Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 276 (2011) 121–143

www.elsevier.com/locate/entcs

A Step-Indexed Kripke Model of Separation Logic for Storable Locks

Alexandre Buisse[1] Lars Birkedal[2]

IT University of Copenhagen, Denmark

Kristian Støvring[3]

DIKU, University of Copenhagen, Denmark

Abstract

We present a version of separation logic for modular reasoning about concurrent programs with dynamically allocated storable locks and dynamic thread creation. The assertions of the program logic are modelled by a Kripke model over a recursively defined set of worlds and the program logic is proved sound through a Kripke relation to the standard operational semantics. This constitutes an elegant solution to the circularity issue arising from lock resource invariants depending on worlds containing lock resource invariants.

Keywords: Semantics, Concurrency, Separation Logic, Step-Indexing

1 Introduction

We present a version of separation program logic for modular reasoning about shared-variable concurrent imperative programs with dynamically allocated storable locks and dynamic thread creation. Following earlier work [8,9,11], the idea of the logic is that a lock can be used to protect a resource invariant which, conceptually speaking, is transferred from one thread to another. The resource invariant follows the movements of the lock itself, changing ownership through acquire and release operations. Since the resource invariants associated with locks describe properties of heaps while locks are themselves (dynamically allocated and) stored in the heap, there is a tricky form of circularity that needs to be resolved in order to develop a model to show soundness of the logic. More specifically, since locks are dynamically

[1] abui@itu.dk

[2] birkedal@itu.dk

[3] stovring@diku.dk

doi:10.1016/j.entcs.2011.09.018

allocated, it is natural to use a Kripke model of assertions, such that the semantics of an assertion is a world-indexed predicate on the set of program heaps, and the set of worlds describes the predicates that the allocated locks protect. In summary, we end up with recursive semantics equations of roughly the following form:

$$\begin{aligned} Pred &= W \to \mathcal{P}(H) \\ W &= Loc \rightharpoonup^{fin} Pred \end{aligned}$$

In earlier work, Gotsman et al. [8] evaded this circularity by restricting the logic and using a static number of classes of locks, so that the circularity could be broken using an indirection through a syntactic name for the class of the lock, and by instrumenting the operational semantics to include such syntactic names of classes of locks in the program heap. In [10,9], Hobor et al. addressed the circularity using step-indexing and instrumenting the operational semantics. The instrumentation of the operational semantics means in both cases that some additional work is required to relate the semantics to the standard operational semantics, as was recently done for the setup used by Hobor et al. [12].

Here instead we (1) solve (a version of) the above recursive equation in a category of complete bounded ultrametric spaces, following ideas that we have used earlier to model higher-order programming languages with dynamically allocated reference cells [5,4]. This gives a real semantic status to predicates, allowing for instance recursive definitions via a fixpoint operator. Moreover (2), we prove the soundness of the logic by giving a Kripke relation to the standard (uninstrumented) operational semantics. This approach to proving soundness is related to logical relation proofs for soundness of models of type systems (e.g., [4]) and was proposed for concurrency by Vafeiadis [13,7]. Here we show how to extend and apply the technique in a richer language with dynamic thread creation and storable locks.

To sum up, we believe our new approach to the semantics and soundness is both semantically simple and useful since it solves the circularity issues directly and soundness is proved w.r.t. standard operational semantics. We demonstrate our approach on a simple but illustrative language and also include an example of using the resulting logic.

2 The language

2.1 Definition

We will use a small, low-level imperative language supporting minimal heap manipulation commands, simple lock operations and concurrency via dynamic thread creation and synchronisation. Its syntax is defined as follows:

Var	$::= l, x, y, \ldots$	Variables
E, F	$::= nil \mid Var \mid E + F \mid \ldots$	Expressions
B	$::= E = F \mid E \neq F$	Boolean Expressions
C	::=`let` x `= new in` C | `let` x `=` E `in` C | `let` x `=` $[E]$ `in` C | $[E] = F$ | `if` B `then` C `else` C' | `while` $[E]$ `do` C | `skip` | C`;`C' | `makelock_P(`E`)` | `acquire(`E`)` | `release(`E`)` | `let` x `= fork(f) in` C | `join(`E`)`	Commands

Expressions include integer constants and arithmetic operators, which we will not detail here.

A program consists of a static definition of sub-procedures, `f0` to `fn`, followed by a main command, `C`:

```
let f0 = C0 and f1 = C1 and ...  and fn = Cn in C
```

Here the main command `C` can create new instances of the sub-procedures via calls to `fork`.

We have made several choices in the syntax, trying to simplify the language whenever features were orthogonal to the problems we are interested in. In particular, we use immutable stack variables via continuation passing style commands (`let` x `= new in` C and `let` x `=` E `in` C for allocation and lookup, respectively), following [6]. Any command of the form `let` x `= ... in` C uses a fresh variable x, obtained if necessary via alpha-conversion.

Also, in order to keep things relatively simple, the language does not support any freeing of resources, be it regular memory cells or locks. The latter permits us to omit the entire system of fractional permissions found in previous works (e.g. [8] and [9]).

2.2 Operational semantics

We will use a simple non-deterministic operational semantics for the language, close to what would actually be executed by a real machine. Our abstract locks behave similarly to spinlocks in that an `acquire` instruction on a lock that is not available will simply "hang" and will be allowed to try to acquire the lock again after another thread took an execution step.

A state in our semantics is a triple (h, s, tp) where h is a heap in $H = Loc \to_{fin} Val$, s a stack in $Var \to Val$ and tp a thread pool mapping each thread identifier to the code it should execute, i.e. $tp : \{1, \ldots, n\} \to Cmd$ for some n.

It is important to note that this operational semantics is very low-level and doesn't deal with some of the more "semantic" features. In particular, neither the heap nor the stack are split between threads and there is no notion of private or shared space. Similarly, locks are implemented in a very simple fashion: as an integer which contains 0 if the lock is available, and k if it is currently owned by the thread with identifier k. Since there is no notion of world, nothing really distinguishes a lock or a thread handle from a regular memory cell, and an incorrect program could easily overwrite lock values.

Definition 2.1 The general form of a reduction in our operational semantics is $(h, s, t) \longrightarrow_j (h', s', t')$. However, to improve readability, we will write $(\texttt{c}, h, s, t) \longrightarrow_j (\texttt{c'}, h', s', t')$ as a shorthand notation for "$(h, s, t) \longrightarrow_j (h', s', t')$ and $t(j) = \texttt{c}$ and $t'(j) = \texttt{c'}$".

Furthermore, a state (h, s, t) can also be *stuck* with respect to thread j, written $(h, s, t) \not\longrightarrow_j$, or lead to an execution fault, written $(h, s, t) \longrightarrow_j \texttt{abort}$. In order to save space, we do not list all the conditions for aborting states: rather, we define an aborting state to be one that is neither stuck nor reduces to another state.

$$
\begin{aligned}
(\texttt{let x = new in c}, h, s, t) &\longrightarrow_j (c, h[v \mapsto 0], s[x \mapsto v], t[j \mapsto \texttt{c}]) \\
&\qquad \text{where } v \notin dom(h) \\
(\texttt{let x = [E] in c}, h, s, t) &\longrightarrow_j (c, h, s[x \mapsto v], t[j \mapsto \texttt{c}]), \quad \text{where } h(\llbracket E \rrbracket_s) = v \\
(\texttt{let x = E in c}, h, s, t) &\longrightarrow_j (c, h, s[x \mapsto \llbracket E \rrbracket_s], t[j \mapsto \texttt{c}]) \\
(\texttt{[E] = F}, h, s, t) &\longrightarrow_j (\texttt{skip}, h[\llbracket E \rrbracket_s \mapsto \llbracket F \rrbracket_s], s, t[j \mapsto \texttt{skip}]) \\
(\texttt{if E then c else c'}, h, s, t) &\longrightarrow_j (\texttt{c}, h, s, t[j \mapsto \texttt{c}]), \quad \text{if } \llbracket E \rrbracket_s = True \\
(\texttt{if E then c else c'}, h, s, t) &\longrightarrow_j (\texttt{c'}, h, s, t[j \mapsto \texttt{c'}]), \quad \text{if } \llbracket E \rrbracket_s = False \\
(\texttt{while [E] do c}, h, s, t) &\longrightarrow_j (\texttt{skip}, h, s, t[j \mapsto \texttt{skip}]), \quad \text{if } h(\llbracket E \rrbracket_s) = False \\
(\texttt{while [E] do c}, h, s, t) &\longrightarrow_j (\texttt{c;while [E] do c}, h, s, t[j \mapsto \texttt{c;while [E] do c}]) \\
&\qquad \text{if } h(\llbracket E \rrbracket_s) = True \\
(\texttt{skip;c}, h, s, t) &\longrightarrow_j (\texttt{c}, h, s, t[j \mapsto \texttt{c}]) \\
(\texttt{c;c'}, h, s, t) &\longrightarrow_j (\texttt{c'';c'}, h', s', t'[j \mapsto \texttt{c'';c'}]), \\
&\qquad \text{if } (\texttt{c}, h, s, t) \longrightarrow_j (\texttt{c''}, h', s', t') \\
(\texttt{skip}, h, s, t) &\not\longrightarrow_j \\
(\texttt{make\_lock\_P(E)}, h, s, t) &\longrightarrow_j (\texttt{skip}, h[\llbracket E \rrbracket_s \mapsto j], s, t[j \mapsto \texttt{skip}]), \quad \text{if } h(\llbracket E \rrbracket_s) = \_ \\
(\texttt{acquire(E)}, h, s, t) &\longrightarrow_j (\texttt{skip}, h[\llbracket E \rrbracket_s \mapsto j], s, t[j \mapsto \texttt{skip}]), \quad \text{if } h(\llbracket E \rrbracket_s) = 0 \\
(\texttt{acquire(E)}, h, s, t) &\not\longrightarrow_j \quad \text{if } h(\llbracket E \rrbracket_s) > 0 \\
(\texttt{release(E)}, h, s, t) &\longrightarrow_j (\texttt{skip}, h[\llbracket E \rrbracket_s \mapsto 0], s, t[j \mapsto \texttt{skip}]), \quad \text{if } h(\llbracket E \rrbracket_s) = j \\
(\texttt{let x = fork(f) in c}, h, s, t) &\longrightarrow_j (c, h[v \mapsto k], s[x \mapsto v], t[k \mapsto \texttt{c}]), \\
&\qquad \text{if } v \notin dom(h), k \notin dom(t) \text{ and } f : \texttt{c} \in \Gamma \\
(\texttt{join(E)}, h, s, t) &\longrightarrow_j (\texttt{skip}, h|_{dom(h) - \llbracket E \rrbracket_s}, s, t[j \mapsto \texttt{skip}]|_{dom(t) - \{k\}}), \\
&\qquad \text{if } h(\llbracket E \rrbracket_s) = k \text{ and } t(k) = \texttt{skip} \\
(\texttt{join(E)}, h, s, t) &\not\longrightarrow_j \\
&\qquad \text{if } h(\llbracket E \rrbracket_s) = k \text{ and } t(k) \neq \texttt{skip} \\
(h, s, t) &\longrightarrow_j \texttt{abort} \quad \text{in all other cases}
\end{aligned}
$$

