &\equiv \{\kappa\,\sigma : \tau \otimes \theta\} \\
\{\rho : \tau \setminus \sigma\} \otimes \theta &\equiv \{\rho : \tau \otimes \theta \setminus \sigma\}
\end{aligned}$$

Fig. 9. The equational theory of types.

The first law in Figure 9 states that a recursive type $\mu\tau$ is equal to its unfolding $[\mu\tau/0]\tau$. In informal nominal syntax, one would write: a recursive type $\mu\alpha.\tau$ is equal to its unfolding $[\mu\alpha.\tau/\alpha]\tau$.

Thus, we are working with equi-recursive types (Gapeyev *et al.*, 2002). In the alternative iso-recursive approach, the types $\mu\tau$ and $[\mu\tau/0]\tau$ are considered distinct, but are subtypes of each other: explicit coercions are provided for converting one into the other. We choose the equi-recursive approach because it seems more natural and elegant: it is more pleasant to work with a rich notion of type equality than with painful explicit coercions. In short, the equi-recursive approach requires more work upfront during the construction of type equality, but simplifies the type soundness proof.

The remaining laws in Figure 9 govern the tensor operator and formalize our informal suggestion that "$\tau \otimes \theta$ is a copy of τ where every function carries an additional argument and result of type θ". In particular, the second law of Figure 9 can be written under the following form:

$$(\tau_1 \to \tau_2) \otimes \theta \equiv ((\tau_1 \otimes \theta) * \theta) \to ((\tau_2 \otimes \theta) * \theta)$$

This shows that when "$\cdot \otimes \theta$" reaches a function type, it transforms this function type by introducing "$\cdot * \theta$" in its domain and codomain. In this law as well as in every other law, "$\cdot \otimes \theta$" propagates itself down recursively so as to transform all occurrences of function types in this way.

The laws that govern tensor describe its action on every type construct, with four exceptions. There is no law that describes the action of tensor on a type variable, on a recursive type, on another tensor, or on a region identifier. An application of tensor to a type variable, of the form $x \otimes \theta$, cannot be simplified: it is effectively suspended until x is instantiated. An application of tensor to a recursive type or to another tensor, of the form $(\mu\tau) \otimes \theta$ or $(\tau_1 \otimes \tau_2) \otimes \theta$, can be simplified only by first simplifying the left-hand side until a type constructor other than μ or $\otimes$ appears at its root. An application of tensor to a region identifier, of the form $r \otimes \theta$, cannot be simplified. This last point has no deep significance: we could have adopted the law $r \otimes \theta \equiv r$. This law would serve no useful role, however, because, in practice, tensor is never applied to a region identifier: observe, how in the last three rules of Figure 9 tensor does not propagate down into ρ and σ.

A number of seemingly natural laws are *not* included as part of the equational theory of Figure 9. Consider, for instance, the candidate law $\tau_1 * \tau_2 \equiv \tau_2 * \tau_1$. A reader who is familiar with separation logic (Reynolds, 2002) might understand this law as an assertion that separating conjunction is commutative, and might be inclined to view it as reasonable. Indeed, in separation logic, where assertions are interpreted as predicates over heaps, this law is valid. However, in the present paper, our interpretation of types is strictly syntactic: the type constructor $*$ constructs ordered pairs. Adopting the law $\tau_1 * \tau_2 \equiv \tau_2 * \tau_1$ would be inconsistent. In combination with the fact that the pair type constructor is injective (Lemma 6.6), this law would imply that all types are equal, which in turn would contradict the fact that disjoint type constructors have disjoint ranges (Lemma 6.9). For an analogous reason, the candidate law $(\tau \otimes \theta_1) \otimes \theta_2 \equiv \tau \otimes (\theta_1 \circ \theta_2)$, which appeared in the author's earlier paper (Pottier, 2008), cannot be made part of the equational theory.

Fortunately, the lack of these laws is not an insurmountable problem. Following Charguéraud and Pottier (2008), the commutativity of $*$, for instance, can be considered a subtyping axiom. Thus, the types $\tau_1 * \tau_2$ and $\tau_2 * \tau_1$ are subtypes of each other. This is almost as good as type equality, but not quite as convenient: in the present paper, type equality is transparent, whereas subtyping can be exploited only via explicit coercions. The author must confess to not being fully aware of this distinction until the machine-checked proof revealed it.

The candidate law $(\tau \otimes \theta_1) \otimes \theta_2 \equiv \tau \otimes (\theta_1 \circ \theta_2)$ could also be considered a subtyping axiom. Instead, we find it slightly more economical to use a different subtyping axiom, which we dub "tensor exchange" (Section 6.3). This highly technical subtyping axiom is not expected to be useful to the end user, but plays a role in the subject reduction proof.

The type constructors other than μ and $\otimes$ must be injective. This property is exploited (as usual) in the proof of subject reduction.

Lemma 6.6 *The type constructors other than μ and $\otimes$ are injective. For instance, in the case of the function type constructor, we have:*

$$\frac{\tau_1 \to \tau_2 \equiv \theta_1 \to \theta_2}{\tau_1 \equiv \theta_1 \qquad \tau_2 \equiv \theta_2} \qquad \heartsuit$$

Recursive type equations must have unique solutions. This property is used, in particular, to establish the existence of commutative pairs (Section 6.3).

Lemma 6.7 (Unique solutions) *Provided the type τ is contractive with respect to the type variable 0, the recursive equation $0 \equiv \tau$ (an equation where the type variable 0 represents the unknown) admits a unique solution, to wit, the recursive type $\mu\tau$. That is, we have:*

$$\frac{\theta \equiv [\theta/0]\tau}{\theta \equiv \mu\tau} \qquad \heartsuit$$

In informal nominal syntax, one would say that the recursive type equation $\alpha \equiv \tau$ admits a unique solution, namely, the recursive type $\mu\alpha.\tau$. This lemma requires a contractiveness hypothesis because not all recursive equations admit a unique solution. Here are two representative examples, in informal nominal syntax. First, consider the equation $\alpha \equiv \alpha$. Every type τ is a solution of it. Second, consider the equation $\alpha \equiv \alpha \otimes \tau$. According to the laws of Figure 9, many types satisfy this equation: two such types are $\mathsf{unit}_{\mathsf{Phy}}$ and $[r]$, where r is a region identifier.

Contractiveness is defined in such a way that the unique solutions lemma (Lemma 6.7) implies nothing about the above equations. Furthermore, the recursive types $\mu\alpha.\alpha$ and $\mu\alpha.(\alpha \otimes \tau)$ are considered ill-formed, because they are not the unique solutions of the above equations. The definitions of contractiveness and well-formedness are deferred to Section 9. In short, well-formedness requires every cycle in the type structure to go through at least one type constructor other than μ or the left-hand side of $\otimes$.

Throughout the paper, we adopt the informal convention that "we work with well-formed types only". As a result, we do not explicitly show any of the well-formedness hypotheses. In the Coq formalization, of course, these hypotheses are explicit.

The following property is exploited in the proof of subject reduction for the "tensor exchange" coercion.

Lemma 6.8 *The type constructors other than μ and $\otimes$ are stable under tensor, in the following sense: if $\tau_1 \otimes \tau_2$ exhibits one of these type constructors at its root, then τ_1 exhibits the same type constructor. For instance, in the case of the function type constructor, we have:*

$$\frac{\exists \theta_1\theta_2,\ \tau_1 \otimes \tau_2 \equiv \theta_1 \rightarrow \theta_2}{\exists \theta_1\theta_2,\ \tau_1 \equiv \theta_1 \rightarrow \theta_2} \qquad \heartsuit$$

All of our requirements so far are positive, so the relation that equates all types satisfies them. Of course, such a trivial definition of type equality is unacceptable, because we also have one negative requirement,[11] which is exploited (as usual) in the proof of progress.

Lemma 6.9 *The type constructors other than μ and $\otimes$ have pairwise disjoint ranges. For instance, in the case of the function and reference type constructors, we have:*

$$\neg(\tau_1 \rightarrow \tau_2 \equiv \mathsf{ref}\ \theta) \qquad \heartsuit$$

Remark 6.10 We do not require that type equality be decidable because this property is not exploited in the proof of type soundness. It would be required, however, in the implementation of a type-checker if one wished to stick with equi-recursive types. Efficiently testing recursive polymorphic types for equality is a subtle problem (Glew, 2002; Gauthier & Pottier, 2004). The equational theory of tensor presumably makes it more difficult still. In a practical surface language, one would probably circumvent the problem by using named iso-recursive types instead of equi-recursive types. ⋄

[11] In fact, we have a quadratic number of disjointness requirements. In Coq, one uses a single tactic definition to avoid explicitly stating a quadratic number of lemmas.

6.3 Commutative pairs

The revelation lemma (Lemma 12.4), which plays an important role in the argument of soundness of the anti-frame rule, requires the technical notion of a commutative pair. We define this notion here because it is part of the theory of type equality, but defer a full explanation until we come to the revelation lemma.

Definition 6.11 *We write* $(\tau, \theta) \bowtie (\tau', \theta')$ *when the types* τ, θ, τ', θ' *form a* commutative pair. *This notion is defined as follows:*

$$\frac{\tau' \equiv \tau \otimes \theta' \qquad \theta' \equiv \theta \otimes \tau'}{(\tau, \theta) \bowtie (\tau', \theta')} \qquad \heartsuit$$

For the revelation lemma to go through, it is necessary that commutative pairs exist. That is, it is necessary that, for arbitrary types τ and θ, there exist types τ' and θ' that satisfy the above equations. Because the equations are mutually recursive, this may seem non-obvious. Fortunately, because tensor is contractive in its second argument, these equations are contractive. So by the unique solutions lemma (Lemma 6.7), they admit a solution.

Lemma 6.12 (Commutative pairs) *Commutative pairs exist. For all types* τ *and* θ,

$$\exists \tau'\theta', \; (\tau, \theta) \bowtie (\tau', \theta') \qquad \heartsuit$$

The proof is trivial. In informal nominal syntax, the type τ' is just $\mu\alpha.(\tau \otimes (\theta \otimes \alpha))$, where α is fresh for τ and θ. Symmetrically, θ' is just $\mu\alpha.(\theta \otimes (\tau \otimes \alpha))$. These recursive types are well-formed.

7 Typing

We now come to the central part of the definition of the type system: The typing judgements for values and terms. These judgements have identical forms: the typing judgement for values takes the form $R, M, E \vdash v : \tau$, while the typing judgement for terms takes the form $R, M, E \Vdash t : \tau$.

Within the syntactic category of values, we distinguish between the constructs that are available to the programmer, on the one hand, and the constructs that exist only for the purposes of the reduction semantics, on the other hand. This distinction is traditional. Memory locations are a typical example of a construct that appears in an operational semantics but is not directly available to the programmer. Here, in addition to memory locations $l\%v$, region inhabitants $[v]$, and capabilities $\{\vec{v}\}$ and $\{? :: \vec{v}\}$ are constructs that cannot appear in source programs. Thus, the typing rules for these constructs are of no concern to programmers. Their role is to participate in the definition of the global invariant that the type system enforces. The presentation of these rules is deferred to Section 11.

The component R in a judgement $R, M, E \vdash v : \tau$ indicates what resources are "owned by" the value v, or, to put it slight differently, which references and regions one may access if one owns v. The component R in a judgement $R, M, E \Vdash t : \tau$

indicates what resources are initially "owned by" the term t, or, in other words, which references and regions this term may access when it is executed.

The component R plays a non-trivial role only in the typing rules that concern constructs that are inaccessible to the programmer. The rules that concern programmer-accessible constructs ignore R, or propagate it, or split it, but never actually consult it. As a result, a source program is well-typed under some resource R if and only if it is well-typed under the empty resource. In other words, when type-checking a source program, the component R may be omitted entirely. Programmers need not know about resources and their theory.

For this reason, we defer the definition of resources to Section 10, and defer the presentation of the typing rules for programmer-inaccessible constructs to Section 11. For the moment, the reader can essentially ignore the role played by resources in the typing rules. It should be sufficient to note that multiplicity environments (which are described below) and resources are always treated in the same manner.

Remark 7.1 The role played by resources is analogous to the role played by store typings in a syntactic proof of type soundness for ML with weak references (Harper, 1994). There the typing judgement for *source* programs has three components, to wit, a typing environment, a term, and a type. However, the typing judgement for programs that are *being executed* has one more component, namely, a store typing, which assigns types to memory locations. Even though store typings play an important role in the statement of subject reduction, programmers need not know about them. ◇

The components M and E in a typing judgement are environments. We wish to present an affine variant of DIIL (Barber, 1996). In Barber's presentation, a typing judgement involves two environments Γ and Δ, both of which map variables to types. A variable that appears in Γ is unrestricted: it may be used for an arbitrary number of times. A variable that appears in Δ is affine: it may be used at most once. At a binary node in the type derivation (that is, at a function application, a pair, etc.), the environment Γ is transmitted to both premises, whereas the environment Δ is split as Δ_1, Δ_2 and the first (resp. second) premise receives just Δ_1 (resp. Δ_2). How should we emulate this approach? Recall that we represent variables as de Bruijn indices and environments as lists. In Barber's approach, a variable that is currently in scope may occur in Γ, or in Δ, or in neither of them. (The third situation means that the variable is affine but has been sent down some other branch of the type derivation.) In our setting, this is not very convenient. If a variable is represented as a de Bruijn index x, we would like to be assured that information about this variable is found at the xth slot in the environment. To achieve this, we adopt a formulation where a *multiplicity environment* M maps variables to multiplicities and a *type environment* E maps variables to types. M and E are represented as lists. The environment M records how many times a variable may be used, whereas the environment E records its type. A *multiplicity* m is one of 0, 1, and ∞. The multiplicity ∞ means that the variable is unrestricted ("it appears in Γ"); the multiplicity 1 means that it is affine ("it appears in Δ"); and the multiplicity 0 means that it is in scope, but may not be used ("it appears in neither of them").

$$\frac{M(x) = m \qquad m \neq 0 \qquad E(x) = \tau}{R, M, E \vdash x : \tau} \qquad \frac{R, (M; 1), (E; \tau_1) \Vdash t : \tau_2}{R, M, E \vdash \lambda t : \tau_1 \rightarrow \tau_2}$$

$$\frac{\begin{array}{cc} R_1 * R_2 = R & M_1 * M_2 = M \\ R_1, M_1, E \vdash v_1 : \tau_1 & R_2, M_2, E \vdash v_2 : \tau_2 \end{array}}{R, M, E \vdash {}_{\iota_1}(v_1, v_2)_{\iota_2} : {}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}} \qquad R, M, E \vdash ()_\iota : \mathsf{unit}_\iota \qquad \frac{R, M, (0 \uparrow E) \vdash v : \tau}{R, M, E \vdash \Lambda v : \forall \tau}$$

$$\frac{R, M, E \vdash v : [\theta/0]\tau}{R, M, E \vdash \mathsf{pack}\ v : \exists \tau} \qquad \frac{\widehat{R}, \widehat{M}, E \vdash v : \tau}{R, M, E \vdash\ !v :\ !\tau} \qquad \frac{R, M, E \vdash v : \tau \qquad nil \vdash c : \tau \leqslant \theta}{R, M, E \vdash c\ v : \theta}$$

$$\frac{R, M, E \vdash v : \tau_1 \qquad \tau_1 \equiv \tau_2}{R, M, E \vdash v : \tau_2}$$

Fig. 10. Typing rules for values: programmer-accessible constructs.

Multiplicities form a separation algebra. That is, they are equipped with a three-place conjunction relation, written $m_1 * m_2 = m$, which is defined as follows:

$$0 * 0 = 0 \qquad 1 * 0 = 1 \qquad 0 * 1 = 1 \qquad \infty * \infty = \infty$$

Furthermore, the function that maps a multiplicity m to its "core" $\widehat{m}$ is defined as follows:

$$\widehat{0} = 0 \qquad \widehat{1} = 0 \qquad \widehat{\infty} = \infty$$

These notations are extended pointwise to multiplicity environments, and are used in the typing rules that follow. When applied to a multiplicity environment M, the function $\widehat{\cdot}$ retains all duplicable variables and discards all affine variables.

We will see later in Section 10 that resources are also equipped with a conjunction relation $*$ and a "core" function $\widehat{\cdot}$. In the typing rules, $*$ and $\widehat{\cdot}$ are applied to both resources R and multiplicity environments M, thus emphasizing the fact that R and M are treated in the same manner.

7.1 Values

The typing rules for values appear in Figure 10. We review them in turn.

A variable x is deemed to have type τ if the multiplicity associated with x in the multiplicity environment M is non-zero (that is, it is either 1 or ∞) and the type associated with x in the type environment E is τ.

A function λt has type $\tau_1 \rightarrow \tau_2$ if its body has type τ_2 under extended multiplicity and type environments. The multiplicity environment is extended with the multiplicity 1, because a function argument is, by default, affine. The function is allowed to capture affine variables in its closure: there are no restrictions on M. This is sound because the function itself is considered affine: the type $\tau_1 \rightarrow \tau_2$ is not duplicable.

In order to type-check the components of a pair, the multiplicity environment M is split, while the type environment E is transmitted to the two premises. The layers ι_1 and ι_2 that decorate the pair are reflected in its type.

The layer annotation ι carried by a unit value is reflected in its type.

The rules that introduce a universal or existential quantifier are borrowed from System F. The introduction rule for the "!" type constructor ensures that only duplicable variables are accessed by transmitting $\widehat{M}$ to the premise in place of M. The resource R is treated in the same manner: the resource $\widehat{R}$ intuitively represents the "duplicable part" of R.

The last two rules in Figure 10 are type conversion rules. The subtyping rule allows moving from τ_1 to τ_2 via an explicit coercion c. (The subtyping judgement, $C \vdash c : \tau_1 \leqslant \tau_2$, is defined in Section 8.) The type equality rule allows implicitly moving from τ_1 to τ_2, provided these types are equal. This rule and its counterpart for terms are the only non-syntax-directed typing rules.

Remark 7.2 Our rules correspond to a purely affine variant of DILL. A purely linear variant would be slightly different. In the rule for variables, the resource R, as well as the multiplicity environment M, deprived of x, would be required to be duplicable. In the rule for the unit value and in the "!" introduction rule, R and M would be required to be duplicable. The property that R is duplicable can be written $R * R = R$ or $R = \widehat{R}$; see Section 10. ◇

7.2 *Terms*

The typing rules for terms appear in Figure 11. We review them in turn.

The first rule in Figure 11 permits the injection of values into terms. Its premise is a judgement about the value v, while its conclusion is a judgement about the term v.

The next two rules, for function application and for pairs, are borrowed from DILL (Barber, 1996). Let us comment on the latter one.

Pairs already appear in the syntax of values. One might think that it is not necessary to also include pairs in the syntax of terms. In fact, we do need to allow at least pairs of the particular form ${}_{\mathsf{Phy}}(t_1, v_2)_{\mathsf{Log}}$ in the syntax of terms. These pairs are useful, both in practice and in the subject reduction proof: their typing rule corresponds to the first-order frame rule of separation logic. In a pair ${}_{\mathsf{Phy}}(t_1, v_2)_{\mathsf{Log}}$, the component t_1 represents a computation that takes place at runtime, whereas the component v_2 is a logical value: it is erased at runtime. In other words, v_2 is a capability that is set aside during the evaluation of t_1. This capability is not made available to t_1: it is "framed out". If one specializes the typing rule to the case where v_2 is a variable x, omits the component R, and presents the rule in informal (nominal, DILL-style) notation, one obtains the following version of the rule:

$$\frac{\Gamma, \Delta \Vdash t : \tau_1}{\Gamma, (\Delta; x : \tau_2) \Vdash {}_{\mathsf{Phy}}(t, x)_{\mathsf{Log}} : \tau_1 * \tau_2}$$

The type (or the "capability") τ_2 appears in the left- and right-hand sides of the conclusion, but does not appear in the premise. This is very close to Charguéraud and Pottier's (2008) formulation of the frame rule. There the fact that the frame rule could be viewed as a pairing construct was made evident by the functional translation, whereas here it is evident in the syntax of the instrumented calculus.

$$\frac{R, M, E \vdash v : \tau}{R, M, E \Vdash v : \tau} \qquad \frac{\begin{array}{c} R_1 * R_2 = R \qquad M_1 * M_2 = M \\ R_1, M_1, E \vdash v : \tau_1 \to \tau_2 \qquad R_2, M_2, E \Vdash t : \tau_1 \end{array}}{R, M, E \Vdash v\ t : \tau_2}$$

$$\frac{\begin{array}{c} R_1 * R_2 = R \qquad M_1 * M_2 = M \\ R_1, M_1, E \Vdash t_1 : \tau_1 \qquad R_2, M_2, E \vdash v_2 : \tau_2 \end{array}}{R, M, E \Vdash {}_{\mathsf{Phy}}(t_1, v_2)_{\mathsf{Log}} : \tau_1 * \tau_2} \qquad \frac{\begin{array}{c} \textit{anti-frame disabled} \\ R, M, (0 \uparrow E) \Vdash t : \tau \end{array}}{R, M, E \Vdash \Lambda t : \forall \tau}$$

$$\frac{\begin{array}{c} R_1 * R_2 = R \qquad M_1 * M_2 = M \\ R_1, M_1, E \vdash v : \exists \tau_1 \qquad R_2, (M_2 ; 1), (0 \uparrow E ; \tau_1) \Vdash t : 0 \uparrow \tau_2 \end{array}}{R, M, E \Vdash \mathsf{unpack}\, v\, \mathsf{in}\, t : \tau_2}$$

$$\frac{\begin{array}{c} R_1 * R_2 = R \qquad M_1 * M_2 = M \\ R_1, M_1, E \vdash v : !\tau_1 \qquad R_2, (M_2 ; \infty), (E ; \tau_1) \Vdash t : \tau_2 \end{array}}{R, M, E \Vdash \mathsf{let!}\, v\, \mathsf{in}\, t : \tau_2}$$

$$\frac{\begin{array}{c} R_1 * R_2 = R \qquad M_1 * M_2 = M \\ R_1, M_1, E \vdash v : {}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2} \qquad R_2, (M_2 ; 1 ; 1), (E ; \tau_1 ; \tau_2) \Vdash t : \theta \end{array}}{R, M, E \Vdash \mathsf{letpair}_{\iota_1, \iota_2}\, v\, \mathsf{in}\, t : \theta} \qquad \frac{\begin{array}{c} R, M, E \vdash v : \tau_1 \\ \vdash p : \tau_1 \to \tau_2 \end{array}}{R, M, E \Vdash p\ v : \tau_2}$$

$$\frac{\begin{array}{c} \textit{anti-frame enabled} \qquad R_1, M, E \vdash v : \tau_1 \\ R_2, (nil ; 1), (nil ; \tau_1 \otimes \theta) \Vdash t : \tau_2 \circ \theta \qquad R_1 * R_2 = R \end{array}}{R, M, E \Vdash \mathsf{let}\, v\, \mathsf{in}\ \mathsf{hide}\, t : \tau_2} \qquad \frac{R, M, E \Vdash t : \tau \qquad nil \vdash c : \tau \leqslant \theta}{R, M, E \Vdash c\ t : \theta}$$

$$\frac{R, M, E \Vdash t : \tau_1 \qquad \tau_1 \equiv \tau_2}{R, M, E \Vdash t : \tau_2}$$

Fig. 11. Typing rules for terms.

The next rule concerns type abstractions of the form Λt. Recall that we already have a rule, presented in Figure 10, for abstracting a value v over a type variable. Here, however, we are looking at a more powerful rule, which abstracts an arbitrary term t over a type variable. The presence of this typing rule amounts to the absence of the value restriction (Wright, 1995). As explained earlier, we need the value restriction only when the anti-frame rule is enabled. In other words, we are able to accept type abstractions of the form Λt when the anti-frame rule is disabled.

The construct $\mathsf{unpack}\, v\, \mathsf{in}\, t$ is type-checked in a standard way. As in System F, the value v must have an existential type $\exists \tau$. One type variable, which stands for an unknown type, and one term variable, which is assumed to have type τ, are then bound in t. As in DILL, the multiplicity environment M is split between v and t.

The construct $\mathsf{let!}\, v\, \mathsf{in}\, t$ is type-checked in a standard way. As in DILL, the value v must have a duplicable type $!\tau_1$. One term variable is then bound in t, with multiplicity ∞ and type τ_1. This is the only way for a variable to be bound with multiplicity ∞. As usual, the multiplicity environment M is split.

The typing rule for $\mathsf{letpair}_{\iota_1, \iota_2}\, v\, \mathsf{in}\, t$ is mostly standard. As in DILL, two term variables are bound in t, while M is split. A non-standard aspect resides in the layer annotations ι_1 and ι_2, which are explicitly carried by the letpair construct and are reflected in the type of the pair v.

The typing rule for the application of a primitive operation p to a value v relies on the auxiliary judgement $\vdash p : \tau_1 \to \tau_2$, which fixes the types of the primitive operations. This judgement is defined further in Section 7.3.

Next comes the anti-frame rule. In the instrumented calculus, the use of this rule is explicitly signaled by the construct let v in hide t. The purpose of this rule is to allow the term t to hide a certain capability, of type θ, from the outside world. Thus, the term t internally has type $\tau_2 \circ \theta$, as indicated by the third premise, but it is allowed to hide the existence of θ and to pretend that it has type τ_2, as indicated by the conclusion. Recall that $\tau_2 \circ \theta$ stands for $(\tau_2 \otimes \theta) * \theta$ (Section 6). In this conjunction, the left-hand conjunct $\tau_2 \otimes \theta$ indicates that θ holds at every interaction between the term t and its environment: Indeed, the type $\tau_2 \otimes \theta$ can be thought of as a copy of the type τ_2, where every function type requires θ as an extra argument and produces θ as an extra result. The right-hand conjunct θ indicates that the term t is responsible for initially establishing the invariant θ: the evaluation of t must produce a pair of some value and a capability of type θ. This capability, once hidden, remains inaccessible forever. More precisely, it becomes temporarily accessible whenever control enters the scope of the hide construct, and becomes inaccessible again upon exit. There is no way of permanently recovering this capability.

We have adopted the convention that, within t, exactly one term variable, namely, the variable 0, is in scope. At runtime, this variable is bound to the value v. Any term variables that were previously in scope are inaccessible within t. Thus, the term t is type-checked under multiplicity and type environments of length 1. If the value v has type τ_1, then the term t has access to it, via the variable 0, at type $\tau_1 \otimes \theta$. Again, this means that the invariant θ must hold at every interaction between the term t and its environment. This reflects the informal idea that if the term t wishes to invoke a "callback" provided by its environment, then it must restore its hidden invariant θ before doing so, and it can assume that θ still holds when the evaluation of the callback returns.

