The Type Discipline of Behavioral Separation

Luís Caires João C. Seco

CITI and Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

(long version with Appendix - TR DI FCT UNL draft of September, 2012)

Abstract

We introduce the concept of behavioral separation as a general principle for disciplining interference in higher-order imperative concurrent programs, and present a type-based approach that systematically develops the concept in the context of an ML-like language extended with concurrency and synchronization primitives. Behavioral separation builds on notions originally introduced for behavioral type systems and separation logics, but shifts the focus from the separation of static program state properties towards the separation of dynamic usage behaviors of runtime values. Behavioral separation types specify how values may be safely used by client code, and can enforce fine-grained interference control disciplines while preserving compositionality, information hiding, and flexibility. We illustrate how our type system, even if based on a small set of general primitives, is already able to tackle fairly challenging program idioms, involving aliasing at various types, concurrency with first-class threads, manipulation of linked data structures, behavioral borrowing, and invariant-based separation.

1. Introduction

The purpose of this work is to introduce and develop the concept of *behavioral separation* as a general principle for disciplining interference in higher-order imperative concurrent programs.

Statically verifying that higher-order imperative programs do not go wrong in the presence of possible interference has proven to be a challenging task, and a fertile ground for research since the seminal work of Reynolds [32]. In general, two program fragments interfere when the effects generated by one fragment may change the state visible to the other, typically due to aliasing or to concurrency. Some forms of interference are "bad", and may cause catastrophic failure, such as read/write races when accessing the same memory cell. Other forms of interference are "good" and even required, such as the interference between producer and consumer running concurrently, and sharing a thread-safe stateful queue, or the interference between the head and tail references of a linked list data structure. "Interference is the essence of concurrency", as Cliff Jones has frequently written [24].

An ongoing challenge to overcome is then to find techniques for disciplining interference between different usages of the same objects, so to ensure safety while being able to address increasingly sophisticated programming idioms.

Significant advances have been achieved recently towards approaching these general goals, in particular by the separation logics of O'Hearn and Reynolds [33, 30] and by substructural type and effect systems, e.g., [1, 8, 3]. In particular, separation logic supports expressive forms of local reasoning, based on the use of the separating conjunction in combination with fractional permissions [9, 5] to characterize the fine structure of program states. Extending such state-based techniques to tackle the sophisticated program idioms arising in modern higher-order imperative concurrent programming is thus both promising and challenging [36].

In this work, we radically depart from a state-based view towards a behavioral view of program assertions, by introducing a notion of behavioral separation type structure. Behavioral separation builds on notions originally introduced for behavioral type systems and separation logics, but shifts the focus from the separation of static program state properties towards the separation of dynamic usage behaviors of runtime values. We thus introduce a type-based approach that systematically develops the concept of behavioral separation to enforce safety of programs, ruling out "bad" interferences in the presence of aliasing and concurrency. Our presentation is grounded on a core ML-like language, with higher-order functions, heap allocated variables, tuples, variants, and concurrency, which we pick as a convenient abstraction for other languages supporting higher-order imperative concurrent programming.

Behavioral types [19, 14, 22] based on process algebras, have been introduced with the aim of characterizing the interface of a process not just as a specification of the static type of exchanged messages, but also as specification of its dynamic behavior. Likewise, our behavioral separation types specify how program values may be safely used by client code, but are able to enforce fine-grained interference control disciplines. A key novelty of our approach consists in uniformly combining in the same type structure "temporal" operations, such as sequential separation, important to capture trace constraints, with "spatial" operations, such as parallel separation and isolation, important to capture aliasing and concurrency. Remarkably, we carry out our development in the context of a clean substructural type theory, in which all type operators satisfy natural algebraic properties, and is based on a λ-calculus extended with imperative references and concurrency constructs.

Behavioral separation types also promote information hiding, compositionality, and flexibility, since type assertions talk about separation constraints on usage behaviors as externally perceived by the programs which use them, rather than about the internal structure of program state or program code. As will be clear from our examples, behavioral separation types are also expressive and flexible enough for proving safety of programs combining features still challenging for (and even out of reach of) existing proof methods, including general higher-order store, aliasing / sharing at all types, linked data structures, borrowing of local behavior, first-class threads, and even invariant-based separation, based on typed synchronization constructs.

2. Overview

In this section, we motivate the general concept of behavioral separation, informally introducing on the way our core programming language and explaining the various behavioral separation type operators. We proceed by going through a sequence of examples. Consider the following implementation of a collection abstract data type, where we assume the list elements to be natural numbers, and the representation data structure to be a linked list.

$$\begin{array}{l}\textsf{let } newNode = \lambda[].\textsf{var } next, elt \textsf{ in}\\ \qquad\qquad [\ setElt = \lambda e.(elt := e),\\ \qquad\qquad\ \ getElt = elt,\\ \qquad\qquad\ \ setNext = \lambda p.(next := p),\\ \qquad\qquad\ \ getNext = next\ ] \textsf{ in}\end{array}$$

$$\begin{array}{l}\textsf{let } newColl =\\ \quad \lambda[].\textsf{var } hd, id \textsf{ in}\\ \quad\ [\ init = \lambda i.(hd := \texttt{NULL}; id := i)\\ \quad\ \ getId = id,\\ \quad\ \ add = \lambda e.\textsf{let } n = (newNode\ \textsf{nil}) \textsf{ in}\\ \qquad\qquad ((n.setElt\ e); (n.setNext\ hd); hd{:=}\texttt{NODE}(n)),\\ \quad\ \ scan = \textsf{var } s \textsf{ in } (\\ \qquad\qquad s := hd;\\ \qquad\qquad \textsf{rec } L.\textsf{case } s \textsf{ of}\\ \qquad\qquad\qquad \texttt{NULL} \to \textsf{nil}\\ \qquad\qquad\qquad \texttt{NODE}(n) \to (s := n.getNext; L))]\end{array}$$

We define four operations on collections: the initializer $init$, which sets the collection identifier (a string); the $getId$ operation, which returns the collection identifier; the add operation, which adds a new element to the collection; and the $scan$ operation, which traverses the linked list, visiting each node in sequence. We model ADT "objects" by tuples of closures sharing memory locations, and "classes" by object generating functions, along standard lines. In our language, tuple fields are bound to expressions (quoted code), to be evaluated only after field selection (as e.g. in [35]). The **var** x **in** e block creates a ML-style heap allocated variable, where the created cell can survive the lifetime of the body e, embedded in the value returned (cf. **let** $x = \textsf{ref}(\textsf{nil})$ **in** e in ML).

In the code shown above, the private hd variable refers to the head of the linked list, and is shared by the add and $scan$ operations. We represent references to list elements by variant values, with options $\texttt{NULL}(\textsf{nil})$ abbreviated NULL and representing the null reference, and $\texttt{NODE}(n)$, representing a list node. In $\texttt{NODE}(n)$, n is a tuple with fields $setElt$, $getElt$, $setNext$, and $getNext$. Notice that the latter two fields access the heap variable $next$, which references the next node, if any, and is local to the given node.

Using standard functional and product types we would assign to $newColl$ a type as $\texttt{0}{\to}SC$, where 0 is a "unit" type, and SC a record type representing the collection ADT "objects". Such a record type would essentially specify a flat interface, listing the operations available, each one modeled as a typed record field. In our system, types specify (fine grained) value *usage behaviors*, rather than value structure. As a first example consider the type

$$SC \triangleq init{:}\texttt{str}{\mapsto}\texttt{0}\,;(getId{:}\texttt{str} \,\&\, add{:}\texttt{nat}{\mapsto}\texttt{0} \,\&\, scan{:}\texttt{0})^*$$

The type SC specifies a possible usage behavior offered by a collection. Intuitively, the type SC says that collections may be used by first calling the $init$ "method", and then the $getId$, add, and $scan$ "methods", in iterated choice. First, it offers a label selection usage, denoted by the label selection type $init{:}\texttt{str}{\mapsto}\texttt{0}$. The usage consists in selecting the $init$ label to get a value of type $\texttt{str} \mapsto \texttt{0}$. The stop type 0 specifies that no usage is available.

The operator $(-){\mapsto}(-)$ is our primitive functional type. $U \mapsto V$ is a type for functions which do not interfere unsafely with their argument, and specifies a *single usage* of a value as a function, at the appropriate argument U and return type V. The $U{\mapsto}V$ does not correspond to a standard function type $U{\to}V$, which can nevertheless be encoded in our system as will become clear later. It does neither correspond to the linear arrow, nor to the arrows of separation [33, 30] or bunched logic [29], even if it is closely connected to all of these. In a stateful, concurrent programming world, there are too many ways of using a function object. In conjugation with other type constructors, the functional type $U{\mapsto}V$ allows much of such variety to be modularly approached.

Sequencing of behaviors is expressed by the sequential separation type constructor $(-)\,;(-)$. A value typed by $U\,;V$ first offers to clients the usage behavior U and only after V. In our example, after the $init{:}\texttt{str}{\mapsto}\texttt{0}$ usage behavior, the collection value offers a usage $(getId{:}\texttt{str} \,\&\, add{:}\texttt{nat}{\mapsto}\texttt{0} \,\&\, scan{:}\texttt{0})^*$. This last type specifies the iterated choice between the selection of fields $getId$, add, and $scan$, each one yielding a value of respectively type str, $\texttt{nat} \mapsto \texttt{0}$, and 0. Choice between alternative behaviors is expressed using intersection types. A value typed by $U \,\&\, V$ offers to clients the choice between behaviors U and V, since it can provide both U and V, alternatively. The star type U^* denotes the iteration of U, defined by $U^* \triangleq rec(X)(0 \,\&(U\,;X))$.

Clearly, the type operators just described may express rich sequential protocols for program values. Still, they are not expressive enough to address aliased or concurrent usages, due to the strict linearity they enforce. A more flexible specification would allow, after initialization, the $getId$ operation to be always available for execution, concurrently with a add or a $scan$ operation. On the other hand, a concurrent execution of the add and $scan$ operations may lead to unsafe interference, due to a read/write race on hd. To express this behavioral usage we use the parallel separation type constructor $(-)\,|(-)$.

In general, a $U \mid V$ type asserts that behaviors U and V may be safely used by causally independent clients, either due to aliasing or concurrency, without incurring in unsafe interference. Such a parallel usage only completes when both behaviors U and V complete. Exploring the parallel type, we may assign to function $newColl$ the more flexible type $\texttt{0}{\mapsto}CC$ where

$$CC \triangleq (init{:}\texttt{str}{\mapsto}\texttt{0})\,;(!getId{:}\texttt{str}\,|(!scan{:}\texttt{0}\,;add{:}\texttt{nat}{\mapsto}\texttt{0})^*)$$

The type CC asserts that, after initialization, a collection provides two independently usable behaviors, one of type $!getId{:}\texttt{str}$ and other of type $(!scan{:}\texttt{0}\,;add{:}\texttt{nat}{\mapsto}\texttt{0})^*$, composed using the parallel separation type constructor $(-)\,|(-)$.

The type constructor $!(-)$, used in $!getId{:}\texttt{str}$ and $!scan{:}\texttt{0}$, specifies an unbounded number, possibly zero, of separated parallel usages (parallel in the same sense of $(-)\,|(-)$). In particular, the type $!getId{:}\texttt{str}$ allows an unbounded number of (possibly concurrent) aliases to access the field $getId{:}\texttt{str}$. Then, the type $(!scan{:}\texttt{0}\,;add{:}\texttt{nat}{\mapsto}\texttt{0})^*$ specifies a usage consisting of the interleaved repetition of some parallel usages of the $scan{:}\texttt{0}$ behavior followed by the $add{:}\texttt{nat}{\mapsto}\texttt{0}$ behavior. Only after all the $scan$ operations selected in the $!scan{:}\texttt{0}$ phase conclude, will $add{:}\texttt{nat}{\mapsto}\texttt{0}$ become again available. Since there is no obligation to select some $scan{:}\texttt{0}$ operation in the $!scan{:}\texttt{0}$ phase, the type CC also allows any number of $add{:}\texttt{nat}{\mapsto}\texttt{0}$ operations to be sequentially performed.

Notice that the behavior of a newly created collection c is completely separated, or isolated, from context: no behavioral dependencies exist between $c{:}CC$ and the behavior of other values in a running program using the collection. We denote isolation by a type operator $\circ(-)$, which also plays an important role in our framework. We thus assign to the function $newColl$ the type $\texttt{0} \mapsto \circ CC$.

Let us now consider some code snippets using the collection type just defined, and discuss valid (and invalid) typings.

$$\textsf{let } c = newColl\ [] \textsf{ in } (c.init\ \text{“}my\text{”}); c.scan; (c.add\ 1)$$

$$\textsf{let } c = newColl\ [] \textsf{ in } (c.init\ \text{“}my\text{”}); (c.add\ 1); c.getId; c.scan$$

Both code fragments are validated by our type system. In the first one, it is clear that the usage of c follows the intended type. In the second one, the intended usage type of c is also not violated, even if behaviors that appear parallel separated in the type (e.g., $c{:}getId{:}\texttt{str}$ and $c{:}scan{:}\texttt{0}$) are sequentially used in the code. Clearly, if a value may be safely used according to $U \mid V$, it may also be safely used according to $U\,;V$. Subsumption principles as this one are captured by subtyping, a pre-order on types written $U <: V$. In particular, subtyping satisfies the exchange law [17]

$$(A\,;C) \mid (B\,;D) <: (A \mid B)\ ;(C \mid D)$$

of which $U \mid V <: U\,;V$ is a special case.

The next examples illustrates behavioral "borrowing", where fragments of the behavior of c are temporarily used by some function, before being given back to the caller context.

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad \textsf{let } f = \lambda x.(x.init\ \text{“}your\text{”})\ \textsf{in}\ (f\ c); (c.add\ 2)\end{array}$$

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad \textsf{let } g = \lambda x.(x.scan)\ \textsf{in}\\ \qquad (c.init\ \text{“}my\text{”}); (g\ c); c.scan; (c.add\ 2); (g\ c)\end{array}$$

In the second case the borrowing function is used twice, at different places of the global behavior. The borrowed type is declared in the function domain, e.g., $f{:}(init{:}\texttt{str} \mapsto 0) \mapsto \texttt{0}$. On the other hand

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad \textsf{let } h = \lambda x.(x.init\ \text{“}your\text{”})\ \textsf{in}\ (c.add\ 2); (h\ c)\end{array}$$

attempts to use add before $init$, and is rejected by our type system.

More interesting examples illustrate borrowing of behavior through the store, which our type system is able to handle in a natural way, even in a higher-order setting. Consider the following code snippet. It respects the expected behavioral constraints on the value c and the heap allocated variable a, even if the behavior of c is temporarily accessible at a, and is in fact typable in our system.

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in var } a \textsf{ in}\\ \quad (a := c; (a.init\ \text{“}my\text{”}); (a.add\ 1); (a.add\ 1); c.scan))\end{array}$$

Heap variables are assigned behavioral separation types expressed in terms of use $\texttt{use}$, read $\texttt{rd}(U)$ and write $\texttt{wr}(U)$ capabilities, and related by subtyping axioms. We show some of them here

$$\texttt{var} <: \texttt{use}\,;\texttt{var} \quad \texttt{use} <: \texttt{wr}(U)\,;\texttt{rd}(U) \quad \texttt{rd}(!U) <: !\texttt{rd}(U)$$

$$\texttt{rd}(U\,;V) <: \texttt{rd}(U)\,;\texttt{rd}(V) \quad \texttt{rd}(U \mid V) <: \texttt{rd}(U) \mid \texttt{rd}(V)$$

The first two axioms say that a single use of a variable consists in writing on it a value of type U, followed by a matching read phase. The following axioms specify how the reading phase may be behaviorally separated, depending on the type of the stored value.

The next example illustrates higher-order borrowing through the store, the function attached to the add field is itself stored in memory, before being called.

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in } ((c.init\ \text{“}my\text{”});\\ \quad \textsf{var } a \textsf{ in } (a := c.add; (a\ 1); c.scan))\end{array}$$

This code does not violate any constraints imposed by the type of c, even if a collection "method" (a functional value) is extracted by selecting $c.add$ and stored in the temporary heap variable a.

These last two examples get past our typing discipline, because the type system keeps track of global separation constraints between all the values in the scope, and relies on sequential and parallel frame reasoning to *locally* replace behaviors in behavioral separation type assertions. In the last example, the function of type $\texttt{nat} \mapsto \texttt{0}$ is required to be used (exactly once) before $scan$ is selected in c, even if its behavior is temporarily stored in the variable a, respecting the initial footprint of $c.add$ of in the global behavioral separation type. The type pre-condition of $a := c.add$ is $a{:}\texttt{use} \mid (c{:}add{:}\texttt{nat} \mapsto \texttt{0}\,;c{:}scan{:}\texttt{0}\,;\cdots)$, and the post-condition is $a{:}\texttt{rd}(add{:}\texttt{nat} \mapsto \texttt{0})\,;c{:}scan{:}\texttt{0}\,;\cdots$. Notice that this last type sequentially constrains the behaviors of a and c, forcing a to be read before using c. The ability to specify global separation constraints, involving several values, seems essential for the expressiveness of our system. As a further illustration of this point, consider

$$\textsf{let } c = newColl\ [] \textsf{ in let } m = c.init \textsf{ in } c.scan$$

Here, the $init$ qualifier is selected, but since the associated functional behavior (bound to m) is not actually exercised, the initialization of the local hd variable is not performed, causing the $scan$ operation to "crash" in the **case** expression. Of course, this code does not typecheck under our current assumptions, since it does not preserve the frame conditions imposed by the intended behavioral separation protocols. Indeed, $c.scan$ would need to be typed under the pre-condition $(m{:}\texttt{str} \mapsto \texttt{0})\,;c{:}scan{:}\texttt{0}\,;\cdots$, which states that m must be used (as a function) before progressing with the continuation behavior of collection c, which is not possible.

The type system systematically uses local reasoning and frame principles on behavioral separation assertions to compute the effects of program fragments in a modular way. As a further example, consider a case of function application, as in

$$\begin{array}{l}\textsf{var } s \textsf{ in } (s := \text{“}hi\text{”};\\ \quad \textsf{let } F = \lambda x.(\textsf{let } c = newColl\ [] \textsf{ in } (c.init\ x; c)) \textsf{ in}\\ \quad (\textsf{let } u = (F\ s) \textsf{ in } (u.add\ 1)))\end{array}$$

Function F returns an initialized collection. Before typing $(F\ s)$ the type assertion is $s{:}\texttt{rd}(\texttt{str})\,;\texttt{var} \mid F{:}(\texttt{str} \mapsto \circ CC)$. We assume that the type of strings $\texttt{str}$ is shareable ($\texttt{str} <: !\texttt{str}$), so that reading from s does not "empty" the variable. To type function application, we collect the footprints of argument and function as $(s{:}\texttt{rd}(\texttt{str}) \mid F{:}(\texttt{str} \mapsto \circ CC))\,;s{:}\texttt{rd}(\texttt{str})\,;\texttt{var}$. After typing $\textsf{let } u = (F\ s)$, the type assertion is $u{:}\circ CC\,;s{:}\texttt{rd}(\texttt{str});\texttt{var}$. Apparently, this says that u must be fully used as $\circ CC$ before the variable s can be read again, which is not sensible. However, since the behavior $\circ CC$ is isolated, as expressed by $\circ(-)$, any use of s cannot causally depend upon it: by the subtyping principle $(\circ U)\,;V <: (\circ U) \mid V$, we actually reach $u{:}\circ CC \mid s{:}\texttt{rd}(\texttt{str});\texttt{var}$. Isolated types offer a safe escape from the strict locality discipline, allowing isolated behaviors to be fully (and soundly) separated from a global type in which they might appear embedded. As a further illustration, the following code is typable by assigning $\circ CC \mapsto 0$ to f so that it captures the argument full behavior (and stores it in heap variable a) rather than borrowing it.

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad \textsf{var } a \textsf{ in let } f = \lambda x.a := x \textsf{ in } ((f\ c); (a.init\ \text{“}y\text{”}))\end{array}$$

We now consider some examples with concurrency. The parallel expression $(e_1 \| e_2)$ clearly brings up the possibility of interference. The next couple of examples are safe, and type-check in our system

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad ((c.init\ \text{“}my\text{”}); (c.add\ 1); (c.scan \| c.scan))\end{array}$$

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in let } f = \lambda x.(x.scan) \textsf{ in}\\ \quad ((c.init\ \text{“}my\text{”}); ((f\ c) \| c.scan); (c.getId \| (c.add\ 2)); (f\ c))\end{array}$$

On the other hand, the code snippets

$$\textsf{let } c = newColl\ [] \textsf{ in } ((c.init\ \text{“}my\text{”}); ((c.add\ 1) \| (c.scan))$$

$$\begin{array}{l}\textsf{let } c = newColl\ [] \textsf{ in}\\ \quad \textsf{let } f = \lambda x.((x.add\ 0) \| (c.add\ 1)) \textsf{ in } ((c.init\ \text{“}my\text{”}); (f\ c))\end{array}$$

violate the intended behavioral separation constraints, and are rejected by the type system. In the last case, although the function f may be given type $(add{:}\texttt{nat} \mapsto 0) \mapsto 0$, the application $(f\ c)$ is not typable, since the type CC cannot provide footprints for separately typing function and argument (no parallel separated add capabil-

ities are available on c). On the other hand, the following similar looking code is safe and well-typed.

$$\textbf{let } c = newColl\ [] \textbf{ in}$$
$$\quad \textbf{let } f = \lambda x.(x.scan \| c.scan) \textbf{ in } ((c.init\ \text{“}my\text{”}); (f\ c))$$

The form $(e_1 \| e_2)$ is actually derived from primitive **fork** and **wait** thread-based concurrency constructs. Thread references are first-class values in our language, created by the **fork**(e) expression. The interesting operation on threads is to wait for their return value. Let

$$\textbf{let } c = newColl\ [] \textbf{ in } ((c.init\ \text{“}my\text{”});$$
$$\quad \textbf{var } a \textbf{ in } (a := \textbf{fork}(c.scan); c.scan; \textbf{wait}(a); (c.add\ 1)))$$

This code is well-typed under the current typing assumptions for c, as enforced by the parallel and sequential frame conditions, since the footprints of the **fork** and **wait** expressions match the expectations of the global behavioral type. On the other hand,

$$\textbf{let } c = newColl\ [] \textbf{ in } ((c.init\ \text{“}my\text{”});$$
$$\quad \textbf{var } a \textbf{ in } (a := \textbf{fork}(c.scan); (c.add\ 1); \textbf{wait}(a)))$$

is not well-typed under the same assumptions: it breaks the separation constraints required by the type of c. Such type requires $c.scan$ and $(c.add\ 1)$ to be sequentially separated, but overlapping may occur at runtime, causing unsafe interference.

As noticed before, it is not sound to assign to our collection a type allowing the add operation to be used concurrently with $scan$ operations. That would violate the intended usage protocol of the internal state, causing a write/read race on heap variable hd. However, our language allows “critical regions” in the code to be sequentialized, and eventually typed by invariant-based separation. Invariant-based separation allows isolated behaviors to be repeatedly interleaved in the global behavior, as far as the associated invariant conditions, expressed by a conveniently chosen type assertion, are preserved. In our example, this could be achieved, e.g., by adding a new local heap variable inv to collection, and wrapping the uses of hd in add and $scan$ as follows

$$add = \lambda e.\textbf{sync}(inv)(\ldots)$$
$$scan = \textbf{var } s \textbf{ in } (\textbf{sync}(inv)s := hd; \textbf{rec } L.\textbf{case } s \textbf{ of } \cdots)$$

To type this code, we associate an assertion $hd{:}\texttt{rd}(!\circ PNode)\,\texttt{;var}$ to the heap variable inv, which expresses an invariant condition protecting its footprint. Our type system is then able to assign to the concurrent collection the following type, which allows, after initialization, operations $getId$, add and $scan$ to be unboundedly aliased, or shared by several active threads.

$$C \triangleq (init{:}\texttt{str} \mapsto \texttt{0})\,\texttt{;}(!getId{:}\texttt{str} \mid !scan{:}\texttt{0} \mid !add{:}\texttt{nat} \mapsto 0)$$

Of course, the main novelty to highlight here is not the familiar reasoning technique for lock invariants, but the way our type discipline elegantly captures it. Even if based on a few fairly general principles, it can be effectively used to reason about safety properties of higher-order concurrent programs involving difficult to handle scenarios of aliasing and concurrency. We are not aware of related proposals, able to address the same set of (realistic) programming idioms, and based on a a similarly general foundation, as we have achieved here. This paper makes the following contributions:

- We motivate and introduce the concept of behavioral separation as a general principle for disciplining interference in higher-order imperative concurrent programs.
- We present a behavioral separation type system for a λ-calculus with imperative and concurrency constructs. We show soundness of the system, proving type preservation under reduction (Theorems 4.3 and 4.5) and progress (Theorem 4.4).
- We illustrate, by means of many examples, how our type system, even if based on a small set of very general primitives, is already able to tackle fairly challenging program idioms.

$m, n, t \ldots \in$		Λ	(*Names*)
$x, y, z \ldots \in$		$\mathcal{V}$	(*Variables*)
$a, b, c \ldots \in$		$\Lambda \cup \mathcal{V}$	(*Identifiers*)
$l, s \ldots \in$		$\mathcal{L}$	(*Labels*)
$X, Y \ldots \in$		χ	(*Expression Vars*)
e, f	::=	x	(*Variable)*
	|	$\lambda x.e$	(*Abstraction)*
	|	$e_1 e_2$	(*Application)*
	|	$\textbf{let } x = e_1 \textbf{ in } e_2$	(*Definition)*
	|	$\textbf{var } a \textbf{ in } e$	(*Heap variable decl)*
	|	$a := v$	(*Assignment)*
	|	a	(*Dereference)*
	|	$[l_1 = e_1, \ldots]$	(*Tupling)*
	|	$e.l$	(*Selection)*
	|	$l(e)$	(*Variant)*
	|	$\textbf{case } e \textbf{ of } l_i(x_i) \to e_i$	(*Conditional)*
	|	$\textbf{rec}(X)e$	(*Recursion)*
	|	X	(*Recursion variable)*
	|	$\textbf{fork } e$	(*New thread)*
	|	$\textbf{wait } e$	(*Wait)*
	|	$\textbf{sync}(a)e$	(*Synchronized block)*
	|	$sy(a)e$	(*inSynchronized block)*

Figure 1. Programming Language.

