Type-Based Safe Resource Deallocation for Shared-Memory Concurrency

Kohei Suenaga
Kyoto University
ksuenaga@kuis.kyoto-u.ac.jp

Ryota Fukuda
Kyoto University
rfukuda@kuis.kyoto-u.ac.jp

Atsushi Igarashi
Kyoto University
igarashi@kuis.kyoto-u.ac.jp

Abstract

We propose a type system to guarantee safe resource deallocation for shared-memory concurrent programs by extending the previous type system based on *fractional ownerships*. Here, safe resource deallocation means that memory cells, locks, or threads are not left allocated when a program terminates. Our framework supports (1) fork/join parallelism, (2) synchronization with locks, and (3) dynamically allocated memory cells and locks. The type system is proved to be sound. We also provide a type inference algorithm for the type system and a prototype implementation of the algorithm.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verifications; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Theory, Verification

Keywords fork/join parallelism, fractional ownerships, memory leak, race freedom, safe resource deallocation, shared-memory concurrency, SMT solver, type inference, type systems

1. Introduction

Safe resource deallocation is a crucial matter in programming languages with manual resource (e.g., memory and files) management such as C and C++. Failing to dispose a created resource, or accessing an already disposed resource could lead to fatal errors[1]. For example, in the C language, a memory leak occurs if a programmer forgets to deallocate a memory cell allocated by `malloc`. In order to address this problem, various formal verification methodologies [8, 13, 14, 16, 18] have been proposed. Most of them are, however, for *sequential* programs.

Our goal is to establish a technique to statically check safe resource deallocation for *concurrent* programs written in the C language. Not only is it harder to reason about concurrent programs, but also there are more kinds of resources to be considered in concurrent programming languages than sequential languages. For example, IEEE standards describe `pthread_join()`, a function for joining a spawned thread, and `pthread_mutex_destroy()`, a function for destroying a mutex, as follows [9]:

> The `pthread_join()` ... *should eventually be called for every thread that is created* [emphasis added] ... so that storage associated with the thread may be reclaimed.
>
> The `pthread_mutex_destroy()` function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. ... *the results of ... referencing the object after it has been destroyed are undefined* [emphasis added].
>
> It shall be safe to destroy an initialized mutex that is unlocked. *Attempting to destroy a locked mutex results in undefined behavior* [emphasis added].

As a first step towards our goal, we will investigate three basic types of resources—memory cells, locks, and threads—in this paper, though there are other kinds of resources than threads and locks (e.g., barriers and reader/writer locks). The following example written in a C-like language describes the language features we will deal with.

EXAMPLE 1. *The* `main` *function in Figure 1 allocates two memory cells on heap, sets the pointers to the cells to* `p` *and*

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10...$10.00

[1] Some of the modern operating systems automatically reclaim resources that have not been not deallocated appropriately. However, we believe that safe resource deallocation is an important issue because not all operating systems have such advanced feature.

```
main() {                    loop(p, q, l) {
  p = malloc();               n = 10;
  q = malloc();               while (n > 0) {
  l = newlock();                acquire(l);
  child =                       *p = 0;
    fork(loop(p,q,l));          release(l);
  loop(p, q, l);                printf("*q=%d",*q);
  wait(child);                  --n;
  freelock(l);                }
  free(p);                  }
  free(q);
}
```

Figure 1. An example of shared-memory concurrent programs.

q*, and creates a lock* l*. Then the program spawns a new thread that executes the body of* loop *inside. The parent and the child threads both write to* *p *and read from* *q *repeatedly. In order to avoid race, they use the lock* l*, which is shared by them. Accesses to* *q *are not guarded because they only read from* *q*. After the child thread terminates, the parent thread reclaims the thread ID (*wait(child)*), deallocates the lock (*freelock(l)*), and then the memory cells (*free(p) *and* free(q)*).*

This program includes (1) dynamically spawned threads, (2) dynamically allocated locks and memory cells, (3) synchronization using locks, and (4) deallocation of threads, locks, and memory cells.

Our approach is to extend the previous flow-sensitive type system of Suenaga and Kobayashi [13] to concurrency. The basic idea of their previous type system is to assign a *fractional ownership* to each pointer-type constructor. A fractional ownership is a rational number that represents capability/obligation on the usage of the pointers. For example, type $\mathbf{int}\ \mathbf{ref}_1$ is the type for pointers to an integer that can be used for reading/writing and should be used for deallocation before termination. Type $\mathbf{int}\ \mathbf{ref}_0$ is for pointers that cannot be used at all and type $\mathbf{int}\ \mathbf{ref}_{0.5}$ is for read-only pointers that should be used for deallocation. By checking that no ownership is left after the execution of a program, the type system guarantees safe memory deallocation for sequential programs.