As usual, the resource R is split between v and t. The multiplicity environment M is not split: all of it is made available to v, since, as per our convention, any term variables that were previously in scope are no longer in scope inside t.

The last two rules in Figure 11 are the rules for subtyping and type equality. They are analogous to the last two rules of Figure 10.

7.3 Primitive operations

The typing rules for primitive operations appear in Figures 12 and 13. Every primitive operation p is considered of arity one, so the typing judgement takes the form $\vdash p : \tau_1 \to \tau_2$. Among the primitive operations, we distinguish two groups.

The operations listed in Figure 12 allow allocating, reading, and writing references. They have an effect at runtime: that is, they are not erased.

The operations listed in Figure 13 are operations on regions. They do not have any effect at runtime: they are erased. However, they do have a side effect, in a

$$\vdash \mathsf{new} : \tau \rightarrow \exists([0] * \{\mathsf{S}\,0 : \mathsf{ref}\ (0\uparrow\tau)\}) \qquad \vdash \mathsf{read} : [\sigma] * \{\mathsf{S}\,\sigma : \mathsf{ref}\ (!\,\tau)\} \rightarrow (!\,\tau) * \{\mathsf{S}\,\sigma : \mathsf{ref}\ (!\,\tau)\}$$

$$\vdash \mathsf{write} : ([\sigma] \times \tau_2) * \{\mathsf{S}\,\sigma : \mathsf{ref}\ \tau_1\} \rightarrow \mathsf{unit} * \{\mathsf{S}\,\sigma : \mathsf{ref}\ \tau_2\}$$

Fig. 12. Typing rules: primitive operations: references.

$$\frac{\vdash \pi : \mathscr{T}}{\vdash \mathsf{focus}\,\pi : \mathscr{T}[\tau] \rightarrow \exists(((0\uparrow\mathscr{T})[[0]]) * \{\mathsf{S}\,0 : 0\uparrow\tau\})}$$

$$\frac{\vdash \pi : \mathscr{T}}{\vdash \mathsf{defocus\text{-}dup}\,\pi : (\mathscr{T}[[\sigma]]) * \{\kappa\,\sigma : !\,\tau\} \rightarrow (\mathscr{T}[!\,\tau]) * \{\kappa\,\sigma : !\,\tau\}}$$

$$\vdash \mathsf{newgroup} : \mathsf{unit} \rightarrow \exists(\mathsf{unit} * \{\mathsf{G}\,0 : \tau\}) \qquad \vdash \mathsf{adopt} : \tau * \{\mathsf{G}\,\sigma : \tau\} \rightarrow [\sigma] * \{\mathsf{G}\,\sigma : \tau\}$$

$$\vdash \mathsf{focusgroup} : [\rho] * \{\mathsf{G}\,\rho : \tau\} \rightarrow \exists([0] * (\{\mathsf{S}\,0 : 0\uparrow\tau\} * \{0\uparrow\rho : 0\uparrow\tau \setminus 0\}))$$

Fig. 13. Typing rules: primitive operations: regions.

certain sense: each of them either allocates a fresh region or enlarges the population of an existing region. Because they are erased, one might wonder whether these operations could be viewed as coercions. We discuss this point in Remark 8.3.

7.3.1 Primitive operations on references

The primitive operations new, read, and write (Figure 12) allow creating, reading, and writing references. Their typing rules are in the style of Charguéraud and Pottier (2008), which itself follows earlier works (Smith *et al.*, 2000). In these rules, one distinguishes between the memory location, which receives a duplicable type of the form $[\sigma]$, and an affine capability for the region σ. The operations read and write require the capability, and return it, since it would otherwise be lost. However, they need not return the reference itself: the reference is duplicable, and it is up to the caller to keep a copy of it if needed. Thus, the runtime behavior of read and write coincides with the behavior of the operations ! and := of ML.

In informal nominal notation, the type of new is $\tau \rightarrow \exists\sigma.[\sigma] * \{\mathsf{S}\,\sigma : \mathsf{ref}\ \tau\}$, where σ is a type variable and is fresh for τ. This operation expects a value of type τ. It produces a fresh region, represented by the (existentially bound) type variable σ (or 0, in the de Bruijn notation). It also produces a fresh memory location, whose type is $[\sigma]$, meaning that this memory location is an inhabitant of the new region. Finally, it produces a capability $\{\mathsf{S}\,\sigma : \mathsf{ref}\ \tau\}$. This capability indicates that σ is a singleton region and its inhabitant has type ref τ. This capability represents the ownership of the region and the reference that it contains.

read expects a pair of an inhabitant of some region $[\sigma]$ and a capability for this region, $\{\mathsf{S}\,\sigma : \mathsf{ref}\ (!\,\tau)\}$. This capability indicates that the inhabitant of this region is a reference to a value of type $!\,\tau$. Reading is restricted to duplicable types $!\,\tau$. The fact that the type $!\,\tau$ occurs twice in the return type of read clearly shows that there is duplication and that this restriction is necessary. read returns a pair of this value and the unmodified capability. The reference itself is not returned.

$$\vdash \textsf{path-root} : [] \qquad \frac{\vdash \pi : \mathscr{T}}{\vdash \textsf{path-left}\,\iota_1\,\iota_2\,\pi : {}_{\iota_1}(\mathscr{T} \times \tau_2)_{\iota_2}} \qquad \frac{\vdash \pi : \mathscr{T}}{\vdash \textsf{path-right}\,\iota_1\,\iota_2\,\pi : {}_{\iota_1}(\tau_1 \times \mathscr{T})_{\iota_2}}$$

$$\frac{\vdash \pi : \mathscr{T}}{\vdash \textsf{path-ref}\,\pi : \textsf{ref}\,\mathscr{T}} \qquad \frac{\vdash \pi : \mathscr{T}}{\vdash \textsf{path-singleton}\,\pi : \{\textsf{S}\,\sigma : \mathscr{T}\}}$$

Fig. 14. Typing rules: focus paths.

write expects three arguments: a reference, a value to be written into the reference, and a capability for the reference. It produces a pair of the unit value and a modified capability. write allows strong updates: the previous type τ_1 and the new type τ_2 may be distinct.

7.3.2 Primitive operations on regions

The primitive operation focus π (Figure 13) and the subtyping axiom defocus π (to be presented in Section 8.6) are inverses of each other. In short, they allow converting back and forth between structural and nominal views of data. Say some value v has type τ, which could be a function type, a pair type, etc. This is a structural view of v. If we create a fresh singleton region σ, which v inhabits, as well as a capability $\{\textsf{S}\,\sigma : \tau\}$, then we can also say that v has type $[\sigma]$. This is a nominal view of v: the type variable σ acts as a name for the value v. This view may be useful, in particular, because $[\sigma]$ is a duplicable type, whereas τ might not be duplicable. We can create several copies of v at type $[\sigma]$. In order to actually use one of these copies, we need the capability $\{\textsf{S}\,\sigma : \tau\}$, of which there exists just one copy.

Focusing is the act of switching from a structural view to a nominal view. It produces a fresh region as well as a capability for this region. De-focusing is the converse operation. Both operations are used in the encoding of weak references with affine content (Section 3).

We have just suggested that focusing allows creating a name σ for a value v. However, if v is a composite value, one may wish to name one of its components. For instance, if v is a pair, one may wish to name its first component; if v is a reference, one may wish to name its content; and so on. One may wish to go down by several levels at once: for instance, if v is a capability for a singleton region whose inhabitant is a reference, one may wish to name the content of this reference. In general, we allow focusing on a component of v that is determined by a *path* π. The syntax of paths was given in Figure 5. The special case where one creates a name for the entire value v corresponds to the empty path path-root.

The judgement $\vdash \pi : \mathscr{T}$ (Figure 14) tells where a path π leads. $\mathscr{T}$ is a type context, that is, a type with one hole. The idea is that if the value v has type $\mathscr{T}[\tau]$, then starting at the root of v and following the path π leads us down to a component of type τ.

In the simplest case, π is the empty path path-root, and the type context $\mathscr{T}$ is the empty context []. Thus, the primitive operation focus path-root accepts a value v of type τ, and returns the same value, now viewed at type $\exists([0] * \{\textsf{S}\,0 : 0{\uparrow}\tau\})$. That is, it produces a fresh region, represented by the (existentially bound) type variable 0, as well as a pair of the value v, now viewed as an inhabitant of the new region (as indicated by the type [0]), and of a capability over the new region, $\{\mathsf{S}\,0 : 0{\uparrow}\tau\}$. This operation is of particular interest when τ is an affine type: then, the rule states that an affine value can be split into a pair of a duplicable value and an affine capability. This allows an affine value to be copied. Because the capability remains affine, only one copy of the value can be used at a time, so the system remains sound.

In more elaborate cases, a non-trivial path π is used. Then the path π serves to select a sub-value v' of the value v to which the primitive operation focus π is applied. If the value v has type $\mathscr{T}[\tau]$, where the type context $\mathscr{T}$ is dictated by the path π (Figure 14), then the value v' has type τ. As before, a fresh singleton region is created, which v' inhabits. The operation returns the value v, at a new type, where [0] replaces τ in the hole of the type context $\mathscr{T}$.

One must not assign a name to something that does not exist, and one must not assign one single name to multiple different things. For these reasons, a path π must exactly lead to one component, as opposed to zero or more than one component. As a result, there is no path for descending into a group region because a group region may have zero, one, or more inhabitants. We do offer an operation (namely, focusgroup, described further on) for focusing on an inhabitant of a group region, but it is not a special case of focus π.

The inverse of focus π is defocus π (Section 8.6). We make defocus π a coercion, as opposed to a primitive operation like focus π, because it does not have any side effect. This issue is discussed in greater depth in Remark 8.3.

The primitive operation defocus-dup π (Figure 13) states that if we hold the capability over some region σ and if this capability indicates that the inhabitants of this region have type $!\tau$, then we may convert a value of type $[\sigma]$ – an inhabitant of the region – to the type $!\tau$. This is a form of defocusing. It is restricted to duplicable types precisely because τ is duplicated in the right-hand side. It was known as SNG-EXTRACT in Charguéraud and Pottier's paper (2008). Ideally, it should be a coercion, like defocus π. Unfortunately, making it a coercion breaks our proof that value reduction terminates (see Remark 13.5).

The primitive operation newgroup creates a fresh group region and produces a capability $\{\mathsf{G}\,0 : \tau\}$ for this new region. The new region initially has no inhabitants, so the type τ of the region's inhabitants can be freely chosen. In particular, τ may refer to the type variable 0. That is, the description of the region's inhabitants may refer to the region itself. This allows creating and working with graph-like structures in the heap.

The primitive operation adopt adopts a value of type τ into an existing group region σ whose inhabitants have type τ. The value becomes a new inhabitant of the region, and is thereafter viewed at type $[\sigma]$. The operation requires a capability for the group region σ and returns it. This operation is found in Charguéraud and Pottier's work (2008), and is originally due to Fähndrich and DeLine (2002).

The primitive operation focusgroup isolates an inhabitant of an existing group region ρ into a fresh singleton region, represented by the type variable 0. The inhabitant, initially of type $[\rho]$, is returned at type [0], together with a capability

$\{\mathsf{S}\,0 : 0{\uparrow}\tau\}$ for the new singleton region. The capability $\{\mathsf{G}\,\rho : \tau\}$ over the group region ρ is transformed into a punched capability $\{0{\uparrow}\rho : 0{\uparrow}\tau \setminus 0\}$, which indicates that the group region ρ is disabled until the inhabitant is returned via the coercion $\mathsf{defocus\text{-}group}$ (Section 8.6). Again, this operation is found in Charguéraud and Pottier's work (2008) and is originally due to Fähndrich and DeLine (2002).

It is worth noting that, after $\mathsf{focusgroup}$ is applied to a value v, this value inhabits both the group region ρ and the fresh singleton region 0. Because $[\rho]$ is a duplicable type, the population of a region can only grow with time, and we cannot revoke the fact that v inhabits ρ. Regions can overlap: here, ρ and 0 have a common inhabitant. Adoption is another operation that introduces overlap between regions.

8 Subtyping

Subtyping is a relation between types. Coercions c serve as witnesses for subtyping assertions. Thus, subtyping judgements take the form $\vdash c : \tau_1 \leqslant \tau_2$. At the level of the instrumented calculus, this means that if v is a value of type τ_1, then the coercion application $c\ v$ is a value of type τ_2. According to the semantics of the instrumented calculus (Section 13), the value $c\ v$ may take one or more reduction steps until it reaches a canonical form. At the level of the raw calculus, subtyping has no computational content. The erasure of $c\ v$ is the erasure of v. A raw value of type τ_1 is a raw value of type τ_2.

In general, coercions have free variables, which denote coercions. In this paper, this is due to the presence of recursive coercions of the form μc. This construct binds a coercion variable within c. Certain type systems give coercions first-class status and allow abstracting over coercions: this would be another reason why coercions have free variables. Thus, in general, the subtyping judgement takes the form $C \vdash c : \tau_1 \leqslant \tau_2$, where a *coercion environment* C maps coercion variables to *coercion types*. A coercion type is of the form $\tau_1 \leqslant \tau_2$. Variables are represented as de Bruijn indices and environments are represented as lists.

The subtyping judgement is inductively defined in a style pioneered by Brandt and Henglein (1998). Because the type system is quite rich, there are many subtyping rules. In the following, we divide these rules in several groups: reflexivity and transitivity (Section 8.1); congruence (Section 8.2); quantifier introduction and elimination (Section 8.3); quantifier movement (Section 8.4); affinity (Section 8.5); regions (Section 8.6); movement of stars (Section 8.7); and recursive coercions (Section 8.8).

Remark 8.1 Because we do not have a semantic model of subtyping, there is no sense in which our inductive definition of subtyping can be said to be complete. In fact, although we have tried to give a wide array of subtyping rules, we have possibly omitted a few valid and useful rules. $\diamond$

Remark 8.2 Extending the system with new rules is usually not difficult, but can be cumbersome. In short, the recipe involves: introducing a new coercion form; introducing a new subtyping rule; introducing one or more new reduction rules in the instrumented semantics; updating the proofs that value reduction terminates

(Lemma 13.4), the raw and instrumented semantics agree (Lemma 15.18), and value reduction enjoys subject reduction and progress (Lemmas 15.9 and 15.10). This administrative burden, as well as the above-mentioned lack of a completeness guarantee, are shortcomings of the syntactic approach to type soundness. ◇

Remark 8.3 (Conversion mechanisms) We have three mechanisms that convert a value from type τ_1 to type τ_2 without performing any computation at runtime. The most transparent one is type equality. It is applicable when $\tau_1 \equiv \tau_2$ holds, and requires no computation at all, not even in the instrumented semantics. The second mechanism is subtyping. It is applicable when $\tau_1 \leqslant \tau_2$ holds, and requires computation at the level of the instrumented calculus, but not at the level of the raw calculus. The last mechanism is the application of a primitive operation that exists in the instrumented calculus but is erased in the raw calculus: an example is newgroup, which creates a fresh group region (Section 7.3.2). Similar to subtyping, this mechanism requires computation at the level of the instrumented calculus, but not at the level of the raw calculus.

The distinction between the first and second mechanisms is justified by the fact that subtyping involves computation at the level of the instrumented calculus, while type equality does not. We have noted earlier (Section 6.2) that $\tau_1 \leqslant \tau_2 \wedge \tau_2 \leqslant \tau_1$ does not imply $\tau_1 \equiv \tau_2$. Certain laws, such as the commutativity of $*$, cannot be made part of type equality and must therefore be subtyping axioms. This is in a sense unfortunate but seems inherent in our interpretation of $*$ as a pair.

The distinction between the second and third mechanisms is justified by the fact that a coercion has no side effect whatsoever, while a primitive operation can "have a conceptual side effect". For instance, newgroup creates a fresh region; adopt adds a new inhabitant to an existing region. Of course, these side effects are not real: at the level of the raw calculus, newgroup and adopt are erased. Nevertheless, whether coercions are or are not allowed to have such side effects has an impact on the statement of the property that "the reduction of values preserves types" (Lemma 15.9). Assume that v_1 reduces to v_2 and v_1 is well-typed with respect to a certain resource R. If "a coercion cannot have a side effect", then we can state that v_2 must be well-typed with respect to R. If, on the other hand, "a coercion can have a side effect", then we must be more flexible and state that v_2 must be well-typed with respect to some resource R' such that $R \blacktriangleleft R'$ holds. (The ordering $\blacktriangleleft$, which describes the "active" evolution of resources during execution, is introduced in Section 10.) Each of these interpretations makes sense. Unfortunately, because $R \blacktriangleleft R'$ does not imply $\widehat{R} \blacktriangleleft \widehat{R'}$, only the first interpretation validates the property that "!" is covariant. If, for instance, one could apply newgroup under "!", then one would obtain a duplicable capability for the new region, which would be unsound. Thus, although we could adopt the point of view that "a coercion can have a side effect", we would have to distinguish anyway between the coercions that do not have a side effect and can be applied under "!", and those that do have a side effect and cannot be applied under "!". Here we adopt a slightly coarser approach, and distinguish between coercions, which do not have a side effect and can be applied

$$\frac{\tau_1 \equiv \tau_2}{C \vdash \mathsf{id} : \tau_1 \leqslant \tau_2} \qquad \frac{C \vdash c_1 : \tau_1 \leqslant \tau_2 \qquad C \vdash c_2 : \tau_2 \leqslant \tau_3}{C \vdash c_1 ; c_2 : \tau_1 \leqslant \tau_3}$$

Fig. 15. Subtyping: reflexivity and transitivity.

$$\frac{C \vdash c_1 : \theta_1 \leqslant \tau_1 \qquad C \vdash c_2 : \tau_2 \leqslant \theta_2}{C \vdash c_1 \to c_2 : \tau_1 \to \tau_2 \leqslant \theta_1 \to \theta_2} \qquad \frac{C \vdash c_1 : \tau_1 \leqslant \theta_1 \qquad C \vdash c_2 : \tau_2 \leqslant \theta_2}{C \vdash {}_{\iota_1}(c_1 \times c_2)_{\iota_2} : {}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2} \leqslant {}_{\iota_1}(\theta_1 \times \theta_2)_{\iota_2}}$$

$$\frac{0{\uparrow}C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash \forall c : \forall \tau_1 \leqslant \forall \tau_2} \qquad \frac{0{\uparrow}C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash \exists c : \exists \tau_1 \leqslant \exists \tau_2} \qquad \frac{C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash\, !c : !\tau_1 \leqslant !\tau_2} \qquad \frac{C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash \mathsf{ref}\ c : \mathsf{ref}\ \tau_1 \leqslant \mathsf{ref}\ \tau_2}$$

$$\frac{C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash \{c\} : \{\kappa\,\sigma : \tau_1\} \leqslant \{\kappa\,\sigma : \tau_2\}} \qquad \frac{C \vdash c : \tau_1 \leqslant \tau_2}{C \vdash \{c\backslash\} : \{\rho : \tau_1 \setminus \sigma\} \leqslant \{\rho : \tau_2 \setminus \sigma\}}$$

Fig. 16. Subtyping: congruence.

under an arbitrary context, and primitive operations, which can have a side effect, but cannot be applied under any context. ◇

8.1 *Reflexivity and transitivity*

It is natural for subtyping to be reflexive and transitive. The simplest way to enforce these properties is to build them into the inductive definition of subtyping (Figure 15). The statement of reflexivity is standard. Of course, it relies on our notion of type equality (Section 6), as opposed to syntactic equality. That is, $\tau_1 \equiv \tau_2$ implies $\tau_1 \leqslant \tau_2$, and the witness of this fact is the coercion id. The statement of transitivity is standard. If the coercions c_1 and c_2 respectively convert τ_1 to τ_2 and τ_2 to τ_3, then the coercion $c_1 ; c_2$, which can be understood as the sequential composition of c_1 and c_2, converts τ_1 to τ_3.

8.2 *Congruence*

Subtyping is a congruence. This is made precise by the rules in Figure 16. As usual, the function type constructor is contravariant in its domain and covariant in its codomain. The pair type constructor is covariant in both of its arguments. The types $\forall\tau$, $\exists\tau$, and $!\,\tau$ are covariant with respect to τ. The type constructor ref τ is *covariant* with respect to τ (Charguéraud & Pottier, 2008). As demonstrated by the mechanized proof, this is sound. In order to understand intuitively why this is so, recall that ref describes *strong* references: writing to a reference changes its type. As a result, ref τ_1 must be understood as the type of a reference that *currently* holds a value of type τ_1. If τ_1 is a subtype of τ_2, then a value of type τ_1 is also a value of type τ_2, so a reference that currently holds a value of type τ_1 is also a reference that currently holds a value of type τ_2.

Remark 8.4 In contrast, in ML, where references are weak, ref τ_1 must be understood as the type of a reference that is forever constrained to hold a value of type τ_1.

$$C \vdash \forall\mathsf{I} : \tau \leqslant \forall(0{\uparrow}\tau) \qquad C \vdash \forall\mathsf{E} : \forall\tau \leqslant [\theta/0]\tau$$

$$C \vdash \exists\mathsf{I} : [\theta/0]\tau \leqslant \exists\tau \qquad C \vdash \exists\mathsf{E} : \exists(0{\uparrow}\tau) \leqslant \tau$$

Fig. 17. Subtyping: quantifier introduction and elimination.

Weak references can be encoded in terms of strong references and the anti-frame rule (Pottier, 2008). A weak reference of type τ is encoded as a pair of accessor methods, of type $(\mathsf{unit} \rightarrow \tau) \times (\tau \rightarrow \mathsf{unit})$. The fact that τ appears in both covariant and contravariant positions in this encoding explains why, under a weak-reference interpretation, $\mathsf{ref}\ \tau$ must be considered invariant with respect to τ. ⋄

The types $\{\kappa\,\rho : \tau\}$ and $\{\rho : \tau \setminus \sigma\}$, which respectively represent full and punched capabilities over regions, are covariant with respect to τ (Charguéraud & Pottier, 2008). This is consistent with a view of regions as sets of values: If τ_1 is a subtype of τ_2, then a region that currently contains values of type τ_1 is also a region that currently contains values of type τ_2.

Remark 8.5 The type constructor $\mu\cdot$ does not come with a congruence rule of its own. A plausible candidate for such a rule would be Amadio and Cardelli's subtyping rule (1993). However, this rule is admissible (Section 8.8). ⋄

Remark 8.6 The type constructor $\cdot \otimes \cdot$ does not come with a congruence rule either. If such a rule existed, what would it be? The meaning of tensor, as encoded in its equational theory (Figure 9 in Section 6), suggests that $\tau \otimes \theta$ could be considered covariant with respect to τ and invariant with respect to θ.

The revelation lemma (Lemma 12.1) shows that, in a limited way, tensor is indeed covariant in its first argument, even in the absence of an explicit congruence rule. This lemma states that if there exists a closed coercion that establishes $\tau_1 \leqslant \tau_2$, then there exists a closed coercion that establishes $\tau_1 \otimes \theta \leqslant \tau_2 \otimes \theta$. This is good enough for our purposes. ⋄

8.3 Quantifier introduction and elimination

Following Mitchell (1988), we note that certain (albeit not all) quantifier introduction and elimination rules can be presented as subtyping axioms. A subtyping axiom is visually lighter than a typing rule. Furthermore, it is more expressive: indeed, because subtyping is a congruence, a subtyping axiom can be applied under a context. Four axioms for introducing and eliminating universal and existential quantifiers appear in Figure 17.

The statement of the universal introduction axiom $\forall\mathsf{I}$, in nominal notation, would be $\tau \leqslant \forall\alpha.\tau$, where the type variable α is fresh for τ. This is a degenerate form: it allows introducing only an unused quantifier. Thus, it does not suppress the need for Λ-abstractions in the syntax of terms. The universal elimination axiom $\forall\mathsf{E}$, on the other hand, is general: it eliminates the need for type applications in the syntax of values and terms. Its statement in nominal notation would be $\forall\alpha.\tau \leqslant [\theta/\alpha]\tau$. Together, $\forall\mathsf{I}$ and $\forall\mathsf{E}$ allow deriving Mitchell's axiom (1988) for comparing universal

$$\frac{\textit{anti-frame disabled}}{C \vdash \mathsf{distrib} : \forall(\tau_1 \to \tau_2) \leqslant (\forall\tau_1) \to (\forall\tau_2)} \qquad C \vdash \exists\mathsf{LI} : \forall(\tau_1 \to (0{\uparrow}\tau_2)) \leqslant (\exists\tau_1) \to \tau_2$$

$$C \vdash \forall\text{-}\mathsf{pair} : \forall({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}) \leqslant {}_{\iota_1}((\forall\tau_1) \times (\forall\tau_2))_{\iota_2} \qquad C \vdash \forall\text{-}\mathsf{bang} : \forall(!\,\tau) \leqslant !\,(\forall\tau)$$

$$C \vdash \forall\text{-}\mathsf{ref} : \forall(\mathsf{ref}\ \tau) \leqslant \mathsf{ref}\ (\forall\tau) \qquad C \vdash \forall\text{-}\mathsf{regioncap} : \forall\{\kappa\, 0{\uparrow}\sigma : \tau\} \leqslant \{\kappa\, \sigma : \forall\tau\}$$

$$C \vdash \forall\text{-}\mathsf{regioncappunched} : \forall\{0{\uparrow}\rho : \tau \setminus 0{\uparrow}\sigma\} \leqslant \{\rho : \forall\tau \setminus \sigma\}$$

$$C \vdash \mathsf{pair\text{-}exists\text{-}left} : {}_{\iota_1}((\exists\tau_1) \times \tau_2)_{\iota_2} \leqslant \exists({}_{\iota_1}(\tau_1 \times (0{\uparrow}\tau_2))_{\iota_2})$$

$$C \vdash \mathsf{pair\text{-}exists\text{-}right} : {}_{\iota_1}(\tau_1 \times (\exists\tau_2))_{\iota_2} \leqslant \exists({}_{\iota_1}((0{\uparrow}\tau_1) \times \tau_2)_{\iota_2}) \qquad C \vdash \mathsf{bang\text{-}exists} : !\,(\exists\tau) \leqslant \exists(!\,\tau)$$

$$C \vdash \mathsf{ref\text{-}exists} : \mathsf{ref}\ (\exists\tau) \leqslant \exists(\mathsf{ref}\ \tau) \qquad C \vdash \mathsf{cap\text{-}exists} : \{\mathsf{S}\,\sigma : \exists\tau\} \leqslant \exists\{\mathsf{S}\,0{\uparrow}\sigma : \tau\}$$

Fig. 18. Subtyping: quantifier movement.

types, which takes the form $\forall\vec{\alpha}.\tau \leqslant \forall\vec{\beta}.[\vec{\theta}/\vec{\alpha}]\tau$, where the type variables $\vec{\beta}$ are fresh for $\forall\vec{\alpha}.\tau$. (We have not formally proved this fact.) Thus, the seemingly uninteresting axiom ∀I is actually useful.