3. Programming Language

Our programming language, presented in Figure 1, is a λ-calculus with mutable heap allocated variables, tuples, variants, and concurrency primitives. To keep it close to familiar high-level languages such as Java, we consider unstructured (fork/join) thread-oriented concurrency primitives, and a synchronization construct. Our language is fairly simple yet expressive enough to support challenging imperative higher-order concurrent programming idioms.

To formally define it, we assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, and an infinite set of method labels $\mathcal{L}$. Names in Λ are used to identify threads and heap locations. For simplicity sake, we omit basic values, and literals for booleans or integers, their addition as primitives is straightforward.

The functional core includes abstraction $\lambda x.e$ and application $e_1 e_2$, following call-by-value evaluation. The tuple expression $[l_1 = e_1, \ldots]$ denotes a record collecting expressions e_i, each one qualified by the label l_i. As in [34], and without any loss of generality, we consider lazy tuples, where the expression e_i is only evaluated after selection of the label l_i. Lazy tuples allow different qualifications of the same entity to be subject to different interference constraints, both along the time and space dimensions, and are convenient for encoding objects as tuples of ”methods”. The empty tuple [] is abbreviated by **nil**.

The **let** expression represents local definition and sequential composition: in $\textbf{let } x = e_1 \textbf{ in } e_2$, the subexpression e_1 is executed first, its result bound to x, and only after e_2 gets evaluated. We abbreviate $\textbf{let } x = e_1 \textbf{ in } e_2$ by $(e_1; e_2)$ if x is not free in e_2.

The construct $l(e)$ injects the value of expression e into the variant label l. The **case** construct corresponds to a standard destructor for labeled sum types. The expression $\textbf{case } e \textbf{ of } l_i(x_i) \to e_i$ first evaluates e to a variant value $l_i(v)$ (if this is not the case, execution will get stuck). Then, v is bound to x_i, e_i evaluated and its value returned as the result of the whole **case** expression.

Variable declaration $\textbf{var } a \textbf{ in } e$, variable access v, and assignment $a := x$ are interpreted as usual. We adopt a simple form of assignment $a := v$ where v must be a value, and often use $a := e$ as an abbreviation of $\textbf{let } x = e \textbf{ in } a := x$.

The expression **fork** e spawns a new thread dedicated to the evaluation of expression e, and immediately returns the new thread identifier to the caller. Both the calling thread and the newly created one proceed execution concurrently. The expression **wait** e suspends the caller until the thread resulting from evaluating e terminates with some result (if ever). Such result will then be returned as the result of the **wait** expression.

Our language includes a simple synchronization primitive. The primitive relies on endowing each heap variable with a lock. Such locks are available for flexible use in programs, pretty much as object locks are used, e.g., in Java programs (either to lock the variable itself, or to protect any other relevant region of the state). At each moment, a lock may be either taken or free. The expression $\mathbf{sync}(n)e$ evaluates the expression e in exclusion, using the lock associated to heap variable n (cf. the Java synchronized block); only one thread may acquire the lock of n in linear ("write") mode.

To track entry and exit of synchronization blocks in the operational semantics, we use the auxiliary construct $sy(n)e$. No occurrences of this construct, or of location or thread names are expected to appear in source programs, these elements belong to the runtime syntax of the full language, as shown in Figure 1.

The operational semantics of our programming language is defined by a reduction system using evaluation contexts. A state consists of a pair $h;\ T$, where h is a heap and T is a multiset of threads. A reduction step has the form

$$\boxed{h;\ T \to h';\ T'} \quad (h;\ T \text{ reduces to } h';\ T')$$

expressing a computation step from state $h;\ T$ to state $h';\ T'$. In any such step, new heap cells may be allocated, threads may be created, evolve, or terminate. Each thread in T is represented by an element of the form $t\langle e\rangle$, where t is the thread name (from Λ), and e is the runtime expression under execution by the thread. We write $t\langle e\rangle \cdot T$ for the multiset union of T and $t\langle e\rangle$. A heap h is a mapping from heap locations (names) to values. Each heap binding $n_k \mapsto v$ also has an integer-valued counter k associated, to be used as a semaphore, important to support the synchronization primitives. We write $h(n_k \mapsto v)$ to denote a heap such that $n_k \mapsto v \in h$, and $h[n_k \mapsto v]$ to denote the heap obtained from h after storing v at location n, with lock value k. We now introduce values $\mathcal{V}$ and evaluation contexts $\mathcal{E}$, given by

$$\begin{array}{lcl} v, u \in \mathcal{V} & ::= & \lambda x.e \mid [\ l_1 = e_1, \dots\ ] \mid l(v) \mid t \mid x \\ \mathcal{E} & ::= & \Box \mid \mathbf{let}\ x = \mathcal{E}\ \mathbf{in}\ e \mid \mathcal{E}.l \mid \mathcal{E}e \mid v\mathcal{E} \mid l(\mathcal{E}) \\ & \mid & \mathbf{wait}(\mathcal{E}) \mid sy(n)\mathcal{E} \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ \overline{l_i(x_i) \to e_i} \end{array}$$

A value in our language is either an abstraction, a tuple (including the empty tuple **nil**), a variant value, a thread name, or a variable. The rules defining the reduction relation are presented in Figure 2. We write $\{v/x\}$ for the capture avoiding substitution of v for x, defined as expected. In all reduction rules, we denote by $t\langle e\rangle \cdot T$ a multiset that contains a thread $t\langle e\rangle$ and a rest T. Notice that there is no order assumed between elements in the thread multiset, so any thread may be (non-deterministically) scheduled at each reduction step. Most reduction rules are easy to interpret, and do not deserve much explanation. In rules (Red var) and (Red fork) the conditions $(\boldsymbol{\nu} n)$ and $(\boldsymbol{\nu} s)$ state that names n and s must be fresh in the respective left hand side.

Rules (Red sync*) rely on the integer-valued lock associated to the each heap location n. When the lock is zero, the lock is free. Rule (Red syncin) checks that the lock k associated to v is free, before decrementing it to -1, and allowing execution to enter the critical region e: the expression $sy(n)e$ signals that the execution of e is taking place inside a critical region protected by the lock of n. The lock is released after the body of the $sy(n)e$ block reduces to a value u, in rule (Red syncout).

(Red rec)
$h;\ t\langle \mathcal{E}[\ \mathbf{rec}(X)e\ ]\rangle \cdot T \to h;\ t\langle \mathcal{E}[\ e\{\mathbf{rec}(X)e/X\}\ ]\rangle \cdot T$
(Red let)
$h;\ t\langle \mathcal{E}[\ \mathbf{let}\ x = v\ \mathbf{in}\ e\ ]\rangle \cdot T \to h;\ t\langle \mathcal{E}[\ e\{v/x\}\ ]\rangle \cdot T$
(Red beta)
$h;\ t\langle \mathcal{E}[\ (\lambda x.e)v\ ]\rangle \cdot T \to h;\ t\langle \mathcal{E}[\ e\{v/x\}\ ]\rangle \cdot T$
(Red sel)
$h;\ t\langle \mathcal{E}[\ [\ \overline{l = e}\ ].l_i\ ]\rangle \cdot T \to h;\ t\langle \mathcal{E}[\ e_i\ ]\rangle \cdot T$
(Red case)
$h;\ t\langle \mathcal{E}[\ \mathbf{case}\ l_i(v)\ \mathbf{of}\ \overline{l(x) \to e}\ ]\rangle \cdot T \to h;\ t\langle \mathcal{E}[\ e_i\{v/x_i\}\ ]\rangle \cdot T$
(Red var)
$h;\ t\langle \mathcal{E}[\ \mathbf{var}\ a\ \mathbf{in}\ e\ ]\rangle \cdot T \to h[n \mapsto \mathbf{nil}];\ t\langle \mathcal{E}[\ e\{n/a\}\ ]\rangle \cdot T \quad (\boldsymbol{\nu} n)$
(Red assign)
$h(n_k \mapsto v);\ t\langle \mathcal{E}[n := u]\rangle \cdot T \to h[n_k \mapsto u];\ t\langle \mathcal{E}[\mathbf{nil}]\rangle \cdot T$
(Red deref)
$h(n_k \mapsto v);\ t\langle \mathcal{E}[n]\rangle \cdot T \to h(n_k \mapsto v);\ t\langle \mathcal{E}[v]\rangle \cdot T$
(Red fork)
$h;\ t\langle \mathcal{E}[\mathbf{fork}\ e]\rangle \cdot T \to h;\ t\langle \mathcal{E}[s]\rangle \cdot s\langle e\rangle \cdot T \quad (\boldsymbol{\nu} s)$
(Red wait)
$h;\ t\langle \mathcal{E}[\mathbf{wait}\ s]\rangle \cdot s\langle v\rangle \cdot T \to h;\ t\langle \mathcal{E}[v]\rangle \cdot T$
(Red syncin)
$h(n_0 \mapsto v);\ t\langle \mathcal{E}[\mathbf{sync}(n)e]\rangle \cdot T \to h[n_{-1} \mapsto v];\ t\langle \mathcal{E}[sy(n)e]\rangle \cdot T$
(Red syncout)
$h(n_{-1} \mapsto u);\ t\langle \mathcal{E}[sy(n)v]\rangle \cdot T \to h[n_0 \mapsto u];\ t\langle \mathcal{E}[v]\rangle \cdot T$

Figure 2. Reduction.

4. Type System

In this section, we technically present our type system. As already discussed, types describe behavioral usages of values. We start by systematically introduce each type operator, discussing on the way their basic algebraic properties and related subsumption laws.

DEFINITION 4.1 (Types). *Type operators are given by*

$$\begin{array}{lcl} T, U & ::= & \mathtt{0} \quad \textit{(stop)} \mid T \mapsto V \quad \textit{(function)} \\ & \mid & T\,;U \quad \textit{(sequential)} \mid T \mid U \quad \textit{(parallel)} \\ & \mid & T \,\&\, U \quad \textit{(intersection)} \mid l{:}T \quad \textit{(qualification)} \\ & \mid & \oplus_{l \in I} l{:}T_l \quad \textit{(sum)} \mid !T \quad \textit{(shared)} \\ & \mid & \circ T \quad \textit{(isolated)} \mid \tau(T) \quad \textit{(thread)} \\ & \mid & \mathtt{rec}(X)T \quad \textit{(recursion)} \mid X \quad \textit{(recursion var)} \end{array}$$

We assume given some primitive type constructors $\mathtt{c}$, $\mathtt{c}(U)$, such as `str`, `nat`, to represent basic data types, and `var`, `rd`(U), etc, to represent behavioral separation types for heap allocated variables. The *stop type* `0` types any value exposing no behavioral capability, in particular it types **nil**. The *sequential type* $T\,;U$ asserts of a value that it can be safely used first according to type T, and only afterwards according to type U. The sequential type expresses behavioral separation along the temporal dimension. Sequential types induce a monoid with identity `0` in the type structure, expressed by

$$U\,;(V\,;T) \mathrel{<:>} (U\,;V)\,;T \quad U\,;\mathtt{0} \mathrel{<:>} U \quad \mathtt{0}\,;U \mathrel{<:>} U$$

The *parallel type* $T \mid U$ asserts of a value that it can be subject to two safe independent parallel usages, specified by type T and type U respectively. By "independent parallel usage" of a value we mean any form of sharing, arising not just in concurrent programs, but also in sequential programs, due to aliasing. The parallel type

thus expresses behavioral separation along the spatial dimension. It builds on the fundamental idea of separation (cf. separation logic [33]), but focusing on the independence of usage behaviors, as perceived from a "client" viewpoint , rather than on the disjointness of underlying resources. To highlight this understanding of $T \mid U$, we refrain from using the notation $T * U$. A key insight on $(- \mid -)$ is that behaviors typed by parallel separation do not interfere in unsafe ways, even if they rely on shared underlying resources.

An usage of type $U \mid V$ only concludes when both U and V conclude. In a type such as $(U \mid V)\,;T$, the usage T is only available when $(U \mid V)$ conclude. So our type language provides an abstract way of splitting "permissions", without using explicit fractions (cf. [9]). Parallel types induce a commutative monoid with identity $\mathtt{0}$

$$U \mid (V \mid T) \mathrel{<:>} (U \mid V) \mid T \quad U \mid V \mathrel{<:>} V \mid U \quad U \mid \mathtt{0} \mathrel{<:>} U$$

Sequential and parallel composition are related by the exchange law [4, 17], the following causality preserving distribution principle

$$(A\,;C) \mid (B\,;D) \mathrel{<:} (A \mid B)\ ;(C \mid D)$$

A special case is the familiar interleaving law $U \mid V \mathrel{<:} V\,;U$.

The *recursive* type $rec(X).A$, where X must occur guarded in A, is interpreted in a standard (co-inductive) sense.

The *shared* type $!T$ asserts of a value that it can be safely subject to an unbounded number of parallel separated usages (cf. $(- \mid -)$), each one specified by type T. In particular, it may be unboundedly aliased at type T. The following laws hold for the type $!T$

$$!U \mathrel{<:} U \quad !U \mathrel{<:} !!U \quad \mathtt{0} \mathrel{<:} !\mathtt{0} \quad !U \mid !V \mathrel{<:} !(U \mid V)$$
$$!U \mathrel{<:} \mathtt{0} \quad !U \mathrel{<:} !U \mid !U$$

Notice that $!(-)$ satisfies the fundamental co-monadic laws for the linear logic exponential; so our notation highlights the connection.

The *function* type $T \mapsto V$ asserts of a value that it can be safely used (once) as a function that when given as argument a value of type T, exercises on it a usage of type T, and returns a result of type V. Type $U \mapsto V$ is adjoint to $U \mid V$, so that the behavioral separation interpretation ensures the intended safety property: no unsafe interference can arise even if function and argument share state-full resources, since they are behaviorally separated. So $U \mapsto V$ is a type for functions that do not unsafely interfere with their arguments, as aimed in the sharing interpretation of $\rightarrow\!\!*$ in [29], but does not completely forbid sharing to enforce safety of interference. Moreover, unlike in the linear logic interpretation of the arrow $(U \multimap V)$, a function of type $U \mapsto V$ can use its parameter more than once, as long as it globally respects the behavioral type U.

The *isolated* type $\circ T$ asserts of a value that it may be used as specified by the type T, but, more crucially, that such usage is fully isolated, not subject to any external (global) constraints. A value of type $\circ T$ is completely separated (in terms of behavior) from the rest of the "world". In particular, $\circ T$ says that the usage T is not borrowed from some larger computation. We may see a value of type $\circ T$ as offering a self-contained suspended behavior of type T, that may be used at any future step in the computation. No liveness commitments are imposed on client code to use a value of type $\circ T$, unless it actually starts to use it at type T. In particular, a value of type $\circ T$ may be safely dropped out, since nothing causally depends on it: a safe use for a value of type $\circ T$ is not to use it at all. Moreover, since nothing can causally depend on a value of type $\circ T$, we expect the law $(\circ U)\,;V \mathrel{<:} (\circ U) \mid V$ to hold. We also have

$$\mathtt{0} \mathrel{<:} \circ\mathtt{0} \quad \circ A \mid \circ B \mathrel{<:} \circ(A \mid B) \quad \circ A \mathrel{<:} A \quad \circ A \mathrel{<:} \circ\circ A$$
$$\circ A \mathrel{<:} \mathtt{0} \quad !\circ A \mathrel{<:} \circ !A \quad (\circ A \mid B)\,;C \mathrel{<:} \circ A \mid (B\,;C)$$

The first five laws express familiar algebraic principles (cf. the basic laws for $!(-)$). The last two laws are proper to $\circ(-)$. In particular, the last one expresses the key property of $\circ(-)$, global behavioral isolation: a behavior of type $\circ T$ is isolated, and can be freely used anytime, concurrently with anything. No other behavior can causally depend from a behavior of type $\circ T$. By the exchange law, we may derive the "postponing" principle $(\circ A)\,;B \mathrel{<:} B\,;(\circ A)$.

The (linear) *intersection* type $U \,\&\, V$ asserts of a value that it may be safely used according to type U and according to type V. The client code using such a value can therefore freely decide to pick either the U or the V behavior (but not both, since we exploit a linear interpretation of $\&$). The following basic laws hold.

$$U \,\&\, V \mathrel{<:} U \quad U \,\&\, V \mathrel{<:} V \quad U \mathrel{<:} U \,\&\, U$$

The *qualified* type $l{:}T$ asserts of a value that it offers a usage of type T under the label l. It describes a label selection capability of a tuple, classifying the usage type of the value in field l. Following [35], general tuple types may be defined by combining qualified types with other type constructors, e.g. $l_1{:}T_1 \,\&\, \ldots \,\&\, l_n{:}T_n$.

The *sum* type $\oplus_{l \in i} l_i{:}T_i$ asserts of a value that it is a labeled value that can be used according to type T_i if it is labeled with l_i. Client code using such a value must branch on the possible labels, before actually using the selected behavior. The sum type thus corresponds to a standard labeled disjoint union, useful to describe variants or options. We do not assume specific subtyping principles for sum types. We abbreviate $(\mathtt{NULL}{:}\mathtt{0} \oplus \mathtt{NODE}{:}U)$ by $\mathtt{Opt}(U)$.

The *thread* type $\tau(T)$ asserts of a value that it references a running thread that upon termination returns a value of type T.

Types for heap allocated variables are conveniently described in our system by specific primitives, expressing usage, write and read capabilities. The type of a freshly allocated heap variable is $\mathtt{var}$. The type $\mathtt{var}$ denotes the generic heap variable usage protocol, and is axiomatized by several subtyping laws, presented in Section 4.1.

As previously discussed, a type classifies a single value. In order to type program expressions, which may use in general several different values, we introduce a notion of *type assertion*. A type assertion corresponds to the usual notion of type environment, assigning types to the various free identifiers in a program. However, our type assertions finely describe behavioral dependencies between the several identifiers in its domain, by placing basic type assignments of the form $x{:}T$ embedded in a larger global type.

DEFINITION 4.2. *Type assertions are given by*

$$A, B ::= x{:}T \mid A\,;B \mid A|B \mid A \,\&\, B \mid\ !A \mid \circ A \mid X \mid \mathtt{rec}(X)A$$

For an example, under the assumptions expressed by the type assertion $(f{:}U \mapsto V\,;y{:}U) \mid z{:}U$, the function f can be applied to z but not to y, so $(f\ z)$ is well typed but $(f\ y)$ is not, since the behavior $y{:}U$ is only available after $f{:}U \mapsto V$ is used.

We denote by $Dom(A)$ the (finite) set of variables appearing in a type assertion A. If A has a singleton domain $\{x\}$ (that is, refers to a single variable x), we implicitly identify it with the singular assertion $x{:}(A)_x$ where the type $(A)_x$ is given by

$$\begin{array}{ll} (x{:}T)_x = T & (A\,;B)_x = (A)_x\,;(B)_x \\ (A \mid B)_x = (A)_x \mid (B)_x & (A \,\&\, B)_x = (A)_x \,\&\, (B)_x \\ (!A)_x = !(A)_x & (\circ A)_x = \circ(A)_x \\ (X)_x = X & (\mathtt{rec}(X)A)_x = \mathtt{rec}(X)(A)_x \end{array}$$

Therefore, we identify, e.g., $x{:}(up\,;dn)$ with $(x{:}up)\,;(x{:}dn)$, and $\mathtt{rec}(X)((x{:}up \mid x{:}dn)\,;X)$ with $x{:}\mathtt{rec}(X)((up \mid dn)\,;X)$.

We define type assertion *contexts* $\mathcal{A}[-], \mathcal{E}[-]$ to be the one hole syntactic contexts associated to type assertions. We also consider *active assertion contexts*, where the hole occurs unguarded, defined

$$\mathcal{E}[-] ::= \Box \mid \mathcal{E}[-]\,;A \mid \mathcal{E}[-] \mid A \mid A \mid \mathcal{E}[-]$$

4.1 Subtyping

Type assertions are related by a subtyping relation. We write $A \mathrel{<:} B$ to state that A is a subtype of B, meaning that the usage behavior B is subsumed by usage behavior A. Intuitively, $A \mathrel{<:} B$ means that

$A \mathrel{<:>} A \mid \mathtt{0} \quad A \mid B \mathrel{<:} B \mid A \quad A \mid (B \mid C) \mathrel{<:>} (A \mid B) \mid C$

$A \mathrel{<:>} A \,;\mathtt{0} \quad A \mathrel{<:>} \mathtt{0} \,; A \quad A \,;(B \,; C) \mathrel{<:>} (A \,; B) \,; C$

$(A \,; C) \mid (B \,; D) \mathrel{<:} (A \mid B) \,; (C \mid D)$

$A \,\&\, B \mathrel{<:} A \quad A \,\&\, B \mathrel{<:} B \quad A \mathrel{<:} A \,\&\, A$

$\mathtt{0} \mathrel{<:} \circ\mathtt{0} \quad \circ A \mathrel{<:} \mathtt{0} \quad \circ A \mathrel{<:} A \quad \circ A \mathrel{<:} \circ\circ A \quad \circ A \mid \circ B \mathrel{<:} \circ(A \mid B)$

$!\circ A \mathrel{<:} \circ !A \quad (\circ A \mid B) \,; C \mathrel{<:} \circ A \mid (B \,; C) \quad !A \mid !B \mathrel{<:} !(A \mid B)$

$\mathtt{0} \mathrel{<:} !\mathtt{0} \quad !A \mathrel{<:} \mathtt{0} \quad !A \mathrel{<:} A \quad !A \mathrel{<:} !!A \quad !A \mathrel{<:} !A \mid !A$

$\mathtt{rec}(X)A \mathrel{<:>} A\{\mathtt{rec}(X)A/X\} \quad \mathtt{var} \mathrel{<:} \mathtt{use} \,; \mathtt{var}$

$\mathtt{use} \mathrel{<:} \mathtt{use} \,; \mathtt{use} \quad \mathtt{use} \mathrel{<:} \mathtt{wr}(U) \,; \mathtt{rd}(U) \quad \mathtt{wr}(\mathtt{0}) \mathrel{<:} \mathtt{0} \quad \mathtt{rd}(\mathtt{0}) \mathrel{<:} \mathtt{0}$

$\mathtt{rd}(U; V) \mathrel{<:} \mathtt{rd}(U); \mathtt{rd}(V) \quad \mathtt{rd}(U \mid V) \mathrel{<:} \mathtt{rd}(U) \mid \mathtt{rd}(V)$

$\mathtt{rd}(!U) \mathrel{<:} !\mathtt{rd}(!U) \quad \mathtt{rd}(\circ U) \,; \mathtt{var} \mathrel{<:} \circ(\mathtt{rd}(\circ U) \,; \mathtt{var})$

Figure 3. Subtyping.

if some value may be safely used according to A then it may also be safely used according to B.

$$\boxed{A \mathrel{<:} B} \quad (A \text{ is a subtype of } B)$$

Notice that subtyping also apply to types by letting $U \mathrel{<:} V$ if and only if $x{:}U \mathrel{<:} x{:}V$. Subtyping axioms, defined in Figure 3, express the basic algebraic laws of the type operators discussed above. We abbreviate by $A \mathrel{<:>} B$ the fact that $A \mathrel{<:} B$ and $B \mathrel{<:} A$. To save space, we abbreviate rules of the form $x{:}U \mathrel{<:} x{:}V$ by $U \mathrel{<:} V$, and omit subtyping congruence rules. All type operators satisfy the expected (covariant) subtyping congruence principles, with some exceptions, e.g., the arrow $U \mapsto V$, which is contravariant in the domain U, and $\mathtt{wr}(V)$, which is contravariant on V (see Appendix).

Particularly interesting are the axioms defining the `var` behavior. We may derive $\mathtt{var} \mathrel{<:} \circ\mathtt{var}$: clearly a fresh heap allocated variable offers an isolated behavior. The first axioms state that a variable can be subject to an unbounded number of uses, each use composed by a write and a read phase. Other axioms specify how the reading phase may be behaviorally separated, depending on the type of the stored value. For example, the axiom for $\mathtt{rd}(U \mid V)$ says that if a value of type $U \mid V$ can be read from the variable, then the variable can also be subject to independent reading at types U and V. This point is extremely important: e.g., a heap variable may be shared or aliased, only if the stored value also may be. Notice that the axiom for `use`, allowing a different type U to be picked at different unfoldings of `var`, naturally support strong updates [2] (updating a heap allocated variable to hold values of unrelated types at different points in time).

4.2 Typing

Type judgments of our system have the form

$$\boxed{A \vdash_z e :: B} \quad (e \text{ types from } A \text{ to } z \text{ in } B)$$

where A and B are type assertions, e is an expression, and the index z is a variable. We refer to A as the pre-condition, and to B as the post-condition of the typing judgment. The behavioral type of (the value of) e, as determined by the type system, appears embedded in assertion B. The variable z stands for such a value, and can only occur free in B (not in A or e). This idea of scoping the return value z over the post-condition already appears in the Hoare triple type of [28], although here the type of z cannot be given apart from B.

For an example, consider

$$a{:}\mathtt{use} \vdash_z (\lambda x.a := x) :: z{:}\circ U \mapsto \mathtt{0} \,; a{:}\mathtt{rd}(\circ U)$$

This judgment asserts that evaluating the expression $(\lambda x.a := x)$ in a state providing $a{:}\mathtt{use}$ returns a functional value (identified by z in the post-condition) that must be used exactly once before the heap variable a can be read. We now progressively present the several rules of our type system, discussing each one on the way.