2.3 Assertion language

Our assertion language is built upon separation logic, with extra assertions dealing with locks and dynamic thread creation and synchronisation. In addition, the intrinsic circularity of storable locks is expressed via a fixpoint operator μ.

$$\begin{aligned}
PredVar &::= p, q, r, \ldots \\
P &::= E_1 = E_2 \mid E_1 \mapsto E_2 \mid P \wedge P \mid P \vee P \mid P * P \mid emp \mid \\
&\quad\; Locked(E,P) \mid Ex(E,P) \mid tid(f,E) \mid \\
&\quad\; r\,(E) \mid (\mu r.\lambda x.P)(E) \; {\scriptstyle (*)} \\
Val &= \; Loc \uplus \mathbb{Z} \uplus (Val \times Val)
\end{aligned}$$

Notice that we do not include separating implication. This is a limitation of our current model; see Proposition 4.6 below.

In order to express lock properties, we use two predicates: $Ex(E,P)$ and $Locked(E,P)$. The first affirms that the interpretation of expression E points to a lock in the heap, with resource invariant P. No hypothesis is made as to whether the lock is available or not. $Locked(E,P)$ is similar, with the additional affirmation that the current thread owns the lock.

It is necessary for a thread to know about the existence of a lock before it can try to acquire it, but not to *know* that the lock is available. Similarly, the postcondition of a lock release simply says that the lock still exists, but not that it is available, in order to protect against interferences from other threads, which can now successfully acquire the lock.

The side condition (*) stipulates that all occurrences of the predicate variable r in P are *guarded*, i.e. that they only appear under a *Locked* or *Ex* predicate. More formally, using the notation that $P[\,]$ is a formula with a hole which can be filled by a predicate Q, written $P[Q]$:

Definition 2.2 A predicate variable r is guarded in P if $\exists P_1, E, Q$ such that $P = P_1[Ex(E,Q)]$ or $P = P_1[Locked(E,Q)]$ and $r \notin fv(P_1)$.

Additionally, we also have the usual intuitionistic logic rules and some additional rules for the assertion logic, such as

$$Locked(E,P) \Rightarrow Locked(E,P) * Ex(E,P)$$
$$(\mu r.\lambda x.P)(E) \Leftrightarrow P[(\mu r.\lambda x.P)/r, E/x]$$

2.4 Selected proof rules

We now present some of the proof rules for the specification logic, choosing to omit most of the structural rules. We need the following definitions: a predicate P is *thread-independent* if it does not contain any sub-predicates of the form $Locked(E,P')$. Intuitively, the meaning of such a predicate will be the same for all threads in a given configuration, whereas, e.g., a $Locked(E,P')$ predicate is true for at most one of the threads. A predicate is *mobile* if it is thread-indepedent and precise.

$$\frac{\{P * x \mapsto 0\}\texttt{c}\{Q\}}{\{P\}\texttt{let x = new in c}\{Q\}}\ x \notin fv(P) \cup fv(Q) \qquad \frac{\{P * E \mapsto x\}\texttt{c}\{Q\}}{\{\exists x, P * E \mapsto x\}\texttt{let x = [E] in c}\{Q\}}\ x \notin fv(Q)$$

$$\frac{\{P \wedge x = E\}\texttt{c}\{Q\}}{\{P\}\texttt{let x = E in c}\{Q\}}\ x \notin fv(Q) \qquad \frac{}{\{[E \mapsto \_]\}\texttt{[E] = F}\{[E \mapsto F]\}}$$

$$\frac{\{E = \mathit{true} \wedge P\}\texttt{c}\{Q\} \qquad \{E = \mathit{false} \wedge P\}\texttt{c'}\{Q\}}{\{P\}\texttt{if E then c else c'}\{Q\}} \qquad \frac{\{P \wedge E \mapsto \mathit{true}\}\texttt{c}\{P \wedge E \mapsto \_\}}{\{P \wedge E \mapsto \_\}\texttt{while [E] do c}\{P \wedge E \mapsto \mathit{false}\}}$$

$$\frac{}{\{P\}\texttt{skip}\{P\}} \qquad \frac{\{P\}\texttt{c}\{Q\} \qquad \{Q\}\texttt{c'}\{R\}}{\{P\}\texttt{c;c'}\{R\}} \qquad \frac{}{\{E \mapsto \_\}\texttt{makelock\_P(E)}\{\mathit{Locked}(E,P)\}}\ (*)$$

$$\frac{}{\{\mathit{Ex}(E,P)\}\texttt{acquire(E)}\{\mathit{Locked}(E,P) * P\}}\ (*) \qquad \frac{}{\{\mathit{Locked}(E,P) * P\}\texttt{release(E)}\{\mathit{Ex}(E,P)\}}\ (*)$$

$$\frac{\Gamma, \{R\}\texttt{f}\{S\} \vdash \{P * \mathit{tid}(f,x)\}\texttt{c}\{Q\}}{\Gamma, \{R\}\texttt{f}\{S\} \vdash \{P * R\}\texttt{let x = fork(f) in c}\{Q\}}\ (*) \qquad \frac{}{\Gamma, \{R\}\texttt{f}\{S\} \vdash \{\mathit{tid}(f,E)\}\texttt{join(E)}\{S\}}\ (*)$$

$$\frac{f : \{R\}\texttt{c'}\{S\} \in \Gamma \qquad \Gamma \vdash \{R\}\texttt{c'}\{S\} \qquad \Gamma \vdash \{P\}\texttt{c}\{Q\}}{\vdash \texttt{let } f : \{R\}\texttt{c'}\{S\} \texttt{ in } \{P\}\texttt{c}\{Q\}}\ (*)$$

The rules marked (*) have the following general provisos:

- All predicates P occurring in $\mathit{Locked}(E,P)$ or $\mathit{Ex}(E,P)$ in the rules are mobile.
- For every triple $\{R\}\texttt{f}\{S\}$ in the assumptions (i.e., on the left of "$\vdash$"), R is mobile while S is thread-indepedent.

The precision requirements for the locks' resource invariants and the preconditions of forkable procedures are necessary to prove soundness of, respectively, the `release` case and the frame rule, since there are two operations, `release` and `fork`, which perform heap splitting of the private space of the active thread, about which we need to be able to reason.

The predicates which are required to be thread-independent are precisely those describing parts of the heap that, conceptually speaking, is transfered between different threads. The meaning of these predicates needs to be the same before and after such a transfer.

3 Example: Lock coupling

The lock coupling algorithm can be considered the seminal example for storable locks and is treated in a very similar fashion by Gotsman et al. [8]. The idea is to allow multiple threads to simultaneously add and remove elements from an ordered singly linked list. Instead of having a single general lock for the entire list, which would require all threads to wait for the active one to finish its operation, there is one lock per node. By requiring threads to hold the lock for both the current node and the previous one whenever they search through the list or try to modify it, we can ensure that the data structures are never in an incoherent state, while the fine-grained locks allow for a great gain in efficiency.

Each node contains three fields: a lock protecting the entire node, with resource invariant P, a data field `val` containing an integer and a pointer to the next node or `nil`. We also use the convention that the head of the list always contains the value $-\infty$ and the tail $+\infty$. We present here code for initializing the list and for adding a new node with value e in the correct location in the heap. A procedure to remove a node with value e would be very similar.

Since the only way to know about the existence of a particular lock is by acquiring its predecessor in the list, the invariant itself enforces the particular locking procedure for iterating through a list. We present below a simplified version of the annotated proof for `add`.