The introduction and elimination axioms for existential quantifiers are dual. Here the axiom ∃I is general, and suppresses the need for a "pack" construct in the syntax of terms. (We do keep a "pack" construct in the syntax of values: it plays a role in the operational semantics of ∃I.) The axiom ∃E is degenerative: it allows eliminating of only an unused quantifier.

8.4 Quantifier movement

While certain type constructors, such as the function and pair type constructors, describe structure that exists at runtime, the universal and existential type constructors are of a pure logical nature. As a result, it often makes sense for them to commute with other type constructors. In Figure 18, we present a set of such commutation rules. This set seems reasonably complete in an informal sense, although there can be no formal statement of this fact. None of these rules is particularly interesting: we include these only because we wish to ensure that "everything works". The reader is encouraged to jump ahead to the next section (Section 8.5).

The first group of rules indicate how the universal quantifier can be pushed down into some other type constructor.

The coercion distrib pushes a universal quantifier into (both sides of) a function type. This axiom appears in Mitchell's work (1988). There is a subtle point about this axiom: it conflicts with the value restriction. (To see why this is so, consider the reduction rule for this axiom, which appears in Figure A 4. The right-hand side of the rule uses a Λ-abstraction whose body is not a value.) Furthermore, as explained earlier (Section 2), enabling the anti-frame rule requires imposing the value restriction. Thus, the use of the coercion distrib is permitted only in the variant of our system where the anti-frame rule is disabled.

$$C \vdash \mathsf{dereliction} : !\,\tau \leqslant \tau \qquad C \vdash \mathsf{bang\text{-}idempotent} : !\,\tau \leqslant !\,(!\,\tau)$$

$$C \vdash \mathsf{pair\text{-}bang} : {}_{\iota_1}((!\,\tau_1) \times (!\,\tau_2))_{\iota_2} \leqslant !\,({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2})$$

$$C \vdash \mathsf{bang\text{-}pair} : !\,({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}) \leqslant !\,({}_{\iota_1}((!\,\tau_1) \times (!\,\tau_2))_{\iota_2}) \qquad C \vdash \mathsf{unit\text{-}bang} : \mathsf{unit}_\iota \leqslant !\,(\mathsf{unit}_\iota)$$

$$C \vdash \mathsf{bang\text{-}ref} : !\,(\mathsf{ref}\ \tau_1) \leqslant \tau_2 \qquad C \vdash \mathsf{bang\text{-}regioncap} : !\,\{\kappa\,\sigma : \tau_1\} \leqslant \tau_2$$

$$C \vdash \mathsf{bang\text{-}regioncappunched} : !\,\{\rho : \tau_1 \setminus \sigma\} \leqslant \tau_2 \qquad C \vdash \mathsf{at\text{-}bang} : [\sigma] \leqslant !\,[\sigma]$$

Fig. 19. Subtyping: affinity.

The coercion ∃LI also pushes a universal quantifier into a function type. In informal nominal syntax, its statement would be $\forall\alpha.(\tau_1 \to \tau_2) \leqslant (\exists\alpha.\tau_1) \to \tau_2$, where α is fresh for τ_2. This axiom is applicable only when the type variable α bound by the universal quantifier does not occur in the codomain; then, the quantifier can be pushed into the domain, where it becomes an existential quantifier. This rule is analogous to a left-introduction rule for the existential quantifier.

The coercions ∀-pair, ∀-bang, ∀-ref, ∀-regioncap, and ∀-regioncappunched push a universal quantifier into various type constructors.

A universal quantifier can be pushed down into another universal quantifier. That is, in informal nominal notation, $\forall\alpha.\forall\beta.\tau$ is a subtype of $\forall\beta.\forall\alpha.\tau$. Similarly, a universal quantifier can be hoisted out of an existential quantifier: that is, $\exists\alpha.\forall\beta.\tau$ is a subtype of $\forall\beta.\exists\alpha.\tau$. These properties are the consequences of the axioms that we have given. In fact, one can prove that $\mathscr{T}[\forall\alpha.\tau]$ is a subtype of $\forall\alpha.\mathscr{T}[\tau]$, where $\mathscr{T}$ is a covariant (possibly multi-hole) type context and α is fresh for $\mathscr{T}$. The proof is purely based on axioms ∀I and ∀E. We omit the details.

In summary, we have listed several axioms and derived rules that tell how a universal quantifier can be pushed into and hoisted out of many type constructors. These quantifier movement rules can be useful in practice. Furthermore, they help emphasize the meaning of certain type constructors: for instance, the fact that the types $\forall(\mathsf{ref}\ \tau)$ and $\mathsf{ref}\ (\forall\tau)$ are inter-convertible indicates that we are considering strong references. Under a weak-reference interpretation, they would be unrelated.

A dual program can be carried out for the existential quantifier. We give a number of axioms that allow hoisting an existential quantifier out of some other type constructor (Figure 18). These axioms indicate that an existential quantifier can be hoisted out of either side of a pair, out of a "!" or "ref" constructor, and out of a capability for a singleton region. It cannot be hoisted out of a group region: in general, a group region has multiple inhabitants, and each has its own witness for the existential type, so it is impossible to exhibit one single witness.

A consequence of axioms ∃I and ∃E is that an existential quantifier can be pushed into any covariant (possibly multi-hole) context. That is, in nominal notation, $\exists\alpha.\mathscr{T}[\tau]$ is a subtype of $\mathscr{T}[\exists\alpha.T]$, where $\mathscr{T}$ is a covariant context and α is fresh for $\mathscr{T}$.

$$\frac{\vdash \pi : \mathscr{T}}{C \vdash \mathsf{defocus}\,\pi : (\mathscr{T}[[\sigma]]) * \{\mathsf{S}\,\sigma : \tau\} \leqslant \mathscr{T}[\tau]}$$

$$C \vdash \mathsf{defocus\text{-}group} : \{\mathsf{S}\,\sigma : \tau\} * \{\rho : \tau \setminus \sigma\} \leqslant \{\mathsf{G}\,\rho : \tau\}$$

$$C \vdash \mathsf{singleton\text{-}to\text{-}group} : \{\mathsf{S}\,\sigma : \tau\} \leqslant \{\mathsf{G}\,\sigma : \tau\}$$

Fig. 20. Subtyping: regions.

8.5 Affinity

Like the universal and existential quantifiers, the "!" type constructor is of pure logical nature. As a result, it interacts with itself and with other type constructors in interesting ways. The corresponding axioms appear in Figure 19.

As usual (Barber, 1996), "!" can disappear spontaneously, and is idempotent.

The axiom unit-bang indicates that the type unit_ι is duplicable.

The axioms pair-bang and bang-pair together with dereliction indicate that the three types ${}_{\iota_1}((!\,\tau_1) \times (!\,\tau_2))_{\iota_2}$ and $!\,({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2})$ and $!\,({}_{\iota_1}((!\,\tau_1) \times (!\,\tau_2))_{\iota_2})$ are interconvertible. That is, a pair of duplicable things, a duplicable pair of things, and a duplicable pair of duplicable things are the same. In particular, a pair whose components are duplicable is itself duplicable. This reflects the fact that even though a pair is a heap-allocated object, we do not consider inherently affine. Hence, we do not keep track of who "owns" it. This is true, in general, of all immutable objects. This design choice simplifies the language and encourages a purely functional programming style. It does imply that we need a garbage collector: duplicable objects cannot be explicitly de-allocated.

The axiom bang-ref indicates that the type $!\,(\mathsf{ref}\ \tau)$ is a subtype of every type, which means that it is empty. In other words, there is no such thing as a duplicable reference: references are inherently affine. Analogously, the capabilities that govern regions are inherently affine. We do not expect the axioms bang-ref and bang-regioncap to be useful in practice: We include them only because we wish to document the interaction of "!" with every other type of constructor.

The axiom at-bang indicates that the type $[\sigma]$ is duplicable. Indeed, this type describes a value that inhabits the region σ, but does not represent the ownership of this value. It is sound to duplicate a value (say, a memory location) as long as one does not duplicate the permission that governs it (say, the right to read and write at this location). This axiom was used in Section 3 to justify that a reference whose content has type $[\sigma]$ has duplicable content and hence can be read.

8.6 Regions

We have explained that focus π (Section 7.3.2) is considered a primitive operation because it has the "side effect" of creating a fresh region. The inverse operation, defocus π, has no such side effect, so we are able to consider it as a coercion (Figure 20).

The coercion defocus π requires a capability of the form $\{\mathsf{S}\,\sigma : \tau\}$, which represents the ownership of a singleton region σ as well as the information that the inhabitant

$$\frac{\iota_1 = \mathsf{Log} \vee \iota_2 = \mathsf{Log}}{C \vdash \mathsf{star\text{-}comm} : {}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2} \leqslant {}_{\iota_2}(\tau_2 \times \tau_1)_{\iota_1}}$$

$$\frac{\iota_1 = \mathsf{Log} \vee \iota_2 = \mathsf{Log} \vee \iota_3 = \mathsf{Log}}{C \vdash \mathsf{star\text{-}assoc} : {}_{(\iota_1.\iota_2)}(({}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}) \times \tau_3)_{\iota_3} \leqslant {}_{\iota_1}(\tau_1 \times ({}_{\iota_2}(\tau_2 \times \tau_3)_{\iota_3}))_{(\iota_2.\iota_3)}}$$

$$C \vdash \mathsf{star\text{-}ref} : (\mathsf{ref}\ \tau_1) * \tau_2 \leqslant \mathsf{ref}\ (\tau_1 * \tau_2) \qquad C \vdash \mathsf{ref\text{-}star} : \mathsf{ref}\ (\tau_1 * \tau_2) \leqslant (\mathsf{ref}\ \tau_1) * \tau_2$$

$$C \vdash \mathsf{star\text{-}singleton} : \{\mathsf{S}\,\sigma : \tau_1\} * \tau_2 \leqslant \{\mathsf{S}\,\sigma : \tau_1 * \tau_2\}$$

$$C \vdash \mathsf{singleton\text{-}star} : \{\mathsf{S}\,\sigma : \tau_1 * \tau_2\} \leqslant \{\mathsf{S}\,\sigma : \tau_1\} * \tau_2$$

$$\frac{(\alpha, \beta) \bowtie (\alpha', \beta') \qquad |\vec{\theta}| = n}{C \vdash \otimes\text{-}\mathsf{exch}_n : ((\tau \otimes \alpha) \otimes \beta') \otimes \vec{\theta} \leqslant ((\tau \otimes \beta) \otimes \alpha') \otimes \vec{\theta}}$$

Fig. 21. Subtyping: movement of stars.

of this region has type τ. It enables the replacement of one occurrence of the singleton type $[\sigma]$ with the type τ. This occurrence appears under a type context $\mathscr{T}$, which is determined by the path π. The capability $\{\mathsf{S}\,\sigma : \tau\}$ is consumed. If it were preserved, the type τ would appear twice in the right-hand side of the axiom, which would be unsound, since τ is not in general duplicable. In the special case where τ is duplicable, the operation defocus-dup π (Section 7.3.2), which does not consume the capability, can be used instead.

The axiom defocus-group allows returning to a group region an inhabitant that has previously been borrowed via the focusgroup operation (Section 7.3.2). The capability $\{\mathsf{S}\,\sigma : \tau\}$ represents the ownership of this particular inhabitant, while the capability $\{\rho : \tau \setminus \sigma\}$ represents the ownership of the group region deprived of this particular inhabitant. If they agree on a common type τ, then these two capabilities can be consumed to produce a capability for the full group region, $\{\mathsf{G}\,\rho : \tau\}$.

The axiom singleton-to-group allows a singleton region to degenerate into a group region.

8.7 Movement of stars

In separation logic, a basic consequence of the semantic interpretation of separating conjunction is that it is commutative and associative. Here the role of separating conjunction is played by the pair type constructor.

Is a pair type constructor commutative? This depends on the layer annotations that it carries. In a pair of two physical values, the ordering of the pair components matters, as it describes the layout of the pair in memory. However, in a pair of a physical value and a logical value, or in a pair of two logical values, the ordering does not matter because logical values are erased. Thus, we provide a subtyping axiom for exchanging the two components of a pair when at least one of them is in the logical layer (Figure 21). This subtyping axiom is its own inverse: the types ${}_{\iota_1}(\tau_1 \times \tau_2)_{\iota_2}$ and ${}_{\iota_2}(\tau_2 \times \tau_1)_{\iota_1}$ are inter-convertible if at least one of ι_1 and ι_2 is Log. Analogously, the pair type constructor can be considered associative, provided that

$$\frac{C(x) = \tau \leqslant \theta}{C \vdash x : \tau \leqslant \theta} \qquad \frac{C ; \tau \leqslant \theta \vdash c : \tau \leqslant \theta}{C \vdash \mu c : \tau \leqslant \theta}$$

Fig. 22. Subtyping: recursive coercions.

at least one of the three components involved is in the logical layer. The conjunction $\iota_1.\iota_2$ is defined as Phy if at least one of ι_1 and ι_2 is Phy and Log otherwise.

We have already studied how the pair-type constructor commutes with the universal and existential quantifiers (Figure 18), and with the "!" modality (Figure 19). There are two more type constructors with which "$*$" freely commutes, namely, references and singleton regions. This is stated by the subtyping axioms star-ref, ref-star, star-singleton, and singleton-star (Figure 21). These axioms allow storing a capability into and retrieving a capability out of a reference or a singleton region.

The last axiom in Figure 21 is more complex. It is not meant to be of interest to a programmer: it is included because it plays a role in the proof of type soundness. (More specifically, it is used in the revelation lemma; see Section 12.) Upon first reading, take the natural integer n to be zero and the vector of types $\vec{\theta}$ to be empty. In this simplified case, the axiom states that if the types α, β, α', and β' form a commutative pair, then the types $(\tau \otimes \alpha) \otimes \beta'$ and $(\tau \otimes \beta) \otimes \alpha'$ are inter-convertible. (Commutative pairs are defined in Section 6.3.) Because commutative pairs are symmetric, this axiom is its own inverse.

This property is at the heart of the soundness argument for the anti-frame rule. It appears in the conference paper (Pottier, 2008), where it is used in the proof of Lemma 3.1. It also appears in Schwinghammer *et al.*'s (2010) semantic proof, where it corresponds to item (1) of Definition 1. In the semantic approach, this property takes the form of a type equality, whereas in the syntactic approach, a more cumbersome explicit coercion is necessary.

The general form of this axiom, where $\vec{\theta}$ is an arbitrary vector of types, is required for the axiom itself to satisfy revelation (Lemma 12.1).

The coercion $\otimes$-exch$_0$, which exchanges two tensors, can be used to define a coercion $\circ$-exch, which exchanges two composition operators.

Lemma 8.7 *There exists a coercion* $\circ$-exch *that satisfies the following typing rule:*

$$\frac{(\alpha, \beta) \bowtie (\alpha', \beta')}{C \vdash \circ\text{-exch} : (\tau \circ \alpha) \circ \beta' \leqslant (\tau \circ \beta) \circ \alpha'} \qquad \heartsuit$$

The coercion $\circ$-exch is used in the semantics of the instrumented calculus. It appears, for instance, when $\otimes$-exch$_n$ is applied to a λ-abstraction (Section 13).

8.8 Recursive coercions

Our treatment of subtyping in the presence of recursive types follows Brandt and Henglein (1998). We introduce coercion variables x and recursive coercions μc, where one variable is bound in c. The rules for ascribing types to these constructs are simple (Figure 22). A coercion variable x has type $\tau \leqslant \theta$ if the coercion environment C says so. A recursive coercion μc has type $\tau \leqslant \theta$ if c has type $\tau \leqslant \theta$ under the assumption that the bound variable has type $\tau \leqslant \theta$.

The ability to fold and unfold recursive types is built into the definition of type equality, which itself is included in the subtyping relation via the coercion id. Thus, we do not need "fold" and "unfold" coercions.

It is worth noting that the typing rules for recursive *coercions* do not explicitly refer to recursive *types*. Even though recursive types are the reason why we need recursive coercions, the two concepts remain orthogonal. To illustrate the expressiveness of this approach, we prove that Amadio and Cardelli's (1993) well-known rule for comparing recursive types is admissible.

Lemma 8.8 *There exists a coercion transformer* amadio-cardelli · *such that the following property holds:*

$$\frac{0{\uparrow}0{\uparrow}C; 0 \leqslant 1 \vdash c : 1{\uparrow}\tau \leqslant 0{\uparrow}\theta}{C \vdash \mathsf{amadio\text{-}cardelli}\ c : \mu\tau \leqslant \mu\theta} \qquad \heartsuit$$

There is however a technical pitfall. Some recursive coercions are dangerous. The coercion $\mu 0$, for instance, has type $\tau \leqslant \theta$ for every τ and θ. If we admit this coercion, then subtyping becomes the full relation, and the type system becomes unsound. Technically, the coercion $\mu 0$ breaks neither subject reduction nor progress at the level of the instrumented calculus. It does, however, break the property that the reduction of values terminates, because a value of the form $(\mu 0)\ v$ reduces to itself. This makes it impossible to transport subject reduction and progress down to the raw calculus.

In order to avoid this problem, we restrict our attention to certain well-formed coercions. The basic idea is as follows: In order to guarantee that a value of the form $(\mu c)\ v$ cannot reduce forever, we allow the coercion to invoke itself recursively only after it has descended at least one level down in the raw value that underlies v. Because the structure of raw values is finite and unaffected by the reduction of coercions, this is enough to ensure that coercions cannot diverge.

A coercion μc is well-formed if and only if c is contractive in the coercion variable 0. The inductive definition of contractiveness, which extends Brandt and Henglein's (1998) definition, is omitted. In short, the coercions $c_1 \rightarrow c_2$, $_{\mathsf{Phy}}(c_1 \times c_2)_{\mathsf{Phy}}$, and ref c are contractive in every variable, while every other composite coercion is contractive in a variable x only if its immediate constituents are contractive in this variable. The bottom line is that every cycle in a recursive coercion must cross a type constructor that exists at runtime.

PART THREE
Proof of type soundness

In this part, we explicitly construct the type equality relation (Section 9), which so far was only axiomatized. We complete the definition of the system by defining monotonic separation algebras, whose elements we refer to as "resources" (Section

10), and by proposing typing rules for the values that do not appear in source programs, such as memory locations and capabilities (Section 11). Then, we study revelation, that is, the transformation of a hidden invariant into an explicit one (Section 12). We prove that the system supports revelation, or, more technically, that a higher-order frame rule is admissible. We present the operational semantics of the instrumented calculus (Section 13), which exploits revelation. We define the well-layeredness judgement, which guarantees that the two "layers" (the values that exist at runtime and those that do not) co-exist in a consistent manner (Section 14). Finally, we establish type soundness for the instrumented calculus and for the raw calculus (Section 15).

9 Type equality: construction

Earlier (Section 6), we have listed the properties that the type equality relation must satisfy. We now construct this relation. This part of formalization comprises roughly 3,000 lines of Coq definitions, statements, and proofs.

Contractiveness and well-formedness. In short, well-formedness requires every cycle in the type structure to go through at least one type constructor other than μ or the left-hand side of $\otimes$.

More precisely, we use an inductive definition of well-formedness (not shown), whereby a type τ is well-formed only if its sub-terms are well-formed, and, in addition, a type $\mu\tau$ is well-formed only if its body τ is contractive in the type variable 0. A type τ is contractive in a type variable x if and only if every occurrence of x within τ appears under at least one type constructor other than μ or the left-hand side of $\otimes$.

This is made precise via an inductive definition of contractiveness, of which we show only the following four representative rules:

$$\frac{k \neq x}{x \text{ contractive in } k} \qquad \frac{\tau \text{ contractive in } k+1}{\mu\tau \text{ contractive in } k}$$

$$\tau_1 \rightarrow \tau_2 \text{ contractive in } k \qquad \frac{\tau_1 \text{ contractive in } k}{\tau_1 \otimes \tau_2 \text{ contractive in } k}$$

Schwinghammer *et al.* (2009, 2010) define a semantic model of separation logic assertions, which correspond roughly to our types. The model forms a metric space. Contractiveness is given a direct definition in terms of the metric, and the fact that the assertion $\mu\tau$ exists when τ is contractive in the type variable 0 follows directly from Banach's fixed point theorem. More strikingly, the tensor operator is also given a direct definition: It is (roughly speaking) a function of a pair of assertions to an assertion. One can prove that this function is contractive in its second argument.

Toward a definition of type equality. It would be nice if one could define type equality just by orienting the laws in Figure 9 from left to right and by considering two types equal if and only if their normal forms are equal. If these laws formed a terminating and confluent rewrite system, then such a definition would make perfect sense, as each type would admit a unique normal form. Unfortunately, although this system is indeed confluent, it is clearly not terminating: by unfolding recursive types, its first rule creates new structure.

One could address this problem by proving that, although the rewriting process out of a type τ does not in general terminate, it produces, in the limit, a possibly infinite tree, known as the "infinite unfolding" of τ. One would first define a function that maps a well-formed type to its infinite unfolding, then define equality of types as equality of infinite unfoldings.

We follow this route in spirit, but decide to fuse the definitions of the unfolding function and that of the equality relation over infinite trees. This allows us to give a direct definition of type equality and to never explicitly work with infinite trees.

Should this definition be inductive or co-inductive? It cannot be purely inductive, because we are dealing with recursive types and wish to allow for equality proofs between them that are "infinitely deep". However, it seems that it cannot be purely co-inductive either. For one thing, we need a rule that allows a rewriting step to take place before the comparison continues. As noted by Gapeyev *et al.* (2002), including such a rule as part of a co-inductive definition allows equality proofs that are "infinitely wide", and leads to a relation that equates too many types.

In summary, we seem to need a mixed inductive–co-inductive definition of type equality. Other researchers have come to this conclusion as well (Danielsson & Altenkirch, 2010). Unfortunately, although certain modern proof assistants, most notably Agda, support the mixing of induction and co-induction, Coq does not. A standard work-around consists in using an inductive definition of *approximate equality down to depth* k, where k is an integer index, and in viewing equality as the limit of the approximate equality relations. The index k can be used to precisely control what kind of infinite proofs are permitted: in short, for each premise of each proof rule, inductive behavior is imposed by leaving k unmodified, while co-inductive behavior is permitted by decrementing k.

Defining type equality. Here is how approximate equality is defined. All types are 0-equal:

$$\tau \equiv_0 \theta$$

Two function types are $k+1$-equal if and only if their respective domains and codomains are k-equal. The decrease in the approximation index k means that a function type constructor contributes one to the depth of a type. This reflects the fact that this type constructor is considered contractive in both arguments.

$$\frac{\tau_1 \equiv_k \theta_1 \qquad \tau_2 \equiv_k \theta_2}{\tau_1 \to \tau_2 \equiv_{k+1} \theta_1 \to \theta_2}$$

An analogous congruence rule is given for every type constructor except μ. This includes type variables, region identifiers, and tensor. In particular, following is the the congruence rule for tensor. The treatment of the approximation index k reflects the fact that tensor is considered contractive in its right-hand argument only.

$$\frac{\tau_1 \equiv_{k+1} \theta_1 \qquad \tau_2 \equiv_k \theta_2}{\tau_1 \otimes \tau_2 \equiv_{k+1} \theta_1 \otimes \theta_2}$$

The equational theory of Figure 9 is built into the definition of approximate equality via the following two symmetric rules. There, the reduction relation $\cdot \rightsquigarrow \cdot$ corresponds to the rewrite system of Figure 9, oriented from left to right; reduction is permitted at the root and in the left-hand side of a tensor. The following two rules may be viewed as transitivity rules, where one premise is constrained to perform a directed rewrite step $\cdot \rightsquigarrow \cdot$ as opposed to an arbitrary equality step $\cdot \equiv_k \cdot$.

$$\frac{\tau_1 \rightsquigarrow \tau_2 \qquad \tau_2 \equiv_k \theta}{\tau_1 \equiv_k \theta} \qquad\qquad \frac{\theta_1 \rightsquigarrow \theta_2 \qquad \tau \equiv_k \theta_2}{\tau \equiv_k \theta_1}$$

The above rules allow unfolding a recursive type in situations where the type constructor μ prevents the application of any other rule. For this reason, no explicit congruence rule is needed for μ. These rules also allow moving tensors out of the way in many situations. For instance, if a tensor is applied to a pair type, it can be moved down into the pair components. There are, however, situations where tensors cannot be moved out of the way. For instance, an application of a tensor to a type variable cannot be simplified. For this reason, a congruence rule for tensor (which was shown above) is required. This gives rise to a critical pair in the proof rules: there can be multiple successful ways of proving two applications of tensor equal. This in turn leads to technical difficulties in the proof that approximate equality is transitive: we found this proof surprisingly difficult. This may be the price to pay for adopting a purely syntactic view of types.

It is not difficult to check that the sequence of relations $\cdot \equiv_k \cdot$ decreases as k increases. We then define type equality $\cdot \equiv \cdot$ as its limit, that is, as the intersection of the approximate type equality relations.

Except for transitivity, it is relatively easy to establish all of the properties that were listed previously (Section 6.2). Many of these properties require well-formedness hypotheses (which, by convention, we do not show). Reflexivity, for instance, requires one. The ill-formed type $\mu 0$ is not equal to itself, because a proof of this equality would be "infinitely wide" and has been ruled out.

10 Resources/monotonic separation algebras

An affine type system controls how many times variables are used. However, as far as we are concerned, this is not an end in itself. It is only a means of governing the ownership of *resources*. That is, the reason why we wish to control the use of variables is that variables denote values; and the reason why we wish to control the use of values is that values encapsulate and provide access to resources. But what are resources, and why control them?

Let us offer some examples.

A memory cell is a resource. Read and write access to memory cells must be controlled so as to ensure that strong updates are sound. A memory cell should have at most one owner, and only the owner should be allowed to read and write the cell. (This informal sentence in our affine setting means that the permission to read and write the cell must not be duplicated. It may, however, be discarded.)

A region is a resource. Regions support various logical operations, such as focus, defocus, and adoption, which can be thought of as strong updates, because they alter the set of region inhabitants or the extent to which the region owns its inhabitants. For this reason, regions too should have at most one owner, and only the owner should be allowed to perform certain operations on regions.