4.2.1 Structural Rules

The identity axiom

$$x{:}U \vdash_z x :: z{:}U \ (Id)$$

asserts that access to the identifier x simply returns the associated value, usable according to the type in the pre-condition (N.B: (Id) has the proviso that U is not an heap variable type: typing rules for heap variable dereference are given below). The type system includes four other structural rules. A crucial one is subtyping (Sub), which embeds into typing the basic subsumption principles. It allows assertions in type rules to be considered up to $\mathrel{<:>}$, and plays a role similar to the consequence rule in Hoare logics.

$$\frac{A \mathrel{<:} A' \quad A' \vdash_x e :: B' \quad B' \mathrel{<:} B}{A \vdash_x e :: B} \ (Sub)$$

The rule for **let** corresponds to cut (x not free in the conclusion)

$$\frac{A \vdash_x e_1 :: B \quad B \vdash_y e_2 :: C}{A \vdash_y \mathsf{let}\ x \ = \ e_1\ \mathsf{in}\ e_2 :: C} \ (Let)$$

The following parallel and sequential structural rules express basic "frame" principles. Rule (Par) allows the footprint of an expression to be enlarged along the spatial dimension, while rule (Seq) allows the footprint to be enlarged along the temporal dimension.

$$\frac{A \vdash_x e :: B}{A \mid C \vdash_x e :: B \mid C} \ (Par) \qquad \frac{A \vdash_x e :: B}{A \,; C \vdash_x e :: B \,; C} \ (Seq)$$

Given these two rules, the following "deep" frame rule is admissible for any active type assertion context $\mathcal{E}[-]$.

$$\frac{A \vdash_x e :: B}{\mathcal{E}[A] \vdash_x e :: \mathcal{E}[B]} \ (Frame)$$

4.2.2 Functional Type

We have the following typing rules for the λ-calculus core.

$$\frac{A | x{:}U \vdash_y e :: y{:}T}{A \ \vdash_z \lambda x.e :: z{:}U \mapsto T} \ (VAbs)$$

$$\frac{A \vdash_z e_1 :: z{:}U \mapsto T \quad B \vdash_x e_2 :: x{:}U}{A \mid B \ \vdash_y e_1 e_2 :: y{:}T} \ (App)$$

These rules are similar to the arrow rules in linear or bunched [29] type systems, even if our semantics for $\mapsto$ is different. Given the interpretation of $A \mid B$, (App) ensures that functions do not interfere with their arguments unsafely upon application. Notice that the type of the argument x is left $\mathtt{0}$ in the postcondition of the premise of $(VAbs)$, forcing the function body to fully exercise the behavior U of its parameter, consistently with the "argument-borrowing" semantics of our functional type. The type of an "argument-capturing" function may be rendered $(\circ U) \mapsto V$, and the type of a function that can safely share the behavior of its argument with its own behavior may specified $(!U) \mapsto V$. Behavioral separation types allow many fine-grained variations of functional behavior to be specified (e.g., $!(U \mapsto V)$ - a shareable function; $(U \mapsto V)^*$ - a non shareable but repeatedly usable function, etc). Notice that in our typed language (as, e.g., in the monadic λ-calculus) one cannot encode the **let** by application and abstraction.

4.2.3 Tuple Type

The rules for tuples and field selection have the expected form.

$$\frac{A \vdash_x e :: x{:}U}{A \vdash_z [\ldots l = e \ldots] :: z{:}l{:}U} \ (Tuple) \qquad \frac{A \vdash_z e :: z{:}l{:}T}{A \vdash_x e.l :: x{:}T} \ (Sel)$$

Recall that field contents of tuples are evaluated lazily, as in [34], so $(Tuple)$ allows a single field to be type checked. As explained above, richer behavioral separation types for multi-field records may be expressed using the various type constructors available.

4.2.4 Intersection Type

We include as primitive the introduction rule And.

$$\frac{A \vdash_y e :: B \quad A \vdash_y e :: C}{A \vdash_y e :: B \ \& \ C}(And) \quad \frac{A \vdash_y e :: B_1 \ \& \ B_2}{A \vdash_y e :: B_i}(AndE)$$

Technically, we choose to absorb the elimination principles for intersection in the subtyping relation (e.g., $A \,\&\, B <: A$). However, familiar elimination rules $AndE$ are admissible (using (Sub)).

4.2.5 Behavioral-Separation Types

Structured behavioral-separation usages are assigned to basic values (abstractions, tuples) by the following type rules:

$$\mathtt{0} \vdash_y v :: \mathtt{0}\ (VStop) \quad \frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A \,;\, B \vdash_y v :: C \,;\, D}(VSeq)$$

$$\frac{!A_1 \mid \ldots \mid !A_n \vdash_x v{::}B}{!A_1 \mid \ldots \mid !A_n \vdash_x v{::}!B}\ (VShr) \quad \frac{A \vdash_y v{::}C \quad B \vdash_y v{::}D}{A \mid B \vdash_y v :: C \mid D}(VPar)$$

Rule $(VShr)$ expresses that a value can be subject to any number of shared usages, if it only relies on resources which may also be safely used by any number of shared usages. Interestingly, these rules allow values to satisfy crisper frame principles than the structural rules in Section 4.2.1, which apply to general expressions. For example, the following "fat" identity axiom and left-sequential frame rule turn out to be admissible for values, even if the corresponding principles are not sound for arbitrary expressions.

$$A \vdash_y v :: A\ (VId) \quad \frac{B \vdash_y v :: C}{A \,;\, B \vdash_y v :: A \,;\, C}\ (VLPar)$$

4.2.6 Isolated Type

The rule (Iso) assigns to the postcondition of an expression an isolated type if it only depends on values of isolated type.

$$\frac{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x e :: B}{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x e :: \circ B}\ (Iso)$$

The type rules for $!A$ and $\circ A$ are therefore similar, and express the basic comonadic principle associated to these type constructors (cf. the introduction rule for ! in intuitionistic linear logic), even if their meaning is quite different (sharing versus isolation).

A remarkable property of any type T of the form $!\circ U$ is that $T <:> T \mid T$ and $T <:> \circ T$, so that T is both shared and isolated.

4.2.7 Sum Type

Sum types are also handled by familiar looking typing rules.

$$\frac{A \vdash_y e_c :: y : \oplus_{l \in I} l{:}T_l \quad x_i{:}T_i \mid B \vdash_z e_i :: C}{A \mid B \vdash_z \textsf{case}\ e_c\ \textsf{of}\ l(x) \to e :: C}\ (Case)$$

$$\frac{A \vdash_z e :: z{:}T_i}{A \vdash_z l_i(e) :: z{:} \oplus_{l \in I} l{:}T_l}\ (Option)$$

As for function application, the type rule for case ensures that the matched value is separated from the corresponding case branch, so to avoid unsafe interference.

4.2.8 Heap Variable Types

We have already explained how heap allocated variables are modeled in our system as special values, subject to a specific usage protocol defined by certain subtyping axioms. The type rule for a variable declaration types the body under the assumption of a separated complete protocol for the new variable, specified by `var`.

$$\frac{a{:}\mathtt{var} \mid A \vdash_x e :: C}{A \vdash_x \textsf{var}\ a\ \textsf{in}\ e :: C}\ (Var)$$

Rules for dereference and assignment are more interesting. We consider two typing rules for dereference, and two typing rules for assignment. The alternative typings express boundary cases on usage of the variable protocol, which are not naturally captured by a single typing rule. We distinguish between reading just a "piece" of the behavior stored in the variable $(RdVB)$, from reading the whole remaining stored behavior $(RdVF)$.

$$a{:}\mathtt{rd}(U) \vdash_x a :: x{:}U\ (RdVB)$$

$$a{:}\mathtt{rd}(U); \mathtt{use} \vdash_x a :: x{:}U \mid a{:}\mathtt{use}\ (RdVF)$$

Rule $(RdVF)$ states that even if the precondition states that the next `use` of variable a is guarded by a read usage $\mathtt{rd}(U)$, the variable content $x{:}U$ is separated of the residual variable behavior $a{:}\mathtt{use}$, specifying an "empty" variable. Rule $(RdVF)$ expresses an important invariant ensured by the type system: the behavior stored in any heap variable is always separated from the continuation behavior of the variable object itself (after all of its content gets read off). So the postcondition in the conclusion of $(RdVF)$ always holds, even if the type of the heap variable content is not explicitly declared as isolated (not of the form $\circ U$).

We also have two rules for assignment, depending on weather the behavior stored in the heap variable is isolated or borrowed.

$$\frac{A \vdash_z v :: z{:}\circ U \mid a{:}\mathtt{wr}(\circ U)}{A \vdash_z a := v :: \mathtt{0}}\ (WrVF)$$

$$\frac{A \vdash_z v :: z{:}U \mid a{:}\mathtt{use}}{A \vdash_z a := v :: a{:}\mathtt{rd}(U)}\ (WrVB)$$

In both rules, the stored value is required to be parallel separated from the heap variable, which must be in a ready-for-write state. Rule (*WrVF*) handles the case in which the value behavior to be stored is isolated. Here, we only require the write capability, as the stored value may be used anytime later. Rule (*WrVB*) handles the case in which the value behavior to be stored may be not isolated. In this case, one must ensure that all associated reads will happen before any sequential continuation of $z{:}U$, so a whole `use` is required in the premise, leaving the associated read usage active in the post-condition. It is interesting to see why a rule as $(WrVF)$, but considering a non-isolated type for the stored value, would not be sound. Let us consider a simple counterexample.

$$\begin{array}{l} r{:}U \mid a{:}\mathtt{wr}(U) \vdash_x r :: x{:}U \mid a{:}\mathtt{wr}(U) \\ r{:}U \mid a{:}\mathtt{wr}(U) \vdash_x a := r :: 0 \\ r{:}U \,;\, V \mid a{:}\mathtt{wr}(U) \vdash_x a := r :: r{:}V \\ r{:}U \,;\, V \mid a{:}\mathtt{wr}(U); \mathtt{rd}(U) \vdash_x a := r :: r{:}V \,;\, a{:}\mathtt{rd}(U) \\ r{:}U \,;\, V \mid a{:}\mathtt{use} \vdash_x a := r :: r{:}V \,;\, a{:}\mathtt{rd}(U) \end{array}$$

This candidate derivation states that after executing $a := r$ the behavior $r{:}V$ is available before reading the variable a, thus violating the requirement that r must be used as $U \,;\, V$. A correct typing is:

$$\begin{array}{l} r{:}U \mid a{:}\mathtt{use} \vdash_x r :: x{:}U \mid a{:}\mathtt{use} \\ r{:}U \mid a{:}\mathtt{use} \vdash_x a := r :: a{:}\mathtt{rd}(U) \\ r{:}U \,;\, V \mid a{:}\mathtt{use} \vdash_x a := r :: a{:}\mathtt{rd}(U) \,;\, r{:}V \end{array}$$

Typing rules for assignment require separation between heap variable and stored value. This may suggest that typing of circular chains of references through the heap may be difficult, if not impossible. Although it is clear that linear behaviors cannot refer circularly to themselves, that is not the case for general behavioral separation types: some safe circular chains may still support separated behaviors, due to the presence of qualified tuples, or just

because of sharing (including invariant based separation). We illustrate this point in Section 4.5, by typing a version of Landin's knot. The following rules are admissible (using the abbreviation $a := e$ for **let** $x = e$ **in** $a := x$, assuming $\mathcal{E}[-]$ an active type context).

$$\frac{A \vdash_z e :: z{:}\circ U \mid a{:}\mathtt{wr}(\circ U)}{A \vdash_x a := e :: \mathtt{0}} \qquad \frac{A \vdash_z e :: \mathcal{E}[z{:}U] \mid a{:}\mathtt{use}}{A \vdash_x a := e :: \mathcal{E}[a{:}\mathtt{rd}(U)]}$$

The rules of the basic type system are shown in Figure 4. We avoid introducing here rules for recursion and recursive types whose treatment require the addition of extra information to typing judgments; details may be found in the Appendix. For clarity's sake, we present the type system in two steps, first without synchronization constructs, which is extended in Section 4.5 to the full language.

4.3 Typing the Collection Implementation

We get back to the running example in the Introduction. We have intuitively argued that a collection value may be assigned the type $\circ CC$, and the function $newColl$ the type $\mathtt{0} \mapsto \circ CC$. We now discuss how such type is actually validated by our type system. We first type the list nodes. Consider the following abbreviations

$$\begin{array}{l} \mathit{InitNode} \triangleq \mathit{setElt}{:}(\mathtt{nat} \mapsto 0)\,;\,\mathit{setNext}{:}(!\circ \mathit{PNode} \mapsto 0) \\ \mathit{INode} \triangleq !\mathit{getNext}{:}\mathit{PNode} \mid !\mathit{getElt}{:}\mathtt{nat} \\ \mathit{Node} \triangleq \mathit{InitNode}\,;\,!\circ \mathit{INode} \qquad \mathit{PNode} \triangleq !\mathtt{Opt}(\mathit{INode}) \end{array}$$

The type $PNode$ defines the behavior of a pointer to a list of (initialized) nodes, as created by the function $newNode$. We use option types to type list pointers: either the "null" pointer, tagged $\mathtt{NULL}(-)$ or a value of type $INode$, tagged by $\mathtt{NODE}(-)$.

Directed by rules $VSeq$, $VShr$ and $Tuple$, the system assigns type $Node$ to the tuple $[\mathit{setElt} = \cdots]$ by checking that it can be subject to the given behavioral separation usage, while safely using its local resources (the variables $next$ and elt). The variable elt gets assigned type $\mathtt{wr}(\mathtt{nat})\,;!\mathtt{rd}(\mathtt{nat})$: it is written just once (in the operation $setElt$), and available for shared reading from then on. Notice that $\mathtt{var} <: \mathtt{wr}(\mathtt{nat})\,;!\mathtt{rd}(\mathtt{nat})$, since $\mathtt{nat} <: !\mathtt{nat}$). Variable $next$ is typed $\mathtt{wr}(\circ \mathit{PNode})\,;!\mathtt{rd}(\mathit{PNode})\,;\mathtt{var}$. After initialization (execution of the behavior $InitNode$), the node type evolves to $!\circ \mathit{INode}$; only operations $getNext$ and $getElt$ are available, usable by an unbounded number of aliases or concurrent clients. The shared behavior $INode$ of the list nodes supports sharing of the linked list and allows the type $!\mathit{scan}{:}\mathtt{0}$ to be assigned within CC. Recall

$$CC \triangleq (\mathit{init}{:}\mathtt{str} \mapsto \mathtt{0})\,;(!\mathit{getId}{:}\mathtt{str}\,|\,(!\mathit{scan}{:}\mathtt{0}\,;\,\mathit{add}{:}\mathtt{nat} \mapsto 0)^*)$$

Checking $newCollection$ against type $0 \mapsto \circ CC$, involves verifying that the "object" tuple $[\mathit{init} = \cdots]$ representing a collection can be safely subject to the behavioral separation usage specified by CC. After $init$, the variable hd is assigned type $\mathtt{rd}(\circ \mathit{PNode})\,;\mathtt{var}$. This type is kept invariant between iterated executions of the add and $scan$ operations.

It is particularly interesting to see how the $scan$ operation is typed. A sequentially separated prefix $PNode$ is borrowed from hd to the local heap variable s, ($\circ \mathit{PNode} <: \mathit{PNode}\,;\circ \mathit{PNode}$). After typing the assignment $s := hd$, the type assertion is

$$s{:}(\mathtt{rd}(\mathit{PNode})\,;\mathtt{var})\,;\mathit{hd}{:}(\mathtt{rd}(\circ \mathit{PNode})\,;\mathtt{var})$$

which by subtyping ($\mathtt{var} <: \circ\mathtt{var}$ and postponing) leads to

$$s{:}\mathtt{rd}(\mathit{PNode})\,;(\mathit{hd}{:}(\mathtt{rd}(\circ \mathit{PNode})\,;\mathtt{var}) \mid s{:}\mathtt{var})$$

where $PNode$ is the behavior of s actually used up by the various loop iterations. After typing the whole loop, the type assertion is

$$\mathit{hd}{:}(\mathtt{rd}(\circ \mathit{PNode})\,;\mathtt{var}) \mid s{:}\mathtt{var}$$

which after closing the scope of s (removing $s{:}\mathtt{var}$), gets us back to the invariant $\mathit{hd}{:}\mathtt{rd}(\circ \mathit{PNode})\,;\mathtt{var}$.

4.4 Type Preservation and Progress

We now state the main correctness results for the basic type system, namely the subject reduction property and progress for well-typed programs. Type preservation and progress ensure that in a well typed program all values are properly used according to their assigned behavioral-separation types. In particular, given the structure of types assigned to variables, no write/write or read/write races while writing to heap variables are possible (it is not the case that $\mathtt{use} <: (\mathtt{wr}(U) \mid \mathtt{wr}(V))\,;T$ or $\mathtt{use} <: (\mathtt{rd}(U) \mid \mathtt{wr}(V))\,;T$).

We first introduce a notion of typing for runtime configurations $h\ ;\ S$ defined by the following rules

$$\emptyset\ ;\ \emptyset\ \triangleright\ \mathtt{0}\ (E) \qquad \frac{h\ ;\ S \triangleright \mathcal{E}[C] \quad C \vdash_x e :: x{:}T}{h\ ;\ S \cdot t\langle e\rangle\ \triangleright\ \mathcal{E}[t{:}\tau(T)]}\ (T)$$

$$\frac{h\ ;\ S\ \triangleright A \quad A <: B}{h\ ;\ S\ \triangleright\ B}\ (S) \qquad \frac{h\ ;\ S\ \triangleright A \quad A \vdash_x v :: B}{h, n \mapsto v\ ;\ S\ \triangleright\ B\{x/n{:}\mathtt{var}\}}\ (H)$$

In rule (T), $\mathcal{E}$ is an active type assertion context. The notation $B\{x/n{:}\mathtt{var}\}$ (with n fresh in B) represents an update to assertion B in which the behavior pieces assigned to $x \in Dom(B)$ are substituted in place by reads to a new heap variable n: essentially, all occurrences of the form $x{:}U$ in type B are replaced by $n{:}\mathtt{rd}(U)$ and a $n{:}\mathtt{var}$ is inserted in sequential and linear position relative to all $n{:}\mathtt{rd}(U)$'s. We can state our type preservation result.

THEOREM 4.3. *If $h\ ;\ S\ \triangleright A$ and $h\ ;\ S \to h'; S'$ then $h'; S'\ \triangleright A$.*

An expression e is live, noted $live(e)$, if it is not a value. A set S of threads is live, noted $live(S)$ if there is some thread $t\langle e\rangle$ in S such that $live(e)$. We can then prove

THEOREM 4.4. *If $h\ ;\ S\ \triangleright A$ and $live(S)$ then $h\ ;\ S \to h'; S'$.*

Detailed proofs and definitions are given in the Appendix.

4.5 Invariant-based Separation

We now extend the basic type system to cover the full core language with $\mathsf{sync}(a)e$ blocks, and invariant-based reasoning. As explained in Section 3, each heap variable is equipped with an associated lock (pretty much like a Java object is). To each lock, a resource invariant, expressed by an isolated typing assertion, is associated for verification purposes. We only accept for lock invariant a heap assertion R such that $R <: \circ R$ (let us call "heap assertion" any type assertion that just refers to heap variable types - `var`, `use`, etc). Handling lock invariants requires some additional structure in our type system: we add to typing judgments an invariant mapping ι, that associates to heap location locks their invariants:

$$\boxed{A \vdash^{\iota}_{z} e :: B} \qquad (e \text{ types from } A \text{ to } z \text{ in } B \text{ under } \iota)$$

The invariant mapping is propagated untouched by all typing rules, except in the new rule for variable declaration, which may introduce a lock invariant, and in the rules for $\mathsf{sync}(n)e$ and $sy(n)e$, which make use the lock invariant associated to heap location n:

$$\frac{\iota(a) \mid A \vdash^{\iota\setminus a}_{x} e :: \iota(a) \mid B}{A \vdash^{\iota}_{x} \mathsf{sync}(a)e :: B}\ (Sync) \qquad \frac{A \vdash^{\iota\setminus a}_{x} e :: \iota(a) \mid B}{A \vdash^{\iota}_{x} sy(a)e :: B}\ (Sy)$$

$$\frac{A <: B \mid R \quad a{:}\mathtt{var} \mid\ B \vdash^{\iota\{R/a\}}_{x} e :: C}{A \vdash^{\iota}_{x} \mathbf{var}\ a\ \mathbf{in}\ e :: C}\ (Var)$$

Without loss of generality, we assume that the invariant associated to some heap variable's lock does not talk about the variable itself, but only about other heap variables in scope. Consider the code snippet describing an "atomic" variable.

$$\begin{array}{l} \mathbf{let}\ \mathit{atomic} = \lambda v. \\ \quad \mathbf{var}\ s\ \mathbf{in}\ s := v; \\ \quad\quad \mathbf{var}\ \mathit{lock}\ \mathbf{in}\ [\ \mathit{set} = \lambda x.\mathsf{sync}(\mathit{lock})s := x, \\ \qquad\qquad\qquad \mathit{get} = \mathsf{sync}(\mathit{lock})s\ ]\ \mathbf{in}\ \ldots \end{array}$$

$$\mathtt{0} \vdash_y v :: \mathtt{0}\ (VStop) \quad x{:}U \vdash_y x :: y{:}U\ (Id) \quad \frac{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x e :: B}{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x e :: \circ B}\ (Iso) \quad \frac{!A_1 \mid \ldots \mid !A_n \vdash_x v :: B}{!A_1 \mid \ldots \mid !A_n \vdash_x v :: !B}\ (VShr)$$

$$\frac{A <: A' \quad A' \vdash_x e :: B' \quad B' <: B}{A \vdash_x e :: B}\ (Sub) \quad \frac{A \vdash_y e :: B \quad A \vdash_y e :: C}{A \vdash_y e :: B\ \&\ C}\ (And) \quad \frac{A \vdash_x e_1 :: B \quad B \vdash_y e_2 :: C}{A \vdash_y \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 :: C}\ (Let)$$

$$\frac{A \vdash_x e :: B}{A\ ;\ C \vdash_x e :: B\ ;\ C}\ (Seq) \quad \frac{A \vdash_x e :: B}{A\ \mid\ C \vdash_x e :: B\ \mid\ C}\ (Par) \quad \frac{A \vdash_x e_1 :: x{:}U \mapsto T \quad B \vdash_y e_2 :: y{:}U}{A\ \mid\ B\ \vdash_z e_1 e_2 :: z{:}T}\ (App)$$

$$\frac{A \vdash_x e :: x{:}U}{A \vdash_x [\ldots l = e \ldots] :: x{:}l{:}U}\ (Tuple) \quad \frac{A \vdash_x e :: x{:}l{:}T}{A \vdash_x e.l :: x{:}T}\ (Sel) \quad \frac{A \mid x{:}U \vdash_z e :: z{:}T}{A\ \vdash_z \lambda x.e :: z{:}U \mapsto T}\ (VAbs)$$

$$\frac{A \vdash_y e :: (y{:} \oplus_{l \in I} l{:}T_l) \quad x_l{:}T_l \mid B \vdash_z e_l :: C}{A \mid B \vdash_z \mathsf{case}\ e\ \mathsf{of}\ \overline{l(\vec{x}) \to e} :: C}\ (Case) \quad \frac{A \vdash_z e :: z{:}T_m \quad (m \in I)}{A \vdash_z l_m(e) :: z{:} \oplus_{l \in I} l{:}T_l}\ (Option)$$

$$\frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A\ ;\ B \vdash_y v :: C\ ;\ D}\ (VSeq) \quad \frac{A \vdash_y v :: C \quad B \vdash_y v :: D}{A \mid B \vdash_y v :: C\ \mid\ D}\ (VPar)$$

$$\frac{a{:}\mathtt{var} \mid\ A \vdash_x e :: C}{A \vdash_x \mathsf{var}\ a\ \mathsf{in}\ e :: C}\ (Var) \quad \frac{A \vdash_z v :: z{:}U \mid a{:}\mathtt{use}}{A \vdash_z a := v :: a{:}\mathtt{rd}(U)}\ (WrVB) \quad a{:}\mathtt{rd}(U); \mathtt{use} \vdash_x a :: x{:}U \mid a{:}\mathtt{use}\ (RdVF)$$

$$\frac{A \vdash_z v :: z{:}\circ U \mid a{:}\mathtt{wr}(\circ U)}{A \vdash_z a := v :: \mathtt{0}}\ (WrVF) \quad a{:}\mathtt{rd}(U) \vdash_x a :: x{:}U\ (RdVB) \quad \frac{A \vdash_x e :: x{:}T}{A \vdash_x \mathsf{fork}\ e :: x{:}\tau(T)}\ (Fork) \frac{A \vdash_x e :: x{:}\tau(T)}{A \vdash_x \mathsf{wait}\ e :: x{:}T}\ (Wait)$$

Figure 4. Typing Rules.

Let U be some shared isolated type (e.g. a type such that U <: $!\circ U$). We can then derive the typing

$$atomic{:}U \mapsto (!set{:}(U \mapsto 0) \mid !get{:}U)$$

by associating to *lock* the invariant $s{:}\mathtt{rd}(U)\,;\mathtt{var}$. This type states that *atomic* is a function that returns a stateful variable "object" that can be safely used concurrently by an arbitrary number of setters and getters. Notice that if any of the two sync blocks is removed, *atomic* would only be typed by a behavioral-separation type that sequentializes the *get* and *set* operations somehow. For example, if both sync blocks are removed, a possible typing is

$$atomic{:}U \mapsto (!get{:}U\ ;\ set{:}(U \mapsto 0))^*$$

which would still allow sharing (aliasing, or concurrent usage) of the *get* "method" but not of the *set* "method".

This example illustrates how the monitor construction can be explained as a type coercion operation in our type structure, e.g., coercing $A\,;B$ to $A \mid B$, or even $(A\,;B)\,\&(B\,;A)$ to $A \mid B$. Recall that by the exchange law we can derive $A \mid B$ <: $(A\,;B)\,\&(B\,;A)$, expressing the basic interleaving principle that a value of type $A \mid B$ can be used according to $(A\,;B)\,\&(B\,;A)$. Conversely, given a value providing the behavior $(A\,;B)\,\&(B\,;A)$, we may in general coerce it to the behavior $A \mid B$, by wrapping within a monitor that enforces the appropriate usage protocol by means of locking. The monitored object then exports two behaviorally parallel separated interfaces A and B, even if there is potentially sharing / interference between the implementations of A and B. Our type system naturally assigns $A \mid B$ to the monitored object, relying on the modular type rules for locking and on standard invariant-based reasoning.