The resource invariant is a recursively defined predicate:

$$P(n,e) = (\mu r.\lambda(n,e).\,[(n.val \mapsto e \wedge e = \infty)\vee \\ (\exists x, e', n.val \mapsto e * n.next \mapsto x * Ex(x.lock, r(x,e')) \wedge e < e'))])(n,e)$$

```
initialize() {
  let tail = new NODE in
  [tail.val] = $+\infty$;
  [tail.next] = NULL;
  makelock$_{P(tail,+\infty)}$(tail.lock);
  release(tail.lock);
  let head = new NODE in
  [head.val] = $-\infty$;
  [head.next] = tail;
  makelock$_{P(head,-\infty)}$(head.lock);
  release(head.lock);
}

add(e) {
$Ex(head.lock, P(head,-\infty))$
  let prev = head in
  acquire(prev.lock);
$Locked(prev.lock, P(prev,-\infty)) * P(prev,-\infty)$
  let curr = prev.next in
  acquire (curr.lock);
$\exists v'$, $Locked(prev.lock, P(prev,-\infty))*$
$prev.val \mapsto -\infty * prev.next \mapsto curr*$
$Locked(curr.lock, P(curr,v')) * P(curr,v')\wedge$
$-\infty < v'$
  while (curr.val < e) do {
$\exists x,v,v',v''$, $Locked(prev.lock, P(prev,v))*$
$Locked(curr.lock, P(curr,v')) * prev.val \mapsto v*$
$prev.next \mapsto curr * curr.val \mapsto v' * (v' = \infty\vee$
$(curr.next \mapsto x * Ex(x.lock, P(x,v'')) \wedge v' < v'')$
$\wedge v' < e \wedge v < v'$
    release(prev.lock);
    let prev = curr in
    let curr = curr.next in
    acquire(curr.lock);
  }
$\exists x,v,v',v''$, $Locked(prev.lock, P(prev,v))*$
$Locked(curr.lock, P(curr,v')) * prev.val \mapsto v*$
$prev.next \mapsto curr * curr.val \mapsto v' * (v' = \infty\vee$
$(curr.next \mapsto x * Ex(x.lock, P(x,v'')) \wedge v' < v'')$
$\wedge v < e \leq v'$
  if (curr.val != e) {
    let n = new NODE in
    [n.val] = e;
    [n.next] = curr;
    makelock$_{P(n,e)}$(n.lock);
$\exists x,v,v'$, $Locked(prev.lock, P(prev,v))*$
$Locked(n.lock, P(n,e))*$
$Locked(curr.lock, P(curr,v'))*$
$prev.val \mapsto v * n.val \mapsto e * n.next \mapsto curr*$
$curr.val \mapsto v' * \wedge v < e < v'$
    release(n.lock);
    [prev.next] = n;
  }
  release(prev.lock);
  release(curr.lock);
$\exists v,v'$, $Ex(prev.lock, P(prev,v))*$
$Ex(curr.lock, P(curr,v')) * ((v' = e \wedge v < v')$
$\vee(\exists n, Ex(n.lock, P(n,e) \wedge v < e < v')))$
}
```

As mentioned before, this example was treated already in [8]. Though our model removes the limitation that lock sorts have to be statically defined before the program is run, this increased generality is not needed in this particular example.

We conjecture that the additional flexibility will be useful for making proofs that are generic in the resource invariant predicate.

4 The model

4.1 The Recursive Domain Equations

As described in Section 1, the main issue with storable locks originates from the fact that the resource invariants depend on "worlds" which contain locks and resource invariants. The worlds can be seen as a semantic layer above the heap, containing additional information about the state of the machine. In our case, in order to be able to formulate soundness of the logic, we want to keep track of three different kinds of information:

- Whenever a memory cell is a lock, we want access to its resource invariant.
- Whenever a memory cell is a thread handle (the value returned by a call to `fork`), we want to remember the name of the procedure we are executing, so that we can look up its pre and post-conditions in the context.
- If a memory cell is "regular" (*i.e.* not a lock or a thread handle), we don't need to remember any other information than the fact that it does exist.

This is formulated in the following equation:

$$W = \ Loc \rightharpoonup^{fin} (Fun + Cell + Pred) \tag{1}$$

Here $Loc = \mathbb{N}$ represents locations in the heap, *Fun* is a set of procedure names, and *Cell* is a singleton set. For clarity, we will use the notations $T(f)$ and $Lock(Q)$ to refer to the first and third components of the $Fun + Cell + Pred$ sum type.

Heaps in our semantic layer will be identical to the concrete ones used in the operational semantics, with one exception: we will occasionally use an *unknown* value $\mathcal{U}$ to denote that no assumption is made as to the exact state of a lock.

$$\hat{H} = Loc \rightarrow_{fin} Val \cup \{\mathcal{U}\}$$

This allows us to define a special $*$ operation designed to split a heap into several private sub-heaps. Because the memory cells corresponding to locks are shared by several threads, the regular separating conjunction is not well defined, forcing us to use a variant where starred heaps are allowed to overlap.

Concretely, $h_1 * h_2$ is well defined if and only if for all $l \in dom(h_1) \cap dom(h_2)$ we have that $h_1(l) * h_2(l)$ is well defined. Here we define $*$ on individual memory cells by $\mathcal{U} * \mathcal{U} = \mathcal{U}$ and $\mathcal{U} * j = j * \mathcal{U} = j$ for any $j \in Val$, while $j * k$ is undefined. We then take $(h_1 * h_2)(l) = h_1(l) * h_2(l)$ when $l \in dom(h_1) \cap dom(h_2)$, while $(h_1 * h_2)(l)$ is defined as for regular separating conjuction when $l \notin dom(h_1) \cap dom(h_2)$. We also use an operation to merge a tuple of heaps into a single one: $\overline{hp} = \mathop{\Large *}_{i \in dom(hp)} hp(i)$.

Since, because of interferences, threads can only be certain of the state of a lock when they actually own it, there is a simple way to map full abstract heaps to concrete ones: simply replace all unknown values by 0, as the absence of any claims to hold the lock means that it is available.

$$|.| : \hat{H} \mapsto H$$
$$|h| = \lambda x. \begin{cases} 0 & \text{if} h(x) = \mathcal{U} \\ h(x) & \text{otherwise} \end{cases}$$

This is a departure from the way heap sharing has been handled in concurrent separation logic. Instead of having a general shared space that is not private to any thread and that everybody can manipulate, we include duplicate values of the shared memory cells in relevant private spaces. This reflects more accurately the scope of the lock variables, and permits much finer granularity, with locks being known only to a limited number of threads.

The one remaining set to define is *Pred*, the type of lock resource invariants. A predicate should be a function that takes a world (containing semantic information about the current state) and returns a set of heaps, i.e., those heaps that "satisfy the predicate." Roughly, we want something like:

$$\mathit{Pred} = W \rightarrow \mathcal{P}(\hat{H})\,. \tag{2}$$

(We also require that if a heap $h \in \hat{H}$ satisfies a predicate in a world $w \in W$, then $\mathrm{dom}(h) \subseteq \mathrm{dom}(w)$; see below.) Notice the circularity between W and *Pred* in (1) and (2): the two equations do not in themselves form a good definition of the sets of worlds and heaps.

4.2 Complete, Bounded Ultrametric Spaces

We solve this system of equations using Complete, Bounded, Ultrametric spaces, objects in the category $CBUlt$, following the methods used in [5]. The first step is to define a proper distance on the various spaces we are manipulating, and in particular $\mathcal{P}(\hat{H})$, which we turn into the set of *uniform predicates*, $UPred$, by combining heaps with downwards closed integer indices.

Definition 4.1

$$\mathit{UPred}(\hat{H}) = \Big\{P \in \mathcal{P}(\hat{H} \times \mathbb{N}) \mid \forall h \in \hat{H}, \forall k \in \mathbb{N}, \forall j \leq k, P(h,k) \Rightarrow P(h,j)\Big\}$$

We define the following distance on $\mathit{UPred}(\hat{H})$:

$$d(P,Q) = \begin{cases} 0 & \text{if } P = Q \\ \min\{2^{-n} | \forall k \leq n, \forall h \in \hat{H}, P(h,k) \Leftrightarrow Q(h,k)\} & \text{otherwise.} \end{cases}$$

For any distance d and natural number n, we define "equality up to n", denoted $P =^n Q$, as $d(P,Q) \leq 2^{-n}$.

Let T be a fixed set. We want to solve the following equation in *CBUlt*:

$$X \cong \frac{1}{2}\Big((\mathbb{Z} \rightharpoonup^{fin} (T + X)) \rightarrow_{\mathrm{n}} \mathit{UPred}(\hat{H})\Big) \tag{3}$$

Here the function space $\to_{\mathrm{n}}$ consists of those functions p which are non-expansive (in the metric-space sense) and satisfy the following property: If $(h, n) \in p(w)$, then $\mathrm{dom}(h) \subseteq \mathrm{dom}(w)$.