One may imagine many other kinds of resources. For instance, a *ghost* memory cell could be a resource. A ghost cell does not exist at runtime but appears in the source program, where it may be written, read, and referred to within logical assertions. A *monotonic* memory cell could be a resource. Updates to a monotonic cell are constrained by a fixed pre-order, and this property can be exploited when reasoning about programs. A *time credit* could be a resource. A time credit represents a permission to perform a constant amount of computation. We have explored these ideas in other work (Pilkiewicz & Pottier, 2011) and hope to verify in the future that the present framework directly supports them. In an extension of the system with primitive weak references, a *store typing* could be a resource. In a multi-threaded setting, a dynamically allocated *lock* could be a resource. A memory cell that supports *fractional permissions* could be a resource.

Although usually one is ultimately interested in specific kinds of resources (in this paper, just references and regions), it is important to manipulate resources abstractly wherever possible. Many of our typing rules implement mechanisms for imposing and preserving affinity in a manner that is independent of the particular kinds of resources that are being controlled.

This idea has been investigated by multiple authors. Calcagno *et al.* (2007) axiomatize separation algebras and present an abstract version of separation logic which they prove sound with respect to an arbitrary separation algebra. Dockins *et al.* (2009) relax the treatment of the unit element, study certain additional axioms, and discuss systematic methods of constructing separation algebras.

Our own axiomatization arises out of a desire to better highlight the abstract structure of Charguéraud and Pottier's paper (2008) proof of type soundness. This axiomatization turns out to be quite closely related with Calcagno *et al.*'s (2007) and Dockins *et al.*'s (2009) ideas. The main novelty, perhaps, is that our resources are equipped with two ordering relations, which allow us to constrain (and reason about) the manner in which an active thread of computation may affect the state of a suspended thread. (The details follow.) For this reason, we refer to the models of our axiomatization as *monotonic separation algebras*.

In the following, we first axiomatize monotonic separation algebras (Section 10.1). In other words, we list a number of properties that resources must satisfy. In the Coq formalization, this takes the form of a type class definition. Then, we present systematic means of constructing separation algebras (Section 10.2). In Coq, this corresponds to a number of parameterized instance definitions. Last, we indicate which concrete instance is used in this paper to deal with references and regions (Section 10.3).

Resources have already appeared in the definition of the typing judgement (Section 7), but they are actively exploited only by the typing rules that deal with programmer-inaccessible constructs, which we have not given yet (Section 11).

10.1 Axiomatization

The most prominent property of resources is that they can be split and combined: that is, they support a form of conjunction. This operation is usually formalized either as a partial function of two arguments (Calcagno *et al.*, 2007; Nanevski *et al.*, 2010) or as a ternary relation (Dockins *et al.*, 2009). We choose the latter approach and use the notation $R_1 * R_2 = R$. (Calcagno *et al.* write $\bullet$, while Dockins *et al.* write $\oplus$.) This statement means that the resources R_1 and R_2 combine to form the resource R, or, adopting an opposite view, R can be split into R_1 and R_2.

Axioms 10.1 *Conjunction is commutative and associative.*

$$\frac{R_1 * R_2 = R}{R_2 * R_1 = R} \qquad \frac{R_1 * R_2 = R_{12} \qquad R_{12} * R_3 = R_{123}}{\exists R_{23},\, R_2 * R_3 = R_{23} \wedge R_1 * R_{23} = R_{123}}$$

Remark 10.2 The statement of associativity involves an existential quantifier. In practice, it turns out that this is quite painful, and makes reasoning modulo commutativity and associativity very difficult. We end up developing a special-purpose tactic, called `reconfigure`, for this kind of reasoning. Within the implementation of `reconfigure`, we temporarily revert to a view of conjunction as a binary function. In hindsight, it might have been more convenient to follow other authors (Nanevski *et al.*, 2010) and view conjunction as a binary function from the very beginning. A perceived advantage of our approach (and the reason why we chose it) is that we do not need to introduce artificial "undefined" resources in order for conjunction to be defined everywhere, and we do not need to pollute the type system with side conditions that require all resources to be "defined". ◇

Remark 10.3 Dockins *et al.* (2009) require conjunction to be a functional relation. Our implementation of `reconfigure` also exploits this property (and a few more). However, the only task of `reconfigure` is to rearrange resources up to commutativity and associativity, so, in principle, it should be possible to implement `reconfigure` without using any axioms other than commutativity and associativity. The property that conjunction is a functional relation is never explicitly used outside the implementation of `reconfigure`, so we do not consider it a part of the axiomatization of resources. ◇

Two resources R_1 and R_2 are *compatible* if and only if there exists a resource R such that $R_1 * R_2 = R$ holds. A resource R is *duplicable* if and only if $R * R = R$ holds.

Remark 10.4 One might expect the following axiom to hold: if R is compatible with itself, then R is duplicable. Dockins *et al.* (2009) refer to this axiom as *disjointness.* This property can be valid in certain instances: for example, if resources represent the exclusive ownership of heap fragments, then the only resource that is compatible with itself is the empty heap, which is duplicable. However, it is false in some other instances: for example, two fractional permissions combine to yield a greater fractional permission, yet they are not duplicable; two permissions to spend a unit of time combine to yield a permission to spend two units of time, yet they are not duplicable. Thus, we omit this axiom. It is not required in the type soundness proof. ⋄

A resource R can be composite: It can be a conjunction of "smaller" resources, some of which are duplicable, some of which are not. In such a case, it is useful to be able to refer to the "duplicable part" of R. Thus, we require the existence of a function $\widehat{\cdot}$ that maps a resource R to its "duplicable core" (or just "core") $\widehat{R}$. In order to spell out our intuition that $\widehat{R}$ is the "duplicable part" of R, we require the following.

Axioms 10.5 *$\widehat{R}$ is a unit for R. Two compatible resources have a common core. A duplicable resource is its own core. Splitting a core yields duplicable parts.*

$$R * \widehat{R} = R \qquad \frac{R_1 * R_2 = R}{\widehat{R_1} = \widehat{R}} \qquad \frac{R * R = R}{R = \widehat{R}} \qquad \frac{R_1 * R_2 = \widehat{R}}{R_1 * R_1 = R_1}$$

These axioms have a somewhat disparate appearance. Perhaps one could find a more aesthetically pleasing set of axioms. We view these axioms as technically satisfactory insofar as they have the following four consequences.

Lemma 10.6 *A core is duplicable.*

$$\widehat{R} * \widehat{R} = \widehat{R}$$ ♡

Lemma 10.7 *If a resource is its own core, then it is duplicable.*

$$\frac{R = \widehat{R}}{R * R = R}$$ ♡

Lemma 10.8 *The function "core" is idempotent.*

$$\widehat{\widehat{R}} = \widehat{R}$$ ♡

Lemma 10.9 *The function "core" preserves conjunction.*

$$\frac{R_1 * R_2 = R}{\widehat{R_1} * \widehat{R_2} = \widehat{R}}$$ ♡

As a result of these properties, we now have three equivalent characterizations of duplicability. Indeed, "R is duplicable" ($R * R = R$) is equivalent to "R is a core" ($\exists R_1,\ R = \widehat{R_1}$), which itself is equivalent to "R is its own core" ($R = \widehat{R}$).

Remark 10.10 Calcagno *et al.*'s separation algebras (2007) have a single unit element. Dockins *et al.* (2009) relax this condition and only require that each resource has its own unit. They state this using existential quantification: $\forall x, \exists u_x, u_x * x = x$. They further prove that units are unique, that is, u_x is really a function of x. To a certain extent, our mapping of R to $\widehat{R}$ is related to Dockins *et al.*'s (2009) mapping of x to u_x. Indeed, if one reads Axioms 10.5 in this light, one finds that the first three axioms are part of (or consequences of) Dockins *et al.*'s basic axioms (2009), while the last one is Dockins *et al.*'s positivity axiom. However, our units are not unique (see Remark 10.11 below), so $\widehat{R}$ is not "the" unit for R, but the strongest unit for R. ◇

Remark 10.11 We depart from Calcagno *et al.* (2007) and Dockins *et al.* (2009) in that our units are not unique and our conjunction is not cancellative, that is, $R_1 * R = R'$ and $R_2 * R = R'$ do not imply $R_1 = R_2$. (The uniqueness of units is a consequence of cancellativity.) We find that cancellativity is not necessary for soundness. Furthermore, it is quite a strong axiom. Indeed, combined with the above axioms, it implies that duplicable resources are atomic, that is, they cannot be split in a non-trivial way: $\widehat{R_1} * \widehat{R_2} = \widehat{R}$ implies $\widehat{R_1} = \widehat{R}$ and $\widehat{R_2} = \widehat{R}$. This is quite unpleasant. We do have in mind instances where duplicable resources admit non-trivial splits. For example, logical assertions, viewed as resources, are duplicable (it is sound for them to be duplicable because they are true forever), yet admit non-trivial splits, since $P \wedge Q$ is in general a strictly stronger assertion than P or Q alone. Similarly, observations (Pilkiewicz & Pottier, 2011) are duplicable, yet admit non-trivial splits: When $j > i$ holds, an observation of a "fate" (a monotonic ghost variable) in state j is strictly stronger than an observation of it in state i: so, they are distinct, yet the two combine to yield just the former. ◇

We now come to what constitutes perhaps the most original part of our axiomatization of resources. We introduce two pre-orders, written $\blacktriangleleft$ and $\triangleleft$, which describe how resources evolve in time.

Axioms 10.12 *The relations* $\blacktriangleleft$ *and* $\triangleleft$ *are reflexive and transitive.* ◇

Why *two* pre-orders? This question is perhaps best answered by thinking in a concurrent setting, where each thread of execution owns a certain resource, and these resources join together to form a resource that describes the global state of the system. In an interleaving semantics, as one thread of execution makes progress, all other threads are suspended.

The active thread performs operations that have an effect on its own resource. For instance, it may allocate (and claim ownership of) new memory, update pre-existing memory cells that it owns, and so on. We use the *active execution pre-order*, written $\blacktriangleleft$, to reflect this evolution. That is, $R_1 \blacktriangleleft R_2$ holds if the active thread can cause its own resource to evolve from R_1 to R_2.

Meanwhile, what does a suspended thread perceive? One might at first imagine that it must not perceive any change, that is, its own resource must remain unchanged as long as it remains suspended. One would formalize this via the following candidate axiom:

$$\frac{R_1^a * R_1^b = R_1 \qquad R_1^a \blacktriangleleft R_2^a}{\exists R_2,\ R_2^a * R_1^b = R_2 \wedge R_1 \blacktriangleleft R_2}$$

Here, one should consider R_1^a as the resource initially owned by the active thread and R_1^b as the resource initially owned by some passive thread. The two resources are compatible: this is expressed by the first premise. The active thread makes one execution step, and moves to R_2^a: this is expressed by the second premise. The conclusion of the candidate axiom states that, after this execution step, the two threads are still in compatible states – that is, R_2^a and R_1^b can be joined to yield a new global state R_2 – and, furthermore, this group of two threads, one of which takes an execution step, can be considered abstractly as a single thread that takes an execution step – that is, the initial global state R_1 and the new global state R_2 are in the active execution pre-order $\blacktriangleleft$. The latter condition is required for the candidate axiom to remain useful in a situation where there is an arbitrary number of passive threads: it allows iterated applications of the axiom.

This candidate axiom makes sense – so it was worth explaining – but is too strong. For example, when resources represent heap fragments, it is natural for every resource to carry an *allocation limit*, which represents the number of the next available memory location, and for two resources to be compatible only if their allocation limits agree. Thus, when the active thread allocates new memory, a passive thread must react by updating its allocation limit. As a result, we cannot require the state of the passive thread, represented by R_1^b, to remain unchanged. We must allow R_1^b to evolve to R_2^b along a certain pre-order, which we refer to as the *passive execution pre-order*, and write $\lhd$.

Thus, the definitive version of the above candidate axiom is as follows.

Axiom 10.13 *Consider an active thread and a passive thread in compatible states. If the active thread moves (along the active execution pre-order), then the passive thread is able to move as well (along the passive execution pre-order) so that the two threads remain in compatible states; furthermore, their combined state has evolved along the active execution pre-order.*

$$\frac{R_1^a * R_1^b = R_1 \qquad R_1^a \blacktriangleleft R_2^a}{\exists R_2^b R_2,\ R_2^a * R_2^b = R_2 \wedge R_1^b \lhd R_2^b \wedge R_1 \blacktriangleleft R_2}$$

Our informal discussion of the execution pre-orders has been in terms of threads. Nevertheless, these pre-orders make sense in a sequential setting as well. An operational semantics typically splits a term into a pair of a redex and an evaluation context. The redex can be viewed as active and the evaluation context as passive. The type system splits the global state R into a conjunction of R_1, owned by the redex, and R_2, owned by the evaluation context. When the redex makes a step, R_1 evolves along the active execution pre-order $\blacktriangleleft$, while R_2 evolves along the passive execution pre-order $\lhd$.

In the concurrent separation logic, the frame rule can be viewed as a special case of the parallel composition rule, where one thread is active and the other is passive.

In light of this remark, one should not be surprised that thinking in terms of threads also makes sense in a sequential setting.

We need another axiom that looks very much like Axiom 10.13.

Axiom 10.14 *Consider two passive threads in compatible states. If their combined state must move (along the passive execution pre-order) then each thread is able to comply (along the passive execution pre-order).*

$$\frac{R_1^a * R_1^b = R_1 \qquad R_1 \lhd R_2}{\exists R_2^a R_2^b,\ R_2^a * R_2^b = R_2 \wedge R_1^a \lhd R_2^a \wedge R_1^b \lhd R_2^b}$$

Together, Axioms 10.13 and 10.14 allow reasoning about a collection of threads. Imagine these threads are organized in a tree, where a binary internal node represents a conjunction and a leaf node represents a thread and its resource. The root of the tree represents the global system. Imagine one leaf makes an active move. Then, Axiom 10.13 can be iteratively applied along the path from this leaf up to the root: one finds that every node along this path makes an active move. Next, Axiom 10.14 can be iteratively applied along every path down to some other leaf: one finds that every leaf other than the active leaf makes a passive move. This argument appears in the type soundness proof.

In the terminology of rely/guarantee, the active execution ordering $\blacktriangleleft$ is a guarantee (it describes the updates that the active thread can perform), while the passive execution ordering $\lhd$ is a rely (it describes the updates that a passive thread must be able to tolerate). In a concurrent setting, Dinsdale-Young *et al.* (2010) have independently developed ideas that appear very closely related to ours: Our Axiom 10.13 corresponds quite closely to their Lemma B.8, while our Axiom 10.14 corresponds exactly to their Lemma B.7. The more recent paper by Dinsdale-Young *et al.* (2012) is also relevant.

Our last axiom is as follows.

Axiom 10.15 *The function "core" preserves the passive execution pre-order.*

$$\frac{R_1 \lhd R_2}{\widehat{R_1} \lhd \widehat{R_2}}$$

This axiom allows establishing the monotonicity lemma (Lemma 15.4) in the case of the "!" introduction rule (Figure 10).

We refer to a structure that satisfies all of the axioms presented above as a *monotonic separation algebra*. Our use of conjunction together with two execution pre-orders allows us to abstractly and smoothly reason about exclusive ownership (for instance, a memory location is owned by at most one thread), shared information (for instance, the allocation limit is shared among all threads), and monotonic information (for instance, the allocation limit can only grow with time; the set of inhabitants of a region can only grow with time).

Remark 10.16 The reader may be puzzled by the fact that there exist several trivial monotonic separation algebras. For instance, a trivial algebra is obtained by taking both $\blacktriangleleft$ and $\triangleleft$ to be the diagonal relation, which relates every resource R with itself and only itself. Another trivial algebra is obtained by taking both $\blacktriangleleft$ and $\triangleleft$ to be the full relation, which relates all resources. Either of these trivial choices satisfies Axioms 10.12, 10.13, 10.14, and 10.15. What is going on? The axioms that we have presented are sufficient to establish type soundness for the core of our type system, deprived of any concrete resources. (This corresponds to the typing rules in Figures 10 and 11.) That is, they are sufficient to ensure that the control of duplication (which, in our DILL-style formulation, is built into every typing rule) is correctly implemented. However, once we extend the system with concrete resources (such as references and regions), this abstract axiomatization of resources is no longer sufficient to define the type system, let alone to prove its soundness. Some of the typing rules must be aware of the concrete structure of resources. (These rules are presented in Section 11.) As a result, some of the lemmas that participate in the type soundness proof must be aware of the concrete definition of the pre-orders $\blacktriangleleft$ and $\triangleleft$. The proofs of these lemmas would break if the trivial definitions above were adopted. In particular, subject reduction (Lemma 15.13) would break if $\blacktriangleleft$ was the diagonal relation: the "active thread" would be disallowed from affecting its own resources or allocating new resources. Monotonicity (Lemma 15.4) would break if $\triangleleft$ was the full relation: a "passive thread" would be forced to assume that its own resources can be stolen or destroyed by the currently active thread. $\diamond$

Our type system is an extension of an abstract DILL-style formalism (Section 7) with specific typing rules that deal with references and regions (Sections 7, 8, and 11). Subject reduction and progress for the core formalism can in principle be proved in terms of an arbitrary monotonic separation algebra. However, we have not explicitly isolated this aspect. Our type soundness proof is only "semi-abstract": It exploits the properties of monotonic separation algebras where possible, but drops down to the level of one specific separation algebra (which involves references and regions; see Section 10.3) where necessary.

10.2 Instances

We now provide a toolbox for building a variety of monotonic separation algebras.

Credits. The natural numbers form a monotonic separation algebra, where conjunction is addition, "core" is the constant function 0, the active execution pre-order $\blacktriangleleft$ is $\geqslant$, and the passive execution pre-order $\triangleleft$ is equality. This could be used to model time or space credits (Hofmann, 2000; Atkey, 2010; Pilkiewicz & Pottier, 2011). The use of $\geqslant$ as the active execution pre-order indicates that credits can be consumed but not manufactured.

Exclusive permissions. Imagine that there is one object of interest. This object might be a memory cell, a region, or something else. For the moment, we are not interested in the nature of this object; neither are we interested in its name, or address. We are only interested in setting up a monotonic separation algebra whose resources allow claiming either "I do not own the object" or "I do own the object, and it is presently in state x". We would like to think of the object as mutable, and we would like to formalize the idea that only the owner is allowed to mutate the object, or to claim that its current state is x. Here the value x ranges over an arbitrary Coq type X.

We represent permissions as values of the Coq type $\mathit{option}\,X$. Recall that we write $\bot$ for "`None`" and x for "`Some` x", so a value of type $\mathit{option}\,X$ is either $\bot$ or some value x of type X. We take $\bot$ to represent the absence of a permission ("I do not own the object") and x to represent a permission ("I do own the object, and it is presently in state x"). In order to indicate how permissions combine and evolve with time, we equip the type $\mathit{option}\,X$ with the structure of a monotonic separation algebra, as follows.

Conjunction is inductively defined by the axioms $\bot * R = R$ and $R * \bot = R$. Thus, two resources (or permissions) of the form x_1 and x_2 are incompatible. This means that ownership is exclusive: there is at most one owner. Furthermore, only the owner has access to the description x of the object. Last, permissions are atomic: whenever a permission is split, one of the two parts must be $\bot$.

The function "core" is the constant function $\bot$.

The active execution pre-order is defined as follows: $R_1 \blacktriangleleft R_2$ holds if and only if $R_1 = \bot$ implies $R_2 = \bot$. Thus, it is impossible for the active thread to spontaneously acquire ownership: the relation $\bot \blacktriangleleft x$ never holds. However, it is permitted for the owner to perform a strong update: the relation $x_1 \blacktriangleleft x_2$ always holds. It is also permitted for the owner to abandon ownership: the relation $x \blacktriangleleft \bot$ always holds.

The passive execution pre-order is equality: as long as a thread remains inactive, the permissions that it holds do not change.

Monotonic sets. In order to explain regions, we introduce *monotonic sets*. The informal idea is two-fold: first, the contents of a monotonic set can only grow with time; second, a monotonic set has an owner, and only the owner can add new elements to the set.

For an arbitrary type X, a monotonic set over X is a pair of a permission p of type $\mathit{option}\,\mathit{unit}$ and a set s of type $X \to \mathit{Prop}$. In Coq, the value of type unit is written tt, so a permission p is either $\bot$ or tt. (We could in principle use bool instead of $\mathit{option}\,\mathit{unit}$. Using $\mathit{option}\,\mathit{unit}$ allows us to re-use the treatment of exclusive permissions presented above.) We write $\mathit{mSet}\,X$ for the type of monotonic sets over X.

Conjunction of monotonic sets is defined in terms of conjunction of permissions and equality of raw sets: $(p_1, s_1)*(p_2, s_2) = (p, s)$ is defined as $p_1 * p_2 = p \wedge s_1 = s \wedge s_2 = s$. Accordingly, the function "core" over monotonic sets loses any permission but preserves the raw set: the core of (p, s) is defined as $(\widehat{p}, s)$, that is, $(\bot, s)$.

These definitions indicate that, although a monotonic set has an exclusive owner, the underlying raw set is considered duplicable information. In our intended application to regions, this is crucial. We would like the type $[\rho]$, which characterizes the inhabitants of region ρ, to be duplicable. (See the coercion at-bang in Section 8.5.) This is possible only if the knowledge that a certain value inhabits a certain region is considered duplicable. This knowledge must be accessible to everyone, not just to the owner of the region.

The active execution pre-order $\blacktriangleleft$ over monotonic sets is defined as follows:

$$\frac{p_1 \blacktriangleleft p_2 \qquad \forall x, s_1\, x \Rightarrow s_2\, x \qquad p_1 = \bot \Rightarrow \forall x, s_2\, x \Rightarrow s_1\, x}{(p_1, s_1) \blacktriangleleft (p_2, s_2)}$$

The first premise dictates how the permission component evolves with time. (Here, p_1 and p_2 are permissions of type *option unit*, so $p_1 \blacktriangleleft p_2$ is the relation defined in the paragraph entitled "Exclusive permissions", that is, $p_1 = \bot \Rightarrow p_2 = \bot$.) The second premise indicates that a monotonic set can only grow with time. The third premise indicates that new elements can appear only if the set is owned. This is expressed in the contrapositive: if it is not owned, then no new elements can appear. In our intended application to regions, this means that only the owner of a region can adopt new inhabitants into the region.

The passive execution pre-order $\lhd$ over monotonic sets is defined in a dual way:

$$\frac{p_1 \lhd p_2 \qquad \forall x, s_1\, x \Rightarrow s_2\, x \qquad p_1 = tt \Rightarrow \forall x, s_2\, x \Rightarrow s_1\, x}{(p_1, s_1) \lhd (p_2, s_2)}$$

The first premise dictates how the permission component evolves with time. The second premise again indicates that a monotonic set can only grow with time. The third premise indicates that it can grow strictly only if it is *not* owned. This is natural because the definition of $\lhd$ reflects the point of view of a passive thread. While a thread is passive, only the monotonic sets that the thread does *not* own can grow.

Allocation maps. Certain objects can be created fresh, and exist forever thereafter. For instance, memory locations are freshly allocated, and exist forever.[12] Similarly, regions can be freshly allocated, and exist forever.[13] One could give many more examples of this phenomenon, including first-class locks (Gotsman *et al.*, 2007; Hobor *et al.*, 2008), first-class ghost memory cells (Pilkiewicz & Pottier, 2011), and so on.

In order to deal with these situations, we introduce *allocation maps*. If X is a monotonic separation algebra, then an allocation map over X is a pair of an address ℓ, known as the *allocation limit*, and a total mapping m of addresses into X. Addresses are represented as natural numbers. The allocation limit represents the next available address. Although the mapping m is formally a total function, we regard its behavior at and above ℓ as irrelevant. We write *aMap* X for the type of allocation maps over X.

In a typical application, X might be an algebra of exclusive permissions, as described earlier. Then an allocation map over X would be a pair of an allocation limit and a mapping of addresses to permissions.

[12] This, one may argue, is true regardless of whether one intends to rely on a garbage collector and regardless of whether explicit memory de-allocation is permitted. Indeed, in a high-level operational semantics, memory allocation can be considered to always yield fresh memory locations; there is no garbage collector; and memory de-allocation, if permitted, has the effect of marking a memory location as de-allocated, but does not make it available again for allocation.

[13] Our regions do not exist at runtime. They are purely a type-checking device. Nevertheless, we do need a mechanism for allocating fresh region names.

The conjunction of allocation maps is defined in terms of equality of the allocation limits and address-wise conjunction of elements of X.

$$\frac{\ell_1 = \ell \qquad \ell_2 = \ell \qquad \forall l,\ (m_1\, l) * (m_2\, l) = m\, l}{(\ell_1, m_1) * (\ell_2, m_2) = (\ell, m)}$$

Thus, both components of a split must agree on a common allocation limit. In other words, all threads share a consistent view of the allocation limit. This mechanism ensures that, when the active thread decides to allocate the next address, this address is globally fresh.

Accordingly, the function "core" over allocation maps preserves the allocation limit: the core of (ℓ, m) is defined as $(\ell, \lambda r.\widehat{(m\, r)})$. In other words, the allocation limit is considered duplicable information.

The execution pre-orders $\blacktriangleleft$ and $\lhd$ over allocation maps encode the property that the allocation limit can only grow with time. Furthermore, at each address that exists in both initial and final states, they require the appropriate pre-order over X to be satisfied. Their definitions are

$$\frac{\begin{array}{c}\ell_1 \leqslant \ell_2 \\ \forall l,\ l < \ell_1 \Rightarrow (m_1\, l) \blacktriangleleft (m_2\, l)\end{array}}{(\ell_1, m_1) \blacktriangleleft (\ell_2, m_2)} \qquad \frac{\begin{array}{c}\ell_1 \leqslant \ell_2 \\ \forall l,\ l < \ell_1 \Rightarrow (m_1\, l) \lhd (m_2\, l)\end{array}}{(\ell_1, m_1) \lhd (\ell_2, m_2)}$$

Products, sums, sequences. If the types X_1 and X_2 are equipped with monotonic separation algebra structure, then the types $X_1 \times X_2$, $X_1 + X_2$, and *list* X_1 again form monotonic separation algebras. If X_1 is arbitrary and X_2 is a monotonic separation algebra, then the type $X_1 \to X_2$ again forms a monotonic separation algebra. The definitions are omitted: they are straightforward adaptations of Dockins *et al.*'s (2009) definitions.

10.3 A concrete instance

In the remainder of the paper, we fix the type of resources: We pick a particular construction of resources that fits our purposes. We use resources to help us reason about two particular features of the type system: references and regions. The typing rules that have been presented already (Section 7) are formulated in terms of (and are sound with respect to) an arbitrary monotonic separation algebra, while the rules that specifically deal with references and regions (Section 11) are aware of the concrete structure of resources.