Invariant based reasoning is also useful in a non-concurrent setting, in which case we may consider the **sync** operator essentially as a typing device for potentially shared (aliased) usages. We elaborate on this point using a simple, yet non-trivial example: a FIFO queue implemented with a linked list data structure with head and tail pointers (code is presented in Figure 5). We describe the type *Node* assigned to node value in the list. Let

$$\begin{array}{ll} Node \triangleq HeadT \mid TailT & SHeadT \triangleq \mathtt{Opt}(HeadT) \\ HeadT \triangleq \circ unlink{:}\circ SHeadT & TailT \triangleq \circ link{:}\circ SHeadT \mapsto \mathtt{0} \end{array}$$

Our type system assigns to function *new* the type $!(\mathtt{0} \mapsto Node)$.

```
SQueue ≜
let new =
  λ[].var next in
      next := NULL;
      var lock in
        [ unLink = sync(lock)let x = next
                                 in (next := NULL; x)
          link = λx.sync(lock)next := x ]
in var head, tail in (
      head := NULL; tail := NULL;
      [ enq = let n = (new nil) in
                case tail of
                NULL → (head := NODE(n);
                        tail := NODE(n))
                NODE(y) → (y.link NODE(n));
                        tail := NODE(n)),
        deq = case head of
                NULL → head := NULL
                NODE(y) → (head := y.unLink;
                      case head of
                      NULL → tail := NULL; head := NULL
                      NODE(y) → head := NODE(y)) ]
```

Figure 5. A FIFO Queue Implemented with a Linked List

Node is a parallel separation type, exposing, on the one hand, the behavior to be assigned to the head pointer or to the previous node in the list, and, on the other hand, the behavior to be assigned to the tail pointer. The safe separation of behaviors is here enforced by the use of invariant based separation, associating to *lock* the invariant $next{:}\mathtt{rd}(\circ SHeadT)\,;\mathtt{var}$.

We can then derive $\vdash_q SQueue :: SQueueI$ where

$$SQueueI \triangleq (q{:}enq{:}\mathtt{0}\ \&\ q{:}deq{:}\mathtt{0})^*$$

This type says that the declared behavioral separation protocols are enforced, even in the presence of possible interference between the state accessible from *head* and *tail*. Additionally, type *Node* clearly says that both *link* and *unlink* operation are used exactly once in each list node, the first through the *tail* alias, the other by the *head* alias, and that this describes the full behavior of a node.

Type $SQueueI$ above declares a sequential behavior for the queue, where enq and deq operations cannot be selected concurrently. A simple typable concurrent queue can be defined by guarding the sequential implementation described with appropriate **sync** blocks:

$$\begin{array}{l}\textbf{let}\, CQueue = \textbf{var}\ head, tail\ \textbf{in}\ (\\ \qquad head := \texttt{NULL}; tail := \texttt{NULL};\\ \qquad\ \ \textbf{var}\ qinv\ \textbf{in}\\ \qquad\ [\ \ enq = \textbf{sync}(qinv)(\ldots)\\ \qquad\ \ \ deq = \textbf{sync}(qinv)(\ldots)\ \ ]\end{array}$$

For type checking this code we associate to $qinv$ the invariant $head{:}\texttt{rd}(\circ SHeadT)\,\texttt{;var} \mid tail{:}\texttt{rd}(\circ STailT)\,\texttt{;var}$. We then derive $\vdash_q CQueue :: QueueCI$ for $QueueCI \triangleq\ !enq{:}\texttt{0} \mid !q{:}deq{:}\texttt{0}$. The $QueueCI$ interface type explicitly says that the queue can be safely used by many concurrent clients, as in. e.g.,

$$\textbf{let}\ q = CQueue\ \textbf{in}\ (q.enq;\, q.enq \| q.deq;\, q.deq)$$

Notice that in the assertion typing $SQueue$ only sequential types appear, that is, no $(-\mid -)$ or $!(-)$. So only a single thread may be visiting the code of $SQueue$. This means (informal claim) that the lock $lock$ associated to each list node will always be free. So, the **sync** blocks in $Node$ are operationally irrelevant, and may be seen as an auxiliary device for bracketing code regions subject to invariant-based type checking of separation. On the other hand, **sync** blocks are essential to $CQueue$, if it is to be actually used according to the more permissive type $QueueCI$.

As a further example, we present a code fragment tying a Landin's knot, thus creating a circular chain of references in the higher-order store. We may verify that it can be typed by assigning to the (operationally useless) lock $linv$ the invariant $a{:}\texttt{rd}(!\circ(0 \mapsto 0))\,\texttt{;var}$, and to the function f the type $!\circ(0 \mapsto 0)$.

$$\begin{array}{l}\textbf{var}\ a\ \textbf{in}\ (\ a := \lambda x.x;\\ \quad \textbf{var}\ linv\ \textbf{in let}\ f = \lambda y.(\textbf{sync}(linv)(a)\ y)\\ \qquad \textbf{in}\ (\textbf{sync}(linv)(a := f); (f\ \textbf{nil}))\ )\end{array}$$

In principle, our type system could be refined to distinguish between two different scenarios for invariant based reasoning, one already useful to handle interference in sequential code, another one to handle interference in truly concurrent code, only the latter would require real locks to be introduced in the code. We leave this discussion for future consideration, for the issue seems orthogonal to the main purpose of this paper. A key point to highlight here is that our typing principles for the **sync** construct seem to capture a useful and general form of invariant reasoning about safe interference in the context of a behavioral separation type system.

We can now state the type preservation result for the full core language. To that end, typing for runtime configurations is generalized to consider the declaration of invariants. This is achieved by a global invariant mapping ι, which assigns lock invariants to locations: essentially, the rule (H) of Section 4.4 is replaced by the following rules, covering the two possible lock states (see Appendix).

$$\frac{h\ ;\ S \triangleright_\iota A \mid R \quad A \vdash^\iota_x v :: B}{h, n_0 \mapsto v\ ;\ S \triangleright_{\iota\{R/n\}} B\{x/n{:}\texttt{var}\}}\ \text{(HU)}$$

$$\frac{h\ ;\ S \triangleright_\iota A \quad A \vdash^\iota_x v :: B}{h, n_{-1} \mapsto v\ ;\ S \triangleright_{\iota\{R/n\}} B\{x/n{:}\texttt{var}\}}\ \text{(HL)}$$

THEOREM 4.5. *If* $h\ ;\ S \triangleright_\iota A$ *and* $h\ ;\ S \to h'; S'$ *then* $h'; S' \triangleright_\iota A$.

A progress property also holds for the full core language with concurrency and synchronization primitives, but in a restricted sense, due to the possibility of deadlock on **sync** blocks (see Appendix).

5. Related Work

The concept of separation results from a research stream whose origins can be traced back to the seminal works of Reynolds on syntactic control of interference [32, 34]. Separation logics extend classical Hoare logic with new connectives, in particular the separation conjunction, which allows to specify the fine-grained structure of states in programs manipulating references, and enables local reasoning to successfully tackle programs with references [33] and concurrency [30]. More recently, separation logic has motivated the introduction of Hoare Type Theory [28], and has been extended to languages with higher-order store [36]. These works focus on the identification of higher-order frame principles for state-based local reasoning. The idea of assigning a parallel separation type to some value, even when there is (safe) interference between the implementations of the separated behaviors, is reminiscent of concepts explored in fictional separation [23] and concurrent abstract predicates [15]. In our case, we consider such usage behaviors as truly separate, as they can be safely used by independent clients. The focus of behavioral separation is on externally usable protocols of program entities, rather than on the internal state of programs. Although it is clear that the fundamental notion of separation applies to many kinds of computational structures [11, 13, 18], the idea of combining separation with behavioral types to discipline interference in a realistic programming language, as we do here, does not seem to have been considered before.

Various forms of behavioral types has been independently introduced by several authors with the intent of classifying usage patterns of computational objects [19, 14, 22]. Some of these works have motivated more refined verification techniques, for example, to check resource usage disciplines [21] in functional programs. A particular case of behavioral types are the so-called session types, intended to discipline message exchanges between partners in distributed systems. Although initially proposed for systems with interaction between exactly two partners [19], session types have been extended to systems with an arbitrary number of participants [20]. Our notion of type assertion is loosely related with the notion of global type introduced in [20], in the sense that it needs to talk about the joint behavior of several entities. A version of session types to discipline interactions between concurrent objects was developed in [16], but does not attempt to deal with interference or aliasing. More recently, we have developed an interpretation of session types in linear logic [12], which also inspired some aspects of the theory presented here. Connections between session types and separation logic have also been investigated in [37], but focusing on disciplining the transference of resources in process communications. In prior work, we attempted a very preliminary approach to the concept of behavioral separation [10]. However, the developments in this paper clarify the notion of behavioral separation type in the context of a clean substructural type theory, based on a λ-calculus with imperative and concurrency constructs, and are much more general and expressive.

Several works have proposed type-based approaches to discipline aliasing and concurrency control in various programming languages, usually exploiting type and effect systems [1, 8]. Ownership types have also been studied to discipline aliasing and concurrency [7, 6]. Some of these works have led to the development of powerful programming tools [25]. Typically, these works do not focus on capturing the dynamic behavior resources at a deeper level, as we attempt in our work, but on tracking occurrences of identifiers, locks, permissions, regions, and data dependencies, usually resorting to linearity. An important exception is typestate, which uses a state-based approach to specify resource usage protocols in object oriented languages [3]. A key ingredient of the typestate approach is the use of primitive permissions to capture usage idioms, rather than resource behavior. In parallel research we are investigating combinations of separation with typestate [26]. Techniques to support expressive borrowing idioms in the context of typestate have been recently proposed in [27].

6. Conclusions

We have introduced behavioral separation as a general principle for disciplining interference, either due to aliasing or concurrency, by combining concepts from separation logic and behavioral type systems. We have designed a behavioral separation type system that illustrates the concept using a higher-order imperative functional language extended with concurrency and synchronization primitives. Our type system is proven sound using proof theoretic techniques.

We have also shown that the expressiveness of our approach goes beyond the state of the art for type-based verification of aliasing and concurrency, and provided several challenging examples involving fine-grained state manipulation, thread based concurrency, and synchronization constructs. Further examples, including the implementation of a concurrent queue based on a double linked list can be found in the Appendix. In ongoing work, we are studying algorithmic aspects of our type system; in fact, we have already designed an algorithm that can effectively typecheck programs in our core language with a reasonable annotation burden, these results will be reported elsewhere. We are also investigating generalizations to invariant-based separation, along the lines suggested in Section 4.5.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[2] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A Linear Language with Locations. *Fundam. Inform.*, 77(4):397–449, 2007.

[3] K. Bierhoff and J. Aldrich. Modular Typestate Checking of Aliased Objects. In *OOPSLA 2007*, pages 301–320, 2007.

[4] S. L. Bloom and Z. Ésik. Free Shuffle Algebras in Language Varieties. *Theoretical Computer Science*, 163(1&2):55–98, 1996.

[5] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL 2005*, pages 259–270, 2005.

[6] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 2002*, pages 211–230. ACM, 2002.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL 2003*, 2003.

[8] C. Boyapati and M. C. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA 2001*, pages 56–69, 2001.

[9] J. Boyland. Checking Interference with Fractional Permissions. In *SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[10] L. Caires. Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems. *Theoretical Computer Science*, 402(2-3):120–141, 2008.

[11] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[12] L. Caires and F. Pfenning. Session Types as Intuitionistic Linear Propositions. In *CONCUR'10*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer-Verlag, 2010.

[13] C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, 2007.

[14] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: Model Checking Message-Passing Programs. In *POPL 2002*, pages 45–57, 2002.

[15] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP 2010*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer-Verlag, 2010.

[16] S. J. Gay, V. Thudichum Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular Session Types for Distributed Object-oriented Programming. In *POPL 2010*, pages 299–312, 2010.

[17] C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene Algebra. In M. Bravetti and G. Zavattaro, editors, *CONCUR'09*, volume 5710 of *Lecture Notes in Computer Science*, pages 399–414. Springer-Verlag, 2009.

[18] T. Hoare and P. W. O'Hearn. Separation Logic Semantics for Communicating Processes. *ENTCS*, 212:3–25, 2008.

[19] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

[20] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL 2008*, pages 273–284. ACM, 2008.

[21] A. Igarashi and N. Kobayashi. Resource Usage Analysis. In *POPL 2002*, pages 331–342, 2002.

[22] A. Igarashi and N. Kobayashi. A Generic Type System for the Pi-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[23] J. B. Jensen and L. Birkedal. Fictional Separation Logic. In *ESOP 2012*, Lecture Notes in Computer Science, pages 377–396. Springer-Verlag, 2012.

[24] Cliff B. Jones. Splitting Atoms Safely. *Theor. Comput. Sci.*, 375(1-3):109–119, 2007.

[25] R. L. Bocchino Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *POPL 2011*, pages 535–548, 2011.

[26] Militão, F. and Aldrich, J. and Caires, L. Aliasing control with view-based typestate. In *FTFJP '10*, pages 7:1–7:7. ACM, 2010.

[27] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A Type System for Borrowing Permissions. In *POPL 2012*, pages 557–570, 2012.

[28] A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare Type Theory, Polymorphism and Separation. *J. Fun. Prog.*, 18(5-6):865–911, 2008.

[29] Peter W. O'Hearn. On Bunched Typing. *J. Fun. Prog.*, 13(4):747–796, 2003.

[30] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[31] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[32] J. C. Reynolds. Syntactic Control of Interference. In *POPL 78*, pages 39–46, 1978.

[33] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Third Annual Symposium on Logic in Computer Science*, Copenhagen, Denmark, 2002. IEEE Computer Society.

[34] John C. Reynolds. Syntactic Control of Interference, Part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *ICALP 89*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer-Verlag, 1989.

[35] John C. Reynolds. *Design of the programming language FORSYTHE*, pages 173–233. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[36] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare Triples and Frame Rules for Higher-order Store. *Logical Methods in Computer Science*, 7(3), 2011.

[37] J. Villard, E. Lozes, and C. Calcagno. Proving copyless message passing. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209. Springer-Verlag, 2009.

A. Appendix

A.1 Recursion

To accommodate recursion (in both programs and types) in the type system, we add to typing judgements a recursion variable environment η, which maps expression variables Z to type judgments and type variables X to type assertions. This technique allows recursion to be interpreted coinductively; our use of a recursion environment is inspired in familiar techniques (e.g., [31]). We include two pair of rules, one pair for recursive expression and expression variable, and other pair for recursive type and type variable.

$$\frac{\eta(X) = A}{A \vdash^{\eta}_{x} v :: X}\ (VRecVar) \qquad \frac{A \vdash^{\eta\{X/A\}}_{x} v :: B}{A \vdash^{\eta}_{x} v :: \mathtt{rec}(X)B}\ (VRec)$$

$$\frac{\eta(Z) = (A \vdash_x B)}{A \vdash^{\eta}_{x} Z :: B}\ (RecVar) \qquad \frac{A \vdash^{\eta\{Z/(A\vdash_x B)\}}_{x} e :: B}{A \vdash^{\eta}_{x} \mathbf{rec}(Z)e :: B}\ (Rec)$$

We have elided the recursion environment in typing judgments in most of our presentation in the paper main body. For the sake of clarity, we have also stated our technical results for the recursion free language and type system, since recursion is treated along standard lines, and fairly orthogonal to the new aspects of our development. Both the recursion environment η and the invariant mapping ι should be propagated without modification from premises to conclusion of all typing rules in Figure 4. Nevertheless, for completeness of our presentation, we collect in Figure 8 the full set of rules of our type system, based on the complete judgment form

$$\boxed{A \vdash^{\iota,\eta}_{z} e :: B} \quad (e \text{ types from } A \text{ to } z \text{ in } B \text{ under } \iota, \eta)$$

A.2 Runtime Configuration Typing

We summarize the runtime configuration typing rules for the full language with synchronization primitives.

$$\emptyset\ ;\ \emptyset \rhd_{\emptyset} \mathtt{0} \quad \text{(E)}$$

$$\frac{h\ ;\ S \rhd_{\iota} \mathcal{E}[C] \quad C \vdash^{\iota,\emptyset}_{x} e :: x{:}T}{h\ ;\ S \cdot t\langle e\rangle \rhd_{\iota} \mathcal{E}[t{:}\tau(T)]} \quad \text{(T)}$$

$$\frac{h\ ;\ S \rhd_{\iota} A \,|\, R \quad A \vdash^{\iota,\emptyset}_{x} v :: B}{h, n_0 \mapsto v\ ;\ S \rhd_{\iota\{R/n\}} B\{x/n{:}\mathtt{var}\}} \quad \text{(HU)}$$

$$\frac{h\ ;\ S \rhd_{\iota} A \quad A \vdash^{\iota,\emptyset}_{x} v :: B}{h, n_{-1} \mapsto v\ ;\ S \rhd_{\iota\{R/n\}} B\{x/n{:}\mathtt{var}\}} \quad \text{(HL)}$$

$$\frac{h\ ;\ S \rhd_{\iota} A \quad A <: B}{h\ ;\ S \rhd_{\iota} B} \quad \text{(S)}$$

N.B. In (HU), R is a lock invariant (see (Section 4.5). The notation $B\{x/n{:}\mathtt{var}\}$ (with n fresh in B) represents the transformation of type B where the behavior at $x \in Dom(B)$ becomes provided through a new heap cell n. We formally define $A\{x/n{:}\mathtt{var}\}$ using the auxiliary substitution form $A[x/n{:}\mathtt{var}]$:

$$\begin{array}{l}
(A\,;B)\{x/n{:}\mathtt{var}\} = A\{x/n{:}\mathtt{var}\}\,;B \quad (x \notin Dom(B)) \\
(A\,;B)\{x/n{:}\mathtt{var}\} = A[x/n{:}\mathtt{var}]\,;(B\{x/n{:}\mathtt{var}\}) \quad (x \in Dom(B)) \\
(A \,|\, B)\{x/n{:}\mathtt{var}\} = A\{x/n{:}\mathtt{var}\} \,|\, B \quad (x \notin Dom(B)) \\
(A \,\&\, B)\{x/n{:}\mathtt{var}\} = A\{x/n{:}\mathtt{var}\} \,\&\, B \quad (x \notin Dom(B)) \\
(\circ A)\{x/n{:}\mathtt{var}\} = \circ(A\{x/n{:}\mathtt{var}\}) \\
A\{x/n{:}\mathtt{var}\} = A[x/n{:}\mathtt{var}]\,;n{:}\mathtt{var} \quad (\textit{otherwise})
\end{array}$$

$$\begin{array}{l}
(x{:}U)[x/n{:}\mathtt{var}] = n{:}\mathtt{rd}(U) \\
(y{:}U)[x/n{:}\mathtt{var}] = y{:}U \quad (x \neq y) \\
(A\,;B)[x/n{:}\mathtt{var}] = A[x/n{:}\mathtt{var}]\,;(B[x/n{:}\mathtt{var}]) \\
(A \,|\, B)[x/n{:}\mathtt{var}] = A[x/n{:}\mathtt{var}] \,|\, (B[x/n{:}\mathtt{var}]) \\
(A \,\&\, B)[x/n{:}\mathtt{var}] = A[x/n{:}\mathtt{var}] \,\&\, (B[x/n{:}\mathtt{var}]) \\
(!A)[x/n{:}\mathtt{var}] = !(A[x/n{:}\mathtt{var}]) \\
(\circ A)[x/n{:}\mathtt{var}] = \circ(A[x/n{:}\mathtt{var}])
\end{array}$$

(We omit the symmetric cases of $|$ and $\&$). Notice that $B\{x/n{:}\mathtt{var}\}$ preserves the standard variable protocol for n, and correctly inserts the terminal behavior $n{:}\mathtt{var}$ at the earliest possible position after all the $n{:}\mathtt{rd}(-)$ declarations. If $B = (x{:}U \mid z{:}T)\,;a{:}V$ with $x \neq z$ then $B\{x/n{:}\mathtt{var}\} = (n{:}(\mathtt{rd}(U)\,;\mathtt{var}) \mid z{:}T)\,;(a{:}V)$, but if $B = (x{:}U \mid x{:}T)\,;a{:}V$ then $B\{x/n{:}\mathtt{var}\} = n{:}(\mathtt{rd}(U \mid T)\,;\mathtt{var})\,;a{:}V$.

It should be clear that if $h\,;S \rhd A$ then the type A only talks about the usage of heap variables in h and threads in S.

A.3 Further Remarks on Type Safety

By Theorems 4.3, 4.4, and 4.5, well typed programs satisfy all the declared behavioral separation protocols, and the progress property holds even in the presence of strong updates (changing types) of heap variables. We expand on the remark in Section 4.4:

PROPOSITION A.1 (Race absence). *Runtime configurations of the following forms cannot be typed.*

1. *(write/read race)* $h\ ;\ S \cdot t_1\langle\mathcal{F}_1[n := u]\rangle \cdot t_2\langle\mathcal{F}_2[n]\rangle \rhd_{\iota} A$.
2. *(write/write race)* $h\ ;\ S \cdot t_1\langle\mathcal{F}_1[n := u]\rangle \cdot t_2\langle\mathcal{F}_2[n := v]\rangle \rhd_{\iota} A$.

Proof. By induction on typing derivations, using the key fact that for the post-condition type B of a well-typed configuration typing $h\ ;S \rhd_{\iota} B$ we cannot have $B <: (n{:}\mathtt{rd}(V) \mid n{:}\mathtt{wr}(U))\,;B'$ or $B <: (n{:}\mathtt{wr}(V) \mid n{:}\mathtt{wr}(U))\,;B'$, given the subtyping axiomatization of `var` (Section 4.1), and runtime configuration typing. ∎

A.4 A double linked List with Concurrent Iterator

We discuss types for the implementation of a FIFO queue over a double linked list, thus refining the development in Section 4.5. This example complements the discussion in Section 4.2.8, and illustrates how behavioral separation and the typing constraints on heap allocated variables do not forbid all circular chains of references in the heap. It also demonstrates how behavioral separation types precisely capture the safe usage discipline of the queue iterator "method", which in our example supports concurrent transversals of the linked list. The code is shown in Figure 6. Consider first the code for the list nodes, generated by the function $DNode$, and which expose operations $setNext$, $getNext$, $setPrev$, $getPrev$. We propose some type declarations that may be used to check our implementation. These are not the most expressive ones one might consider for the code at hand, but sufficient to make several interesting points.

$$\begin{array}{ll}
G \triangleq getNext{:}!ON & ON \triangleq \mathtt{Opt}(G) \\
OH \triangleq \mathtt{Opt}(\circ H) & H \triangleq (G\,;\circ H)\,\&(P\,;!\circ G) \\
OT \triangleq \mathtt{Opt}(\circ T) & P \triangleq setPrev{:}(\circ OT \mapsto 0) \\
T \triangleq getPrev{:}\circ OT & N \triangleq \circ T\,|(setNext{:}(!\circ ON \mapsto 0)\,;\circ H)
\end{array}$$

We can derive $0 \vdash_z DNode :: z{:}0 \mapsto N$, where N is the behavioral separation type of a single list node. To typecheck the synchronized blocks in operations $getPrev$ and $setPrev$ the lock invariant assertion associated to inv is $\mathtt{rd}(\circ OT)\,;\mathtt{var}$.

The queue provides three operations: enq (enqueue an element), deq (dequeue the oldest element, if any), and $iter$. The operation $iter$ returns an iterator "object", which can be used concurrently by an arbitrary number of clients (each one invoking a $scan$ operation), before being released (by means of a $done$ operation). The iterator object is assigned the following interface type

$$IT \triangleq \mathtt{SOME}(!scan{:}\mathtt{0}\,;done{:}\mathtt{0}) \oplus \mathtt{NONE}$$

The option `NONE` signals that the queue is empty, $\mathtt{SOME}(-)$ that there are indeed some elements to scan.

We set $QT \triangleq (enq{:}\mathtt{0} \,\&\, deq{:}\mathtt{0} \,\&\, iter{:}IT)^*$ as the behavioral separation type describing the safe usage protocol of the queue code. We can derive $0 \vdash_q DList :: q{:}QT$.

```
DNode ≜
  λ[].var next, prev in
      prev := NULL;
      var inv in
        [ getNext = next
          setNext = λx.(next := x)
          getPrev = sync(inv)let x = prev
                                in (prev := NULL; x)
          setPrev = λx.sync(inv)(prev := x)  ]

DList ≜
var head, tail in (
  head := NULL; tail := NULL;
  [ enq = let n = (DNode nil) in
            case head of
              NULL → ((n.setNext NULL);
                      head := NODE(n);
                      tail := NODE(n))
              NODE(y) → ((y.setPrev NODE(n));
                         (n.setNext NODE(y));
                         head := NODE(n)),
    deq = case tail of
            NULL → tail := NULL
            NODE(y) → (tail := y.getPrev;
                       case tail of
                         NULL → tail := NULL; head := NULL
                         NODE(z) → tail := NODE(z)),
    iter = case head of
            NULL → head := NULL; NONE
            NODE(n) → head := nil;
                      let s = n.getNext in
                        let io = [
                          scan =
                            var a in (a := s;
                              rec(L).case a of
                                       NULL → nil
                                       NODE(r) →
                                         (a := r.getNext; L)),
                          done = (head := NODE(n) ]
                        in SOME(io)
  ]
```

Figure 6. Queue over a double linked list, and iterator

After the two initial assignments (of NULL) to $head$ and $tail$, these heap variables keep as invariant (between "method" invocations) the types $\texttt{rd}(\circ OH)\,\texttt{;}\,\texttt{var}$ and $\texttt{rd}(\circ OT)\,\texttt{;}\,\texttt{var}$, respectively.

Consider the client code snippet P_1 in Figure 7. We have $q{:}QT \vdash_x P_1 :: \texttt{0}$, so $\texttt{0} \vdash_x \textbf{let}\ q = DList\ \textbf{in}\ P_1 :: \texttt{0}$.