In order to do solve this equation, we must first specify which metric is used on the right-hand side, assuming that (X, d_X) is a given object of *CBUlt*.[4] This metric is obtained from the following building blocks. We equip the set T with the discrete metric, i.e., every pair of distinct elements has distance 1. The operators $\times$ and $\to_{\mathrm{n}}$ are given by the cartesian closed structure of *CBUlt* (here $\to_{\mathrm{n}}$ is a closed subspace of the exponential), while $+$ is the coproduct (which gives elements from different summands the maximal distance, 1). More details about these operators can be found in, e.g., [5]. It only remains to explain the "finite partial function space" operator $\rightharpoonup^{fin}$. Concretely, the distance function d on $\mathbb{Z} \rightharpoonup^{fin} (T + X)$ is given by

$$d(P, Q) = \begin{cases} 1 & \text{if } dom(P) \neq dom(Q) \\ max_{i \in dom(P)}(d_{T+X}(P(i), Q(i))) & \text{otherwise.} \end{cases}$$

Using well-known existence theorems for recursive metric-space equations [1], we can then find a solution X to (3). If we rename $\mathbb{Z}$ to *Loc*, T to $T(\mathit{Fun}) + \mathit{Cell}$ and X to *Pred*, we obtain a solution to the following equation:

$$\begin{aligned} W =&\ Loc \rightharpoonup^{fin} (T(\mathit{Fun}) + \mathit{Cell} + \mathit{Lock}(\widehat{\mathit{Pred}}))) \\ \mathit{Pred} =&\ W \to_{\mathrm{n}} \mathit{UPred}(\hat{H}) \\ \widehat{\mathit{Pred}} \cong&\ \frac{1}{2}\mathit{Pred} \end{aligned}$$

Let *App* be the isomorphism above, and let *Abs* be its inverse:

$$\begin{aligned} \mathit{App} :& \widehat{\mathit{Pred}} \to \frac{1}{2}\mathit{Pred} \\ \mathit{Abs} :& \frac{1}{2}\mathit{Pred} \to \widehat{\mathit{Pred}}. \end{aligned}$$

World composition (of elements w and w' of W) is defined as follows: $w * w'$ is defined if the domains of w and w' do not overlap, and in this case $w * w'$ is the union of the two partial functions w and w'.

We can then prove that *Pred* models the usual separation logic operations, which is summarized by saying it is equipped with a complete BI-algebra structure.[5] First, for every $w \in W$ we define $\mathit{UPred}_w(\hat{H})$ to be the subset of $\mathit{UPred}(\hat{H})$ where all heaps have domain contained in $\mathrm{dom}(w)$. This set is given a BI-algebra structure as follows. We define $h_1 \perp_w h_2$ to hold if $h_1 * h_2$ is defined, and if furthermore

[4] Actually, we must show that the right-hand side of the equation is a contravariant functor in the metric space X. We omit this argument here; details of a similar case can be found in [5].

[5] Although we do not include separating implication in the syntactic assertation language, the BI-algebra structure on *Pred* does include a "separating implication" operator $-\!\!*$. See the discussion after Proposition 4.6.

$w(l) = Lock(-)$ for all $l \in \mathrm{dom}(h_1) \cap \mathrm{dom}(h_2)$. Then we define $*_w$ on heaps in the same way as $*$, but with additional requirement that $h_1 *_w h_2$ is only defined if $h_1 \perp_w h_2$. (The intuition is that only values which correspond to locks can be shared between heaps.) Now the BI-algebra structure on $UPred_w(\hat{H})$ is given as follows: $\emptyset$ is $\perp$, $\{h \mid \mathrm{dom}(h) \subseteq \mathrm{dom}(w)\} \times \mathbb{N}$ is $\top$, and set-theoretic union and intersection are, respectively, join and meet. The remaining three operations are defined by:

$$(h, n) \in P \to Q = \forall k \leq n, (h, k) \in P \Rightarrow (h, k) \in Q$$
$$(h, n) \in P * Q = \exists h_1, h_2, h = h_1 *_w h_2 \wedge (h_1, n) \in P \wedge (h_2, n) \in Q$$
$$(h, n) \in P \mathrel{-\!\!*} Q = \forall k \leq n, \forall h' \perp_w h, (h', k) \in P \Rightarrow (h *_w h', k) \in Q$$

These operations are then extended pointwise to *Pred*, using the fact that, by construction, every element of *Pred* maps any world w into $UPred_w(\hat{H})$ (and not just into $UPred(\hat{H})$).

Lemma 4.2 *Pred equipped with these operations is a complete BI-algebra [3] on Set.*

4.3 Predicate interpretation

Having solved the recursive domain equations, we can now define interpretations for our assertion language. The interpretation of a predicate P, written $[\![P]\!]^k_s$, takes as arguments a stack s and the identifier k of the current thread, returning an object in *Pred*.

Stacks can store either plain values or predicates with an optional argument:

$$Stack = (Var \to_{fin} Val) \uplus (PredVar \to_{fin} (Val \to Pred))$$

Definition 4.3

$$\begin{aligned}
[\![Ex(E,P)]\!]^k_s &= \lambda w.\{(h,n) \mid \exists Q.\, w([\![E]\!]_s) = Lock(Q) \wedge h = [[\![E]\!]_s \mapsto \mathcal{U}] \wedge \\
&\qquad\qquad \forall w'.App(Q)(w') =^{n-1} [\![P]\!]^k_s(w')\} \\
[\![Locked(E,P)]\!]^k_s &= \lambda w.\{(h,n) \mid \exists Q.\, w([\![E]\!]_s) = Lock(Q) \wedge h = [[\![E]\!]_s \mapsto k] \wedge \\
&\qquad\qquad \forall w'.App(Q)(w') =^{n-1} [\![P]\!]^k_s(w')\} \\
[\![tid(f,E)]\!]^k_s &= \lambda w.\{(h,n) \mid w([\![E]\!]_s) = T(f) \wedge \exists j.\, h = [[\![E]\!]_s \mapsto j]\} \\
[\![r\,(E)]\!]^k_s &= s(r)\,([\![E]\!]_s) \\
[\![(\mu r.\lambda x.P)(E)]\!]^k_s &= \mathit{fix}\left(\lambda P_1^{Val \to Pred}\, \lambda v.\, [\![P]\!]^k_{s[r \mapsto P_1, x \mapsto v]}\right)([\![E]\!]_s) \\
[\![P * Q]\!]^k_s &= \lambda w.\{(h,n) \mid \exists h_1, h_2.\, h = h_1 *_w h_2 \wedge \\
&\qquad\qquad (h_1, n) \in [\![P]\!]^k_s(w) \wedge (h_2, n) \in [\![Q]\!]^k_s(w)\} \\
[\![E_1 \mapsto E_2]\!]^k_s &= \lambda w.\{(h,n) \mid w([\![E_1]\!]_s) = Cell \wedge h = [[\![E_1]\!]_s \mapsto [\![E_2]\!]_s]\} \\
&\ldots
\end{aligned}$$

Again, the intuition behind the use of $*_w$ is that only values which correspond to locks can be shared between heaps.

The most interesting point to note is how the interpretation of the two lock predicates Ex and $Locked$ refers to $(n-1)$-equality of predicates instead of "true" equality. This is necessary by the non-expansiveness requirement of $W \to_{\mathrm{n}} UPred(\hat{H})$ in the domain equation, while the -1 part simply reflects the $\frac{1}{2}$ factor in the final isomorphism.

Theorem 4.4 *For all P, s, and k, the predicate $[\![P]\!]^k_s$ is well-defined.*

Proof Using the fact that recursion can only happen in guarded environments, the interpretation of $(\mu r.\lambda x.P)(E)$ uses the fixpoint of a contractive function on $Val \to Pred$, which is in $CBUlt$ if we view Val as a discrete metric space. □

Semantic predicates which are definable by syntactic predicates satisfy a certain property which is crucial for our soundness proof:

Definition 4.5 A predicate $p \in Pred$ is *local* if for all h, n, and w,

$$(h, n) \in p(w) \iff (h, n) \in p(w|_{dom(h)}).$$

Proposition 4.6 *For every syntactic predicate P containing no free predicate variables, the semantic predicate $[\![P]\!]^k_s$ is local.*

Proposition 4.7 *If p is local, then p is monotone: $(h, n) \in p(w)$ implies $(h, n) \in p(w * w')$ whenever $w * w'$ is defined.*

Proposition 4.6 does *not* hold if the assertion language includes separating implication, and this is the reason we have excluded that operator. A counterexample is given by the predicate $[0 \mapsto 1] \mathrel{-\!\!*} \bot$ which would hold for the empty heap in the empty world, but not in a world which maps 0 to $Cell$. An attempt at avoiding the problem would be to construct a model where all semantic predicates are "local by construction": this could perhaps be done by requiring that $(h, n) \in p(w)$ implies $\mathrm{dom}(h) = \mathrm{dom}(w)$ (instead of just $\mathrm{dom}(h) \subseteq \mathrm{dom}(w)$). Then, in the definition of separating conjunction, one would split worlds as well as heaps. We have not yet investigated this approach in detail.

5 Soundness

5.1 Soundness definitions

Having defined how to interpret assertions, we can now define what it means for a Hoare triple to be semantically valid. Using a similar method to the one developed in [13], we use as our main building brick a *safety predicate*, which stipulates that, for a given number of steps, the current machine state will not lead to an execution fault and will satisfy relevant assertions.

Intuitively, a Hoare triple $\{P\}\mathtt{c}\{Q\}$ is then said to be semantically valid to the j^{th} level if we can add to any state safe to the j^{th} level a thread executing $\mathtt{c}$ and a heap satisfying $\llbracket P \rrbracket$, and obtain a new state that is also safe to the j^{th} level. The fact that the new thread, once finished executing, should satisfy $\llbracket Q \rrbracket$ is already part of the safety predicate and does not need to be specified separately.

This allows the complete abstraction of the interferences between different threads, where the non-deterministic character of the operational semantics reductions is confined to the safety predicate.