Memory locations are dynamically allocated. Furthermore, we would like to set up a system of exclusive permissions over memory locations. That is, we would like each reference cell to have at most one owner. Last, we would like the owner of a memory location to be able to formulate an assertion about the value that is currently stored there. For instance, the owner should be able to say, "I know that this reference cell currently holds a value of type τ", for an arbitrary type τ. Because the type τ could be arbitrarily precise, the only way in general for the owner to justify such an assertion is to have exact knowledge of the value that is currently stored in the cell. Thus, an appropriate type for resources that describe references is *aMap* (*option raw_value*). This is a type of allocation maps where a memory location l is mapped to $\perp$ if it is not owned and to V if it is owned and currently contains the raw value V. In other words, this is essentially a type of raw heap fragments.

Like memory locations, regions are dynamically allocated. Like a memory location, a region has at most one owner. Last, a region denotes a set of raw values. This is a monotonic set: It can only grow with time, and only the owner of a region can make it grow strictly. Thus, an appropriate type for resources that describe regions is *aMap* (*mSet raw_value*). This is a type of allocation maps where a region identifier r is mapped to a pair of a permission (either $\perp$, not owned, or tt, owned) and a set of raw values (the inhabitants of the region).

Finally, we fix the type of resources to be the product of the two types mentioned above. By construction, this type is equipped with the structure of a monotonic separation algebra.

The empty resource, written *void*, is a pair of two empty allocation maps. It is used when type-checking a source program (see, for instance, Lemma 3.1).

We introduce the following notation to project the various components of a resource R. We write *reflimit* R for the reference allocation limit. We write $R(l)$ for the permission (of type *option raw_value*) associated with a memory location l. We write *reglimit* R for the region allocation limit. We write $R(r)$ for the permission (of type *option unit*) associated with a region r. We write R *says* $V \in r$ to indicate that the raw value V is an inhabitant of region r. This notation is used in the typing rules that deal with references and regions (Section 7).

11 Typing: programmer-inaccessible constructs

11.1 Values

Figure 23 extends the typing judgement by giving typing rules for four forms of values that do not exist in source programs, but participate in the operational semantics of the instrumented calculus. These include memory locations $l\%v$, region inhabitants $[v]$, and capabilities for regions, $\{\vec{v}\}$ and $\{? :: \vec{v}\}$. Programmers need not understand these rules; they are used only as a part of the type soundness proof.

Memory locations. The first rule concerns memory locations. Recall that, in the instrumented calculus, a memory location takes the form $l\%v$, where l is the actual memory location and v is the value that it contains.

The first premise, $l <$ *reflimit* R, requires l to be a valid (allocated) memory location. The second premise splits the current resource in two parts. The sub-resource R_1 represents the ownership of the memory location l *per se*. Indeed, the premise $R_1(l) = \lfloor v \rfloor$ indicates that $R_1(l)$ must be defined and must be the erasure of the value v. That is, the sub-resource R_1, viewed as a raw store fragment, contains a binding for l, and, up to erasure, the value stored there is v. The sub-resource R_2 represents whatever is necessary to type-check the value v. This value is type-checked under empty multiplicity and type environments, which indicates that it must be closed.

$$\frac{l < \mathit{reflimit}\ R \qquad R_1 * R_2 = R \qquad R_2, \mathit{nil}, \mathit{nil} \vdash v : \tau \qquad R_1(l) = \lfloor v \rfloor}{R, M, E \vdash l\%v : \mathsf{ref}\ \tau} \qquad \frac{r < \mathit{reglimit}\ R \qquad R\ \mathit{says}\ \lfloor v \rfloor \in r \qquad v \text{ is closed}}{R, M, E \vdash [v] : [r]}$$

$$\frac{\begin{array}{c} r < \mathit{reglimit}\ R \\ \forall V,\ R\ \mathit{says}\ V \in r \Rightarrow \exists v, v \in \vec{v} \wedge \lfloor v \rfloor = V \\ \forall v,\ v \in \vec{v} \Rightarrow R\ \mathit{says}\ \lfloor v \rfloor \in r \\ R_1 * R_2 = R \qquad R_2 \vdash \vec{v} : \tau \qquad R_1(r) = \mathit{tt} \\ \kappa = \mathsf{S} \Rightarrow |\vec{v}| = 1 \end{array}}{R, M, E \vdash \{\vec{v}\} : \{\kappa\, r : \tau\}} \qquad \frac{\begin{array}{c} r_1 < \mathit{reglimit}\ R \qquad r_2 < \mathit{reglimit}\ R \\ \forall V,\ R\ \mathit{says}\ V \in r_1 \Rightarrow \exists v, v \in w :: \vec{v} \wedge \lfloor v \rfloor = V \\ \forall v,\ v \in w :: \vec{v} \Rightarrow R\ \mathit{says}\ \lfloor v \rfloor \in r_1 \\ R_1 * R_2 = R \qquad R_2 \vdash \vec{v} : \tau \qquad R\ \mathit{says}\ \lfloor w \rfloor \in r_2 \\ w \text{ is closed} \qquad R_1(r_1) = \mathit{tt} \end{array}}{R, M, E \vdash \{? :: \vec{v}\} : \{r_1 : \tau \setminus r_2\}}$$

Fig. 23. Typing rules for values: programmer-inaccessible constructs.

$$R \vdash \epsilon : \tau \qquad \frac{R_1, \mathit{nil}, \mathit{nil} \vdash v : \tau \qquad R_2 \vdash \vec{v} : \tau \qquad R_1 * R_2 = R}{R \vdash v :: \vec{v} : \tau}$$

Fig. 24. Typing rules for lists of closed values.

The fact that R_1 and R_2 are conjoined to obtain R reflects the idea that the type ref τ represents the ownership of the memory cell *and* the value that it contains. The fact that the type τ of the value v that is *currently* stored at this address is used to construct the type ref τ is characteristic of strong references. In a system of weak references, the typing rule for a memory location l would not have access to the value v, and would instead consult a store typing that maps memory locations to types (Harper, 1994; Wright & Felleisen, 1994).

Region inhabitants. The second rule in Figure 23 is used to type-check a region inhabitant. Its conclusion ascribes the type $[r]$, which means "inhabitant of region r", to a value of the form $[v]$. The underlying value v is arbitrary: it could be a memory location, a function, etc. The type of v is irrelevant: in fact, it is not even required (here) that v be well-typed. It is in the typing rule for capabilities that the inhabitants of a region are required to be well-typed.

The first premise, $r < \mathit{reglimit}\ R$, requires r to be a valid (allocated) region name. The second premise, $R\ \mathit{says}\ \lfloor v \rfloor \in r$, uses the resource R to interpret the region name r as a set of raw values, and requires the erasure of v to be a member of this set. (This notation was defined in Section 10.3.) The last premise indicates that v must be a closed value.

The information that is extracted out of R by this typing rule is the region allocation limit and the mapping of region names to sets of raw values. As per our definitions (Section 10), this information is duplicable, that is, it is still present in $\widehat{R}$. This remark explains why it is sound to consider $[\sigma]$ a duplicable type, and is indeed used in the subject reduction proof for the coercion at-bang. Furthermore, the premises $r < \mathit{reglimit}\ R$ and $R\ \mathit{says}\ \lfloor v \rfloor \in r$ remain valid as the allocation limit increases and as the set of inhabitants of the region r grows. In other words, this typing judgement remains valid if R evolves to a new resource R' such that $R \lhd R'$ holds. We refer to this property as monotonicity (Lemma 15.4).

Capabilities. The third rule in Figure 23 concerns a capability for a group region of the form $\{\vec{v}\}$, where $\vec{v}$ is a list of the region inhabitants. This capability receives the type $\{\kappa r : \tau\}$, where κ is a region kind (either S, for a singleton region, or G, for a group region), r is a region name, and τ is the common type of the region's inhabitants.

The first premise requires r to be a valid region name.

The somewhat impressive second and third premises simply express the fact that the raw values in the list $\lfloor \vec{v} \rfloor$ are exactly the raw values that (R says) inhabit the region r.

The next premise splits the current resource in two parts. The sub-resource R_1 represents the ownership of the region r *per se*. This is expressed by the premise $R_1(r) = tt$. The sub-resource R_2 represents whatever is necessary to type-check the values $\vec{v}$. This is expressed by the premise $R_2 \vdash \vec{v} : \tau$. This premise exploits an auxiliary judgement, whose definition is straightforward and appears in Figure 24. In short, this judgement indicates that each of the values in the list $\vec{v}$ has type τ under empty multiplicity and type environments and that the conjunction of the resources required to type-check these values is R_2.

The last premise, $\kappa = \mathsf{S} \Rightarrow |\vec{v}| = 1$, indicates that a singleton region must have exactly one inhabitant. In contrast, a group region can have any number of inhabitants.

The last rule in Figure 23 concerns a capability for a group region in which a hole has been punched. By this, we mean that the primitive operation focusgroup has been used to isolate a particular inhabitant of the group region. A capability for a punched region is diminished: It represents the ownership of the region *per se* as well as the ownership of all inhabitants *except* the one that has been focused on.

In the typing rule, the punched group region is r_1. The inhabitant that has been isolated is w. The remaining inhabitants are $\vec{v}$. The value w has been placed in a singleton region r_2.

The third and fourth premises express the fact that the raw values in the list $\lfloor w :: \vec{v} \rfloor$ are exactly the raw values that (R says) inhabit the group region r_1. Even though the capability $\{? :: \vec{v}\}$ over the region r_1 does not include the ownership of w, the value w is still considered an inhabitant of the region r_1. Regions can only grow with time: if w was once an inhabitant of r_1, then w is forever an inhabitant of r_1.

The antepenultimate premise, R *says* $\lfloor w \rfloor \in r_2$, requires w to be an inhabitant of the region r_2. This ensures that defocus-group is sound: When we relinquish the ownership of r_2, we will regain control over w, and will be able to reconstruct a full capability for the group region r_1.

11.2 Stores and configurations

In order to facilitate the statement of type soundness, we define typing judgements about stores and configurations.

The judgement that concerns stores does not involve types in any way. In short, as per Section 10.3, a resource R contains a raw store; furthermore, the image of a store s through erasure is a raw store; so a resource R and a store s are consistent with respect to one another if and only if they describe the same raw store.

Definition 11.1 *A store s, which one can also write "$m\,\mathsf{below}\,\ell$", is consistent with a resource R if and only if the following two conditions hold:*

1. *reflimit $R = \ell$, that is, the allocation limits of R and s coincide;*
2. *$\forall l,\ l < \ell \Rightarrow \lfloor m\, l \rfloor = R(l)$, that is, the raw contents of R and s coincide.* ♡

There is no requirement concerning the part of R that describes ghost state, that is, the part of R that records information about regions.

We write $R \vdash s$ when s is consistent with respect to R.

We now define what it means for a top-level configuration $s/\,t$ to be well-typed. We could perfectly well posit that $s/\,t$ is well-typed (without mentioning its type) if and only if $R \vdash s$ and $R, nil, nil \Vdash t : \tau$ hold for some resource R and some type τ. It seems however somewhat more instructive to define a three-place judgement, of the form $\vdash s/\,t : \tau$, which means that the configuration is well-typed and has type τ.

Definition 11.2 *The judgement $\vdash s/\,t : \tau$ is defined by the following rule:*

$$\frac{R \vdash s \qquad R, nil, nil \Vdash t : \tau \circ \vec{\theta}}{\vdash s/\,t : \tau}$$

This definition presents an interesting twist. We do not require the term t to have type τ: instead, we allow it to have type $\tau \circ \vec{\theta}$. In fact, we are building an (iterated) anti-frame rule into the typing rule for top-level configurations so that a configuration that claims to have type τ really has a more complex type inside. This allows us to consider that when an instance of the anti-frame rule is executed (that is, when a "let/hide" construct appears under an evaluation context), the anti-frame rule extrudes all the way up through the evaluation context until it reaches the top level, where it remains. Stating the definition in this manner allows us to give a simpler statement of subject reduction (Lemma 15.14).

12 Revelation

The operational semantics of the anti-frame rule, to be presented shortly (Section 13.2), is based on scope extrusion. As an instance of the anti-frame rule extrudes, its scope grows and eventually encompasses the entire program. Thus, whereas at type-checking time the hidden invariant θ is visible only within the "hide" construct, at run-time it becomes visible everywhere. We refer to this phenomenon as "revelation".

Technically, the code that initially lay outside of the "hide" construct must be rewritten so as to explicitly thread a capability of type θ (which it never uses, but passes to the code that initially lies inside of the “hide” construct). For instance, a first-order function v of type $\tau_1 \to \tau_2$, where τ_1 and τ_2 are base types, must be transformed into a function $\wr v \int$ of type $(\tau_1 * \theta) \to (\tau_2 * \theta)$, which accepts an extra argument and returns it (unmodified) as an extra result. In this particular case, the type $(\tau_1 * \theta) \to (\tau_2 * \theta)$ is equal to $(\tau_1 \to \tau_2) \otimes \theta$, and indeed, in the general case, a value v of type τ must be transformed into a value $\wr v \int$ of type $\tau \otimes \theta$.

It may seem odd to turn a function of one argument and one result into a function of two arguments and two results, and, in the process, to alter the code of the function. One must keep in mind that this technical trick takes place at the level of the instrumented calculus. At the level of the raw calculus, nothing happens: the values v and $\wr v \int$ are equal up to erasure.

Because the syntactic category of values depends on the syntactic categories of coercions and terms, these must be rewritten as well. Let us begin with coercions.

Lemma 12.1 (Revelation for coercions) *If the coercion c proves that τ_1 is a subtype of τ_2, then a coercion $\wr c \int$ proves that $\tau_1 \otimes \theta$ is a subtype of $\tau_2 \otimes \theta$.*

$$\frac{C \vdash c : \tau_1 \leqslant \tau_2}{C \otimes \theta \vdash \wr c \int : \tau_1 \otimes \theta \leqslant \tau_2 \otimes \theta} \qquad \heartsuit$$

In the above statement, if the environment C maps a coercion variable to the coercion type $\tau_1 \leqslant \tau_2$, then $C \otimes \theta$ maps this variable to the coercion type $\tau_1 \otimes \theta \leqslant \tau_2 \otimes \theta$. This statement is ultimately specialized for closed coercions, so both C and $C \otimes \theta$ are *nil*.

The coercion $\wr c \int$ is defined by induction over the structure of c. We omit this definition and mention only a couple of points. At function types, the tensor causes an extra argument to appear, so $\wr c_1 \to c_2 \int$ is defined as $(\wr c_1 \int * \mathsf{id}) \to (\wr c_2 \int * \mathsf{id})$. At a “tensor exchange” coercion, the tensors pile up: that is, $\wr \otimes\text{-}\mathsf{exch}_n \int$ is defined as $\otimes\text{-}\mathsf{exch}_{(n+1)}$. This is the reason why this coercion carries an integer index n and why its typing rule supports a vector of types $\vec{\theta}$. All other cases are trivial.

Let us now move on to values and state the two properties that $\wr v \int$ must satisfy. The application of tensor to a type environment, $E \otimes \theta$, is defined pointwise.

Lemma 12.2 (Revelation for values) *If v has type τ, then $\wr v \int$ has type $\tau \otimes \theta$. That is, $\wr v \int$ behaves like v, but preserves an additional invariant θ.*

$$\frac{R, M, E \vdash v : \tau}{R, M, (E \otimes \theta) \vdash \wr v \int : \tau \otimes \theta} \qquad \heartsuit$$

Lemma 12.3 *$\wr v \int$ and v coincide up to erasure.*

$$\lfloor \wr v \int \rfloor = \lfloor v \rfloor \qquad \heartsuit$$

The definition of $\wr v \int$ is straightforward. At coercion applications, revelation for coercions (Lemma 12.1) is used, that is, $\wr c\ v \int$ is defined as $\wr c \int\ \wr v \int$. The only interesting case is that of functions. An abstraction λt must be transformed so as to accept a pair of arguments: its original argument and a capability of type θ. The capability must then be threaded through the body t of the function so as to eventually produce a pair of results: the original result and a capability of type θ. Thus, the term t must be transformed too. In informal nominal notation, the definition of $\wr\lambda x.t\S$ is $\lambda z.\mathsf{let}\,(x,k) = z \;\mathsf{in}\; {}^{k}\wr t\S$[14]. The pair z is immediately deconstructed to obtain the original argument x and the capability k. Control is then transferred to the transformed function body ${}^{k}\wr t\S$.

Let us now discuss revelation for terms. This transformation is indexed with a variable k, which indicates under what name the extra capability is available. The transformed term ${}^{k}\wr t\S$ must thread this capability throughout every computation and eventually produce a pair of the original result and a capability. Thus, the two properties that ${}^{k}\wr t\S$ must satisfy are as follows.

Lemma 12.4 (Revelation for terms) *If the term t has type τ, then, under the assumption that k has type θ, the term ${}^{k}\wr t\S$ has type $(\tau\otimes\theta)*\theta$. That is, ${}^{k}\wr t\S$ behaves like t, but preserves an additional invariant θ, which is passed in the variable k and returned in the second component of the result. For the sake of readability, we show the statement only in the case where k is 0:*

$$\frac{R, M, E \Vdash t : \tau}{R, (M;1), (E \otimes \theta; \theta) \Vdash {}^{0}\wr t\S : \tau \circ \theta} \qquad \heartsuit$$

Lemma 12.5 *${}^{k}\wr t\S$ and t coincide up to erasure.*

$$\lfloor {}^{k}\wr t\S \rfloor = k\uparrow\lfloor t\rfloor \qquad \heartsuit$$

The definition of ${}^{k}\wr t\S$ is slightly more involved than that of $\wr v\S$. We describe only the key case and refer the reader to the Coq formalization (Pottier, 2012a, 2012b) for further details.

The key case is that of the anti-frame rule, that is, the definition of ${}^{k}\wr\mathsf{let}\, v \;\mathsf{in}\; \mathsf{hide}\, t\S$. The term t has access to an invariant τ that is invisible outside "hide". In order to satisfy revelation for terms (Lemma 12.4), we must transform "$\mathsf{let}\, v \;\mathsf{in}\; \mathsf{hide}\, t$" so as to reveal another invariant θ. The result will be a transformed "hide" construct within whose body two invariants are visible. Will these invariants still be known there as τ and θ? Not quite. We must transform these invariants so as to express the fact that each of them preserves the other. The new invariants must be τ' and θ', where τ' is "a version of τ that additionally preserves θ'", and (symmetrically) θ' is "a version of θ that additionally preserves τ'". In other words, we need the equations $\tau' \equiv \tau \otimes \theta'$ and $\theta' \equiv \theta \otimes \tau'$ to hold. These equations characterize a "commutative pair".

By the commutative pairs lemma (Lemma 6.12), these equations admit a solution. Furthermore, the coercion $\otimes$-exch_0 (Section 8.7) witnesses the fact that two types of the forms $(\cdot \otimes \tau) \otimes \theta'$ and $(\cdot \otimes \theta) \otimes \tau'$ are interconvertible, while the coercion $\circ$-exch witnesses the fact that two types of the forms $(\cdot \circ \tau) \circ \theta'$ and $(\cdot \circ \theta) \circ \tau'$ are

[14] In the de Bruijn notation, the definition of $\wr\lambda t\S$ is $\lambda(\mathsf{letpair}_{\mathsf{Phy,Log}}\, 0 \;\mathsf{in}\; 2\uparrow^{0}\wr t\S)$.

interconvertible. Both of these properties are required here, and indeed we are able to offer a definition of ${}^{k}\wr \mathsf{let}\, v \,\mathsf{in}\, \mathsf{hide}\, t \wr$ that exploits both of these coercions. (This is the fundamental reason why the coercion $\otimes\text{-}\mathsf{exch}_n$ is introduced.) For further insights, the reader is referred to the conference paper (Pottier, 2008), which offers a detailed account of this particular proof case, or to Schwinghammer *et al.*'s paper (2010), which exploits commutative pairs in an analogous manner.

13 Semantics of the instrumented calculus

The instrumented calculus plays an important role in the type soundness proof. It represents the main artifact that we have to invent. Its syntax and semantics incorporate a number of design choices and compromises that allow controlling where and under what form a certain number of proof obligations arise.

We wish to give the reader an opportunity to understand the basic design principles for the semantics of the instrumented calculus. The complete definition of the semantics, however, consists of a rather large number of rules (over 60 value reduction rules and 25 term reduction rules!), so we place it in Appendix. In the following, we explain only a few of the most important rules.

The semantics consists of two relations: the reduction of values, which takes the form $v_1 \longrightarrow v_2$, and the reduction of terms (or, more accurately, of configurations), which takes the form $s_1 / t_1 \xrightarrow{h} s_2 / t_2$. The integer superscript h, which has to do with the extrusion of the anti-frame rule, is explained later on.

13.1 Reduction of values

The reduction of values gives operational meaning to coercions. There is usually one reduction rule per coercion form, although a few coercion forms do not have any reduction rules and a few have more than one.

The purpose of value reduction is to bring values to a canonical form. A canonical value is one that does not contain any coercion applications, except perhaps within abstractions λv and region inhabitants $[v]$. We omit the inductive definition of this notion.

Many of the reduction rules are very simple. For instance, when applied to a value v, the identity coercion id vanishes, while a composite coercion $c_1 ; c_2$ reduces to a couple of nested coercion applications (Figure A 1).

$$\mathsf{id}\; v \longrightarrow v \qquad\qquad (c_1 ; c_2)\; v \longrightarrow c_2\; (c_1\; v)$$

The coercions that witness the congruence of subtyping typically push themselves down into the structure of values (Figure A 2). For instance, the coercion ${}_{\iota_1}(c_1 \times c_2)_{\iota_2}$, applied to a pair ${}_{\iota_1}(v_1, v_2)_{\iota_2}$, reduces by pushing the coercions c_1 and c_2 down respectively into the first and second components of the pair.

$${}_{\iota_1}(c_1 \times c_2)_{\iota_2}\; {}_{\iota_1}(v_1, v_2)_{\iota_2} \longrightarrow {}_{\iota_1}(c_1\; v_1, c_2\; v_2)_{\iota_2}$$

This also works for memory locations, even though the value reduction relation does not have access to the store. Indeed, a memory location l carries its content v, so the coercion $\mathsf{ref}\ c$ is able to reduce by pushing c down into the reference and applying it to v:

$$(\mathsf{ref}\ c)\ (l\%v) \longrightarrow l\%(c\ v)$$

Similarly, because a capability for a region carries a list of the region inhabitants, the cercion $\{c\}$ is able to reduce by pushing c down into the region and applying it to every inhabitant:

$$\{c\}\ \{\vec{v}\} \longrightarrow \{c\ \vec{v}\}$$

(We write $c\ \vec{v}$ for the pointwise application of the coercion c to the list $\vec{v}$.)

The coercions that introduce, eliminate, and move quantifiers around reduce in a fairly predictable and uninteresting way (Figures A 3 and A 4). Here, for instance, are the reduction rules for the coercions that introduce and eliminate a universal quantifier:

$$\forall\mathsf{I}\ v \longrightarrow \Lambda v \qquad\qquad \forall\mathsf{E}\ (\Lambda v) \longrightarrow v$$

The coercions that concern the "!" modality are also fairly mundane (Figure A 5). Here are a couple of examples:

$$\mathsf{dereliction}\ (!\,v) \longrightarrow v \qquad\qquad \mathsf{bang\text{-}idempotent}\ (!\,v) \longrightarrow !\,(!\,v)$$

It may be worth noting that the coercions whose type takes the form $\tau \leqslant \bot$, namely, $\mathsf{bang\text{-}ref}$, $\mathsf{bang\text{-}regioncap}$, and $\mathsf{bang\text{-}regioncappunched}$, do not have a reduction rule. This means that in the progress lemma (Lemma 15.10) one will need to prove that these coercions cannot be applied to a canonical value.

The coercion $\mathsf{defocus}\,\pi$ behaves in a more complex and original manner. This coercion expects its argument to be a pair of a value v_1 and a capability for the region whose single inhabitant is a value v_2. Its reduction rule is the following (Figure A 6):

$$\frac{v_1(\pi) = [v_1''] \qquad v_1(\pi \mapsto v_2) = v_1'}{(\mathsf{defocus}\,\pi)\ {}_{\mathsf{Phy}}(v_1, \{v_2 :: \epsilon\})_{\mathsf{Log}} \longrightarrow v_1'}$$

The first premise checks that the path π within the value v_1 exists and leads to a value of the form "region inhabitant". The second premise plugs the value v_2 in its place to obtain the reduct v_1'. The auxiliary predicates $v(\pi) = w$ and $v(\pi \mapsto w) = v'$ are defined in Figures A 10 and A 11 in Appendix.

Remark 13.1 The base case in the definition of the "plugging" predicate explicitly requires the old value (which is discarded) and the new value (which is plugged in its place) to coincide up to erasure. As a result, $v(\pi \mapsto w) = v'$ implies $\lfloor v \rfloor = \lfloor v' \rfloor$. This ensures that the left- and right-hand sides of the above reduction rule coincide up to erasure (Lemma 13.3).

By building this requirement into "plugging", one makes it more difficult to apply the above reduction rule. This translates to a proof obligation in the progress lemma for values (Lemma 15.10). Fortunately, there a typing hypothesis is available. For some region r and type τ, the value $[v_1'']$ has type $[r]$, while the capability $\{v_2 :: \epsilon\}$

has type $\{\mathsf{S}\,r : \tau\}$. This implies $\lfloor v_1'' \rfloor = \lfloor v_2 \rfloor$, which shows that "plugging" is indeed permitted. $\diamond$

Remark 13.2 The first premise in the above reduction rule requires that the path π within the value v_1 lead to a value of the form $[v_1'']$. Anticipating the definition of well-layeredness (Section 14), let us note that $[v_1'']$ is a physical value. Furthermore, the value v_2 (which we plug in its place) must be physical too, because it appears as an inhabitant in the capability $\{v_2 :: \epsilon\}$. Hence, this reduction rule preserves well-layeredness (Lemma 14.1). $\diamond$

The coercion defocus-group expects its argument to be a pair of a capability for a singleton region, whose inhabitant is v, and a capability for a group region, which has been deprived of the ownership of one inhabitant, and whose remaining inhabitants are $\vec{v}$. A typing assumption, not explicitly expressed here, implies that the value $|v|$ is (up to erasure) the value whose ownership has been taken away. The idea is to return it, and indeed the reduct is just a capability for a complete group region, $\{v :: \vec{v}\}$.

$$\mathsf{defocus\text{-}group}\ {}_{\mathsf{Log}}(\{v :: \epsilon\}, \{? :: \vec{v}\})_{\mathsf{Log}} \longrightarrow \{v :: \vec{v}\}$$

Most of the coercions that deal with the "movement of stars" have straightforward reduction rules (Figure A 7), with one exception, namely, "tensor exchange" $\otimes$-exch$_n$. The special case where n is 0 is sufficient to illustrate the main ideas. Recall (Section 8.7) that the coercion $\otimes$-exch$_0$ has type $(\tau \otimes \alpha) \otimes \beta' \leqslant (\tau \otimes \beta) \otimes \alpha'$, provided that (α, β) and (α', β') form a commutative pair. Because the type τ is arbitrary, this coercion must be prepared to accept any kind of value as an argument. That is, its argument could be a λ-abstraction, a pair, a memory location, etc. We need one reduction rule for each of these situations. Thus, the number of reduction rules for $\otimes$-exch$_0$ is equal to the number of forms of canonical values.