Thus the composition of the queue and client code typechecks, and thus free from protocol violations, including races on heap variable access. On the other hand, consider the code snippet P_2: it enqueues an element in the queue after requesting an iterator ($i = q.iter$), but before using the iterator i. Due to aliasing, the $s.done$ operation may erroneously interfere with the mentioned enqueue operation, thus breaking the invariant that $head$ should always refer to the first element and $tail$ to the last element of the queue. Defensively, we have safeguarded against such misusage by setting $head$ to **nil** at iterator construction time (line $head := \textbf{nil}$).

So, in fact, P_2 "crashes" during evaluation of the last $q.enq$ expression, since the code for enq expects $head$ to hold a value of sum type $\circ OH$ (such a "crash" is signaled in our semantics by the fact that the $\textbf{case}\ head\ \textbf{of}\cdots$ expression will get stuck).

```
P1 ≜
q.enq; q.enq; q.deq;
let i = q.iter in (
  case i of
    NONE → nil
    | SOME(s) →
      letf = fork(s.scan) in
        (s.scan; wait(f); s.done))

P2 ≜
q.enq; q.enq; q.deq;
let i = q.iter in (
  q.enq;
  case i of
    NONE → nil
    | SOME(s) →
      letf = fork(s.scan) in
        (s.scan; wait(f); s.done))
```

Figure 7. Some Client Code

Obviously, P_2 does not typecheck under the assumption $q{:}QT$! The pre-condition of the body of expression ($\textbf{let}\ i = q.iter\ \textbf{in}\cdots$) is $(i{:}IT\,\texttt{;}\,q{:}QT)$, thus forcing the behavior $i{:}IT$ to be exercised before $q{:}QT$ (sequential separation). Likewise, in P_1 the type pre-condition of expression ($\textbf{let}\ f\ \textbf{in}\cdots$) right after $\texttt{SOME}(s) \rightarrow$ is

$$(s{:}!scan{:}\texttt{0}\,\texttt{;}\,s{:}done{:}\texttt{0}\,\texttt{;}\,q{:}QT)$$

thus allowing multiple threads to perform $scan$ concurrently (parallel separation), but forcing $s{:}done$ to be executed (exactly once) before proceeding with further queue operations, as shown.

Notice that the concurrent usages of the $scan$ operation, expressed by the type $!scan{:}\texttt{0}$ do not require locks in the implementation, since they only rely on read capabilities of shared resources (the cursor variable a is typed $a{:}!ON$). On the other hand, the synchronized blocks in $DNode$ have been introduced to allow invariant-based reasoning justify the safe sharing of list nodes by $head$ and $tail$, and do not play an essential role in operational terms, as discussed in Section 4.5.

A.5 Proofs of Main Results

We present proofs for the main (and some auxiliary) results in the paper. For clarity, we prove type preservation and progress for the basic type system, and afterwards address the full system with synchronization primitives.

We start by stating a few basic inversion principles for the typing relation (which only apply to values v).

LEMMA A.2 (Inversions). *We have*

1. *If $A \vdash_x v{::}B$ and $x \notin Dom(B)$ then $A <: B$ and $B \vdash_x v{::}B$.*
2. *If $A \vdash_x v{::}B_1 \mid B_2$ then there are A_1, A_2 such that $A <: A_1 \mid A_2$ and $A_1 \vdash v{::}B_1$ and $A_2 \vdash v{::}B_2$.*
3. *If $A \vdash_x v{::}B_1\,\texttt{;}\,B_2$ then there are A_1, A_2 such that $A <: A_1\,\texttt{;}\,A_2$ and $A_1 \vdash v{::}B_1$ and $A_2 \vdash v{::}B_2$.*
4. *If $A \vdash_x v{::}!B$ then there are A_i such that $A <: !A_1 \mid \cdots \mid !A_n$ and $!A_1 \mid \cdots \mid !A_n \vdash v{::}B$.*
5. *If $A \vdash_x v{::}\circ B$ then there are A_i such that $A <: \circ A_1 \mid \cdots \mid \circ A_n$ and $\circ A_1 \mid \cdots \mid \circ A_n \vdash v{::}B$.*

Proof. By induction on typing derivations. ∎

A generalized substitution lemma holds for the type system, which corresponds essentially to a natural cut principle in the behavioral separation type structure.

LEMMA A.3. *If $A \vdash_x v::B$ and $B \vdash_z e::C$ then there is D such that $A \vdash_z e\{v/x\}::D$ and $D \vdash_x v :: C$.*

Proof. By induction on the derivation of $B \vdash_z e::C$. Notice that x may occur in B, but not in A. In the statement conclusion, the judgment $D \vdash_x v :: C$ expresses the usage of v that remains after the use of x actually performed by e; this stronger property is particularly useful to handle the **let** expression case.

(Case (Id))
$B = y{:}U, C = z{:}U$ and $e = y$.
(Case $y = x$) $e\{v/x\} = v$.
$A \vdash_x v :: x{:}U$, by assumption
$A \vdash_z v :: C$, by conversion
$C \vdash_x v :: C$, by (VId)
(Case $y \neq x$) $e\{v/x\} = e = y$ and $x \notin Dom(B)$
$A <: B$ and $B \vdash_x v :: B$ by Lemma A.2(1)
$A \vdash_x v :: B$ by (Sub)
$A \vdash_z y :: C$, by conversion
$C \vdash_x v :: C$, by (VId)

(Case (Sub))
$B' \vdash_z e :: C'$, $B <: B'$, $C' <: C$, by inversion
$A \vdash_x v::B'$, from assumption, by (Sub)
$A \vdash_z e\{v/x\}::C''$, and $C'' \vdash_x v :: C'$, by ih.
$C'' \vdash_x v :: C$, by (Sub)

(Case (Par))
$B = B_1 \mid B_2, C = C_1 \mid B_2, B_1 \vdash_z e :: C_1$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \mid A_2$
by Lemma A.2(2)
$A_1 \vdash_z e\{v/x\} :: C_1'$, and $C_1' \vdash_x v :: C_1$, by ih.
$A_1 \mid A_2 \vdash_z e\{v/x\} :: C_1' \mid A_2$, by (Par)
$A \vdash_z e\{v/x\} :: C_1' \mid A_2$, by (Sub)
$C_1' \mid A_2 \vdash_x v :: C_1 \mid A_2$, by (Par)
$x \notin B_2$ so $A_2 <: B_2$ by Lemma A.2(1)
$C_1' \mid A_2 \vdash_x v :: C$, by (Sub)

(Case (Seq))
$B = B_1 \,;\, B_2, C = C_1 \,;\, B_2, B_1 \vdash_z e :: C_1$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \,;\, A_2$
by Lemma A.2(3)
$A_1 \vdash_z e\{v/x\} :: C_1'$, and $C_1' \vdash_x v :: C_1$, by ih.
$A_1 \,;\, A_2 \vdash_z e\{v/x\} :: C_1' \,;\, A_2$, by (Seq)
$A \vdash_z e\{v/x\} :: C_1' \,;\, A_2$, by (Sub)
$C_1' \,;\, A_2 \vdash_x v :: C_1 \,;\, A_2$, by (Seq)
$x \notin B_2$ so $A_2 <: B_2$ by Lemma A.2(1)
$C_1' \,;\, A_2 \vdash_x v :: C$, by (Sub)

(Case $(VAbs)$)
$e = \lambda u.e'$, $C = z{:}U \mapsto T$, $B \mid u{:}U \vdash_y e' :: y{:}T$, by inversion
$A \mid u{:}U \vdash_x v :: B \mid u{:}U$, by (Par)
$A \mid u{:}U \vdash_y e'\{v/x\} :: y{:}T'$, and $y{:}T' \vdash_x v :: y{:}T$, by ih.
$x \notin y{:}T$ so $T' <: T$ by Lemma A.2(1)
$A \mid u{:}U \vdash_y e'\{v/x\} :: y{:}T$, by (Sub)
$A \vdash_z e\{v/x\} :: C$, by $(VAbs)$
$C \vdash_x v :: C$, by (VId)

(Case (App))
$e = e_1 e_2$, $C = z{:}T$, $B = B_1 \mid B_2$,
$B_1 \vdash_z e_1 :: z{:}U \mapsto T$, $B_2 \vdash_z e_2 :: z{:}U$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \mid A_2$,
by Lemma A.2(2)
$A_1 \vdash_z e_1\{v/x\} :: z{:}U \mapsto T$, tby ih.
$A_2 \vdash_z e_2\{v/x\} :: z{:}U$, by ih.
$A \vdash_z e\{v/x\} :: C$, by (App)
$C \vdash_x v :: C$, by (VId)

(Case $(VStop)$)
$e = v'$, $C = B = \mathsf{0}$
$A <: \mathsf{0}$ and $\mathsf{0} \vdash_x v :: \mathsf{0}$ by Lemma A.2(1)
$A \vdash_x v'\{v/x\} :: C$ by $(VStop)$ and (Sub)
$C \vdash_x v :: C$, by (VId)

(Case $(VPar)$)
$e = v'$, $B = B_1 \mid B_2, C = C_1 \mid C_2$,
$B_1 \vdash_z v' :: C_1$, $B_2 \vdash_z v' :: C_2$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \mid A_2$,
by Lemma A.2(2)
$A_1 \vdash_z v'\{v/x\} :: C_1'$, and $C_1' \vdash_x v :: C_1$, by ih.
$A_2 \vdash_z v'\{v/x\} :: C_2'$, and $C_2' \vdash_x v :: C_2$, by ih.
$A_1 \mid A_2 \vdash_z v'\{v/x\} :: C_1' \mid C_2'$, by $(VPar)$
$A \vdash_z e\{v/x\} :: C_1' \mid C_2'$, by (Sub)
$C_1' \mid C_2' \vdash_x v :: C$, by $(VPar)$

(Case $(VSeq)$)
$e = v'$, $B = B_1 \,;\, B_2, C = C_1 \,;\, C_2$,
$B_1 \vdash_z v' :: C_1$, $B_2 \vdash_z v' :: C_2$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \,;\, A_2$,
by Lemma A.2(3)
$A_1 \vdash_z v'\{v/x\} :: C_1$, and $C_1' \vdash_x v :: C_1$, by ih.
$A_2 \vdash_z v'\{v/x\} :: C_2$, and $C_2' \vdash_x v :: C_2$, by ih.
$A_1 \,;\, A_2 \vdash_z v'\{v/x\} :: C_1' \,;\, C_2'$, by $(VSeq)$
$A \vdash_z e\{v/x\} :: C_1' \,;\, C_2'$, by (Sub)
$C_1' \,;\, C_2' \vdash_x v :: C$, by $(VSeq)$

(Case $(VShr)$)
$e = v'$, $B = !B_1 \mid \cdots$, $C = !C'$, and $B \vdash_z v' :: C'$, by inversion
$A <: !A_1 \mid \cdots$ and $!A_1 \mid \cdots \vdash_x v :: B$,
by Lemma A.2(4)
$!A_1 \mid \cdots \vdash_z v'\{v/x\} :: C''$, and $C'' \vdash_x v :: C'$, by ih.
$A \vdash_z v'\{v/x\} ::!C''$, by $(VShr)$ and (Sub)
$!C'' \vdash_x v :: C$, by (Sub) and $(VShr)$

(Case (Let))
$e = \mathbf{let}\ u = e_1\ \mathbf{in}\ e_2$, and $B \vdash_u e_1 :: B'$,
and $B' \vdash_z e_2 :: C$, by inversion
$A \vdash_u e_1\{v/x\} :: B''$, and $B'' \vdash_x v :: B'$, by ih.
$B'' \vdash_z e_2\{v/x\} :: B'''$, and $B''' \vdash_x v :: C$, by ih.
$A \vdash e\{v/x\} :: B'''$, by (Let)

(Case (Iso))
$B = \circ B_1 \mid \cdots$, $C = \circ C'$, and $B \vdash_z e :: C'$, by inversion
$A <: \circ A_1 \mid \cdots$ and $\circ A_1 \mid \cdots \vdash_x v :: B$, by Lemma A.2(5)
$\circ A_1 \mid \cdots \vdash_z e\{v/x\} :: C''$, and $C'' \vdash_x v :: C'$, by ih.
$A \vdash_z v\{v/x\} :: \circ C''$, by (Iso) and (Sub)
$\circ C'' \vdash_x v :: C$, by (Sub) and (Iso)

(Case $(Option)$)
$e = l_i(e')$, $C = z{:}\oplus_{l\in I}\, l{:}T_l$, $B \vdash_z e' :: z{:}T_i$, by inversion
$A \vdash_y e'\{v/x\} :: z{:}T_i$, by ih.
$A \vdash_z e\{v/x\} :: C$, by $(Option)$
$C \vdash_x v :: C$, by (VId)

(Case $(Tuple)$)
$e = [\ldots l = e' \ldots]$, $C = z{:}l{:}U$,
$B \vdash_z e' :: z{:}U$, by inversion
$A \vdash_z e'\{v/x\} :: z{:}U$, by ih.
$A \vdash_z e\{v/x\} :: C$, by $(Tuple)$
$C \vdash_x v :: C$, by (VId)

(Case (Sel))
$e = e'.l$, $C = z{:}T$, $B \vdash_z e' :: z{:}l{:}T$, by inversion
$A \vdash_z e'\{v/x\} :: z{:}l{:}T$, by ih.
$A \vdash_z e\{v/x\} :: C$, by (Sel)
$C \vdash_x v :: C$, by (VId)

(Case (And))
$C = C_1 \,\&\, C_2$, $B \vdash_z e :: z{:}C_1$,
and $B \vdash_z e :: z{:}C_2$, by inversion
$A \vdash_y e\{v/x\} :: C_1'$, and $C_1' \vdash_x v :: C_1$, by ih.
$A \vdash_y e\{v/x\} :: C_2'$, and $C_2' \vdash_x v :: C_2$, by ih.
$A \vdash_z e\{v/x\} :: C_1' \,\&\, C_2'$, by (And)
$C_1' \,\&\, C_2' \vdash_x v :: C$, by (Sub) and (And)

(Case $(Case)$)

$e = \textbf{case}\ e_c\ \textbf{of}\ l(x) \to e'$, $B = B_1 \mid B_2$,
$B_1 \vdash_y e_c :: y : \oplus_{l\in I} l{:}T_l$, $x_i{:}T_i \mid B_2 \vdash_z e'_i :: C$, by inversion
$A_1 \vdash_x v :: B_1$, $A_2 \vdash_x v :: B_2$ and $A <: A_1 \mid A_2$
by Lemma A.2(2)
$A_1 \vdash_y e_c\{v/x\} :: y : \oplus_{l\in I} l{:}T_l$, by ih.
$x_i{:}T_i \mid A_2 \vdash_y e'_i\{v/x\} :: C'$, and $C' \vdash v :: C$, by ih.
$A \vdash_z e\{v/x\} :: C$, by $(Case)$ and (Sub)
(Case (Var))
$e = \textbf{var}\ a\ \textbf{in}\ e'$, $B \mid a{:}\texttt{var} \vdash_z e' :: C$, by inversion
$A \mid a{:}\texttt{var} \vdash_x v :: B \mid a{:}\texttt{var}$, by (Par)
$A \mid a{:}\texttt{var} \vdash_z e'\{v/x\} :: y : C'$ and $C' \vdash_x v :: C$, by ih.
$A \vdash_z e\{v/x\} :: C'$ by (Var)
(Case $(RdVB)$)
$e = a$, $B = a{:}\texttt{rd}(U)$ and $C = z{:}U$, by inversion
Since $x \notin B$, $A <: B$ and $B \vdash_x v :: B$, by Lemma A.2(1).
$A \vdash_z e\{v/x\} :: C$, by (Sub)
$C \vdash_x v :: C$, by (VId)
(Case $(RdVF)$)
$e = a$, $B = a{:}\texttt{rd}(U)\,;\texttt{use}$, $C = z{:}U \mid a{:}\texttt{use}$, by inversion
Since $x \notin B$, $A <: B$ and $B \vdash_x v :: B$, by Lemma A.2(1).
$A \vdash_z e\{v/x\} :: C$, by (Sub)
$C \vdash_x v :: C$, by (VId)
(Case $(WrVF)$)
e is $a := u$ and $C = \texttt{0}$ and
$B \vdash_w u :: w{:}\circ U \mid a{:}\texttt{wr}(\circ U)$, by inversion
Let $D = w{:}\circ U \mid a{:}\texttt{wr}(\circ U)$
$A \vdash_w u\{v/x\} :: C'$ and $C' \vdash_x v :: D$, by ih.
Since $x \notin C$, $C' <: D$, and $D \vdash_x v :: D$ by Lemma A.2(1).
$A \vdash_w u\{v/x\} :: D$, by (Sub)
$A \vdash_w e\{v/x\} :: C$, by $(WrVF)$
$C \vdash_x v :: C$, by (VId)
(Case $(WrVB)$)
e is $a := u$ and $C = a{:}\texttt{rd}(U)$ and
$B \vdash_w u :: w{:}U \mid a{:}\texttt{use}$, by inversion
Let $D = w{:}\circ U \mid a{:}\texttt{use}$
$A \vdash_w u\{v/x\} :: C'$ and $C' \vdash_x v :: D$, by ih.
Since $x \notin C$, $C' <: D$ and $D \vdash_x v :: D$, by Lemma A.2(1).
$A \vdash_w e\{v/x\} :: C$, by (Sub)
$A \vdash_z e\{v/x\} :: C$, by $(WrVB)$
$C \vdash_x v :: C$, by (VId) ∎

By the previous result, it is straightforward to state and proof the following specialized substitution lemma, useful, for example, to show type preservation of beta reduction.

LEMMA A.4. *If $A \vdash_x v :: x{:}U$ and $B \mid x{:}U \vdash_z e :: C$ where $x \notin Dom(C)$ then $B \mid A \vdash_z e\{v/x\} :: C$.*

Proof. Assume $A \vdash_x v :: x{:}U$.
$B \mid A \vdash_x v :: B \mid x{:}U$, by (Par)
There is D such that $B \mid A \vdash_z e\{v/x\} :: D$
and $D \vdash_x v :: C$, by Lemma A.3
$D <: C$ by Lemma A.2(1), hence
$B \mid A \vdash_z e\{v/x\} :: C$ by (Sub). ∎

The next lemma covers critical cases of the preservation theorem. We recall that we use a similar notation for evaluation contexts and type assertion contexts ($\mathcal{E}$, $\mathcal{F}$), without confusion arising.

LEMMA A.5. *Let $\mathcal{F}[-]$ be an evaluation context. We have,*

1. *If $A \vdash_x \mathcal{F}[\ \textbf{let}\ z = v\ \textbf{in}\ e\ ] :: B$ then $A \vdash_x \mathcal{F}[\ e\{v/z\}\ ] :: B$;*
2. *If $A \vdash_x \mathcal{F}[\ (\lambda z.e)v\ ] :: B$ then $A \vdash_x \mathcal{F}[\ e\{v/z\}\ ] :: B$;*
3. *If $A \vdash_x \mathcal{F}[\ [\overline{l = e}].l_i\ ] :: B$ then $A \vdash_x \mathcal{F}[\ e_i\ ] :: B$;*
4. *If $A \vdash_x \mathcal{F}[\ \textbf{case}\ l_i(v)\ \textbf{of}\ \overline{l(x) \to e}\ ] :: B$ then $A \vdash_x \mathcal{F}[\ e_i\{v/x_i\}\ ] :: B$;*
5. *If $A \vdash_x \mathcal{F}[\ \textbf{var}\ a\ \textbf{in}\ e\ ] :: B$ then $A \mid n{:}\texttt{var} \vdash_x \mathcal{F}[\ e\{n/a\}\ ] :: B$;*
6. *If $A \vdash_x \mathcal{F}[\ n := v\ ] :: B$ then there is C such that either*
   (a) $A <: \mathcal{E}[\ n{:}\texttt{use} \mid C\ ]$ and $\mathcal{E}[\ n{:}\texttt{rd}(U)\ ] \vdash_x \mathcal{F}[\ \textbf{nil}\ ] :: B$ and $C \vdash_z v :: z{:}U$.
   (b) $A <: \mathcal{E}[\ n{:}\texttt{wr}(\circ U) \mid C\ ]$ and $\mathcal{E}[\ \texttt{0}\ ] \vdash_x \mathcal{F}[\ \textbf{nil}\ ] :: B$ and $C \vdash_z v :: z{:}\circ U$.
7. *If $A \vdash_x \mathcal{F}[\ n\ ] :: B$ then either*
   (a) $A <: \mathcal{E}[\ n{:}\texttt{rd}(U)\ ]$ and for all $C \vdash_z v :: z{:}U$ we have $\mathcal{E}[C] \vdash_x \mathcal{F}[\ v\ ] :: B$;
   (b) $A <: \mathcal{E}[\ n{:}\texttt{rd}(U); \texttt{use}\ ]$ and for all $C \vdash_z v :: z{:}U$ we have $\mathcal{E}[\ n{:}\texttt{use} \mid C\ ] \vdash_x \mathcal{F}[v] :: B$;
8. *If $A \vdash_x \mathcal{F}[\ \textbf{fork}(e)\ ] :: B$ then $A <: \mathcal{E}[C]$ and $C \vdash_z e :: z{:}U$ and $\mathcal{E}[t{:}\tau(U)] \vdash_x \mathcal{F}[\ t\ ] :: B$, for fresh t ;*
9. *If $A \vdash_x \mathcal{F}[\ \textbf{wait}(t)\ ] :: B$ then $A <: \mathcal{E}[t{:}\tau(U)]$ and for all $C \vdash_z v :: z{:}U$ we have $\mathcal{E}[C] \vdash_x \mathcal{F}[\ v\ ] :: B$;*

Proof. Induction on the typing derivations. We detail some cases of *2*, induction on the derivation of $A \vdash_x \mathcal{F}[\ (\lambda z.e)v\ ] :: B$.
(Case (Par))
$A = A_1 \mid C$, $B = B_1 \mid C$
$A_1 \vdash_x \mathcal{F}[(\lambda z.e)v] :: B_1$, by inversion
$A_1 \vdash_x \mathcal{F}[e\{z/v\}] :: B_1$, by i.h.
$A \vdash \mathcal{F}[e\{v/z\}] :: B$ by (Par)
(Case (Let))
$\mathcal{F}[-] = \textbf{let}\ y = \mathcal{G}[-]\ \textbf{in}\ e_2$
$A \vdash_y \mathcal{G}[(\lambda z.e)v] :: C$, $C \vdash_x e_2 :: B$, by inversion
$A \vdash_y \mathcal{G}[e\{z/v\}] :: C$, by i.h.
$A \vdash \mathcal{F}[e\{v/z\}] :: B$ by (Let)
(Case (App))
$\mathcal{F}[(\lambda z.e)v] = e_1 e_2$, $B = x{:}T$, $A = A_1 \mid A_2$,
$A_1 \vdash_y e_1 :: y{:}U \mapsto T$, $A_2 \vdash_w e_2 :: w{:}U$, by inversion
(Case $\mathcal{F}[-] = \square$)
$e_1 = \lambda z.e$, $e_2 = v$
$A_1 \mid z{:}U \vdash_y e :: y{:}T$, by inversion
$A_2 \vdash_z v :: z{:}U$ by (Par)
$A_1 \mid A_2 \vdash_y e\{v/z\} :: y{:}T$ by Lemma A.4
$A \vdash_x e\{v/z\} :: x{:}T$, by renaming
(Case $\mathcal{F}[-] = \mathcal{G}[-]e_2$, with $\mathcal{G}[(\lambda z.e)v] = e_1$)
$A_1 \vdash_y \mathcal{G}[(\lambda z.e)v] :: y{:}U \mapsto T$.
$A_1 \vdash_y \mathcal{G}[e\{v/z\}] :: y{:}U \mapsto T$, by i.h.
$A \vdash \mathcal{G}[e\{v/z\}]e_2 :: B$, $A \vdash \mathcal{F}[e\{v/z\}] :: B$ by (App)
(Case $\mathcal{F}[-] = v_1\mathcal{G}[-]$, with $e_1 = v_1$, and $\mathcal{G}[(\lambda z.e)v] = e_2$)
$A_2 \vdash_w \mathcal{G}[(\lambda z.e)v] :: w{:}U$
$A_2 \vdash_w \mathcal{G}[e\{v/z\}] :: w{:}U$, by i.h.
$A \vdash e_1\mathcal{G}[e\{v/z\}] :: B$, $A \vdash \mathcal{F}[e\{v/z\}] :: B$ by (App) ∎

The following technical properties are related to the propagation of footprints from the post-condition to pre-condition of typing judgments of a certain form, often useful to prove results below. In all the four next statements $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$ are assumed to be active type assertion contexts. The proof is in all cases a simple induction on typing derivations, we also give some useful justifications.

LEMMA A.6. *Let $A \vdash_x v :: \mathcal{E}[x{:}U]$. Then there is C such that $C \vdash_x v :: x{:}U$ and $A \vdash_x v :: \mathcal{E}[C]$.*

LEMMA A.7. *Let $D \vdash_y v :: B$ and $B <: \mathcal{F}[n{:}\texttt{rd}(U)]$. Then there is $\mathcal{E}[]$ such that $D <: \mathcal{E}[n{:}\texttt{rd}(U)]$ and for all C, $\mathcal{E}[C] \vdash_y v :: \mathcal{F}[C]$.*

Intuitively, Lemma A.7 holds as a consequence of $\mathcal{F}$ being active: $n{:}\texttt{rd}(U)$ is available at the front of the post-condition, and thus cannot be a residual of an usage of the value v, which would need to be either in the front of the context hole (contradiction), or isolated, but then the variable residual n could not have escaped.