We now turn to the definitions. A *configuration* is a tuple (w, hp, s, tp, qp) where for some $n \in \mathbb{N}$ we have $w \in W$, $hp : \{1, \ldots, n, \Omega\} \to \hat{H}$, $s \in \mathit{Stack}$, $tp : \{1, \ldots, n\} \to \mathit{Cmd}$, and $qp : \{1, \ldots, n\} \to \mathit{Pred}$. The intuition is the following. In order to have a notion of ownership, allowing us to reason with resource invariants, we split the heap in many private sub-heaps hp, one for each thread in tp, forbidding anyone to manipulate a memory cell not in its private space. There is also an additional space, denoted with the special thread id Ω and corresponding to the heaps described by the invariants of available locks. A successful lock acquisition will then correspond to the transfer of a sub-heap owned by Ω into the private space of the new owner thread, with release being the reverse operation. Similar heap swaps occur with fork and join operations. Finally, the intended semantic properties of the total heap are described by the world w, and each predicate $qp(j)$ is a postcondition that the private heap of thread j must satisfy when that thread is done executing.

Definition 5.1 Assume that $w \in W$ and $h \in H$ satisfy $\mathrm{dom}(w) = \mathrm{dom}(h)$. The *free predicate* of (w, h) is defined as the separating conjunction of all lock invariants of available locks: $\mathit{free}(w, h) = \ast \mathit{App}(p_i)$ where $l_1, \ldots, l_m$ are all the locations in $\mathrm{dom}(w)$ such that $w(l_i) = \mathit{Lock}(-)$ and $h(l_i) = 0$, and where $w(l_i) = \mathit{Lock}(p_i)$ for each $i = 1, \ldots, m$.

Definition 5.2 A configuration (w, hp, s, tp, qp) is *consistent*, which is written $\mathit{cons}(w, hp, s, tp, qp)$, if the following properties hold:

- $\mathrm{dom}(w) = \mathrm{dom}(\overline{hp})$.
- For every l and all $j \neq k$, if $l \in \mathrm{dom}(hp(j)) \cap \mathrm{dom}(hp(k))$, then $w(l) = \mathit{Lock}(-)$.
- For every j, the predicate $qp(j)$ is local.
- For every l, if $w(l) = \mathit{Cell}$ or $w(l) = T(f_i)$, then $l \in \mathrm{dom}(hp(j))$ implies $hp(j)(l) \neq \mathcal{U}$.
- For every l, if $w(l) = \mathit{Lock}(p)$, then the predicate p is local. Furthermore, if $l \in \mathit{dom}(hp(j))$ then $hp(j)(l) \neq 0$.

A consistent configuration is called complete if, intuitively, all threads are present and $hp(\Omega)$ satisfies the invariants of free locks:

Definition 5.3 A configuration (w, hp, s, tp, qp) is *n-complete*, which is written $\mathit{comp}_n(w, hp, s, tp, qp)$, if the following properties hold:

- (w, hp, s, tp, qp) is consistent.
- For every l, if $w(l) = T(f_i)$, then there exist j and k such that $l \in \mathrm{dom}(hp(j))$

and $hp(j)(l) = k$ and $qp(k) = [\![Q_i]\!]^k$. Here $\{P_i\}f_i\{Q_i\}$ comes from the global environment Γ.

- $(hp(\Omega), n) \in \mathit{free}(w, |\overline{hp}|)(w)$.

We next define a transition relation on consistent configurations. For complete configurations, this relation serves as a semantic layer on top of the standard operational semantics. For configurations which are not complete, however, we add non-deterministic reductions that mimic the "missing parts."

Definition 5.4 The relation

$$(w, hp, s, tp, qp) \overset{n}{\to}_j (w', hp', s', tp', qp')$$

holds if there exists a reduction $(|\overline{hp}|, s, tp) \to_j (h', s', tp')$ such that $|\overline{hp'}| = h'$ and $hp'(k) = hp(k)$ for all $k \notin \{j, \Omega\}$. Furthermore, letting $c = tp(j)$:

- If c is not a reference allocation, lock operation, or thread operation, then $w' = w$, $qp' = qp$, and $hp'(\Omega) = hp(\Omega)$.
- If c is `let x = new in C'`, then $w' = w[v \mapsto \mathit{Cell}]$, $qp' = qp$, and $hp'(\Omega) = hp(\Omega)$.
- If c is `make_lock_P(E)`, then $w([\![E]\!]_s) = \mathit{Cell}$, $w' = w[[\![E]\!]_s \mapsto \mathit{Lock}(\mathit{Abs}([\![P]\!]_s^j))]$, $qp' = qp$, and $hp'(\Omega) = hp(\Omega)$.
- If c is `acquire(E)`, then $w([\![E]\!]_s) = \mathit{Lock}(Q)$ for some Q, and $hp(j)([\![E]\!]_s) = \mathcal{U}$. Furthermore, $w' = w$, $qp' = qp$, and there exists $(h_0, n) \in \mathit{App}(Q)(w)$ such that $hp(\Omega) = h_0 *_w h_1$ while $hp'(\Omega) = h_1$ and $hp'(j) = h_0 *_w (hp(j)[[\![E]\!]_s \mapsto j])$. (Notice that here the superscript "n" on the relation symbol is used.)
- If c is `release(E)`, then $w([\![E]\!]_s) = \mathit{Lock}(Q)$ for some Q, and $hp(j)([\![E]\!]_s) = j$. Furthermore, $w' = w$, $qp' = qp$, and there exists $(h_0, n) \in \mathit{App}(Q)(w)$ such that $hp(j) = h_0 *_w h_1$ while $hp'(j) = h_1[[\![E]\!]_s \mapsto \mathcal{U}]$ and $hp'(\Omega) = h_0 *_w hp(\Omega)$.
- If c is `let x = fork(f) in M`, then $w' = w[v \mapsto T(f)]$ where $s' = s[x \mapsto v]$. Furthermore, $qp' = qp[k \mapsto [\![Q]\!]^k]$ where $\{P\}f\{Q\} \in \Gamma$ and $k \in \mathrm{dom}(tp') \setminus \mathrm{dom}(tp)$. Also, $hp'(\Omega) = hp(\Omega)$, and there exist h_1,h_2, such that $hp(j) = h_1 *_w h_2$ and $(h_2, n) \in [\![P]\!]^k(w)$, while $hp'(j) = h_1[v \mapsto k]$ and $hp'(k) = h_2$.
- Finally, if c is `join(E)`, then $w([\![E]\!]_s) = T(f)$ for some f. Also, $hp(j) = h_1 *_w [[\![E]\!]_s \mapsto k]$ where $k \in \mathrm{dom}(tp) \setminus \mathrm{dom}(tp')$. Furthermore, $w' = w|_{dom(w)\setminus\{[\![E]\!]_s\}}$, $hp'(\Omega) = hp(\Omega)$, and $hp'(j) = h_1 *_w hp(k)$.

Furthermore, the following reductions hold:

$$\begin{aligned}(w, hp[j \mapsto h * [l \mapsto \mathcal{U}]], s, tp[j \mapsto \texttt{acquire(E)}], qp) &\overset{n}{\to}_j \\ &(w, hp[j \mapsto h1 * [l \mapsto j] * h_0], s, tp[j \mapsto \texttt{skip}], qp),\end{aligned}$$

if $w(l) = \mathit{Lock}(Q)$, $[\![E]\!]_s = l$, $\overline{hp}(l) = \mathcal{U}$, $(h_0, n) \in \mathit{App}(Q)(w)$, and there is no sub-heap h'_0 of $hp(\Omega)$ such that $(h'_0, n) \in App(Q)(w)$.

$$(w, hp[j \mapsto h_1 * [l \mapsto k]], s, tp[j \mapsto \texttt{join(E)}], qp) \overset{n}{\rightarrow}_j \\ (w, hp[j \mapsto h_1 * h_0], s, tp[j \mapsto \texttt{skip}], qp),$$

if $k \notin \mathrm{dom}(tp)$, $w(l) = T(f_i)$, $[\![E]\!]_s = l$, and $(h_0, n) \in [\![Q_i]\!]^j_s$. (Here $\{P_i\}f_i\{Q_i\}$ comes from the global environment.)

Notice that the last two reductions can only happen for tuples that are not n-complete. In the first of the two, the idea is that there is no sub-heap in $hp(\Omega)$ which satisfies the lock invariant, so a sub-heap is chosen non-deterministically. In the second of the two, the idea is that the thread to be joined is not present, so a sub-heap satisfying the postcondition of the thread is chosen non-deterministically.

In the following, let d range over configurations.

Proposition 5.5

(i) *If $cons(d)$ and $d \overset{n}{\rightarrow}_j d'$, then $cons(d')$.*

(ii) *If $comp_n(d)$ and $d \overset{n}{\rightarrow}_j d'$, then $comp_n(d')$.*

As promised, reduction of complete configurations can be viewed as a semantic layer on top of standard reduction:

Proposition 5.6 *If (w, hp, s, tp, qp) is n-complete and $(w, hp, s, tp, qp) \overset{n}{\rightarrow}_j (w', hp', s', tp', qp')$, then $(\overline{hp}, s, tp) \rightarrow_j (\overline{hp}', s', tp')$.*

We are now ready to define the semantic safety predicate. The safety predicate stipulates that, up to a given number of steps, the configuration d it considers will be semantically *correct*. Intuitively, this means the following:

- d is consistent.
- d cannot reduce to `abort` in one step.
- The private heap of any thread which has finished execution (*i.e.* it only needs to execute `skip`) satisfies its post-condition.
- If d reduces to another configuration d', then d' is safe up to a smaller number of steps. ("Preservation")
- If d is complete, and if the configuration corresponding to d reduces in the standard operational semantics then d reduces. ("Progress")

Definition 5.7 Let $d = (w, hp, s, tp, qp)$. The predicate $safe_n(d)$ is defined by induction on n as follows. $safe_0(d)$ always holds, and $safe_{n+1}(d)$ holds iff:

- d is consistent, i.e., $cons(d)$ holds.
- $\forall j \in dom(tp), \forall h$ such that $h \perp \overline{hp}$, there is no reduction of the form $(|\overline{hp} * h|, s, tp) \longrightarrow_j \texttt{abort}$.
- $\forall j \in dom(tp).\ tp(j) = \texttt{skip} \implies (hp(j), n) \in qp(j)(w)$
- If $d \overset{m}{\rightarrow}_j d'$, then $safe_{min(n,m)}(d')$.