Among these rules, only the one that concerns λ-abstractions performs non-trivial work. The rule is as follows:

$$\otimes\text{-}\mathsf{exch}_0\ (\lambda t) \longrightarrow \lambda([\circ\text{-}\mathsf{exch}\ 0/1](\circ\text{-}\mathsf{exch}\ (0{\uparrow}t)))$$

The function λt has type $((\tau_1 \to \tau_2) \otimes \alpha) \otimes \beta'$, which by the equational theory of tensor is equal to $((\tau_1 \circ \alpha) \circ \beta') \to ((\tau_2 \circ \alpha) \circ \beta')$. On the other hand, the coercion application $\otimes$-exch$_0$ (λt) is supposed to have type $((\tau_1 \to \tau_2) \otimes \beta) \otimes \alpha'$, which by the equational theory of tensor is equal to $((\tau_1 \circ \beta) \circ \alpha') \to ((\tau_2 \circ \beta) \circ \alpha')$. Thus, we see that the effect of $\otimes$-exch$_0$ should be to pre- and post-compose the function λt with the coercion $\circ$-exch (which was defined in Lemma 8.7). This is done in the right-hand side of the above reduction rule.

The other reduction rules for $\otimes$-exch$_n$ simply push it down into the structure of values. For instance, here is how this coercion acts upon a pair:

$$\otimes\text{-}\mathsf{exch}_n\ {}_{\iota_1}(v_1, v_2)_{\iota_2} \longrightarrow {}_{\iota_1}(\otimes\text{-}\mathsf{exch}_n\ v_1, \otimes\text{-}\mathsf{exch}_n\ v_2)_{\iota_2}$$

Recursive coercions reduce by unfolding (Figure A 8):

$$\frac{c\ \text{contractive in}\ 0}{(\mu c)\ v \longrightarrow ([\mu c/0]c)\ v}$$

Value reduction is permitted under all contexts, except inside λ-abstractions, which suspend computation, and within values of the form $\lfloor v \rfloor$, where value reduction would serve no purpose (Figure A 9).

This concludes our presentation of the reduction of values.

This relation satisfies the following two properties. Note that neither of them requires a well-typedness hypothesis.

Lemma 13.3 *Value reduction has no computational content. The image of a value reduction step through erasure is zero reduction steps.*

$$\frac{v_1 \longrightarrow v_2}{\lfloor v_1 \rfloor = \lfloor v_2 \rfloor} \qquad \heartsuit$$

Lemma 13.4 *The reduction of values terminates.* $\heartsuit$

The proof of the former property is fairly easy: as noted in Remark 13.1, the semantics is designed with this property in mind.

The proof of the latter property is significantly more involved. The difficulty is mainly due to recursive coercions. In their absence, we believe that a simple argument based on a recursive path ordering would work. In their presence, the best we could do was map a value to a multiset of coercion application descriptors, where the descriptor associated with a coercion application $c\ v$ is a (lexicographically ordered) pair of the weight of v and the measure of c, for appropriate notions of weight and measure (described below), and prove that every reduction rule causes this multiset to decrease.[15]

The weight of a value counts the number of "physical" constructors (namely, Phy/Phy pairs and memory locations) that appear in it. Because "logical" constructors are not counted, they can be rearranged without affecting the weight of a value. Thus, when one coercion takes a reduction step, the descriptors associated with the other coercions that might exist elsewhere in the value are not affected.

The measure of a coercion counts the number of constructors that appear in it, but this count stops at "physical" constructors (namely, λ-abstractions, Phy/Phy pairs, and memory locations): it does not examine their children. This definition is designed so that unfolding a contractive recursive coercion causes its measure to decrease, that is, the measure of $[\mu c/0]c$ is strictly less than the measure of μc. (The notion of a contractive coercion was defined in Section 8.8.)

What happens when a coercion propagates down into a "physical" constructor? Consider, for instance, the second rule of Figure A 2, where a coercion propagates down into a pair. Suppose ι_1 and ι_2 are Phy. The measure of the coercion does not necessarily decrease because the measure of ${}_{\mathsf{Phy}}(c_1 \times c_2)_{\mathsf{Phy}}$ is 1, whereas the measure of c_1 and c_2 is arbitrary. However, the weights of v_1 and v_2 are less than the weight of ${}_{\mathsf{Phy}}(v_1, v_2)_{\mathsf{Phy}}$. Thus, this reduction step replaces one coercion descriptor with two smaller descriptors. The multiset of the coercion descriptors decreases.

[15] In fact, the $\otimes$-exch_n reduction rules preserve this multiset, so an additional argument, based on a simple polynomial interpretation, is required.

What happens when a coercion propagates down into a "logical" constructor? Consider, for instance, the third rule of Figure A 2, where a coercion propagates down into a polymorphic value. The weight of v is equal to the weight of Λv, and the measure of c is less than the measure of $\forall c$. Thus, this reduction step replaces one coercion descriptor with one smaller descriptor. The multiset of the coercion descriptors decreases.

This is about as much as we can say about this proof. For more details, the reader is referred to the Coq formalization (Pottier, 2012a, 2012b).

Remark 13.5 This argument takes up about 1,000 lines, which is more than we would like. It is also somewhat brittle. For instance, defocus-dup π should ideally be a coercion, as opposed to a primitive operation. However, because this operation duplicates a value, it causes an increase in the weight. We have been unable to prove that value reduction terminates if defocus-dup π is made a coercion. We view this as a shortcoming of our proof technique. ◇

13.2 Reduction of terms

The reduction judgement for configurations takes the form $s_1/t_1 \xrightarrow{h} s_2/t_2$. The integer index h indicates how many instances of the anti-frame rule are being executed, that is, how many new invariants are being installed. In most reduction rules, h is zero. In the reduction rule associated with the "hide" construct, h is one. In the rule that allows reduction under an evaluation context, h is arbitrary. This rule uses h to adapt the evaluation context to work in the presence of the new invariants.[16]

The reduction rules for functions, pairs, "let!", and "unpack" are unsurprising (Figure A 12). Another set of rules permits the reduction of values wherever they appear (Figure A 13). When a construct appears both in the syntax of terms and in the syntax of values (this is the case, for instance, of Λ-abstractions), there is a rule for reducing one form to the other (also in Figure A 13; these rules appear to do nothing due to our ambiguous notation).

A group of reduction rules give the semantics of the primitive operations on regions (Figure A 14). For instance, the application of focus π to a value v_1 reduces as follows:

$$\frac{v_1(\pi) = v_2 \qquad v_1(\pi \mapsto [v_2]) = v_1'}{s/\,(\mathsf{focus}\,\pi)\; v_1 \xrightarrow{0} s/\,\mathsf{pack}_{\,\mathsf{Phy}}(v_1', \{v_2 :: \epsilon\})_{\mathsf{Log}}}$$

The first premise checks that the path π within the value v_1 exists and leads to a value v_2. The second premise plugs $[v_2]$ in its place, yielding a new value v_1'. The reduct is a pair of v_1' and a capability for a region whose single inhabitant is v_2. This pair is wrapped in a "pack" construct because the freshly created region is existentially quantified. This reduction rule is by no means surprising: once the

[16] In our system, h is always zero or one, but it seemed more general to make an integer index, rather than a Boolean index.

typing rule for $\mathsf{focus}\,\pi$ is fixed (Figure 13) and the spirit of the instrumented calculus is understood, it does not take much ingenuity to predict that the reduction rule must look like this.

A "hide" construct reduces as follows (Figure A 16):

$$s/\,\mathsf{let}\,v\,\mathsf{in}\ \mathsf{hide}\,t \xrightarrow{1} s/\,[\wr v \wr/0]t$$

We wrote earlier (Section 5.1) that, roughly speaking, the semantics of "let $\mathsf{in}\ \mathsf{hide}\,vt$" is just that of a normal "let" definition, that is, the value v is substituted for the variable 0, which is the one and only variable in scope within t. We now see that, in fact, we substitute $\wr v \wr$ for the variable 0. In view of the typing rule associated with "hide" (Figure 11), this makes sense: The value v has type τ_1, while the term t is type-checked under the assumption that the variable 0 has type $\tau_1 \otimes \theta$. Substituting v for 0 would not make sense, but substituting $\wr v \wr$ for 0 does, according to the revelation lemma for values (Lemma 12.2).

One might informally sum this up as follows. The hidden invariant is initially unknown to the value v, because v lies outside of the scope of the "hide" construct. As v is substituted for the variable 0, v enters the scope of the "hide" construct, so the invariant is revealed to it, and v must be adapted.

Remark 13.6 We may now explain why we adopt the construct "$\mathsf{let}\,v\,\mathsf{in}\ \mathsf{hide}\,t$", where only one variable is in scope within t. As noted earlier (Remark 5.2), a more natural alternative would have been to use a construct of the form "$\mathsf{hide}\,t$", without any restrictions on the free variables of t. (This is effectively the approach adopted in the conference paper (Pottier, 2008).) However, we would then have been forced to adopt a non-standard notion of substitution, whereby substituting into a "hide" construct performs revelation on-the-fly:

$$[v/k](\mathsf{hide}\,t) = \mathsf{hide}\,([\wr v \wr/k]t)$$

In principle, this approach should work. It would, however, make the definitions of revelation and substitution mutually recursive, which seems unpleasant. In effect, our approach delays all substitutions into the "hide" construct until it is executed.

Another motivation for our approach is that the definition of revelation for the "hide" construct requires applying a suitable coercion to every free variable. This is made significantly simpler if there is just one such variable. ◇

The semantics of "$\mathsf{let}\,v\,\mathsf{in}\ \mathsf{hide}\,t$" might seem surprising in that the left- and right-hand sides of the reduction rule do not have the same type. Indeed, if the redex has type τ_2, then the reduct has type $\tau_2 \circ \theta$, where θ is the "hidden" invariant. Might this break the subject reduction lemma? No, because we are careful to formulate this lemma in an appropriate manner (Lemma 15.13). However, this does mean that the evaluation context must adapt. The fact that the parameter h takes the value 1 can be understood as a signal that an invariant is being revealed and that the evaluation context must adapt. This will be evident shortly.

We now come to the rule that permits reduction under an evaluation context. The syntax of evaluation contexts is as follows:

$$\mathscr{E} ::= v\ [] \mid {}_{\mathsf{Phy}}([], v)_{\mathsf{Log}} \mid c\ []$$

That is, reduction is permitted within the argument of a function, within an application of the frame rule, and within the argument of a coercion.

We write $\wr\mathscr{E}\wr^h$ for h successive applications of revelation[17] to the context $\mathscr{E}$. So in the particular case where h is zero, the rule that permits reduction under an evaluation context has a standard appearance:

$$\frac{s_1 / t_1 \xrightarrow{0} s_2 / t_2}{s_1 / \mathscr{E}[t_1] \xrightarrow{0} s_2 / \mathscr{E}[t_2]}$$

In the general case where h is arbitrary, the evaluation context must adapt to the fact that, in the transition from t_1 to t_2, the term may decide to reveal a number of invariants. Thus, the general form of the rule for reduction under a context (Figure A 17) is

$$\frac{s_1 / t_1 \xrightarrow{h} s_2 / t_2}{s_1 / \mathscr{E}[t_1] \xrightarrow{h} s_2 / \wr\mathscr{E}\wr^h[t_2]}$$

The connection between $\mathscr{E}$ and $\wr\mathscr{E}\wr$ in terms of type-checking will be made explicit further on (Lemma 15.12). The connection between them in terms of erasure is as expected: $\mathscr{E}$ and $\wr\mathscr{E}\wr$ are equal up to erasure. We do not explicitly state this property: instead, we establish and use it on-the-fly inside the proof of the simulation lemma (Lemma 15.18).

Reduction is also permitted under a type abstraction, with a caveat: this is sound only when h is zero. For this reason, we do not make $\Lambda[]$ an evaluation context: instead, we provide the following reduction rule, where h must be zero (Figure A 17):

$$\frac{s_1 / t_1 \xrightarrow{0} s_2 / t_2}{s_1 / \Lambda t_1 \xrightarrow{0} s_2 / \Lambda t_2}$$

What would go wrong if one allowed h to be non-zero? This corresponds to a situation where a "hide" construct is executed under a type abstraction that binds a type variable. Then the reduction rule would dictate that the invariant must be revealed outside the Λ-abstraction. However, if the invariant refers to the type variable that is bound by this abstraction, this is impossible! This reduction rule, generalized to an arbitrary h, would break the subject reduction lemma.

This is not a surprise. It is analogous to the well-known dangerous interaction between weak references and polymorphism. In our setting, weak references can be encoded in terms of strong references and hidden state, and it becomes apparent that only the latter interacts in a dangerous way with polymorphism.

The problem is easily avoided. The above reduction is restricted so as to satisfy subject reduction. There remains to guarantee that progress is also satisfied, that is, to ensure that one never attempts to execute a "hide" construct under a type

[17] Since evaluation contexts $\mathscr{E}$ are built out of values v and coercions c, it is straightforward to define revelation for evaluation contexts $\wr\mathscr{E}\wr$ in terms of revelation for values $\wr v\wr$ and coercions $\wr c\wr$. The definition is omitted.

abstraction if the hidden invariant refers to the Λ-bound type variable. We adopt a coarse sufficient condition: The user must choose between having both the anti-frame rule and the value restriction, or neither of them.

14 Well-layeredness

We have argued at the beginning of the paper in favor of a simple and unambiguous distinction between the "physical" and the "logical" layers, that is, between what exists at runtime and what is erased. At the cost of annotating pairs and the unit value with layers, we have been able to view erasure as a function of terms to raw terms. However, there exist terms whose erasure does not make sense. For instance, a function that accepts a pair of an ordinary value and a capability and attempts to return the second component of this pair does not make sense, because a function must produce a physical result, whereas a capability is logical.

The well-layeredness judgement tells which values and terms do make sense. Like the subtyping and typing judgements, it is a part of the definition of the system. In spite of this, we have delayed its presentation until now because it is so simple as to be rather uninteresting.

The well-layeredness judgement about values takes the form $I \vdash_{\mathrm{wl}} v : \iota$. This judgement asserts that, under the *layer environment* I, which maps variables to layers, the value v is well-layered and belongs to the layer ι. The well-layeredness judgement about terms takes the form $I \vdash_{\mathrm{wl}} t$. This judgement asserts that, under the layer environment I, the term t is well-layered. A term always belongs to the physical layer: at runtime, its evaluation must produce an actual result.

We use an auxiliary well-layeredness judgement about primitive operations, which takes the form $\vdash_{\mathrm{wl}} p$. This judgement asserts that the operation p transforms a physical value to a physical value. (One could perhaps aim at greater generality and allow primitive operations to have arbitrary input and output layers. This was deemed good enough.) Because the primitive operations $\mathsf{defocus\text{-}dup}\,\pi$ and $\mathsf{focus}\,\pi$ refer to a path π, we further use an auxiliary well-layeredness judgement about paths. This judgement takes the form $\vdash_{\mathrm{wl}} \pi : \iota_1 \to \iota_2$ and means that the path π leads from a value rooted in the layer ι_1 to a sub-value in the layer ι_2. The inductive definitions of these two auxiliary judgements are omitted; the reader is referred to the Coq formalization (Pottier, 2012a, 2012b).

The inductive definition of well-layeredness for values and terms appears in Figures 25 and 26. We briefly comment on some aspects. The argument of a λ-abstraction must be a physical value; its result is physical as well, since every well-layered term is physical; and a λ-abstraction is itself a physical value. The first and second components of a pair must respectively inhabit the layers ι_1 and ι_2 that decorate the pair. The pair itself inhabits the layer $\iota_1.\iota_2$, defined as Phy if at least one of ι_1 and ι_2 is Phy and Log otherwise. Universal quantification, existential quantification, and the "!" modality, as well as subtyping, are mechanisms that make sense in both physical and logical layers. The constructs related to references and regions, on the other hand, are layer-specific. A memory location is physical, and its content must be a physical value. A capability for a region is logical. Every

$$\frac{I(x) = \iota}{I \vdash_{\mathrm{wl}} x : \iota} \qquad \frac{I\,;\mathsf{Phy} \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \lambda t : \mathsf{Phy}} \qquad \frac{I \vdash_{\mathrm{wl}} v_1 : \iota_1 \qquad I \vdash_{\mathrm{wl}} v_2 : \iota_2 \qquad \iota = \iota_1.\iota_2}{I \vdash_{\mathrm{wl}} {}_{\iota_1}(v_1, v_2)_{\iota_2} : \iota} \qquad I \vdash_{\mathrm{wl}} ()_\iota : \iota$$

$$\frac{I \vdash_{\mathrm{wl}} v : \iota}{I \vdash_{\mathrm{wl}} \Lambda v : \iota} \qquad \frac{I \vdash_{\mathrm{wl}} v : \iota}{I \vdash_{\mathrm{wl}} \mathsf{pack}\ v : \iota} \qquad \frac{I \vdash_{\mathrm{wl}} v : \iota}{I \vdash_{\mathrm{wl}} c\ v : \iota} \qquad \frac{I \vdash_{\mathrm{wl}} v : \iota}{I \vdash_{\mathrm{wl}} !\,v : \iota} \qquad \frac{nil \vdash_{\mathrm{wl}} v : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} l\%v : \mathsf{Phy}}$$

$$\frac{nil \vdash_{\mathrm{wl}} \vec{v} : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} \{\vec{v}\} : \mathsf{Log}} \qquad \frac{nil \vdash_{\mathrm{wl}} \vec{v} : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} \{? :: \vec{v}\} : \mathsf{Log}} \qquad \frac{I \vdash_{\mathrm{wl}} v : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} [v] : \mathsf{Phy}}$$

Fig. 25. Well-layeredness: values.

$$\frac{I \vdash_{\mathrm{wl}} v : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} v} \qquad \frac{\begin{array}{c}I \vdash_{\mathrm{wl}} v : \mathsf{Phy}\\ I \vdash_{\mathrm{wl}} t\end{array}}{I \vdash_{\mathrm{wl}} v\ t} \qquad \frac{I \vdash_{\mathrm{wl}} v : \mathsf{Log} \qquad I \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} {}_{\mathsf{Phy}}(t, v)_{\mathsf{Log}}} \qquad \frac{I \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \Lambda t} \qquad \frac{I \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} c\ t}$$

$$\frac{I \vdash_{\mathrm{wl}} v : \iota \qquad I\,;\iota \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \mathsf{unpack}\,v\,\mathsf{in}\,t} \qquad \frac{I \vdash_{\mathrm{wl}} v : \iota \qquad I\,;\iota \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \mathsf{let}!\,v\,\mathsf{in}\,t} \qquad \frac{I \vdash_{\mathrm{wl}} v : \iota_1.\iota_2 \qquad I\,;\iota_1\,;\iota_2 \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \mathsf{letpair}_{\iota_1,\iota_2}\,v\,\mathsf{in}\,t}$$

$$\frac{\vdash_{\mathrm{wl}} p \qquad I \vdash_{\mathrm{wl}} v : \mathsf{Phy}}{I \vdash_{\mathrm{wl}} p\ v} \qquad \frac{I \vdash_{\mathrm{wl}} v : \mathsf{Phy} \qquad I\,;\mathsf{Phy} \vdash_{\mathrm{wl}} t}{I \vdash_{\mathrm{wl}} \mathsf{let}\,v\,\mathsf{in}\,\mathsf{hide}\,t}$$

Fig. 26. Well-layeredness: terms.

inhabitant of a region must be a physical value. At a letpair construct, the layer annotations ι_1 and ι_2 are used to extend the layer environment.

A store s is well-layered if and only if every value v in the image of s satisfies $nil \vdash_{\mathrm{wl}} v : \mathsf{Phy}$. A configuration s/t is well-layered (and we write $\vdash_{\mathrm{wl}} s/t$) if and only if the store s and the closed term t are well-layered.

The next lemmas guarantee that if a source program is well-layered, then all of its reducts are well-layered.

Lemma 14.1 *Value reduction preserves well-layeredness.*

$$\frac{v_1 \longrightarrow v_2 \qquad nil \vdash_{\mathrm{wl}} v_1 : \iota}{nil \vdash_{\mathrm{wl}} v_2 : \iota} \qquad \heartsuit$$

Lemma 14.2 *Reduction preserves well-layeredness.*

$$\frac{s_1/t_1 \xrightarrow{h} s_2/t_2 \qquad \vdash_{\mathrm{wl}} s_1/t_1}{\vdash_{\mathrm{wl}} s_2/t_2} \qquad \heartsuit$$

Well-layeredness is exploited in the lemmas that relate the instrumented calculus and the raw calculus (Lemmas 15.18 and 15.20).

Remark 14.3 The separation between well-typedness and well-layeredness may seem, in hindsight, somewhat awkward and also sometimes limiting. For instance, we are not able to add the subtyping axiom $!\,\tau \leqslant (!\,\tau) * (!\,\tau)$, which is supposed to mean that a duplicable capability can be duplicated. Here is why. The natural reduction rule that accompanies this axiom reduces a value v to a pair ${}_{\mathsf{Log}}(v, v)_{\mathsf{Log}}$. Because this pair is a logical value (its erasure is $\bullet$), this reduction rule preserves well-layeredness if and only if v is also a logical value. Unfortunately, we seem to lack the means of requiring that v be a logical value. We would like to restrict the above subtyping axiom by adding the side condition that τ must be the type of a logical value, as opposed to the type of a physical value; but our types do not distinguish between physical and logical values (in particular, a type variable stands for a completely arbitrary type).

In the system as it stands, a duplicable capability can be duplicated by other means; in particular, a variable of type $!\,\tau$ can be used multiple times. Still, the absence of this subtyping axiom is regrettable.

One alternative approach might be to introduce a modality that denotes "erasability", in the same way that the "!" modality denotes duplicability. Another alternative approach would be to use kinds, instead of modalities, to distinguish which types are duplicable (as opposed to affine) and which are logical (as opposed to physical). This was the approach of Charguéraud and Pottier's paper (2008). Introducing kinds has a certain cost, but this approach seemed to work well on paper. $\diamond$

15 Type soundness

We are now able to march toward the type soundness theorem. We begin with a series of simple auxiliary lemmas. For the sake of readability, we simplify the statement of these lemmas by giving only a statement about terms (there is usually an identical statement about values) and by specializing the statement to the case where some type variable or term variable of interest is 0.

The first three auxiliary lemmas are weakening properties.

Lemma 15.1 (Type variable weakening) *Typing is stable under introduction of a new type variable.*

$$\frac{R, M, E \Vdash t : \tau}{R, M, (0{\uparrow}E) \Vdash t : 0{\uparrow}\tau} \qquad \heartsuit$$

Lemma 15.2 (Term variable weakening) *Typing is stable under introduction of a new term variable.*

$$\frac{R, M, E \Vdash t : \theta}{R, (M\,;m), (E\,;\tau) \Vdash 0{\uparrow}t : \theta} \qquad \heartsuit$$

The more resources one has access to, the better. Similarly, the more variables one has permission to use, the better. This property holds because the system is affine, as opposed to linear.

Lemma 15.3 (Resource and multiplicity weakening) *Typing is stable under addition of resources and multiplicities.*

$$\frac{R_1, M_1, E \Vdash t : \tau \qquad R_1 * R_2 = R \qquad M_1 * M_2 = M}{R, M, E \Vdash t : \tau} \qquad \heartsuit$$

The next auxiliary lemma is a monotonicity property. It is used, in particular, in the proof of Lemma 15.12, to prove that the (passive) evaluation context remains well-typed, while the (active) term in the hole makes a step. Its proof relies in Axioms 10.14 and 10.15.

Lemma 15.4 (Monotonicity) *Typing is stable under the passive execution ordering* $\lhd$.

$$\frac{R_1, M, E \Vdash t : \tau \qquad R_1 \lhd R_2}{R_2, M, E \Vdash t : \tau} \qquad \heartsuit$$

Next come the substitution properties. The first of these concerns type variables.

Lemma 15.5 (Type substitution) *Typing is stable under substitution of a type for a type variable.*

$$\frac{R, M, (0\uparrow E) \Vdash t : \tau}{R, M, E \Vdash t : [\theta/0]\tau} \qquad \heartsuit$$

The next three concern term variables. We distinguish three lemmas, depending on the multiplicity of the variable that is being substituted away. This multiplicity is one of 0, 1, and ∞. The case where it is 0 is trivial, since the variable is then unused. It is nevertheless worth spelling it out, as it is used in the proof of the next case. The cases where it is 1 and ∞ respectively correspond to Barber's linear cut and intuitionistic cut (1996).

The constraints that bear on the value v, which is introduced by the substitution, depend on this multiplicity. If it is 0, the value v is arbitrary. If it is 1, the value v must be well-typed, and the assumptions R_2, M_2, E that are used to type-check v must be compatible with the assumptions R_1, M_1, E that are used to type-check the term t. If it is ∞, the resource R_2 and the multiplicity environment M_2 are further required to be duplicable.

Lemma 15.6 (Unused value substitution) *Typing is stable under substitution of an arbitrary value for a variable of multiplicity* 0.

$$\frac{R_1, (M_1; 0), (E; \theta) \Vdash t : \tau}{R_1, M_1, E \Vdash [v/0]t : \tau} \qquad \heartsuit$$

Lemma 15.7 (Affine value substitution) *Typing is stable under substitution of a well-typed value for a variable of multiplicity* 1.

$$\frac{\begin{array}{c} R_1, (M_1; 1), (E; \tau_1) \Vdash t : \tau_2 \\ R_2, M_2, E \vdash v : \tau_1 \\ R_1 * R_2 = R \qquad M_1 * M_2 = M \end{array}}{R, M, E \Vdash [v/0]t : \tau_2} \qquad \heartsuit$$

Lemma 15.8 (Unrestricted value substitution) *Typing is stable under substitution of a well-typed, duplicable value for a variable of multiplicity ∞.*

$$\frac{\begin{array}{c} R_1, (M_1\,;\infty), (E\,;\tau_1) \Vdash t : \tau_2 \\ R_2 * R_2 = R_2 \qquad M_2 * M_2 = M_2 \\ R_2, M_2, E \vdash v : \tau_1 \\ R_1 * R_2 = R \qquad M_1 * M_2 = M \end{array}}{R, M, E \Vdash [v/0]t : \tau_2} \qquad \heartsuit$$

This concludes the series of auxiliary lemmas, and brings us to the core statements of type soundness for the instrumented calculus. We begin with subject reduction and progress statements for the reduction of values.