LEMMA A.8. *Let $A <: \mathcal{E}[n{:}\texttt{rd}(U)]$. Then there are $\mathcal{F}$, $\mathcal{G}$ such that $A = \mathcal{F}[n{:}\texttt{rd}(V)]$, $n{:}\texttt{rd}(V) <: \mathcal{G}[n{:}\texttt{rd}(U)]$ and for all C, $\mathcal{F}[\mathcal{G}[C]] <: \mathcal{E}[C]$.*

This last lemma, states that an active $a{:}\mathtt{rd}(U)$ capability in a type assertion selected by subsumption from A must result from some active $a{:}\mathtt{rd}(V)$ capability in A, such that U is active in V. This results from the distributing subtyping axioms for $\mathtt{rd}(U)$, e.g., $\mathtt{rd}(U_1 \mid U_2) <: \mathtt{rd}(U_1) \mid \mathtt{rd}(U_2)$, and so on. The next lemma expresses a basic inversion principle for $B\{x/n{:}\mathtt{var}\}$.

LEMMA A.9. *Let $B\{y/n{:}\mathtt{var}\} <: \mathcal{E}[n{:}\mathtt{rd}(U)]$ with n fresh in B. Then there are $\mathcal{G}$, $\mathcal{F}$ and V such that $B = \mathcal{G}[y{:}V]$, $y{:}V <: \mathcal{F}[y{:}U]$, and for all C, $\mathcal{G}[\mathcal{F}[C]]\{y/n{:}\mathtt{var}\} <: \mathcal{E}[C]$.*

The next lemma states several key inversion principles for the basic typing relation for configurations.

LEMMA A.10 (Configuration Typing Inversions). *Let $\mathcal{E}[-]$ be an active context. We have*

1. *Let $h\ ;\ S\ \triangleright A$ and $A <: \mathcal{E}[\ n{:}\mathtt{rd}(U)\ ]$. Then $h(n) = v$ and there is C such that $C \vdash_x v :: x{:}U$ and $h\ ;\ S\ \triangleright \mathcal{E}[\ C\ ]$.*
2. *Let $h\ ;\ S \triangleright A$ and $A <: \mathcal{E}[\ n{:}\mathtt{rd}(U); \mathtt{use}\ ]$. Then $h(n) = v$ and is C s.t. $C \vdash_x v :: x{:}U$ and $h, n \mapsto v\ ;\ S\ \triangleright \mathcal{E}[\ n{:}\mathtt{use} \mid C\ ]$.*
3. *Let $h\ ;\ S \triangleright A$ and $A <: \mathcal{E}[\ C \mid n{:}\mathtt{use}\ ]$. Then $h = h'[n \mapsto u]$, and $C \vdash_x v :: x{:}U$ implies $h'[n \mapsto v]\ ;\ S\ \triangleright \mathcal{E}[\ n{:}\mathtt{rd}(U)\ ]$.*
4. *Let $h\ ;\ S \triangleright A$ and $A <: \mathcal{E}[\ n{:}\mathtt{wr}(\circ U) \mid C\ ]$. Then $h = h', n \mapsto u$, and $C \vdash_x v :: x{:}\circ U$ implies $h', n \mapsto v\ ;\ S\ \triangleright \mathcal{E}[\ \mathtt{0}\ ]$.*
5. *Let $h\ ;\ S\ \triangleright A$ and $A <: \mathcal{E}[\ t{:}\tau(U)\ ]$. Then $S = S' \cdot t\langle e\rangle$ and there is C such that $C \vdash_x e :: x{:}U$ and $h\ ;\ S\ \triangleright \mathcal{E}[\ C\ ]$.*

Proof. In all cases, the proof is by induction on the derivation of $h\ ;\ S\ \triangleright A$. We detail item 1.

1. Induction on the derivation of $h\ ;\ S\ \triangleright A$. We use two hole active contexts (e.g. $\mathcal{F}[-][-]$), defined as expected.

   (Case (E)) not possible.

   (Case (T))

   $A = \mathcal{E}'[t{:}\tau(T)] <: \mathcal{E}[n{:}\mathtt{rd}(U)]$, $S = S' \cdot t\langle e\rangle$, $h\ ;\ S'\ \triangleright \mathcal{E}'[C]$, and $C \vdash_y e :: y{:}T$, by inversion (for some $\mathcal{E}'$, S', etc).
   $\mathcal{E}'[t{:}\tau(T)] = \mathcal{F}[t{:}\tau(T)][n{:}\mathtt{rd}(V)] <: \mathcal{E}[n{:}\mathtt{rd}(U)]$, for some $\mathcal{F}$
   $\mathcal{E}[n{:}\mathtt{rd}(U)] = \mathcal{E}''[t{:}\tau(T)][n{:}\mathtt{rd}(U)]$
   $\mathcal{E}'[C] = \mathcal{F}[C][n{:}\mathtt{rd}(V)] <: \mathcal{E}''[C][n{:}\mathtt{rd}(U)]$
   $n{:}\mathtt{rd}(V) <: \mathcal{G}[n{:}\mathtt{rd}(U)]$ and
   $\mathcal{F}[C][\mathcal{G}[-]] <: \mathcal{E}''[C][-]$, for some $\mathcal{G}$
   $\mathcal{E}'[C] = \mathcal{F}[C][n{:}\mathtt{rd}(V)] <: \mathcal{F}[C][\mathcal{G}[n{:}\mathtt{rd}(U)]]$
   $h = h', n \mapsto v$ and exists D such that $D \vdash_x v :: x{:}U$
   and $h\ ;\ S' \triangleright \mathcal{F}[C][\mathcal{G}[D]]$, by i.h.
   $h\ ;\ S \triangleright \mathcal{E}''[C][D]$, by above, and $h\ ;\ S \triangleright \mathcal{E}''[t{:}\tau(T)][D]$, by (T)
   $h\ ;\ S \triangleright \mathcal{E}[D]$

   (Case (H))

   $h = h'', m \mapsto u$ and $h''\ ;\ S\ \triangleright D$ and $D \vdash_y u :: B$ and
   $A = B\{y/m{:}\mathtt{var}\} <: \mathcal{E}[n{:}\mathtt{rd}(U)]$, by inversion.
   (Subcase $m = n$)
   $B = \mathcal{B}[y{:}V]$, $y{:}V <: \mathcal{F}[y{:}U]$, and $\mathcal{B}[\mathcal{F}[\ ]]\{y/n{:}\mathtt{var}\} <: \mathcal{E}[\ ]$, by Lemma A.9.
   Hence $D \vdash_y u :: \mathcal{B}[\mathcal{F}[y{:}U]]$.
   $D'' \vdash_y u :: y{:}U$, $D \vdash_y u :: \mathcal{B}[\mathcal{F}[D'']]$, by Lemma A.6(1)
   Set $h'' = h'$, $u = v$. So $h = h', n \mapsto v$ and $D'' \vdash_x v :: x{:}U$
   and $D \vdash_y v :: \mathcal{B}[\mathcal{F}[D'']]$ and $h'\ ;\ S \triangleright D$
   Let $A' = \mathcal{B}[\mathcal{F}[D'']]\{y/n{:}\mathtt{var}\}$, so $h\ ;\ S \triangleright A'$, by (H)
   We have $A' <: \mathcal{E}[D'']$, so $h\ ;\ S \triangleright \mathcal{E}[D'']$, by (S)
   (Subcase $m \neq n$)
   $B <: \mathcal{F}[n{:}\mathtt{rd}(U)]$ where $\mathcal{F}[-]\{y/m{:}\mathtt{var}\} = \mathcal{E}[-]$
   $D <: \mathcal{E}'[n{:}\mathtt{rd}(U)]$ and $\mathcal{E}'[C] \vdash_y u :: \mathcal{F}[C]$ by Lemma A.8.
   $h'' = h''', n \mapsto v$ and exists C such that $C \vdash_x v :: x{:}U$
   and $h''\ ;\ S' \triangleright \mathcal{E}'[C]$, by i.h.
   $h, S \triangleright \mathcal{F}[C]\{y/n{:}\mathtt{var}\}$, by (H)
   $h, S \triangleright \mathcal{E}[C]$, by identity above

   (Case (S)) By i.h. ∎

We prove the basic type preservation and progress Theorems.

THEOREM 4.3. *If $h\ ;\ S\ \triangleright A$ and $h\ ;\ S \to h'; S'$ then $h'; S'\ \triangleright A$.*

Proof. By induction on the derivation of $h\ ;\ S \triangleright A$.
(Case (E)) not possible.
(Case (S)) We have
$h\ ;\ S\ \triangleright A$
$h\ ;\ S\ \triangleright B$ and $B <: A$, by inversion
$h'\ ;\ S'\ \triangleright B$, by ih.
$h'\ ;\ S'\ \triangleright A$, by (S)
(Case (H)) We have
$h_p, n \mapsto v\ ;\ S\ \triangleright C\{x/n{:}\mathtt{var}\}$
$h_p\ ;\ S\ \triangleright B$ and $B \vdash_x v :: C$, by inversion
$h'_p\ ;\ S'\ \triangleright B$, by ih.
$h'_p, n \mapsto v\ ;\ S'\ \triangleright C\{x/n{:}\mathtt{var}\}$, by (H)
(Case (T))
$h\ ;\ R \cdot t\langle e\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$
$h\ ;\ R\ \triangleright \mathcal{E}[C]$ and $C\ \vdash_x e :: x{:}T$, by inversion
There are two subcases:
(Subcase a) $h\ ;\ R \cdot t\langle e\rangle \to h'\ ;\ R' \cdot t\langle e\rangle$
$h'\ ;\ R'\ \triangleright B$, by ih.
$h'\ ;\ R' \cdot t\langle e\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(Subcase b) $h\ ;\ R \cdot t\langle e\rangle \to h'\ ;\ R' \cdot t\langle e'\rangle$.
We consider all possible cases for this reduction step.
(SubCase (Red let)) We have
$e = \mathcal{F}[\textsf{let}\ z = v\ \textsf{in}\ e_1]$ and $e' = \mathcal{F}[e_1\{v/z\}]$
$h' = h$ and $R' = R$
$C\ \vdash_x \mathcal{F}[e_1\{v/z\}] :: x{:}T$, by Lemma A.5(1)
$h'\ ;\ R' \cdot t\langle e'\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red beta)) We have
$e = \mathcal{F}[(\lambda z.e_1)v]$ and $e' = \mathcal{F}[e_1\{v/z\}]$
$h' = h$ and $R' = R$
$C\ \vdash_x \mathcal{F}[e_1\{v/z\}] :: x{:}T$, by Lemma A.5(2)
$h'\ ;\ R' \cdot t\langle e'\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red sel)) We have
$e = \mathcal{F}[\overline{l = e}.l_i]$ and $e' = \mathcal{F}[e_i]$
$h' = h$ and $R' = R$
$C\ \vdash_x \mathcal{F}[e_i] :: x{:}T$, by Lemma A.5(3)
$h'\ ;\ R' \cdot t\langle e'\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red case)) We have
$e = \mathcal{F}[\ \textsf{case}\ l_i(v)\ \textsf{of}\ \overline{l(x) \to e}]$ and $e' = \mathcal{F}[e_i\{v/x_i\}]$
$h' = h$ and $R' = R$
$C\ \vdash_x \mathcal{F}[e_i\{v/x_i\}] :: x{:}T$, by Lemma A.5(3)
$h'\ ;\ R' \cdot t\langle e'\rangle\ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red var)) We have
$e = \mathcal{F}[\textsf{var}\ a\ \textsf{in}\ e_1]$ and $e' = \mathcal{F}[e_1\{n/a\}]$
$h' = h, n \mapsto \textsf{nil}$ and $R' = R$
$C \vdash_x \mathcal{F}[\textsf{var}\ a\ \textsf{in}\ e_1] :: x{:}T$, from above
$C\ \mid n{:}\mathtt{var} \vdash_x \mathcal{F}[e_1\{n/a\}] :: x{:}T$, by Lemma A.5(5)
$C \vdash_w \textsf{nil} :: w{:}\mathtt{0}\,;C$, by (VId)
$h', R' \triangleright C \mid n{:}\mathtt{var}$, by (H)
$h'; R' \cdot t\langle e'\rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red deref)) We have
$e = \mathcal{F}[a]$ and $e' = \mathcal{F}[v]$, $h' = h$ and $h(a) = v$ and $R' = R$
$C \vdash_x \mathcal{F}[a] :: x{:}T$, from above
(subcase *(a)*) $C <: \mathcal{G}[a{:}\mathtt{rd}(U)]$, by Lemma A.5(7a)
$h; R' \triangleright \mathcal{E}[\mathcal{G}[D]]$ and $D \vdash_z v :: z{:}U$, by Lemma A.10(1)
$\mathcal{G}[D] \vdash_x \mathcal{F}[v] :: x{:}T$, by Lemma A.5(7a)
$h'; R' \cdot t\langle e'\rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(subcase *(b)*) $C <: \mathcal{G}[a{:}\mathtt{rd}(U)\,;\mathtt{use}]$, by Lemma A.5(7b)
$h; R' \triangleright \mathcal{G}[D \mid a{:}\mathtt{use}]$ and $D \vdash_z v :: z{:}U$, by Lemma A.10(2)
$\mathcal{G}[D \mid a{:}\mathtt{use}] \vdash_x \mathcal{F}[v] :: x{:}T$, by Lemma A.5(7b)
$h'; R' \cdot t\langle e'\rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)

(SubCase (Red assign))
$e = \mathcal{F}[a := v]$ and $e' = \mathcal{F}[\mathbf{nil}]$
$h = h_p, a \mapsto u$ and $h' = h_p, a \mapsto v$ and $R' = R$
$C \vdash_x \mathcal{F}[a := v] :: x{:}T$, from above
(subcase *(a)*) $C <: \mathcal{G}[a : \mathtt{use} \mid D]$,
$\mathcal{G}[a : \mathtt{rd}(U)] \vdash_x \mathcal{F}[\mathbf{nil}] :: x{:}T$,
$D \vdash_y v :: y{:}U$, by Lemma A.5(6a)
$h'; R' \triangleright \mathcal{E}[\mathcal{G}[a : \mathtt{rd}(U)]]$, by Lemma A.10(3)
$h'; R' \cdot t\langle e' \rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(subcase *(b)*) $C <: \mathcal{G}[a : \mathtt{wr}(\circ U) \mid D]$,
$\mathcal{G}[\mathtt{0}] \vdash_x \mathcal{F}[\mathbf{nil}] :: x{:}T$,
$D \vdash_y v :: y{:}\circ U$, by Lemma A.5(6b)
$h'; R' \triangleright \mathcal{E}[G[\mathtt{0}]]$, by Lemma A.10(4)
$h'; R' \cdot t\langle e' \rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red fork)) We have
$e = \mathcal{F}[\mathbf{fork}(e_1)]$ and $e' = \mathcal{F}[k]$, with fresh k
$h' = h$ and $R' = R \cdot k\langle e \rangle$
$C <: \mathcal{G}[D]$, $D \vdash_y e_1 :: y{:}U$,
$\mathcal{G}[k{:}\tau(U)] \vdash_x \mathcal{F}[k]{::}x{:}T$, by Lemma A.5(8)
$h' \ ; \ R \cdot k\langle e_1 \rangle \ \triangleright \mathcal{E}[\mathcal{G}[k{:}\tau(U)]]$, by (T)
$h' \ ; \ R' \cdot t\langle e' \rangle \ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red wait)) We have
$e = \mathcal{F}[\mathbf{wait}(k)]$ and $e' = \mathcal{F}[v]$
$h' = h$ and $R = R' \cdot k\langle v \rangle$
$C <: \mathcal{G}[k{:}\tau(U)]$, by Lemma A.5(9)
$D \vdash_y v :: y{:}U$, and
$h \ ; \ R' \triangleright \mathcal{E}[\mathcal{G}[D]]$, by Lemma A.10(5)
$\mathcal{G}[D] \vdash_x \mathcal{F}[v] :: x{:}T$, by Lemma A.5(9)
$h' \ ; \ R' \cdot t\langle e' \rangle \ \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)

■

Progress, Theorem 4.4, is a consequence of the following.

PROPOSITION A.11. *If* $h \ ; \ S \ \triangleright \mathcal{E}[C]$ *and* $C \vdash_x e :: D$ *and* $live(e)$ *then* $h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$.

Proof. By induction on the derivation of $C \vdash_x e :: D$. (Cases (Id), $(VStop)$, $(VAbs)$, $(VPar)$, $(VSeq)$, $(VShr)$) Not possible.
(Case (Sub))
$C' \vdash_x e :: D'$, $C <: C'$, $D' <: D$, by inversion
$h \ ; \ S \ \triangleright \mathcal{E}[C']$ by (S)
$h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$. by ih.
(Case (Par))
$C = C_1 \mid C_2$, $D = D_1 \mid C_2$, $C_1 \vdash_x e :: D_1$, by inversion
$h \ ; \ S \ \triangleright \mathcal{G}[C_1]$ where $\mathcal{G}[-] = \mathcal{E}[- \mid C_2]$ by (S)
$h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$. by ih.
(Case (Seq))
$C = C_1 \,;\! C_2$, $D = D_1 \,;\! C_2$, $C_1 \vdash_x e :: D_1$, by inversion
$h \ ; \ S \ \triangleright \mathcal{G}[C_1]$ where $\mathcal{G}[-] = \mathcal{E}[- \,;\! C_2]$ by (S)
$h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$. by ih.
(Case (App))
$e = e_1 e_2$, $D = x{:}V$, $C = C_1 \mid C_2$,
$C_1 \vdash_x e_1 :: z{:}U \mapsto V$, $C_2 \vdash_x e_2 :: z{:}U$, by inversion
(Subcase $(live(e_1))$)
$h \ ; \ S \ \triangleright \mathcal{G}[C_1]$ where $\mathcal{G}[-] = \mathcal{E}[- \mid C_2]$ by (S)
$h \ ; \ S \cdot t\langle e_1 \rangle \to h'; S \cdot t\langle e_1' \rangle$, by i.h.
$h \ ; \ S \cdot t\langle e_1 e_2 \rangle \to h'; S \cdot t\langle e_1' e_2 \rangle$ for evaluation context $\Box e$
(Subcase $(live(e_2), e_1 = v)$)
$h \ ; \ S \ \triangleright \mathcal{G}[C_2]$ where $\mathcal{G}[-] = \mathcal{E}[C_1 \mid -]$ by (S)
$h \ ; \ S \cdot t\langle e_2 \rangle \to h'; S \cdot t\langle e_2' \rangle$, by i.h.
$h \ ; \ S \cdot t\langle v e_2 \rangle \to h'; S \cdot t\langle v e_2' \rangle$ for evaluation context $v\Box$
(Subcase (e_1, e_2 values)) $e_1 = \lambda z.e_1'$ and $e_2 = v$
$h; \ t\langle (\lambda z.e_1') v \rangle \to h; \ t\langle e_1'\{v/z\} \rangle$ by (Red beta)
(Case (Iso))
$C = \circ C_1 \mid \cdots$, $D = \circ D'$, and $C \vdash_z e :: D'$, by inversion
$h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$. by ih.
(Case (And))
$D = D_1 \,\&\, D_2$, and $C \vdash_z e :: D_1$, and $C \vdash_z e :: D_2$, by inversion
$h \ ; \ S \cdot t\langle e \rangle \to h'; S \cdot t\langle e' \rangle$. by ih.
(Case (Sel))
e is $e_s.l$, $D = z : l : T$, and $C \vdash_z e_s :: z : T$, by inversion
(Subcase $(live(e_s))$)
$h \ ; \ S \cdot t\langle e_s \rangle \to h'; S \cdot t\langle e_s' \rangle$. by ih.
$h \ ; \ S \cdot t\langle e_s.l) \rangle \to h'; S \cdot t\langle e_s'.l \rangle$ for evaluation context $\Box.l$
(Subcase $(e_s = v)$)
$v = [\ldots l = e_l \ldots]$
$h \ ; \ S \cdot t\langle [\ldots l = e_l \ldots].l \rangle \to h'; S \cdot t\langle e_l \rangle$ by (Red del)
(Case $(Case)$)
$e = \mathbf{case}\ e_c\ \mathbf{of}\ l(x) \to e'$, $C = C_1 \mid C_2$,
$C_1 \vdash_y e_c :: y : \oplus_{l \in I} l{:}T_l$ and $x_i{:}T_i \mid C_2 \vdash_z e_i' :: C$, by inversion
(Subcase $(live(e_c))$)
$h \ ; \ S \ \triangleright \mathcal{G}[C_1]$ where $\mathcal{G}[-] = \mathcal{E}[- \mid C_1]$ by (S)
$h \ ; \ S \cdot t\langle e_c \rangle \to h'; S \cdot t\langle e_c' \rangle$. by ih.
$h; \ t\langle \mathbf{case}\ e_c\ \mathbf{of}\ l(x) \to e' \rangle \to h'; \ t\langle \mathbf{case}\ e_c'\ \mathbf{of}\ l(x) \to e' \rangle$
for evaluation context $\mathbf{case}\ \Box\ \mathbf{of}\ l(x) \to e'$
(Subcase $(e_c = v)$)
$e_c = l_i(v')$
$h; \ t\langle \mathbf{case}\ l_i(v')\ \mathbf{of}\ l(x) \rangle \to t\langle e' \to h'; \ e_i'\{v'/x_i\} \rangle$
by (Red case)
(Case $(Option)$)
$e = l_i(e_c)$, $D = \oplus_{l \in I} l{:}T_l$, $C \vdash_z e' :: T_i$, by inversion
$live(e_c)$
$h \ ; \ S \cdot t\langle e_c \rangle \to h'; S \cdot t\langle e_c' \rangle$. by ih.
$h; \ S \cdot t\langle l(e_c) \rangle \to h'; \ S \cdot t\langle l(e_c') \rangle$ for evaluation context $l(\Box)$
(Case (Let))
e is $\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2$, and $C \vdash_y e_1 :: C'$
and $C' \vdash_x e_2 :: D$, by inversion
(Subcase $live(e_1)$)
$h \ ; \ S \cdot t\langle e_1 \rangle \to h'; S \cdot t\langle e_1' \rangle$, by ih.
$h; \ S \cdot t\langle \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2 \rangle \to h'; \ S \cdot t\langle \mathbf{let}\ y = e_1'\ \mathbf{in}\ e_2 \rangle$
for evaluation context $\mathbf{let}\ y = \Box\ \mathbf{in}\ e_2$
(Subcase $e_1 = v$)
$h; \ S \cdot t\langle \mathbf{let}\ y = v\ \mathbf{in}\ e_2 \rangle \to h'; \ S \cdot t\langle e_2\{x/y\} \rangle$ by (Red let)
(Case (Var))
e is $\mathbf{var}\ a\ \mathbf{in}\ e_v$, by inversion
$h; \ S \cdot t\langle e \rangle \to h'; \ S \cdot t\langle e' \rangle$ by (Red var)
(Case $(RdVB)$)
$e = a$, $C = a{:}\mathtt{rd}(U)$ and $D = z{:}U$, by inversion
$h(a) = w$, by A.10 Lemma (1)
$h; \ S \cdot t\langle e \rangle \to h'; \ S \cdot t\langle e' \rangle$ by (Red deref)
(Case $(RdVF)$)
$e = a$, $C = a{:}\mathtt{rd}(U)\,;\mathtt{use}$, $D = z{:}U \mid a{:}\mathtt{use}$, by inversion
$h(a) = w$, by Lemma A.10 (2)
$h; \ S \cdot t\langle e \rangle \to h'; \ S \cdot t\langle e' \rangle$ by (Red deref)
(Case $(WrVF)$)
e is $a := u$ and $C \vdash_w u :: w{:}\circ U \mid a{:}\mathtt{wr}(\circ U)$, by inversion
$C <: \mathcal{F}[a{:}\mathtt{wr}(\circ U)]$
$h(a) = w$, by Lemma A.10(4)
$h; \ S \cdot t\langle e \rangle \to h'; \ S \cdot t\langle e' \rangle$ by (Red assign)
(Case $(WrVB)$)
e is $a := u$ and $C \vdash_w u :: w{:}U \mid a{:}\mathtt{use}$, by inversion
$C <: \mathcal{F}[a{:}\mathtt{use}]$, $h(a) = w$, by Lemma A.10(3)
$h; \ S \cdot t\langle e \rangle \to h'; \ S \cdot t\langle e' \rangle$ by (Red assign)

■

THEOREM 4.4. *If* $h \ ; \ S \ \triangleright A$ *and* $live(S)$ *then* $h \ ; \ S \to h'; S'$.