- If d is n-complete and $(|\overline{hp}|, s, tp) \to_j (h', s', tp')$, then $d \overset{n}{\to}_j d'$ for some d'.

Next we define semantic validity of Hoare triples.

Definition 5.8

$$\Gamma \vDash_j \{P\}\mathtt{c}\{Q\}$$

is defined as

$$\begin{gathered}\forall m < j, \forall w, hp, s, tp, qp,\\ \text{if } (\forall i \in \mathit{dom}(\Gamma), \Gamma \vDash_m \{P_i\}\mathtt{c\_i}\{Q_i\} \wedge \mathit{safe}_m(w, hp, s, tp, qp)) \text{ then}\\ \forall k \notin \mathit{dom}(hp), \forall w' \text{ such that } w * w' \text{ is well defined,}\\ \forall (h, n) \in [\![P]\!]_s^k(w * w'),\\ (h \perp \overline{hp} \;\wedge\; \mathit{cons}(w * w', hp[k \mapsto h], s, tp[k \mapsto \mathtt{c}], qp[k \mapsto [\![Q]\!]^k])) \Rightarrow\\ \mathit{safe}_{\mathit{min}(m,n)}(w * w', hp[k \mapsto h], s, tp[k \mapsto \mathtt{c}], qp[k \mapsto [\![Q]\!]^k]).\end{gathered}$$

Definition 5.9

$$\vDash_j \mathtt{let}\ f_i : \{P_i\}\mathtt{c\_i}\{Q_i\}\ \mathtt{in}\ \{P\}\mathtt{c}\{Q\}$$

is defined as

$$\forall i, \Gamma \vDash_j \{P_i\}\mathtt{c\_i}\{Q_i\} \wedge \Gamma \vDash_j \{P\}\mathtt{c}\{Q\}$$

Theorem 5.10 (Soundness)

$$\vdash \mathtt{let}\ f_i : \{P_i\}\mathtt{c\_i}\{Q_i\}\ \mathtt{in}\ \{P\}\mathtt{c}\{Q\} \Rightarrow \forall j, \vDash_j \mathtt{let}\ f_i : \{P_i\}\mathtt{c\_i}\{Q_i\}\ \mathtt{in}\ \{P\}\mathtt{c}\{Q\}$$

5.2 The proof

We prove soundness separately for each proof rule, in each case using strong induction on the index of the safety predicate. The proofs are relatively straightforward, consisting mostly in unfolding various definitions, considering all reduction steps that can be taken from the initial state and then proving safety of the new state. The following result takes care of the cases where a reduction happens in a thread other than the one we have just added.

Proposition 5.11 *Assume that* $\mathit{safe}_j(w, hp, s, tp, qp)$, $\mathit{cons}(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto q])$ *where* $k \notin \mathrm{dom}(tp)$, *and*

$$(w * w', hp[k \mapsto h], s, tp[k \mapsto c], qp[k \mapsto q]) \overset{n}{\to}_j d$$

for some $j \neq k$. *Assume furthermore that* $c \neq \mathtt{skip}$. *Then there exist* w_0, hp_0, s_0, tp_0, *and* qp_0 *such that*

$$d = (w_0 * w', hp_0[k \mapsto h], s_0, tp_0[k \mapsto c], qp_0[k \mapsto q])$$

and

$$(w, hp, s, tp, qp) \overset{n}{\to}_j (w_0, hp_0, s_0, tp_0, qp_0).$$

Proof (sketch). By case analysis following the definition of the reduction relation. The locality requirements in the definition of consistent configurations are needed to obtain the reduction starting from (w, hp, s, tp, qp); intuitively, we need to ensure that the relevant predicates which hold in world $w * w'$ already hold in world w. □

By this proposition, it is enough to consider reductions in the newly added thread k when proving safety. Here the case of the "fork" rule is the most challenging, since (informally speaking) after a "fork" reduction there are *two* threads that need to be added to the "base" configuration, while the definition of soundness only allows adding one at a time. By using the locality property (Proposition 4.6) of preconditions of forkable threads, one can add the newly forked thread first, and then the old thread. Using precision of the considered predicates, we can then prove that the heap splitting operated by the transition relation yields a new configuration suitable for using our induction hypotheses.

6 Conclusion

We have provided a new approach to models of program logics for concurrent programs with storable locks, using ultrametric spaces to solve directly the recursive domain equations that arise when resource invariants are allowed to describe heaps in which their associated locks reside. In addition, we have also proposed conceptually simple proof methods for soundness of our logic with respect to the standard operational semantics.

Future work includes extending the language with new features, for instance storable procedures, which we believe would be straightforward to add. The soundness proof also appears to lend itself particularly well to formalisation and automated machine proving, e.g. using the work by Benton et al. [2].

We thank Mike Dodds, Derek Dreyer, Jacob Thamsborg, and Viktor Vafeiadis for discussions about this work.

References

[1] P. America and J. J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989.

[2] N. Benton, L. Birkedal, A. Kennedy, and C. Varming. Formalizing domains, ultrametric spaces and semantics of programming languages. Available from `http://research.microsoft.com/en-us/um/people/nick/domultra.pdf`, July 2010.

[3] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Transactions on Programming Languages and Systems*, 2007.

[4] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.

[5] L. Birkedal, K. Støvring, and J. Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *Mathematical Structures in Computer Science*, 20(4):655–703, 2010.

[6] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *CoRR*, 2006.

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[8] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, pages 19–37, 2007.

[9] A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, Department of Computer Science, Princeton, NJ, October 2008.

[10] A. Hobor, A. Appel, and F. Zappa-Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[11] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004.

[12] G. Stewart and A. Appel. Local actions for a Curry-style operational semantics. In *Proceedings of PLPV'11*, 2011.

[13] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.

A Detailed soundness proof

A.1 Technical lemmas

In order for the soundness proofs to go through, we need a number of small results on the operation semantics:

Lemma A.1 *If $k \neq j$ and (if $tp(j) = \texttt{join}(E)$ then $h(\llbracket E \rrbracket_s) \neq k$), then $(h, s, tp) \longrightarrow_j abort \Leftrightarrow (h, s, tp[k \mapsto c]) \longrightarrow_j abort$*

Lemma A.2 *If $(h * h', s, tp) \longrightarrow_j \texttt{abort}$, then $(h, s, tp) \longrightarrow_j \texttt{abort}$.*

A.2 Sequential composition

Before proving soundness of sequential composition, we first need a technical lemma:

Lemma A.3 *$\forall j, \forall w, hp, s, tp, qp, k, Q, R, \texttt{c1}, \texttt{c2}$, noting $qp_Q = qp[k \mapsto \llbracket Q \rrbracket^k]$ and $qp_R = qp[k \mapsto \llbracket R \rrbracket^k]$, if*

- *$safe_j(w, hp, s, tp[k \mapsto \texttt{c1}], qp_Q)$ (1)*
- *$\models_j \{Q\}\texttt{c2}\{R\}$*

then $safe_j(w, hp, s, tp[k \mapsto \texttt{c1;c2}], qp_R)$.

Proof We proceed by induction on j. The only truly interesting case is when $(w, hp, s, tp[k \mapsto c1; c2], qp_R) \overset{n}{\to}_k (w', hp', s', tp[k \mapsto c1'; c2], qp_R)$, for which we want to prove $safe_{min(j-1,n)}(w', hp', s', tp[k \mapsto \texttt{c1;c2}], qp_R)$.

From the operational semantics, we get that $(w, hp, s, tp[k \mapsto \texttt{c1}], qp_Q) \overset{n}{\to}_k (w', hp', s', tp[k \mapsto \texttt{c1'}], qp_Q)$ which, in conjunction with (1), gives $safe_{min(n,j-1)}(w', hp', s', tp[k \mapsto \texttt{c1'}], qp_Q)$. Since $min(j-1, n) < j$, we can apply the induction hypothesis and conclude.

□

The proof rule for sequentical composition is

$$\frac{\vdash \{P\}\texttt{c1}\{Q\} \qquad \vdash \{Q\}\texttt{c2}\{R\}}{\vdash \{P\}\texttt{c1;c2}\{R\}}$$

Following the definition of soundness, we assume that $\forall j, \vDash_j \{P\}\texttt{c1}\{Q\}$ (1) and $\forall j, \vDash_j \{Q\}\texttt{c2}\{R\}$ (2) and aim to prove that $\forall j, \vDash_j \{P\}\texttt{c1;c2}\{R\}$.

Proof We proceed by induction on j. The case where $j = 0$ is trivial since the definition of $\vDash$ relies on properties for some $m < j$.