Lemma 15.9 (Subject reduction for values) *Value reduction preserves well-typedness.*

$$\frac{R, nil, nil \vdash v_1 : \tau \qquad v_1 \longrightarrow v_2}{R, nil, nil \vdash v_2 : \tau} \qquad \heartsuit$$

Lemma 15.10 (Progress for values) *A well-typed value is canonical or reduces.*

$$\frac{R, nil, nil \vdash v : \tau}{v\ canonical \vee (\exists w,\ v \longrightarrow w)} \qquad \heartsuit$$

The proof of Lemma 15.9 involves one case per reduction rule in the instrumented semantics. It is reasonably straightforward. The proof of Lemma 15.10 involves a potentially tedious case analysis, which fortunately can be almost completely automated. Both proofs require a number of auxiliary inversion lemmas for the typing judgement (not shown). The number of these lemmas is linear in the number of constructs in the instrumented language.

Our next step (and, in this paper, a key step) is to state an auxiliary lemma that allows reasoning about the deconstruction and reconstruction of a term-in-context. In a traditional type system, such a lemma usually takes a very simple form: "If $\mathscr{E}[t_1]$ has type τ, then there exists a type θ such that t_1 has type θ and, for every closed term t_2 of type θ, $\mathscr{E}[t_2]$ has type τ". In other words, the term in the hole can be replaced with any term of the same type without affecting the type of the hole. Such a statement appears, for instance, in Wright and Felleisen's paper (1994), where it is known as the Replacement Lemma.

Here, the statement of this property is made significantly more complex by two aspects, namely, the treatment of resources and the treatment of the anti-frame rule. We focus on each of these aspects in turn, and give two successive versions of the lemma. The first version deals with resources, and is valid, but too weak; the second generalizes the first version to also deal with the anti-frame rule.

The first version of the lemma accounts for the fact that the resource R_1 that corresponds to $\mathscr{E}[t_1]$ is split as $R_1^a * R_1^b$, where R_1^a and R_1^b respectively correspond to $\mathscr{E}$ and t_1. As we replace the term t_1 with a new term t_2, we cannot require t_2 to be well-typed with respect to the same resource R_1^b. We must instead allow t_2

to be well-typed under a new resource R_2^b. This new resource cannot be completely arbitrary, however: In order to guarantee that the term t_2 can be placed in the evaluation context $\mathscr{E}$, we require that there exists a split $R_2^a * R_2^b = R_2$, where $R_1^a \lhd R_2^a$ holds. That is, we require the evolution that is imposed on the evaluation context to follow the passive execution ordering $\lhd$. (If R_1^b and R_2^b are related by the active execution ordering $\blacktriangleleft$, then, by Axiom 10.14, this is the case.) Then by monotonicity (Lemma 15.4), the evaluation context remains well-typed. Thus, it is possible to place the term t_2 in the evaluation context $\mathscr{E}$, yielding a term that is well-typed with respect to the new resource R_2.

Lemma 15.11 (Term-in-context, preliminary) *Let $R_1, nil, nil \Vdash \mathscr{E}[t_1] : \tau$. Then, there are resources R_1^a and R_1^b and a type θ such that:*

1. *The resource R_1 is split between R_1^a, which is "owned by" the evaluation context, and R_1^b, which is "owned by" the term.*

$$R_1^a * R_1^b = R_1$$

2. *The "type of the hole" of the evaluation context is θ.*

$$R_1^b, nil, nil \Vdash t_1 : \theta$$

3. *In the place of t_1, it is possible to plug a new term t_2 of type θ, provided the evolution that is imposed on the evaluation context respects the passive execution ordering $\lhd$. That is, for all t_2, R_2^a, R_2^b, and R_2, the following implication holds:*

$$\frac{R_2^b, nil, nil \Vdash t_2 : \theta \qquad R_1^a \lhd R_2^a \qquad R_2^a * R_2^b = R_2}{R_2, nil, nil \Vdash \mathscr{E}[t_2] : \tau} \qquad \heartsuit$$

The second version of the lemma accounts for the fact that we cannot require the terms t_1 and t_2 to have the same type. In general, if t_1 has type θ, we must allow t_2 to have type $\theta \circ \vec{\theta}$, where $\vec{\theta}$ is a vector of "hidden" invariants that are being revealed. In the most common case, which we have studied up to now, the vector $\vec{\theta}$ has length zero, so that both t_1 and t_2, in fact, have type θ. However, if t_1 reduces to t_2 by executing a "let/hide" construct, then the vector $\vec{\theta}$ has length one. In the statements that follow, we assume that $\vec{\theta}$ has an arbitrary length h. Then the term t_2 cannot be plugged in the evaluation context $\mathscr{E}$. It can, however, be plugged in the context $\wr\mathscr{E}\wr^h$ obtained by applying revelation h times to $\mathscr{E}$. This yields a term $\wr\mathscr{E}\wr^h[t_2]$ of type $\tau \circ \vec{\theta}$, whereas the original term $\mathscr{E}[t_1]$ has type τ. In summary, if the step from t_1 to t_2 is type-preserving up to the revelation of $\vec{\theta}$, then the step from $\mathscr{E}[t_1]$ to $\wr\mathscr{E}\wr^h[t_2]$ is also type-preserving up to the revelation of $\vec{\theta}$.

This is the basic idea behind the proof of type-preservation for the anti-frame rule, and a contribution of the present paper. When a term decides to install a new invariant by executing a "let/hide" construct, this invariant is immediately revealed to the entire evaluation context. Whereas the static semantics of the anti-frame is concerned with hiding the invariant, its dynamic semantics reveals it!

This syntactic phenomenon seems intuitively closely related to what happens in the Kripke model (Levy, 2002; Birkedal *et al.*, 2009, 2011; Schwinghammer *et al.*, 2012), where allocating a fresh (weak) reference or installing a fresh hidden invariant causes the entire system (that is, both the term and the evaluation context) to move to a new world.

Here is the generalized statement of the previous lemma.

Lemma 15.12 (Term-in-context) *Assume $R_1, nil, nil \Vdash \mathscr{E}[t_1] : \tau$. Then, there are resources R_1^a and R_1^b and a type θ such that:*

1. *The resource R_1 is split between the evaluation context and the term.*
$$R_1^a * R_1^b = R_1$$
2. *The "type of the hole" of the evaluation context is θ.*
$$R_1^b, nil, nil \Vdash t_1 : \theta$$
3. *In the place of t_1, it is possible to plug a new term t_2 of type $\theta \circ \vec{\theta}$, for an arbitrary vector of types $\vec{\theta}$. This vector represents a number of invariants that are being revealed. The evaluation context $\mathscr{E}$ must then be adapted by applying revelation as many times as there are invariants, that is, h times, where h is the length of the vector $\vec{\theta}$. The new complete term $\wr \mathscr{E} \wr^h [t_2]$ thus obtained no longer has type τ, but $\tau \circ \vec{\theta}$: it itself reveals the invariants. That is, for all t_2, R_2^a, R_2^b, R_2, h, and $\vec{\theta}$, the following implication holds:*
$$\frac{R_2^b, nil, nil \Vdash t_2 : \theta \circ \vec{\theta} \qquad R_1^a \lhd R_2^a \qquad R_2^a * R_2^b = R_2 \qquad |\vec{\theta}| = h}{R_2, nil, nil \Vdash \wr \mathscr{E} \wr^h [t_2] : \tau \circ \vec{\theta}} \quad \heartsuit$$

We continue with subject reduction and progress statements for the main reduction relation, that is, for the reduction of configurations. Again, the statement of the subject reduction lemma takes on a more complex form than usual, because of the treatment of resources and the anti-frame rule. In the statement that follows, we allow the term t_1 to be well-typed with respect to a fragment R_1^a of the resource R_1 that corresponds to the entire store s_1. The idea is that the term t_1 appears under an evaluation context $\mathscr{E}$ (which is not explicitly mentioned), and the resource R_1 can be split as $R_1^a * R_1^b$, where the fragment R_1^b is "owned by" $\mathscr{E}$.

Lemma 15.13 (Subject reduction, inductive form) *Assume that the configuration s_1 / t_1 takes a reduction step:*
$$s_1 / t_1 \xrightarrow{h} s_2 / t_2$$
Assume that the store s_1 is consistent with a global resource R_1:
$$R_1 \vdash s_1$$
Assume that the term t_1 is well-typed under R_1^b, presumably a fragment of R_1:
$$R_1^b, nil, nil \Vdash t_1 : \tau$$
Then, there exists a vector of types $\vec{\theta}$, whose length is h, such that, whatever the fragment R_1^a owned by the evaluation context (that is to say, for every R_1^a such that $R_1^a * R_1^b = R_1$ holds), there exist new resources R_2^a, R_2^b, and R_2 such that

- *The term t_2 has type τ, where the invariants $\vec{\theta}$ are revealed:*

$$R_2^b, nil, nil \Vdash t_2 : \tau \circ \vec{\theta}$$

- *The evolution imposed on the evaluation context respects the passive execution ordering:*

$$R_1^a \lhd R_2^a$$

- *The new resources owned by the evaluation context and by the term combine:*

$$R_2^a * R_2^b = R_2$$

- *The store s_2 is consistent with this combination:*

$$R_2 \vdash s_2 \qquad \heartsuit$$

The above statement is quite complex. This seems to be the price to pay for a formulation that lends to itself to a proof by structural induction over the first hypothesis. Lemma 15.12 is used in the proof case that deals with reduction under a context. Fortunately, this statement leads to the following much simpler corollary.

Lemma 15.14 (Subject reduction) *The reduction of configurations preserves well-typedness.*

$$\frac{\vdash s_1/t_1 : \tau \qquad s_1/t_1 \xrightarrow{h} s_2/t_2}{\vdash s_2/t_2 : \tau} \qquad \heartsuit$$

The premise and conclusion of the above lemma are able to refer to a common type τ, thanks to the manner in which the typing judgement for configurations was defined (Definition 11.2).

Let us now move on to the progress lemma.

The usual statement of this property is that every well-typed configuration is acceptable, where a configuration s/t is acceptable if either t is a value or s/t is able to reduce. We make two minor amendments to this notion. First, because our values are supposed to reduce to a canonical form, we consider only canonical values acceptable. Second, we build in the fact that if the anti-frame rule is disabled, then s/t is able to reduce without revealing any new invariants.

Definition 15.15 *Whether a configuration s/t is acceptable is defined as follows:*

$$\frac{v \text{ canonical} \qquad t = v}{s/t \text{ acceptable}} \qquad \frac{s/t \xrightarrow{h} s'/t' \qquad \text{anti-frame disabled} \Rightarrow h = 0}{s/t \text{ acceptable}}$$

We first state progress under a form that is amenable to an inductive proof.

Lemma 15.16 (Progress, inductive form) *If the store s is consistent with the resource R and if the term t is well-typed under a fragment of R, then the configuration s/t is acceptable.*

$$\frac{R \vdash s \qquad R_1, nil, nil \Vdash t : \tau \qquad R_2 * R_1 = R}{s/t \text{ acceptable}} \qquad \heartsuit$$

The proof is routine. Only the case of a term of the form Λt may be worth mentioning. The induction hypothesis guarantees that either t is a value (in which case Λt reduces to a value, and we are done) or t is able to step. Because Λt is well-typed, the anti-frame rule must be disabled. Thus, the step out of t must have $h = 0$. This means that reduction under a Λ-abstraction is permitted: Λt is able to make one step.

The above lemma has the following corollary.

Lemma 15.17 (Progress) *Every well-typed configuration is acceptable.*

$$\frac{\vdash s/t : \tau}{s/t \text{ acceptable}} \qquad \heartsuit$$

Together, Lemmas 15.14 and 15.17 show that, in the instrumented calculus, well-typed configurations cannot go wrong. There remains to transport this result down to the level of the raw calculus.

In order to do so, we first relate the instrumented calculus and the raw calculus by proving that erasure is a weak simulation. That is, one step of reduction at the level of the instrumented calculus corresponds, through erasure, to zero or more steps of reduction at the level of the raw calculus. We further prove that a computationally irrelevant step (one that corresponds to zero steps at the level of the raw calculus) causes a decrease in some well-founded ordering. (The well-founded ordering $t_1 \succ t_2$ is a lexicographic combination of a couple of suitable term measures and the value reduction relation, which by Lemma 13.4 is well-founded.) This means that every sequence of computationally irrelevant steps must be finite, or, in other words, divergence at the level of the instrumented calculus implies divergence at the level of the raw calculus.

Lemma 15.18 (Erasure is a simulation) *Assume that a well-layered configuration s_1/t_1 reduces to s_2/t_2. The image of this reduction step through erasure is either zero reduction steps (and, in that case, $t_1 \succ t_2$ holds) or a non-empty sequence of reduction steps.*

$$\frac{s_1/t_1 \xrightarrow{h} s_2/t_2 \qquad \vdash_{\mathrm{wl}} s_1/t_1}{\begin{array}{c}(\lfloor s_1 \rfloor = \lfloor s_2 \rfloor \wedge \lfloor t_1 \rfloor = \lfloor t_2 \rfloor \wedge t_1 \succ t_2) \vee \\ \lfloor s_1 \rfloor / \lfloor t_1 \rfloor \longrightarrow^{+} \lfloor s_2 \rfloor / \lfloor t_2 \rfloor\end{array}} \qquad \heartsuit$$

The proof of this lemma does not require a well-typedness hypothesis, but does require well-layeredness.

Then we define what it means for a raw configuration S/T to be well-typed. The first definition that comes to mind is to consider S/T well-typed if and only if it is the erasure of some well-layered and well-typed configuration s/t. However, because one step of reduction in the instrumented calculus may correspond to several steps in the raw calculus, there are intermediate configurations at the raw level which are not the erasure of a well-typed configuration and which we would nevertheless like to consider valid. Thus, we generalize this definition slightly.

Definition 15.19 *A raw configuration S / T is well-typed if and only if it is able to reach the erasure of some well-typed, well-layered configuration s / t.*

$$\frac{\vdash s / t : \tau \qquad \vdash_{\mathrm{wl}} s / t \qquad S / T \longrightarrow^{\star} \lfloor s \rfloor / \lfloor t \rfloor}{\vdash S / T : \tau}$$

It may be worth noting that this definition is appropriate only because the raw calculus is deterministic. The existence of a path from S / T to $\lfloor s \rfloor / \lfloor t \rfloor$ means that S / T will inevitably reduce to $\lfloor s \rfloor / \lfloor t \rfloor$.

In order to establish a subject reduction property for this notion of well-typedness, we must prove that a reduction step in the raw calculus corresponds to a number of reduction steps in the instrumented calculus, so as to then exploit subject reduction in the instrumented calculus. In other words, we need a backward simulation lemma, whereas Lemma 15.18 is a forward simulation lemma. Fortunately, in a deterministic setting, one follows from the other. We prove the following intermediate result.

Lemma 15.20 *If the erasure of a well-typed, well-layered configuration makes a step to S_2 / T_2, then S_2 / T_2 is able to reach the erasure of a well-typed, well-layered configuration.*

$$\frac{\vdash s_1 / t_1 : \tau \qquad \vdash_{\mathrm{wl}} s_1 / t_1 \qquad \lfloor s_1 \rfloor / \lfloor t_1 \rfloor \longrightarrow S_2 / T_2}{\vdash S_2 / T_2 : \tau} \qquad \heartsuit$$

The proof exploits the fact that erasure is a simulation (Lemma 15.18), the determinism of the raw semantics, subject reduction and progress for the instrumented calculus (Lemmas 15.14 and 15.17), and the preservation of well-layeredness (Lemma 14.2). It also exploits the fact that the erasure of a value is a raw value, hence cannot reduce.

Subject reduction for the raw calculus is an immediate corollary.

Lemma 15.21 (Raw subject reduction) *The reduction of raw configurations preserves well-typedness.*

$$\frac{\vdash S_1 / T_1 : \tau \qquad S_1 / T_1 \longrightarrow S_2 / T_2}{\vdash S_2 / T_2 : \tau} \qquad \heartsuit$$

The proof of progress exploits the same ingredients. We state the result directly.

Lemma 15.22 (Raw progress) *A well-typed raw configuration either exhibits a value or is able to reduce.*

$$\frac{\vdash S_1 / T_1 : \tau}{(\exists V,\ T_1 = V) \vee (\exists S_2 T_2,\ S_1 / T_1 \longrightarrow S_2 / T_2)} \qquad \heartsuit$$

Lemmas 15.21 and 15.22 together lead to the final type soundness result, where divergence is co-inductively defined in the simplest possible manner. (In a deterministic setting, the notions of may-diverge and must-diverge coincide.)

Lemma 15.23 (Type soundness) *A well-typed raw configuration either eventually yields a value or diverges.*

$$\frac{\vdash S_1 /\, T_1 \,:\, \tau}{(\exists S_2 V_2,\; S_1 /\, T_1 \longrightarrow^{\star} S_2 /\, V_2) \vee S_1 /\, T_1 \textit{ diverges}} \qquad \heartsuit$$

16 Conclusion

We have presented a definition and type soundness proof for an expressive type-and-capability system. At the core of the system lies an affine version of DILL, extended with references, capabilities, and regions. On top of this rests a notion of hidden state, in the form of an anti-frame rule. The system also incorporates a number of features that are required by the very statement of the anti-frame rule (such as the type constructor $\otimes$ and its equational theory) or by its soundness proof (such as recursive types). This is the first syntactic soundness proof, and the first machine-checked soundness proof, for the type-and-capability system with a hidden state.

The formulation of the type-and-capability system involves a number of design choices, some of which we are fairly happy with, others that appear in hindsight are more questionable. Isolating the notion of a monotonic separation algebra (Section 10) and formulating the typing rules in the style of DILL (Sections 7 and 11) seem to have been good decisions, which have led to elegant abstract definitions. On the other hand, separating the notions of well-typedness and well-layeredness (Section 14) seems in the end somewhat awkward and also sometimes limiting, as noted in Remark 14.3.

The type soundness proof involves a number of technical choices, the most prominent of which is the choice of a syntactic proof technique, based on subject reduction and progress lemmas. We find that establishing subject reduction for the instrumented semantics was fairly easy: there is a large number of rules, each of which does relatively little work, so the proof consists of a large number of relatively simple cases. Establishing progress was fairly easy as well: although there is again a large number of cases, their analysis can be automated in Coq. One weakness of this approach seems to appear in the proof that value reduction terminates (Lemma 13.4), where a single termination criterion must be found for a system that involves several dozen rules (Remark 13.5). Working with equirecursive types seems to have been a reasonable choice: although this approach makes it more difficult to construct the type equality relation and establish its properties, it makes the rest of the system simpler. Another weakness of our approach is the fact that certain laws, including the commutativity and associativity of $*$ and the "tensor exchange" law, cannot be built into the type equality relation, but must be viewed as coercions (Section 6.2). This is cumbersome and complicates the definition of the revelation operation $\wr\cdot\wr$. Finally, let us briefly comment on the fact that, among the operations that do nothing at runtime, there must be a distinction between those that "have no side effect", such as $\mathsf{defocus}\,\pi$, and those that "have a side effect", such as $\mathsf{focus}\,\pi$.

This is not an inherent shortcoming of the syntactic approach; some distinction is required in order to achieve type soundness (Remark 8.3). With some more work, it would be possible to view both $\mathsf{defocus}\,\pi$ and $\mathsf{focus}\,\pi$ as coercions, provided one distinguishes between "pure" and "effectful" coercions and forbids the use of an effectful coercion under the "!" modality.

The type-and-capability system presented in this paper is of very low level. From a purely type-theoretic point of view, this is rather pleasant. Most of the type constructors involved in the definition of the system are standard. Only a few new type constructors, such as "region inhabitant" and "capability for a region", are added, whose meaning does not overlap with that of the pre-existing type constructors. From a practical point of view, however, the system is extremely unwieldy, as evidenced by the detailed example that we have given (Section 3). In a surface language, we would suggest (1) using enough inference to ensure that the flow of capabilities remains completely implicit; (2) getting rid of the distinction between values and regions, which causes much noise by often imposing the introduction of two names (a variable x and a type variable σ) for each value; and (3) getting rid of the anti-frame rule, which seems difficult to explain to programmers and is sound only in a sequential setting, whereas dynamically allocated locks seem easier to explain and are sound in a concurrent setting (albeit with a risk of deadlock). Pottier and Protzenko (2012) present a preliminary design for such a surface language.

Appendix Semantics of the instrumented calculus

This appendix provides the definitions of the value reduction relation (Figures A 1 to A 9) and the term reduction relation (Figures A 12 to A 17). The definition is complete, except for the fact that the definition of revelation (which is used in Figures A 16 and A 17) has been omitted. The full Coq formalization is available online for browsing (Pottier, 2012a) and downloading (Pottier, 2012b).

$$\mathsf{id}\; v \longrightarrow v \qquad\qquad (c_1\,;c_2)\; v \longrightarrow c_2\;(c_1\; v)$$

Fig. A 1. Value reduction: reflexivity and transitivity.

$$(c_1 \to c_2)\;(\lambda t) \longrightarrow \lambda(c_2\;([c_1\;0/1](0{\uparrow}t))) \qquad {}_{\iota_1}(c_1 \times c_2)_{\iota_2}\;{}_{\iota_1}(v_1, v_2)_{\iota_2} \longrightarrow {}_{\iota_1}(c_1\;v_1, c_2\;v_2)_{\iota_2}$$

$$(\forall c)\;(\Lambda v) \longrightarrow \Lambda(c\;v) \qquad (\exists c)\;(\mathsf{pack}\;v) \longrightarrow \mathsf{pack}\;(c\;v) \qquad (!\,c)\;(!\,v) \longrightarrow !\,(c\;v)$$

$$(\mathsf{ref}\;c)\;(l\%v) \longrightarrow l\%(c\;v) \qquad \{c\}\;\{\vec{v}\} \longrightarrow \{c\;\vec{v}\} \qquad \{c\backslash\}\;\{?::\vec{v}\} \longrightarrow \{?::c\;\vec{v}\}$$

Fig. A 2. Value reduction: congruence.

$$\forall\mathsf{I}\; v \longrightarrow \Lambda v \qquad \forall\mathsf{E}\;(\Lambda v) \longrightarrow v \qquad \exists\mathsf{I}\; v \longrightarrow \mathsf{pack}\; v \qquad \exists\mathsf{E}\;(\mathsf{pack}\; v) \longrightarrow v$$

Fig. A 3. Value reduction: quantifier introduction and elimination.

$$\textsf{distrib}\ (\Lambda(\lambda t)) \longrightarrow \lambda(\Lambda([\forall \textsf{E}\ 0/1](0\uparrow t))) \qquad \exists\textsf{LI}\ (\Lambda(\lambda t)) \longrightarrow \lambda(\textsf{unpack}\,0\,\textsf{in}\,1\uparrow t)$$

$$\forall\textsf{-pair}\ (\Lambda_{\iota_1}(v_1, v_2)_{\iota_2}) \longrightarrow {}_{\iota_1}(\Lambda v_1, \Lambda v_2)_{\iota_2} \qquad \forall\textsf{-bang}\ (\Lambda(!\,v)) \longrightarrow !\,(\Lambda v)$$

$$\forall\textsf{-ref}\ (\Lambda(l\%v)) \longrightarrow l\%(\Lambda v) \qquad \forall\textsf{-regioncap}\ (\Lambda\{\vec{v}\}) \longrightarrow \{\Lambda\vec{v}\}$$

$$\forall\textsf{-regioncappunched}\ (\Lambda\{?\ ::\ \vec{v}\}) \longrightarrow \{?\ ::\ \Lambda\vec{v}\}$$

$$\textsf{pair-exists-left}\ {}_{\iota_1}(\textsf{pack}\ v_1, v_2)_{\iota_2} \longrightarrow \textsf{pack}\ {}_{\iota_1}(v_1, v_2)_{\iota_2}$$

$$\textsf{pair-exists-right}\ {}_{\iota_1}(v_1, \textsf{pack}\ v_2)_{\iota_2} \longrightarrow \textsf{pack}\ {}_{\iota_1}(v_1, v_2)_{\iota_2}$$

$$\textsf{bang-exists}\ (!\,(\textsf{pack}\ v)) \longrightarrow \textsf{pack}\ (!\,v) \qquad \textsf{ref-exists}\ (l\%(\textsf{pack}\ v)) \longrightarrow \textsf{pack}\ (l\%v)$$

$$\textsf{cap-exists}\ \{(\textsf{pack}\ v)\ ::\ \epsilon\} \longrightarrow \textsf{pack}\ \{v\ ::\ \epsilon\}$$

Fig. A 4. Value reduction: quantifier movement.

$$\textsf{dereliction}\ (!\,v) \longrightarrow v \qquad \textsf{bang-idempotent}\ (!\,v) \longrightarrow !\,(!\,v)$$

$$\textsf{pair-bang}\ {}_{\iota_1}(!\,v_1, !\,v_2)_{\iota_2} \longrightarrow !\,{}_{\iota_1}(v_1, v_2)_{\iota_2} \qquad \textsf{bang-pair}\ (!\,{}_{\iota_1}(v_1, v_2)_{\iota_2}) \longrightarrow !\,{}_{\iota_1}(!\,v_1, !\,v_2)_{\iota_2}$$

$$\textsf{unit-bang}\ ()_\iota \longrightarrow !\,()_\iota \qquad \textsf{at-bang}\ [v] \longrightarrow !\,[v]$$

Fig. A 5. Value reduction: affinity.

$$\frac{v_1(\pi) = [v_1''] \qquad v_1(\pi \mapsto v_2) = v_1'}{(\textsf{defocus}\ \pi)\ {}_{\textsf{Phy}}(v_1, \{v_2\ ::\ \epsilon\})_{\textsf{Log}} \longrightarrow v_1'} \qquad \textsf{defocus-group}\ {}_{\textsf{Log}}(\{v\ ::\ \epsilon\}, \{?\ ::\ \vec{v}\})_{\textsf{Log}} \longrightarrow \{v\ ::\ \vec{v}\}$$

$$\textsf{singleton-to-group}\ \{\vec{v}\} \longrightarrow \{\vec{v}\}$$

Fig. A 6. Value reduction: regions.