Proof. By induction on the derivation of $h \ ; \ S \ \triangleright A$ we can show that there is $h' \ ; \ S' \triangleright \mathcal{E}[C]$ (with $S = S' \cdot S''$ and $h = h', h''$) and $C \vdash_x e :: D$ where $live(e)$. We conclude by Proposition A.11. ■

We now extend our results to the full language with synchronization constructs and invariant-based separation. The reduction cases of Lemma A.5 are extended to the appropriate judgment form

$A \vdash^{\iota}_{x} e :: B$ as stated in Lemma A.13 below. Recall that a lock invariant is a heap assertion R such that $R <: \circ R$ (Section 4.5). An useful commutation property is the following:

LEMMA A.12. *Let R be a lock invariant. If $A \vdash^{\iota}_{x} v :: R \mid B$ then there is C such that $A <: C \mid R$ and $C \vdash^{\iota}_{x} v :: B$.*

LEMMA A.13. *Let $\mathcal{F}$ be an evaluation context. Then*

1. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{let}\ z = v\ \mathbf{in}\ e\ ] :: B$ then $A \vdash^{\iota}_{x} \mathcal{F}[\ e\{v/z\}\ ] :: B$;*
2. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ (\lambda z.e)v\ ] :: B$ then $A \vdash^{\iota}_{x} \mathcal{F}[\ e\{v/z\}\ ] :: B$;*
3. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \overline{[l = e]}.l_i\ ] :: B$ then $A \vdash^{\iota}_{x} \mathcal{F}[\ e_i\ ] :: B$;*
4. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{case}\ l_i(v)\ \mathbf{of}\ \overline{l(x) \to e}\ ] :: B$ then $A \vdash^{\iota}_{x} \mathcal{F}[\ e_i\{v/x_i\}\ ] :: B$;*
5. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{var}\ a\ \mathbf{in}\ e\ ] :: B$ then exists C and lock invariant R such that $A <: C \mid R$ and $C \mid n{:}\mathtt{var} \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B$;*
6. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ n := v\ ] :: B$ then there is C such that either*
   (a) $A <: \mathcal{E}[\ n{:}\mathtt{use} \mid C\ ]$ and $\mathcal{E}[\ n{:}\mathtt{rd}(U)\ ] \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{nil}\ ] :: B$ and $C \vdash^{\iota}_{z} v :: z{:}U$.
   (b) $A <: \mathcal{E}[\ n{:}\mathtt{wr}(\circ U) \mid C\ ]$ and $\mathcal{E}[\ \mathtt{0}\ ] \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{nil}\ ] :: B$ and $C \vdash^{\iota}_{z} v :: z{:}\circ U$.
7. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ n\ ] :: B$ then either*
   (a) $A <: \mathcal{E}[\ n{:}\mathtt{rd}(U)\ ]$ and for all $C \vdash^{\iota}_{z} v :: z{:}U$ we have $\mathcal{E}[C] \vdash^{\iota}_{x} \mathcal{F}[\ v\ ] :: B$;
   (b) $A <: \mathcal{E}[\ n{:}\mathtt{rd}(U); \mathtt{use}\ ]$ and for all $C \vdash^{\iota}_{z} v :: z{:}U$ we have $\mathcal{E}[\ n{:}\mathtt{use} \mid C\ ] \vdash^{\iota}_{x} \mathcal{F}[v] :: B$;
8. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{fork}(e)\ ] :: B$ then $A <: \mathcal{E}[C]$ and $C \vdash^{\iota}_{z} e :: z{:}U$ and $\mathcal{E}[t{:}\tau(U)] \vdash^{\iota}_{x} \mathcal{F}[\ t\ ] :: B$, for fresh t*
9. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{wait}(t)\ ] :: B$ then $A <: \mathcal{E}[t{:}\tau(U)]$ and for all $C \vdash^{\iota}_{z} v :: z{:}U$ we have $\mathcal{E}[C] \vdash^{\iota}_{x} \mathcal{F}[\ v\ ] :: B$;*
10. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{sync}(n)e\ ] :: B$ then $A \mid \iota(n) \vdash^{\iota}_{x} \mathcal{F}[sy(n)e] :: B$.*
11. *If $A \vdash^{\iota}_{x} \mathcal{F}[\ sy(n)v\ ] :: B$ then $A <: C \mid \iota(n)$ and $C \vdash^{\iota}_{x} \mathcal{F}[v] :: B$.*

Proof. By induction on the typing derivations. We detail some cases of *5* and *11*.

5. Induction on the derivation of $A \vdash^{\iota}_{x} \mathcal{F}[\ \mathbf{var}\ a\ \mathbf{in}\ e\ ] :: B$.
(Case (Par))
$A = A_1 \mid D$, $B = B_1 \mid D$
$A_1 \vdash^{\iota}_{x} \mathcal{F}[\mathbf{var}\ a\ \mathbf{in}\ e] :: B_1$, by inversion
Exists C and lock invariant R such that $A_1 <: C \mid R$
and $C \mid n{:}\mathtt{var} \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B_1$, by i.h.
$A <: C \mid R \mid D$
$D \mid C \mid n{:}\mathtt{var} \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B$, by (Par)
(Case (Var))
$A <: A_1 \mid R$, with lock invariant R
$A_1 \mid n{:}\mathtt{var} \vdash^{\iota\{R/a\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B$
by inversion, and fresh renaming
(Case (Seq))
$A = A_1\ ;D$, $B = B_1\ ;D$
$A_1 \vdash^{\iota}_{x} \mathcal{F}[\mathbf{var}\ a\ \mathbf{in}\ e] :: B_1$, by inversion
Exists C and lock invariant R such that $A_1 <: C \mid R$
and $C \mid n{:}\mathtt{var} \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B_1$, by i.h.
$(C \mid n{:}\mathtt{var})\ ;D \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B$, by (Seq)
$C \mid n{:}\mathtt{var} \mid D \vdash^{\iota\{R/n\}}_{x} \mathcal{F}[\ e\{n/a\}\ ] :: B$, by (Sub)
$A <: (C \mid R)\ ;D <: C \mid D \mid R$ (since $R <: \circ R$)

11. Induction on the derivation of $A \vdash^{\iota}_{x} \mathcal{F}[\ sy(n)v\ ] :: B$.
(Case (Par))
$A = A_1 \mid C$, $B = B_1 \mid C$
$A_1 \vdash^{\iota}_{x} \mathcal{F}[sy(n)v] :: B_1$, by inversion
$A_1 <: D \mid \iota(n)$ and $D \vdash^{\iota}_{x} \mathcal{F}[v] :: B_1$, by i.h.
$A <: D \mid C \mid \iota(n)$
$D \mid C \vdash^{\iota}_{x} \mathcal{F}[v] :: B$, by (Par)
(Case (Let))
$\mathcal{F}[-] = \mathbf{let}\ y = \mathcal{G}[-]\ \mathbf{in}\ e_2$
$A \vdash_y \mathcal{G}[sy(n)v] :: C$, $C \vdash_x e_2 :: B$, by inversion
$A <: D \mid \iota(n)$ and $D \vdash_y \mathcal{G}[v] :: C$, by i.h.
$D \vdash \mathcal{F}[v] :: B$ by (Let)
(Case (App))
$\mathcal{F}[sy(n)v] = e_1e_2$, $B = x{:}T$, $A = A_1 \mid A_2$,
$A_1 \vdash_y e_1 :: y{:}U \mapsto T$, $A_2 \vdash_w e_2 :: w{:}U$, by inversion
(Case $\mathcal{F}[-] = \mathcal{G}[-]e_2$, with $\mathcal{G}[sy(n)v] = e_1$)
$A_1 \vdash_y \mathcal{G}[sy(n)v] :: y{:}U \mapsto T$.
$A_1 <: D \mid \iota(n)$ and $D \vdash_y \mathcal{G}[v] :: y{:}U \mapsto T$, by i.h.
$A <: D \mid A_2 \mid \iota(n)$
$D \mid A_2 \vdash \mathcal{G}[v]e_2 :: B$, $D \mid A_2 \vdash \mathcal{F}[v] :: B$ by (App)
(Case $\mathcal{F}[-] = v_1\mathcal{G}[-]$, with $e_1 = v_1$, and $\mathcal{G}[sy(n)v] = e_2$)
$A_2 \vdash_w \mathcal{G}[sy(n)v] :: w{:}U$
$A_2 <: D \mid \iota(n)$ and $D \vdash_w \mathcal{G}[v] :: w{:}U$, by i.h.
$A <: A_1 \mid D \mid \iota(n)$
$A_1 \mid D \vdash e_1\mathcal{G}[v] :: B$, $A_1 \mid D \vdash \mathcal{F}[v] :: B$ by (App)
(Case (Sy))
$\mathcal{F}[-] = sy(a)e_1$
(Case $\mathcal{F}[-] = \Box$, $a = n$, $e_1 = v$)
$A \vdash^{\iota\setminus n}_{y} v :: \iota(n) \mid B$, by inversion
$A <: D \mid \iota(n)$ and $D \vdash^{\iota\setminus n}_{y} v :: B$ by Lemma A.12
$D \vdash^{\iota}_{y} \mathcal{F}[v] :: B$
(Case $\mathcal{F}[-] = sy(a)\mathcal{G}[-]$, $e_1 = \mathcal{G}[sy(n)v]$)
$A \vdash^{\iota\setminus a}_{y} \mathcal{G}[sy(n)v] :: \iota(a) \mid B$, by inversion
$A <: D \mid \iota(n)$ and $D \vdash^{\iota\setminus a}_{y} \mathcal{G}[v] :: \iota(a) \mid B$, by i.h.
$D \vdash^{\iota}_{y} sy(a)\mathcal{G}[v] :: B$, $D \vdash^{\iota}_{y} \mathcal{F}[v] :: B$ by (Sy)

■

All properties listed in Lemma A.5 also hold of the configuration typing $h\ ;\ S \triangleright_{\iota} A$ (see Section A.2).

Lemma A.14 pinpoints the interplay between the state of variable n lock and its invariant assertion $\iota(n)$ at the level of typing. Acquiring the lock transfers ownership of the invariant from the lock to post-condition (1), and releasing the lock transfers ownership of the invariant back to the lock (2).

LEMMA A.14. *Let $h\ ;\ S \triangleright_{\iota} A$, $n_k \mapsto v \in h$, and no thread in S holds lock n (no thread in S has an active $sy(n)e$ subexpression).*

1. *If $k = 0$ then $h[n_{-1}]\ ;\ S \triangleright_{\iota} A \mid \iota(n)$.*
2. *If $A <: C \mid \iota(n)$, and $k = -1$ then $h[n_0]\ ;\ S \triangleright_{\iota} C$.*

Proof. Induction on the configuration typing derivation, the base case being the step that introduces n in the heap h. In (1), we simply replace (HU) by (HL) in such step. In (2) we move $\iota(n)$ back over all heap cells in h (using Lemma A.12), and over all threads in S (using the assumption), until the step where n is introduced (base case), where we replace the instance of rule (HL) by (HU).

We now state and prove our final main result, the type preservation property for the full language with synchronization primitives and invariant-based reasoning.

THEOREM 4.5. *If $h; S \triangleright_{\iota} A$ and $h; S \to h'; S'$ then $h'; S' \triangleright_{\iota} A$.*

Proof. By induction on the derivation of $h\ ;\ S \triangleright_{\iota} A$. We detail some key cases.
(Case (HU)) We have
$h_p, n_0 \mapsto v\ ;\ S \triangleright_{\iota\{R/n\}} C\{x/n{:}\mathtt{var}\}$
$h_p\ ;\ S \triangleright_{\iota} B \mid R$ and $B \vdash^{\iota}_{x} v :: C$, by inversion
$h'_p\ ;\ S' \triangleright_{\iota} B \mid R$, by ih.
$h'_p, n_0 \mapsto v\ ;\ S' \triangleright_{\iota\{R/n\}} C\{x/n{:}\mathtt{var}\}$, by (HU)
(Case (HL)) We have
$h_p, n_{-1} \mapsto v\ ;\ S \triangleright_{\iota\{R/n\}} C\{x/n{:}\mathtt{var}\}$
$h_p\ ;\ S \triangleright_{\iota} B$ and $B \vdash^{\iota}_{x} v :: C$, by inversion
$h'_p\ ;\ S' \triangleright_{\iota} B$, by ih.
$h'_p, n_{-1} \mapsto v\ ;\ S' \triangleright_{\iota\{R/n\}} C\{x/n{:}\mathtt{var}\}$, by (HL)

(Case (T))
$h\ ;\ R \cdot t\langle e\rangle \ \triangleright_\iota \mathcal{E}[t{:}\tau(T)]$
$h\ ;\ R \ \triangleright_\iota \mathcal{E}[C]$ and $C \ \vdash^\iota_x e :: x{:}T$, by inversion
There are two subcases:
(Subcase a) $h\ ;\ R \cdot t\langle e\rangle \to h'\ ;\ R' \cdot t\langle e\rangle$
$h'\ ;\ R' \ \triangleright_\iota B$, by ih.
$h'\ ;\ R' \cdot t\langle e\rangle \ \triangleright_\iota \mathcal{E}[t{:}\tau(T)]$, by (T)
(Subcase b) $h\ ;\ R \cdot t\langle e\rangle \to h'\ ;\ R' \cdot t\langle e'\rangle$.
We consider some cases for this reduction step.
(SubCase (Red var)) We have
$e = \mathcal{F}[\mathbf{var}\ a\ \mathbf{in}\ e_1]$ and $e' = \mathcal{F}[e_1\{n/a\}]$
$h' = h, n_0 \mapsto \mathbf{nil}$ and $R' = R$
$C \vdash^\iota_x \mathcal{F}[\mathbf{var}\ a\ \mathbf{in}\ e_1] ::\ x{:}T$, from above
There are B and R such that $C <: B \mid S$ and $S <: \circ S$
$B \mid n{:}\mathtt{var} \vdash^{\iota\{S/n\}}_x \mathcal{F}[e_1\{n/a\}] :: x{:}T$
by Lemma A.13(1)
$B \vdash_w \mathbf{nil} :: w{:}\mathsf{0}\ ;\ B$, by (VId)
We have $\mathcal{E}[C] <: \mathcal{E}[B \mid S] <: \mathcal{E}[B] \mid S$
$h', R' \triangleright_{\iota\{S/n\}} \mathcal{E}[B] \mid n{:}\mathtt{var}$, by (HU)
$h', R' \triangleright_{\iota\{S/n\}} \mathcal{E}[B \mid n{:}\mathtt{var}]$, by (S)
$h'; R' \cdot t\langle e'\rangle \triangleright \mathcal{E}[t{:}\tau(T)]$, by (T)
(SubCase (Red syncin)) We have
$e = \mathcal{F}[\mathbf{sync}(n)e_1]$ and $e' = \mathcal{F}[sy(n)e_1]$, $h = h_1(n_0 \to u)$
and $h' = h_1[n_{-1} \to u]$ and $R' = R$
We have $C \mid \iota(n) \vdash^\iota_x \mathcal{F}[sy(n)e_1] :: x{:}T$
by Lemma A.13(2)
$h'\ ;\ R \triangleright_\iota \mathcal{E}[C] \mid \iota(n)$, by Lemma A.14(1)
$h'\ ;\ R \triangleright_\iota \mathcal{E}[C \mid \iota(n)]$, by (S)
$h'\ ;\ R' \cdot t\langle e'\rangle \ \triangleright_\iota \mathcal{E}'[t{:}\tau(T)]$, by (T)
(SubCase (Red syncout)) We have
$e = \mathcal{F}[sy(n)v]$ and $e' = \mathcal{F}[v]$, $h = h_1(n_{-1} \to u)$
and $h' = h_1[n_0 \to u]$ and $R' = R$
We have $C <: \mathcal{G}[n{:}v] \mid \iota(n)$ and $\mathcal{G}[n{:}v] \vdash^\iota_x \mathcal{F}[v] :: x{:}T$
by Lemma A.13(3)
$h'\ ;\ R \triangleright_\iota \mathcal{G}[n{:}v]$, by Lemma A.14(2)
$h'\ ;\ R' \cdot t\langle e'\rangle \ \triangleright_\iota \mathcal{E}[t{:}\tau(T)]$, by (T)

■

A.6 Progress up to locking

A progress property can be stated and proved for the full type system with synchronization primitives, by filtering out the situations where progress is hindered by the impossibility of acquiring a lock. Apart from this, well-typed programs are stuck free, and absent of races in the sense explained in Sections 4.4 and A.3.

We say that an expression e is *live up to locking* in heap h, noted $liveh(e, h)$, if it is neither a value, nor of the form $\mathcal{F}[\mathbf{sync}(a)e_1]$ and $a_{-1} \mapsto v \in h$ for some evaluation context $\mathcal{F}$.

A multiset S of threads is live up to locking in heap h, noted $liveh(S, h)$ if there is some thread $t\langle e\rangle$ in S such that $liveh(e, h)$. We can then prove the following "progress up to locking property".

THEOREM A.15. *If $h; S \triangleright_\iota A$ and $liveh(S, h)$ then $h; S \to h'; S'$.*

THEOREM A.16. *If $h; S \triangleright_\iota \mathcal{E}[C]$ and $C \vdash^\iota_x e :: D$ and $liveh(e, h)$ then $h; S \cdot t\langle e\rangle \to h'; S \cdot t\langle e'\rangle$.*

Proof. Identical to proof of Theorem 4.4, by induction on the typing derivation $C \vdash^\iota_x e :: D$. We show a few interesting cases.
(Case $(Sync)$)
$e = \mathbf{sync}(a)e_1$, $\iota(a) \mid C \vdash^{\iota\setminus a}_x e_1 :: \iota(a) \mid D$, by inversion
Since $liveh(e, h)$, we have $a_0 \mapsto u \in h$.
$h;\ t\langle \mathbf{sync}(a)e_1\rangle \to h[a_{-1}];\ t\langle sy(a)e_1\rangle$ by (Red syncin)
(Case (Sy))
$e = sy(a)e_1$, $C \vdash^{\iota\setminus a}_x e_1 :: \iota(a) \mid D$, by inversion
(Subcase $(liveh(e_1, h))$)
$h\ ;\ S \cdot t\langle e_1\rangle \to h'; S \cdot t\langle e'_1\rangle$, by i.h.
$h\ ;\ S \cdot t\langle sy(a)e_1\rangle \to h'; S \cdot t\langle sy(a)e'_1\rangle$
for evaluation context $sy(a)\Box$
(Subcase (e_1 value))
$e_1 = v$
$h;\ t\langle sy(a)v\rangle \to h[a_0];\ t\langle v\rangle$
where $a_{-1} \mapsto u \in h$, by (Red syncout)

■

A.7 Subtyping congruences

As discussed in Section 4.1, we have omitted in Figure 3 the rules expressing subtyping congruences. All type operators satisfy the expected covariant subtyping congruence principles, except $(-) \mapsto (-)$ and $\mathtt{wr}(-)$ which are contravariant, e.g.,

$$\frac{A <: A' \quad B <: B'}{A\,;B <: A'\,;B'} \qquad \frac{U <: V}{\mathtt{rd}(U) <: \mathtt{rd}(V)}$$

$$\frac{A' <: A \quad B <: B'}{A \mapsto B <: A' \mapsto B'} \qquad \frac{V <: U}{\mathtt{wr}(U) <: \mathtt{wr}(V)}$$

A.8 Some Detailed Typing Derivations

We complement the presentation in Section 4.3 and show in detail (decomposed in Figures 9, 10, 11, 12, and 13) the complete typing derivation for the motivating collection example presented in Section 2. The derivation for the atomic cell is shown in Figure 14.

$$\mathsf{0} \vdash_y^{\iota,\eta} v :: \mathsf{0}\ (VStop) \quad x{:}U \vdash_y^{\iota,\eta} x :: y{:}U\ (Id) \quad \frac{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x^{\iota,\eta} e :: B}{\circ A_1 \mid \ldots \mid \circ A_n \vdash_x^{\iota,\eta} e :: \circ B}\ (Iso) \quad \frac{!A_1 \mid \ldots \mid !A_n \vdash_x^{\iota,\eta} v :: B}{!A_1 \mid \ldots \mid !A_n \vdash_x^{\iota,\eta} v :: !B}\ (VShr)$$

$$\frac{A <: A' \quad A' \vdash_x^{\iota,\eta} e :: B' \quad B' <: B}{A \vdash_x^{\iota,\eta} e :: B}\ (Sub) \quad \frac{A \vdash_y^{\iota,\eta} e :: B \quad A \vdash_y^{\iota,\eta} e :: C}{A \vdash_y^{\iota,\eta} e :: B \ \&\ C}\ (And) \quad \frac{A \vdash_x^{\iota,\eta} e_1 :: B \quad B \vdash_y^{\iota,\eta} e_2 :: C}{A \vdash_y^{\iota,\eta} \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2 :: C}\ (Let)$$

$$\frac{A \vdash_x^{\iota,\eta} e :: B}{A\ ;\ C \vdash_x^{\iota,\eta} e :: B\ ;\ C}\ (Seq) \quad \frac{A \vdash_x^{\iota,\eta} e :: B}{A \mid C \vdash_x^{\iota,\eta} e :: B \mid C}\ (Par) \quad \frac{A \vdash_x^{\iota,\eta} e_1 :: x{:}U \mapsto T \quad B \vdash_y^{\iota,\eta} e_2 :: y{:}U}{A \mid B \vdash_z^{\iota,\eta} e_1 e_2 :: z{:}T}\ (App)$$

$$\frac{A \vdash_x^{\iota,\eta} e :: x{:}U}{A \vdash_x^{\iota,\eta} [\ldots l = e \ldots] :: x{:}l{:}U}\ (Tuple) \quad \frac{A \vdash_x^{\iota,\eta} e :: x{:}l{:}T}{A \vdash_x^{\iota,\eta} e.l :: x{:}T}\ (Sel) \quad \frac{A \mid x{:}U \vdash_z^{\iota,\eta} e :: z{:}T}{A \vdash_z^{\iota,\eta} \lambda x.e :: z{:}U \mapsto T}\ (VAbs)$$

$$\frac{A \vdash_y^{\iota,\eta} e :: (y{:} \oplus_{l \in I} l{:}T_l) \quad x_l{:}T_l \mid B \vdash_z^{\iota,\eta} e_l :: C}{A \mid B \vdash_z^{\iota,\eta} \mathbf{case}\ e\ \mathbf{of}\ \overline{l(\vec{x}) \to e} :: C}\ (Case) \quad \frac{A \vdash_z^{\iota,\eta} e :: z{:}T_m \quad (m \in I)}{A \vdash_z^{\iota,\eta} l_m(e) :: z{:} \oplus_{l \in I} l{:}T_l}\ (Option)$$

$$\frac{A \vdash_y^{\iota,\eta} v :: C \quad B \vdash_y^{\iota,\eta} v :: D}{A\ ;\ B \vdash_y^{\iota,\eta} v :: C\ ;\ D}\ (VSeq) \quad \frac{A \vdash_y^{\iota,\eta} v :: C \quad B \vdash_y^{\iota,\eta} v :: D}{A \mid B \vdash_y^{\iota,\eta} v :: C \mid D}\ (VPar)$$

$$\frac{A <: B \mid R \quad a{:}\mathtt{var} \mid B \vdash_x^{\iota\{R/a\},\eta} e :: C}{A \vdash_x^{\iota,\eta} \mathbf{var}\ a\ \mathbf{in}\ e :: C}\ (Var) \quad \frac{\iota(a) \mid A \vdash_x^{\iota\setminus a,\eta} e :: \iota(a) \mid B}{A \vdash_x^{\iota,\eta} \mathbf{sync}(a)e :: B}\ (Sync) \quad \frac{A \vdash_x^{\iota\setminus a,\eta} e :: \iota(a) \mid B}{A \vdash_x^{\iota,\eta} sy(a)e :: B}\ (Sy)$$

$$\frac{A \vdash_z^{\iota,\eta} v :: z{:}U \mid a{:}\mathtt{use}}{A \vdash_z^{\iota,\eta} a := v :: a{:}\mathtt{rd}(U)}\ (WrVB) \quad \frac{A \vdash_z^{\iota,\eta} v :: z{:}\circ U \mid a{:}\mathtt{wr}(\circ U)}{A \vdash_z^{\iota,\eta} a := v :: \mathsf{0}}\ (WrVF)$$

$$a{:}\mathtt{rd}(U) \vdash_x^{\iota,\eta} a :: x{:}U\ (RdVB) \quad a{:}\mathtt{rd}(U); \mathtt{use} \vdash_x^{\iota,\eta} a :: x{:}U \mid a{:}\mathtt{use}\ (RdVF)$$

$$\frac{A \vdash_x^{\iota,\eta} e :: x{:}T}{A \vdash_x^{\iota,\eta} \mathbf{fork}\ e :: x{:}\tau(T)}\ (Fork) \quad \frac{A \vdash_x^{\iota,\eta} e :: x{:}\tau(T)}{A \vdash_x^{\iota,\eta} \mathbf{wait}\ e :: x{:}T}\ (Wait)$$

$$\frac{\eta(X) = A}{A \vdash_x^{\iota,\eta} v :: X}\ (VRecVar) \quad \frac{A \vdash_x^{\iota,\eta\{X/A\}} v :: B}{A \vdash_x^{\iota,\eta} v :: \mathtt{rec}(X)B}\ (VRec) \quad \frac{\eta(Z) = (A \vdash_x B)}{A \vdash_x^{\iota,\eta} Z :: B}\ (RecVar) \quad \frac{A \vdash_x^{\iota,\eta\{Z/(A \vdash_x B)\}} e :: B}{A \vdash_x^{\iota,\eta} \mathbf{rec}(Z)e :: B}\ (Rec)$$

Figure 8. Typing Rules (Summary).