Let's assume that $\forall k \leq j, \vDash_k \{P\}\texttt{c1;c2}\{R\}$ (3) and prove $\vDash_{j+1} \{P\}\texttt{c1;c2}\{R\}$.

Following the definition of $\vDash$, we need to prove safety for all $m < j+1$, but any $m < j$ is obtained directly via (3), so we only need to consider the case where $m = j$.

Let w, hp, s, tp, qp be fixed such that $\mathit{safe}_j(w, hp, s, tp, qp)$ (4) holds. Let now w' and $k \notin \mathit{dom}(hp)$ be fixed. We choose $(h, n) \in \llbracket P \rrbracket_s^k(w * w')$ (5) such that $h \bot \overline{hp}$ and $\mathit{cons}(q * w', hp[k \mapsto h], s, tp', qp')$ (6). We note $tp' = tp[k \mapsto \texttt{c1;c2}]$ and $qp' = qp[k \mapsto \llbracket R \rrbracket^k]$. Our goal is now to prove $\mathit{safe}_{\mathit{min}(n,j)}(w * w', hp[k \mapsto h], s, tp', qp')$.

- $\mathit{cons}(w * w', hp[k \mapsto h], s, tp', qp')$ is exactly (6).
- Let's assume there is a thread l such that $(|\overline{hp} * h|, s, tp') \longrightarrow_l \texttt{abort}$. If $l \neq k$, we get from lemmas A.1 and A.2 that $(|\overline{hp}|, s, tp) \longrightarrow_l \texttt{abort}$, which is absurd by (4). If $l = k$, the operational semantics entails that $(|\overline{hp} * h|, s, tp[k \mapsto c1]) \longrightarrow_l \texttt{abort}$ which contradicts (1).
- $\forall l \in \mathit{dom}(tp')$ such that $tp'(l) = \texttt{skip}$, we know from (4) that $(hp(l), j-1) \in qp(l)(w)$. Since $tp'(k) \neq \texttt{skip}$, we have that $k \neq l$ and thus, $qp'(l) = qp(l)$ and $hp[k \mapsto h](l) = hp(l)$, hence that $(hp[k \mapsto h](l), j-1) \in qp'(l)(w)$. Since $\mathit{min}(n,j) \leq j$, we obtain $(hp[k \mapsto h](l), \mathit{min}(n,j) - 1) \in qp'(l)(w)$. Finally, with proposition 4.7, we conclude that $(hp[k \mapsto h](l), \mathit{min}(n,j) - 1) \in qp'(l)(w * w')$.
- If $(w * w', hp[k \mapsto h], s, tp', qp') \overset{n}{\rightarrow}_l (w_0, hp_0, s_0, tp_0, qp_0)$, we want to show $\mathit{safe}_{\mathit{min}(j-1,n)}(w_0, hp_0, s_0, tp_0, qp_0)$. If $l \neq k$, we conclude from lemma 5.11. If on the other hand, $l = k$, from (1), we obtain $\mathit{safe}_{\mathit{min}(n,j)}(w * w', hp[k \mapsto h], s, tp[k \mapsto \texttt{c1}], qp[k \mapsto \llbracket Q \rrbracket_s^k])$ (7). There are then two possible situations:
  - If $\texttt{c1} = \texttt{skip}$, we get $(|\overline{hp} * h|, s, tp') \longrightarrow_k (|\overline{hp} * h|, s, tp[k \mapsto \texttt{c2}])$. From (7) and the safety definition, $(h, \mathit{min}(n,j) - 1) \in \llbracket Q \rrbracket^k(w * w')$. From (2) with index $j - 1$, we get $\mathit{safe}_{\mathit{min}(j-1, \mathit{min}(n,j)-1)}(w * w', hp[k \mapsto h], s, tp[k \mapsto \texttt{c2}], qp')$. Since $\mathit{min}(j-1, \mathit{min}(n,j) - 1) = \mathit{min}(n-1, j-1)$, we can conclude.
  - If $\texttt{c1} \neq \texttt{skip}$ and $(|\overline{hp} * h|, s, tp[k \mapsto \texttt{c1}]) \longrightarrow_k (|\overline{hp'}|, s', tp[k \mapsto \texttt{c1'}])$, from (7) we get that for some $j' < j$ and w'', $\mathit{safe}_{j'}(w'', hp', s', tp[k \mapsto \texttt{c1'}], qp[k \mapsto \llbracket Q \rrbracket_s^k])$. Since the thread local operational semantics is deterministic, we want to show that $\mathit{safe}_{j'}(w'', hp', s', tp[k \mapsto \texttt{c1';c2}], qp')$, which is obtained directly by applying lemma A.3.
- If $(|\overline{hp} * h|, s, tp') \longrightarrow_l (h_0, s_0, tp_0)$, the fact that $(w * w', hp[k \mapsto h], s, tp', qp') \overset{n}{\rightarrow}_l d'$ for some d' follows from (4) and the definition of $\overset{n}{\rightarrow}_l$.

□

A.3 Lock release

The proof rule for releasing a lock is

$$\frac{}{\vdash \{Locked(E,P) * P\}\texttt{release(E)}\{Ex(E,P)\}}$$

Let's assume that $\forall k \leq j, \vDash_j \{Locked(E,P) * P\}\texttt{release(E)}\{Ex(E,P)\}$ (1) and let's prove that this holds for $j+1$.

We choose w, hp, s, tp, qp such that $safe_j(w, hp, s, tp, qp)$ (2) holds. We then choose $k \notin dom(hp), w'$ such that $w * w'$ is well defined. Let then $(h, n) \in \llbracket Locked(E,P) * P \rrbracket^k_s(w * w')$ (4), $h \bot \overline{hp}$ and $cons(w * w', hp[k \mapsto h], s, tp[k \mapsto \texttt{release(E)}], qp[k \mapsto \llbracket Ex(E,P) \rrbracket^k])$ (5).

We note $qp' = qp[k \mapsto \llbracket Ex(E,P) \rrbracket^k], tp' = tp[k \mapsto \texttt{release(E)}]$ and choose and want to prove $safe_{min(n,j)}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof We unfold the safety definition:

- Consistency of the state is given by (5).
- If there was a reduction to $\texttt{abort}$ in thread l, we could use the same argument than in the sequential composition proof to show that $l \neq k$ is absurd. If $l = k$, the operational semantics tells us that $h(\llbracket E \rrbracket_s) \neq k$. By (4) and the definition of $\llbracket Locked(E,P) \rrbracket^k_s$, however, we have that $h(\llbracket E \rrbracket_s) = k$, which is absurd.
- Since the command to execute in thread k is not $\texttt{skip}$, we can conclude by (2) and proposition 4.7 that all threads done executing satisfy their postconditions in qp'.
- If there is a transition $(w * w', hp[k \mapsto h], s, tp', qp') \xrightarrow{m}_l (w_0, hp_0, s_0, tp_0, qp_0)$, the case where $l \neq k$ is handled by lemma 5.11.

  If $l = k$, we know from the operational semantics and the definition of $\xrightarrow{n'}_k$ that $w_0 = w*w', s_0 = s, tp_0 = tp[k \mapsto \texttt{skip}], qp_0 = qp$. Furthermore, from (4), we know that $(w*w')(\llbracket E \rrbracket_s) = Lock(Q)$ for some Q, and there exists $(h_1, n') \in App(Q)(w * w')$ (6) such that $h = h_1 *_{w*w'} h_2$ (7) and $hp_0 = hp[k \mapsto h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], \Omega \mapsto hp(\Omega) *_{w*w'} h_1]$ (7). We want to prove $safe_{min(m,n,j-1)}(w * w', hp_0, s, tp_0, qp')$.

  From (4), $\exists h_3, h_4$ such that $(h_3, n) \in \llbracket Locked(E,P) \rrbracket^k_s(w * w')$ (8) and $(h_4, n) \in \llbracket P \rrbracket^k_s(w * w')$ (9).

  From (8) and the interpretation of $Locked(E,P)$, we get that $App(Q)(w * w') =^{n-1} \llbracket P \rrbracket^k_s(w * w')$. From (6), $(h_1, min(m, n-1)) \in \llbracket P \rrbracket^k_s(w * w')$.

  From (9), $(h_4, min(m, n-1)) \in \llbracket P \rrbracket^k_s(w * w')$. Since the resource invariants are required to be precise, we can conclude that $h_4 = h_1$, and further than $h_3 = h_2$.

  Since now $(h_2, min(m, n-1)) \in \llbracket Locked(E,P) \rrbracket^k_s(w * w')$, we get that $(h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], min(m, n-1)) \in \llbracket Ex(E,P) \rrbracket^k_s(w * w')$ (10).

  We can then unfold the safety definition one last time. All cases are trivial, with one exception: we now have $tp_0(k) = \texttt{skip}$. From (10), we get that $(h_2[\llbracket E \rrbracket_s \mapsto \mathcal{U}], min(m, n-1, j-1)) \in \llbracket Ex(E,P) \rrbracket^k_s(w * w') = qp'(k)$. We also know that there will be no more reduction in thread k, so all reduction cases are handled by lemma 5.11, which concludes our proof.

□

A.4 Lock creation

The proof rule for creating a lock is

$$\frac{}{\vdash \{E \mapsto \_\}\texttt{make\_lock\_P(E)}\{\mathit{Locked}(E, P)\}}$$

Let's assume $\forall k \leq j, \vDash_k \{E \mapsto \_\}\texttt{make\_lock\_P(E)}\{\mathit{Locked}(E, P)\}$ (1) and let's prove this also holds for $j + 1$.