$$\frac{\iota_1 = \mathsf{Log} \vee \iota_2 = \mathsf{Log}}{\mathsf{star\text{-}comm}\ {}_{\iota_1}(v_1, v_2)_{\iota_2} \longrightarrow {}_{\iota_2}(v_2, v_1)_{\iota_1}}$$

$$\frac{\iota_1 = \mathsf{Log} \vee \iota_2 = \mathsf{Log} \vee \iota_3 = \mathsf{Log}}{\mathsf{star\text{-}assoc}\ {}_{(\iota_1.\iota_2)}({}_{\iota_1}(v_1, v_2)_{\iota_2}, v_3)_{\iota_3} \longrightarrow {}_{\iota_1}(v_1, {}_{\iota_2}(v_2, v_3)_{\iota_3})_{(\iota_2.\iota_3)}}$$

$$\mathsf{star\text{-}ref}\ {}_{\mathsf{Phy}}(l\%v_1, v_2)_{\mathsf{Log}} \longrightarrow l\%_{\mathsf{Phy}}(v_1, v_2)_{\mathsf{Log}}$$

$$\mathsf{ref\text{-}star}\ (l\%_{\mathsf{Phy}}(v_1, v_2)_{\mathsf{Log}}) \longrightarrow {}_{\mathsf{Phy}}(l\%v_1, v_2)_{\mathsf{Log}}$$

$$\mathsf{star\text{-}singleton}\ {}_{\mathsf{Log}}(\{v_1 :: \epsilon\}, v_2)_{\mathsf{Log}} \longrightarrow \{{}_{\mathsf{Phy}}(v_1, v_2)_{\mathsf{Log}} :: \epsilon\}$$

$$\mathsf{singleton\text{-}star}\ \{{}_{\mathsf{Phy}}(v_1, v_2)_{\mathsf{Log}} :: \epsilon\} \longrightarrow {}_{\mathsf{Log}}(\{v_1 :: \epsilon\}, v_2)_{\mathsf{Log}}$$

$$\otimes\text{-}\mathsf{exch}_n\ (\lambda t) \longrightarrow \lambda([\circ\text{-}\mathsf{exch}_n\ 0/1](\circ\text{-}\mathsf{exch}_n\ (0\uparrow t)))$$

$$\otimes\text{-}\mathsf{exch}_n\ {}_{\iota_1}(v_1, v_2)_{\iota_2} \longrightarrow {}_{\iota_1}(\otimes\text{-}\mathsf{exch}_n\ v_1, \otimes\text{-}\mathsf{exch}_n\ v_2)_{\iota_2} \qquad \otimes\text{-}\mathsf{exch}_n\ ()_\iota \longrightarrow ()_\iota$$

$$\otimes\text{-}\mathsf{exch}_n\ (\Lambda v) \longrightarrow \Lambda(\otimes\text{-}\mathsf{exch}_n\ v) \qquad \otimes\text{-}\mathsf{exch}_n\ (\mathsf{pack}\ v) \longrightarrow \mathsf{pack}\ (\otimes\text{-}\mathsf{exch}_n\ v)$$

$$\otimes\text{-}\mathsf{exch}_n\ (!\,v) \longrightarrow !\,(\otimes\text{-}\mathsf{exch}_n\ v) \qquad \otimes\text{-}\mathsf{exch}_n\ (l\%v) \longrightarrow l\%(\otimes\text{-}\mathsf{exch}_n\ v)$$

$$\otimes\text{-}\mathsf{exch}_n\ \{\vec{v}\} \longrightarrow \{\otimes\text{-}\mathsf{exch}_n\ \vec{v}\} \qquad \otimes\text{-}\mathsf{exch}_n\ \{? :: \vec{v}\} \longrightarrow \{? :: \otimes\text{-}\mathsf{exch}_n\ \vec{v}\}$$

$$\otimes\text{-}\mathsf{exch}_n\ [v] \longrightarrow [v]$$

Fig. A 7. Value reduction: movement of stars.

$$\frac{c \text{ contractive in } 0}{(\mu c)\ v \longrightarrow ([\mu c/0]c)\ v}$$

Fig. A 8. Value reduction: recursive coercions.

$$\frac{v_1 \longrightarrow v_2}{{}_{\iota_1}(v_1, v)_{\iota_2} \longrightarrow {}_{\iota_1}(v_2, v)_{\iota_2}} \qquad \frac{v_1 \longrightarrow v_2}{{}_{\iota_1}(v, v_1)_{\iota_2} \longrightarrow {}_{\iota_1}(v, v_2)_{\iota_2}} \qquad \frac{v_1 \longrightarrow v_2}{c\ v_1 \longrightarrow c\ v_2} \qquad \frac{v_1 \longrightarrow v_2}{\Lambda v_1 \longrightarrow \Lambda v_2}$$

$$\frac{v_1 \longrightarrow v_2}{\mathsf{pack}\ v_1 \longrightarrow \mathsf{pack}\ v_2} \qquad \frac{v_1 \longrightarrow v_2}{!\,v_1 \longrightarrow !\,v_2} \qquad \frac{\vec{v}_1 \longrightarrow \vec{v}_2}{\{\vec{v}_1\} \longrightarrow \{\vec{v}_2\}} \qquad \frac{\vec{v}_1 \longrightarrow \vec{v}_2}{\{? :: \vec{v}_1\} \longrightarrow \{? :: \vec{v}_2\}}$$

$$\frac{v_1 \longrightarrow v_2}{l\%v_1 \longrightarrow l\%v_2}$$

Fig. A 9. Value reduction: reduction under a context.

$$v(\mathsf{path\text{-}root}) = v \qquad \frac{v_1(\pi) = v}{{}_{\iota_1}(v_1, v_2)_{\iota_2}(\mathsf{path\text{-}left}\ \iota_1\ \iota_2\ \pi) = v} \qquad \frac{v_2(\pi) = v}{{}_{\iota_1}(v_1, v_2)_{\iota_2}(\mathsf{path\text{-}right}\ \iota_1\ \iota_2\ \pi) = v}$$

$$\frac{v_1(\pi) = v}{(l\%v_1)(\mathsf{path\text{-}ref}\ \pi) = v} \qquad \frac{v_1(\pi) = v}{\{v_1 :: \epsilon\}(\mathsf{path\text{-}singleton}\ \pi) = v}$$

Fig. A 10. Selecting a value at a path.

$$\frac{\lfloor v_1 \rfloor = \lfloor v_2 \rfloor}{v_1(\textsf{path-root} \mapsto v_2) = v_2} \qquad \frac{v_1(\pi \mapsto v) = v_1'}{{}_{\iota_1}(v_1, v_2)_{\iota_2}(\textsf{path-left}\, \iota_1\, \iota_2\, \pi \mapsto v) = {}_{\iota_1}(v_1', v_2)_{\iota_2}}$$

$$\frac{v_2(\pi \mapsto v) = v_2'}{{}_{\iota_1}(v_1, v_2)_{\iota_2}(\textsf{path-right}\, \iota_1\, \iota_2\, \pi \mapsto v) = {}_{\iota_1}(v_1, v_2')_{\iota_2}} \qquad \frac{v_1(\pi \mapsto v) = v_1'}{(l\%v_1)(\textsf{path-ref}\, \pi \mapsto v) = l\%v_1'}$$

$$\frac{v_1(\pi \mapsto v) = v_1'}{\{v_1 :: \epsilon\}(\textsf{path-singleton}\, \pi \mapsto v) = \{v_1' :: \epsilon\}}$$

Fig. A 11. Updating a value at a path.

$$s/\,(\lambda t)\; v \xrightarrow{0} s/\,[v/0]t \qquad s/\,\textsf{let}\,!\,(!\,v)\,\textsf{in}\,t \xrightarrow{0} s/\,[v/0]t$$

$$s/\,\textsf{letpair}_{\iota_1,\iota_2}\; {}_{\iota_1}(v_1, v_2)_{\iota_2}\;\textsf{in}\,t \xrightarrow{0} s/\,[v_1/0][0{\uparrow}v_2/0]t \qquad s/\,\textsf{unpack}\,(\textsf{pack}\; v)\,\textsf{in}\,t \xrightarrow{0} s/\,[v/0]t$$

Fig. A 12. Term reduction: λ-calculus.

$$\frac{v_1 \longrightarrow v_2}{s/\,v_1 \xrightarrow{0} s/\,v_2} \qquad s/\,{}_{\textsf{Phy}}(v_1, v_2)_{\textsf{Log}} \xrightarrow{0} s/\,{}_{\textsf{Phy}}(v_1, v_2)_{\textsf{Log}} \qquad s/\,\Lambda v \xrightarrow{0} s/\,\Lambda v \qquad s/\,c\; v \xrightarrow{0} s/\,c\; v$$

$$\frac{v_1 \longrightarrow v_2}{s/\,v_1\; t \xrightarrow{0} s/\,v_2\; t} \qquad \frac{v_1 \longrightarrow v_2}{s/\,\textsf{unpack}\,v_1\,\textsf{in}\,t \xrightarrow{0} s/\,\textsf{unpack}\,v_2\,\textsf{in}\,t} \qquad \frac{v_1 \longrightarrow v_2}{s/\,\textsf{let}!\,v_1\,\textsf{in}\,t \xrightarrow{0} s/\,\textsf{let}!\,v_2\,\textsf{in}\,t}$$

$$\frac{v_1 \longrightarrow v_2}{s/\,\textsf{letpair}_{\iota_1,\iota_2}\, v_1\,\textsf{in}\,t \xrightarrow{0} s/\,\textsf{letpair}_{\iota_1,\iota_2}\, v_2\,\textsf{in}\,t} \qquad \frac{v_1 \longrightarrow v_2}{s/\,p\; v_1 \xrightarrow{0} s/\,p\; v_2}$$

Fig. A 13. Term reduction: injection of values into terms.

$$\frac{v_1(\pi) = [v] \qquad !\,w \in \vec{v} \qquad v_1(\pi \mapsto !\,w) = v_2}{s/\,(\textsf{defocus-dup}\,\pi)\;{}_{\textsf{Phy}}(v_1, \{\vec{v}\})_{\textsf{Log}} \xrightarrow{0} s/\,{}_{\textsf{Phy}}(v_2, \{\vec{v}\})_{\textsf{Log}}}$$

$$\frac{v_1(\pi) = v_2 \qquad v_1(\pi \mapsto [v_2]) = v_1'}{s/\,(\textsf{focus}\,\pi)\; v_1 \xrightarrow{0} s/\,\textsf{pack}\;{}_{\textsf{Phy}}(v_1', \{v_2 :: \epsilon\})_{\textsf{Log}}}$$

$$s/\,\textsf{newgroup}\;()_{\textsf{Phy}} \xrightarrow{0} s/\,\textsf{pack}\;{}_{\textsf{Phy}}(()_{\textsf{Phy}}, \{\epsilon\})_{\textsf{Log}}$$

$$s/\,\textsf{adopt}\;{}_{\textsf{Phy}}(v, \{\vec{v}\})_{\textsf{Log}} \xrightarrow{0} s/\,{}_{\textsf{Phy}}([v], \{v :: \vec{v}\})_{\textsf{Log}}$$

$$\frac{\lfloor v \rfloor = \lfloor w \rfloor \qquad w \in \vec{v}}{s/\,\textsf{focusgroup}\;{}_{\textsf{Phy}}([v], \{\vec{v}\})_{\textsf{Log}} \xrightarrow{0} s/\,\textsf{pack}\;{}_{\textsf{Phy}}([v], {}_{\textsf{Log}}(\{w :: \epsilon\}, \{?\, :: \vec{v} \setminus w\})_{\textsf{Log}})_{\textsf{Log}}}$$

Fig. A 14. Term reduction: operations on regions.

$$\dfrac{m_1[l \mapsto v] = m_2}{\begin{array}{l} m_1 \,\mathsf{below}\, l /\, \mathsf{new}\; v \\ \xrightarrow{0} m_2 \,\mathsf{below}\, l+1 /\, \mathsf{pack}_{\,\mathsf{Phy}}([l\%v], \{(l\%v) :: \epsilon\})_{\mathsf{Log}} \end{array}}$$

$$\dfrac{m\, l = v \qquad \lfloor v \rfloor = \lfloor w \rfloor \qquad \lfloor v' \rfloor = l}{\begin{array}{l} m \,\mathsf{below}\, \ell /\, \mathsf{read}_{\,\mathsf{Phy}}([v'], \{(l\%(!\,w)) :: \epsilon\})_{\mathsf{Log}} \\ \xrightarrow{0} m \,\mathsf{below}\, \ell /\, {}_{\mathsf{Phy}}(!\,w, \{(l\%(!\,w)) :: \epsilon\})_{\mathsf{Log}} \end{array}}$$

$$\dfrac{m_1\, l = v_1 \qquad m_1[l \mapsto v_2] = m_2 \qquad \lfloor v_1' \rfloor = l}{\begin{array}{l} m_1 \,\mathsf{below}\, \ell /\, \mathsf{write}_{\,\mathsf{Phy}}({}_{\mathsf{Phy}}([v_1'], v_2)_{\mathsf{Phy}}, \{(l\%w) :: \epsilon\})_{\mathsf{Log}} \\ \xrightarrow{0} m_2 \,\mathsf{below}\, \ell /\, {}_{\mathsf{Phy}}(()_{\mathsf{Phy}}, \{(l\%v_2) :: \epsilon\})_{\mathsf{Log}} \end{array}}$$

Fig. A 15. Term reduction: references.

$$s /\, \mathsf{let}\, v \;\mathsf{in}\; \mathsf{hide}\; t \xrightarrow{1} s /\, [\llparenthesis v \rrparenthesis/0]t$$

Fig. A 16. Term reduction: hide.

$$\dfrac{s_1 / t_1 \xrightarrow{h} s_2 / t_2}{s_1 / \mathscr{E}[t_1] \xrightarrow{h} s_2 / \llparenthesis \mathscr{E} \rrparenthesis^h [t_2]} \qquad\qquad \dfrac{s_1 / t_1 \xrightarrow{0} s_2 / t_2}{s_1 / \Lambda t_1 \xrightarrow{0} s_2 / \Lambda t_2}$$

Fig. A 17. Term reduction: reduction under a context.

Acknowledgments

I would like to thank Arthur Charguéraud for a pleasant and fruitful collaboration and for proof-reading this paper. I would also like to thank anonymous referees for their insightful suggestions, which helped greatly in improving the paper.

Supplementary material

For supplementary material for this article, please visit http://dx.doi.org/10.1017/S0956796812000366.

References

Abadi, M., Pierce, B. & Plotkin, G. (1991) Faithful ideal models for recursive polymorphic types. *Int. J. Found. Comput. Sci* **2**(1), 1–21.

Ahmed, A. J. (2004) *Semantics of Types for Mutable State*. Ph.D. thesis, Princeton University, Princeton, NJ.

Ahmed, A., Appel, A. W., Richards, C. D., Swadi, K. N., Tan, G. & Wang, D. C. (2010) Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* **32**(3), 7:1–7:67.

Ahmed, Amal J., Fluet, M. & Morrisett, G. (2005) A step-indexed model of substructural state. In *ACM International Conference on Functional Programming (ICFP)*, pp. 78–91.

Ahmed, A., Fluet, M. & Morrisett, G. (2007) L^3: A linear language with locations. *Fundam. Inform.* **77**(4), 397–449.

Almeida, P. S. (1997) Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 1241. New York: Springer, pp. 32–59.

Amadio, R. M. & Cardelli, L. (1993) Subtyping recursive types. *ACM Trans. Program. Lang. Syst.* **15**(4), 575–631.

Atkey, R. (2010) Amortised resource analysis with separation logic. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, vol. 6012. New York: Springer, pp. 85–103.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, Benjamin C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005) Mechanized metatheory for the masses: The PoplMark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science, vol. 3603. New York: Springer, pp. 50–65.

Barber, A. (1996) *Dual Intuitionistic Linear Logic*. Tech. Rep. ECS-LFCS-96-347. Laboratory for Foundations of Computer Science, School of Informatics at the University of Edinburgh, Edinburgh, UK.

Bell, C. J., Dockins, R., Hobor, A., Appel, A. W. & Walker, D. (2008) Comparing semantic and syntactic methods in mechanized proof frameworks.*Proceedings of the International Workshop on Proof-Carrying Code (PCC)*, Carnegie Mellon University, Pittsburgh, PA.

Bierhoff, K. & Aldrich, J. (2007) Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 301–320.

Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J. & Yang, H. (2011) Step-indexed Kripke models over recursive worlds. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 119–132.

Birkedal, L., Støvring, K. & Thamsborg, J. (2009) Realizability semantics of parametric polymorphism, general references, and recursive types. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture Notes in Computer Science, vol. 5504. New York: Springer, pp. 456–470.

Birkedal, L., Støvring, K. & Thamsborg, J. (2010) Realisability semantics of parametric polymorphism, general references, and recursive types. *Math. Struct. Comput. Sci.* **20**(4), 655–703.

Birkedal, L., Torp-Smith, N. & Yang, H. (2006) Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods Comput. Sci.* **2**(5).

Blanqui, F. & Koprowski, A. (2011) CoLoR: A coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.* **21**(4), 827–859.

Boyapati, C., Lee, R. & Rinard, M. (2002) Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 211–230.

Boyland, J. T. (2010) Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.* **32**(6), 22:1–22:33.

Boyland, J. T. & Retert, W. (2005) Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 283–295.

Brandt, M. & Henglein, F. (1998) Coinductive axiomatization of recursive type equality and subtyping. *Fundam. Inform.* **33**, 309–338.

Buisse, A., Birkedal, L. & Støvring, K. (2011) A step-indexed Kripke model of separation logic for storable locks. *Electron. Notes Theor. Comput. Sci.* **276**, 121–143.

Calcagno, C., O'Hearn, P. W. & Yang, H. (2007) Local action and abstract separation logic. In *IEEE Symposium on Logic in Computer Science (LICS)*, pp. 366–378.

Charguéraud, A. (2012) The locally nameless representation. *J. Autom. Reasoning* **49**(3), 363–408.

Charguéraud, A. & Pottier, F. (2008) Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pp. 213–224.

Clarke, D. G., Potter, J. M. & Noble, J. (1998) Ownership types for flexible alias protection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 48–64.

Crary, K., Walker, D. & Morrisett, G. (1999) Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 262–275.

Danielsson, N. A. & Altenkirch, T. (2010) Subtyping, declaratively. In *International Conference on Mathematics of Program Construction (MPC)*, Lecture Notes in Computer Science, vol. 6120. New York: Springer, pp. 100–118.

de Bruijn, N. G. (1972) Lambda-calculus notation with nameless dummies: A tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.* **34**(5), 381–392.

DeLine, R. & Fähndrich, M. (2001) Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 59–69.

Detlefs, D. L., Leino, K., Rustan M. & Nelson, G. (1998) *Wrestling with Rep Exposure*. Res. Rep. 156, SRC, Palo Alto, CA.

Dietl, W. & Peter, M. (2005) Universes: Lightweight ownership for JML. *J. Object Technol.* **4**(8), 5–32.

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J. & Yang, H. (submitted) *Views: Compositional Reasoning for Concurrent Programs.*

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. & Vafeiadis, V. (2010) *Concurrent Abstract Predicates.* Tech. Rep., Computer Laboratory, University of Cambridge, Cambridge, UK.

Dockins, R., Hobor, A. & Appel, A. W. (2009) A fresh look at separation algebras and share accounting. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Lecture Notes in Computer Science, vol. 5904. New York: Springer, pp. 161–177.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R. & Levi, S. (2006) Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys*, pp. 177–190.

Fähndrich, M. & DeLine, R. (2002) Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 13–24.

Gapeyev, V., Levin, M. & Pierce, B. (2002) Recursive subtyping revealed. *J. Funct. Program.* **12**(6), 511–548.

Gauthier, N. & Pottier, F. (2004) Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pp. 150–161.

Gifford, D. K., Jouvelot, P., Sheldon, M. A. & O'Toole, J. W. (1992) *Report on the FX-91 Programming Language*. Tech. Rep. MIT/LCS/TR-531, Massachusetts Institute of Technology, Cambridge, MA.

Girard, J.-Y. (1972) *Interprétation Fonctionnelle et Élimination des Coupures de L'arithmétique D'ordre Supérieur*. Thèse d'état, Université Paris 7.

Glew, N. (2002) A theory of second-order trees. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, vol. 2305. New York: Springer, pp. 147–161.

Gotsman, A., Berdine, J., Cook, B., Rinetzky, N. & Sagiv, M. (2007) *Local Reasoning for Storable Locks and Threads*. Tech. Rep. MSR-TR-2007-39. Microsoft Research, .

Harper, R. (1994) A simplified account of polymorphic references. *Inf. Process. Lett.* **51**(4), 201–206.

Hoare, C. A. R. (1972) Proof of correctness of data representations. *Acta Inform.* **4**, 271–281.

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

Hofmann, M. (2000) A type system for bounded space and functional in-place update. *Nord. J. Comput.* **7**(4), 258–289.

Hogg, J. (1991) Islands: Aliasing protection in object-oriented languages. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 271–285.

Ishtiaq, Samin S. & O'Hearn, Peter W. (2001) BI as an assertion language for mutable data structures. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 14–26.

Launchbury, J. & Jones, S. P. (1995) State in Haskell. *LISP Symb. Comput.* **8**(4), 293–341.

Levy, P. B. (2002) Possible world semantics for general storage in call-by-value. *Computer Science Logic*, Lecture Notes in Computer Science, vol. 2471. New York: Springer.

MacQueen, D. B., Plotkin, G. D. & Sethi, R. (1986) An ideal model for recursive polymorphic types. *Inf. Control* **71**(1–2), 95–130.

Mazurak, K., Zhao, J. & Zdancewic, S. (2010) Lightweight linear types in system F°. In *Workshop on Types in Language Design and Implementation (TLDI)*, pp. 77–88.

Mitchell, John C. (1988) Polymorphic-type inference and containment. *Inf. Comput.* **76**(2–3), 211–249.

Monnier, S. (2008) *Statically Tracking State with Typed Regions*. Draft.

Müller, P. & Poetzsch-Heffter, A. (2001) *Universes: A Type System for Alias and Dependency Control*. Tech. Rep. 279, Fernuniversität Hagen, Germany.

Nanevski, A., Morrisett, G. & Birkedal, L. (2008) -type theory, polymorphism and separation. *J. Funct. Program.* **18**(5–6), 865–911.

Nanevski, A., Vafeiadis, V. & Berdine, J. (2010) Structuring the verification of heap-manipulating programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 261–274.

O'Hearn, Peter W. (2007) Resources, concurrency and local reasoning. *Theor. Comput. Sci.* **375**(1–3), 271–307.

O'Hearn, P. W., Yang, H. & Reynolds, J. C. (2004) Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 268–280.

Peyton Jones, S. & Wadler, P. (1993) Imperative functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pp. 71–84.

Pilkiewicz, A. & Pottier, F. (2011) The essence of monotonic state. *Workshop on Types in Language Design and Implementation (TLDI)*, Philadelphia, PA.

Pollack, R., Sato, M. & Ricciotti, W. (2012) A canonical locally named representation of binding. *J. Autom. Reasoning* **49**(2), 185–207.

Pottier, F. (2008) Hiding local state in direct style: A higher-order anti-frame rule. In *IEEE Symposium on Logic in Computer Science (LICS)*, pp. 331–340.

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

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

Pottier, F. (2012a) Accompanying Coq scripts; for browsing [online]. Available at: `http://gallium.inria.fr/~fpottier/ssphs/`. Accessed 21 September 2012.

Pottier, F. (2012b) *Accompanying Coq scripts; for downloading* [online]. Available at: `http://gallium.inria.fr/~fpottier/ssphs/ssphs.tar.gz` and also as an online supplement at `http://www.cambridge.org/...`. Accessed 21 September 2012.

Pottier, F. & Protzenko, J. (2012) Programming with permissions: An introduction to Mezzo. Unpublished manuscript.

Reus, B. & Schwinghammer, J. (2006) Separation logic for higher-order store. In *Computer Science Logic*, Lecture Notes in Computer Science, vol. 4207. New York: Springer, pp. 575–590.

Reynolds, John C. (1974) Towards a theory of type structure. In *Colloque sur la Programmation*, Lecture Notes in Computer Science, vol. 19. New York: Springer, pp. 408–425.

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

Schwinghammer, J., Birkedal, L., Pottier, F., Reus, B., Støvring, K. & Yang, H. (2012) A step-indexed Kripke model of hidden state. *Math. Struct. Comput. Sci.* Available at: `http://dx.doi.org/10.1017/S0960129512000035`.

Schwinghammer, J., Birkedal, L., Reus, B. & Yang, H. (2009) Nested Hoare triples and frame rules for higher-order store. In *Computer Science Logic*, Lecture Notes in Computer Science, vol. 5771. New York: Springer, pp. 440–454.

Schwinghammer, J., Birkedal, L. & Støvring, K. (2011) A step-indexed Kripke model of hidden state via recursive properties on recursively defined metric spaces. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture Notes in Computer Science, no. 6604. New York: Springer, pp. 305–319.

Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F. & Reus, B. (2010) A semantic foundation for hidden state. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, Lecture Notes in Computer Science, vol. 6014. New York: Springer, pp. 2–17.

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

Swamy, N., Hicks, M., Morrisett, G., Grossman, D. & Jim, T. (2006) Safe manual memory management in Cyclone. *Sci. Comput. Program.* **62**(2), 122–144.

Talpin, J.-P. & Jouvelot, P. (1994) The type and effect discipline. *Inf. Comput.* **11**(2), 245–296.

Tan, G., Shao, Z., Feng, X. & Cai, H. (2009) Weak updates and separation logic. In *Asian Symposium on Programming Languages and Systems (APLAS)*, Lecture Notes in Computer Science, vol. 5904. New York: Springer, pp. 178–193.

Tofte, M. & Talpin, J.-P. (1997) Region-based memory management. *Inf. Comput.* **132**(2), 109–176.

Tov, J. A. & Pucella, R. (2010) Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, vol. 6012. New York: Springer, pp. 550–569.

Tov, J. A. & Pucella, R. (2011) Practical affine types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 447–458.

Urban, C. (2008) Nominal techniques in Isabelle/HOL. *J. Autom. Reason.* **40**(4), 327–356.

Vouillon, J. & Melliès, P.-A. (2004) Semantic types: A fresh look at the ideal model for types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pp. 52–63.

Walker, D. (2005) Substructural type systems. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed). Cambridge, MA: MIT Press, Chap. 1, pp. 3–43.

Walker, D. & Morrisett, G. (2000) Alias types for recursive data structures. In *Workshop on Types in Compilation (TIC)*, Lecture Notes in Computer Science, vol. 2071. New York: Springer, pp. 177–206.

Wright, A. K. (1995) Simple imperative polymorphism. *LISP Symb. Comput.* **8**(4), 343–356.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.