1.	$next$:$\mathtt{rd}(!\circ PNode) \vdash_x next :: x$:$PNode$	by (RdVB) and (Sub)
2.	$next$:$\mathtt{rd}(!\circ PNode) \vdash_x [\ldots] :: x$:$getNext$:$PNode$	by (Tuple) on 1.
3.	$!next$:$\mathtt{rd}(!\circ PNode) \vdash_x [\ldots] :: x$:$!getNext$:$PNode$	by (Sub) on 2. and (VShr)
4.	$next$:$\mathtt{rd}(!\circ PNode) \vdash_x [\ldots] :: x$:$!getNext$:$PNode$	by (Sub) on 3.
5.	$next$:$\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash_x [\ldots] :: x$:$!getNext$:$PNode$	by (Sub) on 4.
6.	elt:$\mathtt{rd}(\mathtt{nat}) \vdash_x elt :: x$:$\mathtt{nat}$	by (RdVB)
7.	elt:$\mathtt{rd}(\mathtt{nat}) \vdash_x [\ldots] :: x$:$getElt$:$\mathtt{nat}$	by (Tuple) on 6.
8.	$!elt$:$\mathtt{rd}(\mathtt{nat}) \vdash_x [\ldots] :: x$:$!getElt$:$\mathtt{nat}$	by (Sub) on 7. and (VShr)
9.	elt:$\mathtt{rd}(\mathtt{nat}) \vdash_x [\ldots] :: x$:$!getElt$:$\mathtt{nat}$	by (Sub) on 8.
10.	elt:$\mathtt{rd}(\mathtt{nat})\,;\mathtt{var} \vdash_x [\ldots] :: x$:$!getElt$:$\mathtt{nat}$	by (Sub) on 9.
11.	$next$:$\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \mid elt$:$\mathtt{rd}(\mathtt{nat})\,;\mathtt{var} \vdash_x [\ldots] :: x$:$(!getNext$:$PNode \mid !getElt$:$\mathtt{nat})$	by (Par) on 10. and 5.
12.	$\circ(next$:$\mathtt{rd}(!\circ PNode)\,;\mathtt{var}) \mid \circ(elt$:$\mathtt{rd}(\mathtt{nat})\,;\mathtt{var}) \vdash_x [\ldots] :: x$:$INode$	by (Sub) on 11.
13.	$\circ(next$:$\mathtt{rd}(!\circ PNode)\,;\mathtt{var}) \mid \circ(elt$:$\mathtt{rd}(\mathtt{nat})\,;\mathtt{var}) \vdash_x [\ldots] :: x$:$\circ INode$	by (Iso) on 12.
14.	$next$:$\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \mid elt$:$\mathtt{rd}(\mathtt{nat})\,;\mathtt{var} \vdash_x [\ldots] :: x$:$\circ INode$	by (Sub) on 13.
15.	p:$!\circ PNode \mid next$:$\mathtt{wr}(!\circ PNode) \vdash_z p :: z$:$!\circ PNode \mid next$:$\mathtt{wr}(!\circ PNode)$	by (Id) and (Par)
16.	p:$!\circ PNode \mid next$:$\mathtt{wr}(!\circ PNode) \vdash_x next := p :: \mathsf{0}$	by (WrVF) on 15.
17.	$next$:$\mathtt{wr}(!\circ PNode) \vdash_x \lambda p.(next := p) :: x$:$!\circ PNode \mapsto \mathsf{0}$	by (VAbs) on 16.
18.	$next$:$\mathtt{wr}(!\circ PNode) \vdash_x [\ldots] :: x$:$setNext$:$(!\circ PNode \mapsto \mathsf{0})$	by (Tuple) on 17.
19.	e:$\mathtt{nat} \mid elt$:$\mathtt{wr}(\mathtt{nat}) \vdash_z e :: z$:$\mathtt{nat} \mid elt$:$\mathtt{wr}(\mathtt{nat})$	by (Id) and (Par)
20.	e:$\mathtt{nat} \mid elt$:$\mathtt{wr}(\mathtt{nat}) \vdash_x elt := e :: \mathsf{0}$	by (WrVF) on 19.
21.	elt:$\mathtt{wr}(\mathtt{nat}) \vdash_x \lambda e.(elt := e) :: x$:$(\mathtt{nat} \mapsto \mathsf{0})$	by (VAbs) on 20.
22.	elt:$\mathtt{wr}(\mathtt{nat}) \vdash_x [\ldots] :: x$:$setElt$:$(\mathtt{nat} \mapsto \mathsf{0})$	by (Tuple) on 21.
23.	$next$:$\mathtt{wr}(!\circ PNode) \mid elt$:$\mathtt{wr}(\mathtt{nat}) \vdash_x [\ldots] :: x$:$setElt$:$(\mathtt{nat} \mapsto \mathsf{0})\,;\, setNext$:$(!\circ PNode \mapsto \mathsf{0})$	by (VSeq) on 22. and 18., and (Sub)
24.	elt:$\mathtt{var} \mid next$:$\mathtt{var} \vdash_x [\ldots] :: x$:$InitNode\,;\circ INode$	by (VSeq) on 23. and 14., and (Sub)
25.	$\mathsf{0} \vdash_x \mathbf{var}\ next, elt\ \mathbf{in} \ldots :: x$:$Node$	by (Var) on 24.
26.	$\mathsf{0} \vdash_x \lambda[].\mathbf{var} \ldots :: x : (\mathsf{0} \mapsto Node)$	by (VAbs) on 25.
27.	$\mathsf{0} \vdash_x \lambda[].\mathbf{var} \ldots :: x : \circ(\mathsf{0} \mapsto Node)$	by (Iso) on 26.
28.	$\mathsf{0} \vdash_x \lambda[].\mathbf{var} \ldots :: x : !\circ(\mathsf{0} \mapsto Node)$	by (VShr) on 27.

N.B. $\mathtt{nat}$ is a shared isolated type, so $\mathtt{nat} <: !\circ\mathtt{nat}$, so that $\mathtt{nat} <: \mathtt{nat} \mid \mathtt{nat}$ and $\mathtt{nat} <: \circ\mathtt{nat}$.

Figure 9. Typing derivation for $newNode$ function

$_{n1.}$	$hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \vdash^{\emptyset}_{x} [\ldots] :: x{:}(init{:}\mathtt{str} \mapsto \mathsf{0})$	see Figure 13
$_{n3.}$	$id{:}\mathtt{rd}(\mathtt{str}) \vdash^{\eta}_{x} id :: x{:}\mathtt{str}$	by (RdVB)
$_{n4.}$	$id{:}\mathtt{rd}(\mathtt{str}) \vdash^{\eta}_{x} [\ getId = id \ldots\ ] :: x{:}getId{:}\mathtt{str}$	by (Tuple) on n3.
$_{n5.}$	$!id{:}\mathtt{rd}(\mathtt{str}) \vdash^{\eta}_{x} [\ getId = id \ldots\ ] :: x{:}!getId{:}\mathtt{str}$	by (Sub) and (VShr) on n4.
$_{n6.}$	$id{:}\mathtt{rd}(\mathtt{str})\,;\mathtt{var} \vdash^{\eta}_{x} [\ getId = id \ldots\ ] :: x{:}!getId{:}\mathtt{str}$	by (Sub) on n5.
$_{n7.}$	$\mathsf{0} \vdash^{\eta}_{x} [\ldots] :: \mathsf{0}$	by (VStop)
$_{n8.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\eta}_{x} [\ldots] :: \mathsf{0}$	by (Sub) on n7.
$_{n9.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\eta}_{x} [\ldots] :: X$	by (VRecVar) since $\eta(X) = hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var}$
$_{n10.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{wr}(!\circ PNode) \vdash^{\eta}_{x} [\ldots] :: x{:}add{:}\mathtt{nat}{\mapsto}0$	see Figure 11
$_{n11.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\eta}_{x} [\ldots] :: x{:}add{:}\mathtt{nat}{\mapsto}0; X$	by (VSeq) on n10. and n9. and (Sub)
$_{n12.}$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\eta}_{x} [scan = \ldots] :: x{:}scan{:}\mathsf{0}$	see Figure 12
$_{n13.}$	$!hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\eta}_{x} [scan = \ldots] :: x{:}!scan{:}\mathsf{0}$	by (Sub) and (VShr) on n12.
$_{n14.}$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\eta}_{x} [scan = \ldots] :: x{:}!scan{:}\mathsf{0}$	by (Sub) on n13.
$_{n15.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\eta}_{x} [\ldots] :: x{:}!scan{:}\mathsf{0}\,;(add{:}\mathtt{nat}{\mapsto}0; X)$	by (VSeq) on n14., n11. and (Sub)
$_{n16.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\eta}_{x} [\ldots] :: x{:}\mathsf{0}\&(!scan{:}\mathsf{0}\,;add{:}\mathtt{nat}{\mapsto}0); X$ with $\eta = \{X/newNode : \ldots \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var}\}$	by (And) on n8. and n15.
$_{n17.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \vdash^{\emptyset}_{x} [\ldots] :: x{:}(\mathtt{rec}(X)(\mathsf{0}\&(!scan{:}\mathsf{0}\,;add{:}\mathtt{nat}{\mapsto}0); X)$	by (VRec) on n16.
$_{n18.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\mathtt{var} \mid id{:}\mathtt{rd}(\mathtt{str})\,;\mathtt{var} \vdash^{\emptyset}_{x} [\ldots] :: x{:}(!getId{:}\mathtt{str} \mid (!scan{:}\mathsf{0}\,;add{:}\mathtt{nat}{\mapsto}0)^*)$	by (VPar) on n6. and n17.
$_{n19.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{var} \mid id{:}\mathtt{var} \vdash^{\emptyset}_{x} [\ldots] :: x{:}(init{:}\mathtt{str} \mapsto \mathsf{0}\,;(\ldots))$	by (VSeq) on n1. and n18.
$_{n20.}$	$newNode{:}\circ!\circ(\mathsf{0} \mapsto Node) \mid \circ(hd{:}\mathtt{var}) \mid \circ(id{:}\mathtt{var}) \vdash^{\emptyset}_{x} [\ldots] :: x{:}\circ CC$	by (Sub) and (Iso) on n19.
$_{n21.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{var} \mid id{:}\mathtt{var} \vdash^{\emptyset}_{x} [\ldots] :: x{:}\circ CC$	by (Sub) on n20.
$_{n22.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \vdash^{\emptyset}_{x} \mathbf{var}\ hd, id\ \mathbf{in} \ldots :: x{:}\circ CC$	by (Sub) on n21.
$_{n23.}$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \vdash^{\emptyset}_{x} \lambda[].\mathbf{var} \ldots :: x{:}\mathsf{0} \mapsto \circ CC$	by (VAbs) on n22.

Figure 10. Typing derivation for function $newColl$

$a1.$	$\mathsf{0} \vdash_x \textbf{nil} :: x : \mathsf{0}$	by (VStop)
$a2.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \vdash_x newNode :: x{:}\mathsf{0} \mapsto Node$	by (Id) and (Sub)
$a3.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \vdash_x newNode\ \textbf{nil} :: x{:}Node$	by (App) on a1. and a2.
$a4.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\ldots \mid e{:}\mathtt{nat} \vdash_n newNode\ \textbf{nil} :: n{:}Node \mid hd{:}\ldots \mid e{:}\mathtt{nat}$	by (Par) on a3.
$a5.$	$n{:}setElt{:}(\mathtt{nat} \mapsto \mathsf{0}) \vdash_x n :: x{:}setElt{:}(\mathtt{nat} \mapsto \mathsf{0})$	by (Id)
$a6.$	$n{:}setElt{:}(\mathtt{nat} \mapsto \mathsf{0}) \vdash_x n.setElt :: x{:}(\mathtt{nat} \mapsto \mathsf{0})$	by (Sel) on a5
$a7.$	$e{:}\mathtt{nat} \vdash_x e :: x{:}\mathtt{nat}$	by (Id)
$a8.$	$n{:}setElt{:}(\mathtt{nat} \mapsto \mathsf{0}) \mid e{:}\mathtt{nat} \vdash_x n.setElt\ e :: \mathsf{0}$	by (App) on a6. and a7.
$a9.$	$n{:}setElt{:}(\mathtt{nat} \mapsto \mathsf{0})\,;\, setNext{:}(!\circ PNode \mapsto \mathsf{0})\,;\, !\circ INode \mid e{:}\mathtt{nat} \mid hd{:}\ldots \vdash_x (n.setElt\ e) :: n{:}setNext{:}(!\circ PNode \mapsto \mathsf{0})\,;\, !\circ INode \mid hd{:}\ldots$	by (Seq) and (Par) on a8.
$a10.$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash_x hd :: x{:}!\circ PNode$	by (RdVB)
$a11.$	$n{:}setNext{:}(!\circ PNode \mapsto \mathsf{0}) \vdash_x n :: x{:}setNext{:}(!\circ PNode \mapsto \mathsf{0})$	by (Id)
$a12.$	$n{:}setNext{:}(!\circ PNode \mapsto \mathsf{0}) \vdash_x n.setNext :: x{:}(!\circ PNode \mapsto \mathsf{0})$	by (Sel) on a11.
$a13.$	$n{:}setNext{:}(!\circ PNode \mapsto \mathsf{0}) \mid hd{:}\mathtt{rd}(!\circ PNode) \vdash_x n.setNext\ hd :: \mathsf{0}$	by (App) on a10. and a12.
$a14.$	$n{:}setNext{:}(!\circ PNode \mapsto \mathsf{0})\,;\, !\circ INode \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \vdash_x (n.setNext\ hd) :: n{:}!\circ INode \mid hd{:}\mathtt{wr}(!\circ PNode)$	by (Seq), (Par), and (Sub) on a13.
$a15.$	$n{:}Node \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \mid e{:}\mathtt{nat} \vdash_x (n.setElt\ e)\,;(n.setNext\ hd) :: n{:}!\circ INode \mid hd{:}\mathtt{wr}(!\circ PNode)$	by (Let) on a9. and a14.
$a16.$	$n{:}INode \vdash_y n :: y{:}INode$	by (Id)
$a17.$	$n{:}!\circ INode \vdash_y \mathtt{NODE}(n) :: y{:}!\circ PNode$	by (Option), (Sub), (Iso), and (VShr) on a16.
$a18.$	$n{:}!\circ INode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_y \mathtt{NODE}(n) :: y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode)$	by (Par) on a17.
$a19.$	$y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_z y :: z{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode)$	by (Id) and (Par)
$a20.$	$y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_x hd{:=}y :: \mathsf{0}$	by (WrVF) on a19.
$a21.$	$n{:}!\circ INode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_x \textbf{let}\ y = \mathtt{NODE}(n)\ \textbf{in}\ hd{:=}y :: \mathsf{0}$	by (Let) on a18. and a20.
$a22.$	$n{:}Node \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \mid e{:}\mathtt{nat} \vdash_x ((n.setElt\ e)\,;(n.setNext\ hd))\,;(hd{:=}\mathtt{NODE}(n)) :: \mathsf{0}$	by (Let) on a15.and a21.
$a23.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \mid e{:}\mathtt{nat} \vdash_x \textbf{let}\ n = (newNode\ \textbf{nil})\ \textbf{in}\ \ldots :: \mathsf{0}$	by (Let) on a4. and a22.
$a24.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \vdash_x \lambda e.\textbf{let}\ n = \ldots :: x{:}\mathtt{nat} \mapsto \mathsf{0}$	by (Abs) on a23.
$a25.$	$newNode{:}!\circ(\mathsf{0} \mapsto Node) \mid hd{:}\mathtt{rd}(!\circ PNode)\,;\, \mathtt{wr}(!\circ PNode) \vdash_x [\ldots] :: x{:}add{:}\mathtt{nat} \mapsto \mathsf{0}$	by (Tuple) on a24.

Figure 11. Typing derivation for "method" add

$s1.$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\emptyset}_{y} hd :: y{:}!\circ PNode$	by (RdVB)
$s2.$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\emptyset}_{y} hd :: y{:}PNode$	by (Sub) on s1.
$s3.$	$hd{:}\mathtt{rd}(!\circ PNode) \mid s : \mathtt{var} \vdash^{\emptyset}_{y} hd :: y{:}PNode \mid s : \mathtt{var}$	by (Par) on s2.
$s4.$	$y{:}PNode \mid s : \mathtt{use} \vdash^{\emptyset}_{z} y :: z{:}PNode \mid s{:}\mathtt{use}$	by (Id) and (Par)
$s5.$	$y{:}PNode \mid s : \mathtt{use} \vdash^{\emptyset}_{x} s := y :: s{:}\mathtt{rd}(PNode)$	by (WrVB) on s4.
$s6.$	$y{:}PNode \mid s : \mathtt{var} \vdash^{\emptyset}_{x} s := y :: s{:}\mathtt{rd}(PNode)\,;\mathtt{var}$	by (Seq) and (Sub) on s5.
$s7.$	$hd{:}\mathtt{rd}(!\circ PNode) \mid s : \mathtt{var} \vdash^{\emptyset}_{x} \mathsf{let}\ y = hd\ \mathsf{in}\ s := y :: s{:}\mathtt{rd}(PNode)\,;\mathtt{var}$	by (Let) on s3. and s6.
$s8.$	$s{:}\mathtt{use} \vdash^{\eta}_{x} \mathsf{nil} :: s{:}\mathtt{use}$	by (VId)
$s9a.$	$n{:}getNext{:}PNode \vdash^{\eta}_{y} n :: y{:}getNext{:}PNode$	by (Id)
$s9b.$	$n{:}getNext{:}PNode \vdash^{\eta}_{y} n.getNext :: y{:}PNode$	by (Sel) and (Sub)
$s9.$	$n{:}INode \vdash^{\eta}_{y} n.getNext :: y{:}PNode$	by (Sub)
$s10.$	$n{:}INode \mid s{:}\mathtt{use} \vdash^{\eta}_{y} n.getNext :: y{:}PNode \mid s{:}\mathtt{use}$	by (Par) on s9.
$s11.$	$y{:}PNode \mid s{:}\mathtt{use} \vdash^{\eta}_{x} y :: x{:}PNode \mid s{:}\mathtt{use}$	by (Id) and (Par)
$s12.$	$y{:}PNode \mid s{:}\mathtt{use} \vdash^{\eta}_{x} s := y :: s{:}\mathtt{rd}(PNode)\,;\mathtt{use}$	by (WrVB) on s11. and (Seq) and (Sub)
$s13.$	$n{:}INode \mid s{:}\mathtt{use} \vdash^{\eta}_{x} \mathsf{let}\ y = n.getNext\ \mathsf{in}\ s := y :: s{:}\mathtt{rd}(PNode)\,;\mathtt{use}$	by (Let) on s10. and s12.
$s14.$	$s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash^{\eta}_{x} L :: s{:}\mathtt{use}$	by (RecVar) since $\eta(L) = s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash_x s{:}\mathtt{use}$
$s15.$	$n{:}INode \mid s{:}\mathtt{use} \vdash^{\eta}_{x} \mathsf{let}\ \_ = \ s := n.getNext\ \mathsf{in}\ L :: s{:}\mathtt{use}$	by (Let) on s13. and s14.
$s16.$	$y{:}PNode \mid s{:}\mathtt{use} \vdash^{\eta}_{x} \mathsf{case}\ y\ \mathsf{of}\ \ldots :: s{:}\mathtt{use}$	by (Case) on s8. and s15. and (Sub)
$s17.$	$s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash^{\eta}_{y} s :: y{:}PNode \mid s{:}\mathtt{use}$	by (RdVF)
$s18.$	$s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash^{\eta}_{x} \mathsf{let}\ y = s\ \mathsf{in}\ \mathsf{case}\ y\ \mathsf{of}\ \ldots :: s{:}\mathtt{use}$	by (Let) on s17. and s16. with $\eta = \{L/(s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash_x s{:}\mathtt{use})\}$
$s19.$	$s{:}\mathtt{rd}(PNode)\,;\mathtt{use} \vdash^{\emptyset}_{x} \mathsf{rec}\ L.\mathsf{case}\ \ldots :: s{:}\mathtt{use}$	by (Rec) on s18.
$s20.$	$s{:}\mathtt{rd}(PNode)\,;\mathtt{var} \vdash^{\emptyset}_{x} \mathsf{rec}\ L.\mathsf{case}\ \ldots :: s{:}\mathtt{var}$	by (VSeq) on s19. and (Sub)
$s21.$	$hd{:}\mathtt{rd}(!\circ PNode) \mid s : \mathtt{var} \vdash^{\emptyset}_{x} \mathsf{let}\ \_ = s := hd\ \mathsf{in}\ \ldots :: \mathsf{0}$	by (Let) on s7. and s20. and (Sub)
$s22.$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\emptyset}_{x} \mathsf{var}\ s\ \mathsf{in} \ldots :: \mathsf{0}$	by (Var) on s21.
$s23.$	$hd{:}\mathtt{rd}(!\circ PNode) \vdash^{\emptyset}_{x} [scan = \ldots] :: x{:}scan{:}\mathsf{0}$	by (Tuple) on s22.

Figure 12. Typing derivation for "method" *scan*

$i1.$	$\mathsf{0} \vdash_y \mathsf{nil} :: \mathsf{0}$	by (VStop)
$i2.$	$\mathsf{0} \vdash_y \mathtt{NULL}(\mathsf{nil}) :: y{:}!\mathtt{Opt}(INode)$	by (Option) on i1. and (VShr)
$i3.$	$\mathsf{0} \vdash_y \mathtt{NULL}(\mathsf{nil}) :: y{:}\circ PNode$	by (Iso) on i2.
$i4.$	$\mathsf{0} \vdash_y \mathtt{NULL}(\mathsf{nil}) :: y{:}!\circ PNode$	by (VShr) on i3.
$i5.$	$hd{:}\mathtt{wr}(\ldots) \mid id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_y \mathtt{NULL} :: y{:}!\circ PNode \mid hd{:}\mathtt{wr}(\ldots) \mid id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str}$	by (Par) on i4.
$i6.$	$y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_x y :: x{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode)$	by (Id) and (Par)
$i7.$	$y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode) \vdash_x hd := y :: \mathsf{0}$	by (WrVF) on i6.
$i8.$	$y{:}!\circ PNode \mid hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_x hd := y :: id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str}$	by (Par) on i7.
$i9.$	$hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_x \mathsf{let}\ y = \mathtt{NULL}\ \mathsf{in}\ hd := y :: id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str}$	by (Let) on i5. and i8.
$i9a.$	$id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_x i :: id{:}\mathtt{wr}(\mathtt{str}) \mid x{:}\mathtt{str}$	by (Id) and (Par)
$i10.$	$id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_x id := i :: \mathsf{0}$	by (WrVF) on i9a.
$i11.$	$hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \mid i{:}\mathtt{str} \vdash_x (hd := \mathtt{NULL}); (id := i) :: \mathsf{0}$	by (Let) on i9. and i10.
$i12.$	$hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \vdash_x \lambda i.(hd := \mathtt{NULL}; id := i) :: x{:}\mathtt{str} \mapsto \mathsf{0}$	by (Abs) on i11.
$i13.$	$hd{:}\mathtt{wr}(!\circ PNode) \mid id{:}\mathtt{wr}(\mathtt{str}) \vdash_x [\ldots] :: x{:}(init{:}\mathtt{str} \mapsto \mathsf{0})$	by (Tuple) on i12.

Figure 13. Typing derivation for "method" *init*

$t1.$	$v{:}\mathtt{nat} \mid s{:}\mathtt{use} \vdash_y v :: y{:}\mathtt{nat} \mid s{:}\mathtt{use}$	by (Id) and (Par)
$t1a.$	$v{:}\mathtt{nat} \mid s{:}\mathtt{use} \vdash_y s := v :: s{:}\mathtt{rd}(\mathtt{nat})$	by (WrVB) on t1.
$t2.$	$(v{:}\mathtt{nat} \mid s{:}\mathtt{use}); \mathtt{var} \vdash_y s := v :: s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}$	by (Seq) on t1a.
$t3.$	$v{:}\mathtt{nat} \mid s{:}\mathtt{var} \vdash_y s := v :: s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}$	by (Sub) on t2.
$t3a.$	$x{:}\mathtt{nat} \mid s{:}\mathtt{use} \vdash_y x :: y{:}\mathtt{nat} \mid s{:}\mathtt{use}$	by (Id) and (Par)
$t4.$	$x{:}\mathtt{nat} \mid s{:}\mathtt{use} \vdash_y s := x :: s{:}\mathtt{rd}(\mathtt{nat})$	by (WrVB) on t3a.
$t5.$	$(x{:}\mathtt{nat} \mid s{:}\mathtt{use}) \,;\, \mathtt{var} \vdash_y s := x :: s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}$	by (Seq) on t4.
$t6.$	$x{:}\mathtt{nat} \mid s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var} \vdash_y s := x :: s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}$	by (Sub) on t5.
$t7.$	$x{:}\mathtt{nat} \vdash_y \mathbf{sync}(lock)(s := x) :: \mathtt{0}$	by (Sub) on t6.
$t8.$	$\mathtt{0} \vdash_y \lambda x.\mathbf{sync}(lock)(s := x) :: y : \mathtt{nat} \mapsto \mathtt{0}$	by (Sub) on t7.
$t9.$	$\mathtt{0} \vdash_x [set = ...] :: set{:}(\mathtt{nat} \mapsto \mathtt{0})$	by (Tuple) on t8.
$t10.$	$\mathtt{0} \vdash_x [set = ...] :: !set : (\mathtt{nat} \mapsto \mathtt{0})$	by (VShr) on t9.
$t11.$	$s{:}\mathtt{rd}(\mathtt{nat}) \vdash_x s :: x{:}\mathtt{nat}$	by (RdVb)
$t12.$	$s{:}\mathtt{rd}(\mathtt{nat}) \mid s{:}\mathtt{rd}(\mathtt{nat}) \vdash_x s :: x{:}\mathtt{nat} \mid s{:}\mathtt{rd}(\mathtt{nat})$	by (Par) on t11.
$t13.$	$s{:}\mathtt{rd}(\mathtt{nat}) \vdash_x s :: x{:}\mathtt{nat} \mid s{:}\mathtt{rd}(\mathtt{nat})$	by (Sub) on t12.
$t14.$	$s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var} \vdash_x s :: x{:}\mathtt{nat} \mid s{:}\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}$	by (Seq) and (Sub) on t13.
$t15.$	$\mathtt{0} \vdash_x \mathbf{sync}(lock)(s) :: x : \mathtt{nat}$	by (Sync) on t14.
$t16.$	$\mathtt{0} \vdash_x [set = \ldots] :: x{:}get{:}\mathtt{nat}$	by (Tuple) on t15.
$t17.$	$\mathtt{0} \vdash_x [set = \ldots] :: x{:}!get{:}\mathtt{nat}$	by (VShr) on t16.
$t18.$	$\mathtt{0} \vdash_x [set = \ldots] :: x{:}(!set{:}(\mathtt{nat} \mapsto \mathtt{0}) \mid !get{:}\mathtt{nat})$	by (VPar) on t10,t17.
$t19.$	$s : \mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var} \vdash_x \mathbf{var}\ lock\ \mathbf{in}\ \ldots :: !set : (\mathtt{nat} \mapsto \mathtt{0}) \mid !get : \mathtt{nat}$	by (Var) on t18 with invariant $(s{:}(\mathtt{rd}(\mathtt{nat}) \,;\, \mathtt{var}))$
$t20.$	$v{:}\mathtt{nat} \mid s{:}\mathtt{var} \vdash_x s := v \,;\, \mathbf{var}\ lock\ \mathbf{in} \ldots :: !set : (\mathtt{nat} \mapsto \mathtt{0}) \mid !get{:}\mathtt{nat}$	by (Let) on t3,t19.
$t21.$	$v{:}\mathtt{nat} \vdash_x \mathbf{var}\ s\ \mathbf{in} \ldots :: !set : (\mathtt{nat} \mapsto \mathtt{0}) \mid !get : \mathtt{nat}$	by (Var) on t20.
$t22.$	$\mathtt{0} \vdash_x \lambda v.\mathbf{var}\ s\ \mathbf{in} \ldots :: \mathtt{nat} \mapsto (!set : (\mathtt{nat} \mapsto \mathtt{0}) \mid !get : \mathtt{nat})$	by (Abs) on t21.

Figure 14. Typing derivation for function *atomic*