Let's choose w, hp, s, tp, qp such that $\mathit{safe}_j(w, hp, s, tp, qp)$ (2) holds. We pick $k \notin \mathit{dom}(hp)$ and w' such that $w * w'$ is well defined. Let then $(h, n) \in \llbracket E \mapsto \_\rrbracket^k_s(w * w')$ (3), $h \bot \overline{hp}$ and $\mathit{cons}(w * w', hp[k \mapsto h], s, tp', qp')$. We note $qp' = qp[k \mapsto \llbracket \mathit{Locked}(E, P)\rrbracket^k_s]$ and $tp' = tp[k \mapsto \texttt{make\_lock\_P(E)}]$ and want to prove $\mathit{safe}_{\mathit{min}(n,j)-1}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof

- If there was a reduction to `abort`, it would have to be in thread k by the same argument than in the sequential composition proof. This would however contradict the operational semantics and (3), which implies there is no execution fault.
- All threads finished executing satisfy their postconditions follows directly from (2) and $tp'(k) \neq \texttt{skip}$.
- If $(w * w', hp[k \mapsto h], s, tp', qp') \overset{m}{\to}_k (w_0, hp_0, s_0, tp_0, qp_0)$, we have from the definition of the transition relation that $w_0 = (w * w')[\llbracket E\rrbracket_s \mapsto \mathit{Lock}(\mathit{Abs}(\llbracket P\rrbracket^k_s))], hp_0 = hp[k \mapsto h'], s_0 = s, tp_0 = tp[k \mapsto \texttt{skip}]$ and $qp_0 = qp'$, with the notation $h' - h[\llbracket E\rrbracket_s \mapsto k]$. We now want to show $\mathit{safe}_{\mathit{min}(n-1,j-1,m}(w_0, hp[k \mapsto h'], s, tp[k \mapsto \texttt{skip}], qp')$.

  The only interesting case when we unfold the safety definition comes from the fact that $tp[k \mapsto \texttt{skip}](k) = \texttt{skip}$, which means we need to prove $(hp[k \mapsto h'](k), \mathit{min}(n-2, j-2, m-1) = (h', \mathit{min}(n-2, j-2, m-1) \in qp'(k)(w_0) = \llbracket \mathit{Locked}(E, P)\rrbracket^k_s(w_0)$. $\llbracket E\rrbracket_s$ is in the domains of both h' and w_0, and $h'(\llbracket E\rrbracket_s) = k$. Furthermore, for any $w'', p, \mathit{App}(\mathit{Abs}(\llbracket P\rrbracket^k_s))(w'') =^p \llbracket P\rrbracket^k_s(w'')$.

□

A.5 Thread creation

The proof rule for `fork` is

$$\frac{\Gamma, \{R\}f\{S\} \vdash \{P * \mathit{tid}(f, x)\}\texttt{c}\{Q\}}{\Gamma, \{R\}f\{S\} \vdash \{P * R\}\texttt{let x = fork(f) in c}\{Q\}}$$

We assume $\forall j, \Gamma \vDash_j \{P * \mathit{tid}(f, x)\}\texttt{c}\{Q\}$ (1) and $\{R\}f : \texttt{cf}\{Q\} \in \Gamma$ (2) (note that Γ here is not exactly the same as in the proof rule, as we "factor in" the triple for f).

We now assume that $\forall k \leq j, \Gamma \vDash_k \{P * R\}$`let x = fork(f) in c`$\{Q\}$ (3) and want to show that $\Gamma \vDash_{j+1} \{P * R\}$`let x = fork(f) in c`$\{Q\}$.

Let's fix w, hp, s, tp, qp such that $safe_j(w, hp, s, tp, qp)$ (4) and $\Gamma \vDash_j \{R\}$`f`$\{S\}$ (5). We now choose $k \notin dom(hp), w'$ such that $w * w'$ is well defined, $(h, n) \in \llbracket P * R \rrbracket_s^k(w * w')$ (6), $h \bot \overline{hp}$ and $cons(w * w', hp[k \mapsto h], s, tp', qp')$ (7). We note $qp' = qp[k \mapsto \llbracket Q \rrbracket_s^k]$ and $tp' = tp[k \mapsto$ `let x = fork(f) in c`$]$ and now want to show $safe_j(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof As in the other proof cases, most of the safety conditions follow directly from (4). The only difficult condition is the case where a reduction happens in thread k:

$(w * w', hp[k \mapsto h], s, tp', qp') \overset{m}{\to}_k (w_0, hp_0, s_0, tp_0, qp_0)$ for some $i \notin dom(tp')$, where $s_0 = s[x \mapsto v], w_0 = w[v \mapsto T(f)], qp_0 = qp'[i \mapsto \llbracket S \rrbracket^i], tp_0 = tp[k \mapsto \mathtt{c}, i \mapsto \mathtt{cf}]$. Furthermore, there exists h_1, h_2 such that $h = h_1 *_{w*w'} h_2, h_2 \in \llbracket R \rrbracket^i(w * w')$ and $hp_0 = hp[k \mapsto h_1[v \mapsto i], i \mapsto h_2]$.

By (6), $\exists h_3, h_4$ such that $h = h_3 *_{w*w'} h_4, (h_3, min(m, n-1, j-1)) \in \llbracket P \rrbracket_s^k(w * w')$ (8) and $(h_4, min(m, j-1, n-1)) \in \llbracket R \rrbracket_s^k(w * w')$ (9). Using precision of R, we can deduce that $h_1 = h_3$ and $h_2 = h_4$.

The key step is now to start from the "base state" (w, hp, s, tp, qp) and first add the thread i, which we can do thanks to (5), (4), (7) and (9), which gives us $safe_{min(m,j-1,n-1)}(w_0, hp[i \mapsto h_2], s_0, tp[i \mapsto \mathtt{cf}], qp[i \mapsto \llbracket S \rrbracket^i])$.

In order to be able to use (1) to add thread k and conclude, we first need to apply the locality property of proposition 4.7 to (8), which gives that $h_1[v \mapsto i] \in \llbracket P * tid(f, x) \rrbracket_{s_0}^k(w_0)$. From there, using the soundness definition, we obtain the desired $safe_{min(m,n-1,j-1)}(w_0, hp_0, s_0, tp_0, qp_0)$.

□

A.6 Thread synchronisation

The proof rule for `join` is

$$\frac{}{\Gamma, \{P\}f\{Q\} \vdash \{tid(f, E)\}\mathtt{join(E)}\{Q\}}$$

We assume $\{P\}f\{Q\} \in \Gamma$ (1) and $\forall k \leq j, \Gamma \vDash_j \{tid(f, E)\}$`join(E)`$\{Q\}$ (2).

Let w, hp, s, tp, qp be fixed such that $safe_j(w, hp, s, tp, qp)$ (3). We choose $k \notin dom(hp), w'$ such that $w * w'$ is well defined. Let $(h, n) \in \llbracket tid(f, E) \rrbracket_s^k(w * w')$ (4), $h \bot \overline{hp}$ and $cons(w * w', hp[k \mapsto h], s, tp', qp')$ (5).

We note $qp' = qp[k \mapsto \llbracket Q \rrbracket_s^k], tp' = tp[k \mapsto$ `join(E)`$]$, and want to show $safe_{min(j,n)}(w * w', hp[k \mapsto h], s, tp', qp')$.

Proof

- Hypothesis (3) implies that if a thread reduces to `abort`, it has to be k. From (4) and (7), we know there is a i such that $h(\llbracket E \rrbracket_s) = i$ and $i \in dom(tp)$. The operational semantics allows us to conclude that thread k does not reduce into `abort`.

- We only show the proof for the first of the two kinds of "join" reductions. If $(w * w', hp[k \mapsto h], s, tp', qp') \overset{m}{\rightarrow}_k (w_0, hp_0, s_0, tp_0, qp_0)$, we know from the definition of the reduction relation and (7) that there exists some $i = h(\llbracket E \rrbracket_s)$ and that $w_0 = (w * w')|_{dom(w*w') \setminus \{\llbracket E \rrbracket_s\}}, s_0 = s, tp_0 = tp[k \mapsto \texttt{skip}]|_{dom(tp') \setminus \{i\}}$ and $qp_0 = qp'|_{domqp' \setminus \{i\}}$. Furthermore, $h = h_1 *_{w*w'} [\llbracket E \rrbracket_s \mapsto i]$ for some h_1 and $hp_0 = hp[k \mapsto hp(k) *_{w*w'} h_1]|_{domhp' \setminus \{i\}}$.

  We prove the desired $safe_{min(m,n-1,j-1)}(w_0, hp_0, s, tp_0, qp_0)$ directly by unfolding the safety definition. The only interesting case arises from $tp_0(k) = \texttt{skip}$. Since there was a reduction, we know that $tp(i) = \texttt{skip}$, which from (1) and (3) gives that $(hp(i), j-1) \in \llbracket Q \rrbracket_s^i(w)$.

  From the side condition on the `join` proof rule, we get $(hp(i), j-1) \in \llbracket Q \rrbracket_s^k(w)$ (since the interpretation of *Locked* is the only one making use of the thread identifier). Using the locality property (proposition 4.7), $(hp(i), j-1) \in \llbracket Q \rrbracket_s^k(w_0)$. Finally, we can weaken this into the desired $(hp_0(k), min(n,j) - 2) \in \llbracket Q \rrbracket_s^k(w_0)$.

□