Our contribution is summarized as follows.

- We propose a type system for safe resource deallocation for a programming language in which a programmer has to manually deallocate resources (i.e., memory cells, locks, and threads). Our type system supports (1) dynamic creation of threads, locks, and memory cells and (2) synchronization using locks.
- We prove that the type system is sound—the execution of a well-typed program indeed leaves no resources when it terminates.
- We provide a type inference algorithm for the type system, and an implementation of the algorithm. Readers can try the implementation from http://www.fos.kuis.kyoto-u.ac.jp/~rfukuda/freesafety-con/.

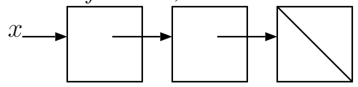
Our type system also guarantees race-freedom and that, for each operation to acquire a lock, there is exactly one corresponding release operation before termination. Race-freedom is naturally achieved by the property inherited from the previous type system that there is at most one pointer that a program can use for writing to each memory cell. Though it is not our main purpose to guarantee these properties, they are interesting on their own because they are important in deadlock-freedom analysis of concurrent languages with non-block-structured lock primitives [12].

The rest of the paper is organized as follows. We first review the previous type system for a sequential language in Section 2 and present an overview of our extension in Section 3. Then, Section 4 defines the language $\mathcal{L}$, a concurrent language with manual resource management, and the safety properties we are to guarantee by the type system. Section 5 introduces the type system and sketches a proof of type soundness, which is followed by a type inference algorithm described in Section 6. After discussing related work in Section 7, Section 8 concludes the paper. For readability, we defer several definitions and detailed proofs to Appendix.

2. Review of Fractional Ownerships

This section briefly reviews the type system of Suenaga and Kobayashi [13]. As mentioned in Section 1, the basic idea of Suenaga and Kobayashi's type system is to assign a *fractional ownership* f (represented by a rational number in $[0, 1]$) to each pointer type constructor $\tau\ \mathbf{ref}_f$. An ownership represents *capability* about how the pointer is used in a program: $f = 1$ means full access capability—capability for reading from, writing to, and deallocate the memory cell that the pointer points to; $f = 0$ means no access capabilities; and other rational numbers between 0 and 1 mean capability only to read. It also represents *obligation*: a non-zero ownership means obligation for deallocation. (A rational number less than 1 does not give capability to deallocate, however. So, ownerships held by aliases have to be merged to recover the full ownership 1 before deallocation.)

EXAMPLE 2. *For the following heap structure, where a reference x points to the first cell,*



if x has type $\tau\ \mathbf{ref}_0\ \mathbf{ref}_1$, the program can read from/write to and has to deallocate the first cell through x, while the program cannot access the second cell through x. The type system also ensures that an alias of the pointer to the second cell exists and the second cell will be deallocated through it.

A type judgment in this type system is of the form $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$, where s is a command, Θ records types of top-level functions, and Γ and Γ' record types of variables. The judgment $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$ intuitively means "under the assumption that (1) each function has the type described in Θ and that (2) each variable has the type described in Γ, the ownerships left after s terminates are described as in Γ'." We call Γ the *pre type environment* and Γ' the *post type environment* of the judgment.

For example, consider a command $\mathbf{let}\ x = *y\ \mathbf{in\ skip}$, which binds x to the value stored in the memory cell that y points to (and does nothing). Then, $\Theta; y : \mathbf{int\ ref}_{0.5} \vdash \mathbf{let}\ x = *y\ \mathbf{in\ skip} \Rightarrow y{:}\mathbf{int\ ref}_{0.5}$ is a valid type judgment because y is given a non-zero ownership to read through. Unlike reading, however, the full ownership is required for writing operation into a memory cell. So, $\Theta; x{:}\mathbf{int\ ref}_f, y{:}\mathbf{int} \vdash *x \leftarrow y \Rightarrow x : \mathbf{int\ ref}_f, y : \mathbf{int}$ (where $*x \leftarrow y$ is a command to store the value of y to the memory cell that x points to) is valid only if $f = 1$. Similarly, deallocation also requires the full ownership; moreover, the ownership becomes zero after deallocation, because the pointer becomes dangling. So, $\Theta; x : \mathbf{int\ ref}_1 \vdash \mathbf{free}(x) \Rightarrow x : \mathbf{int\ ref}_0$ is valid, whereas neither $\Theta; x : \mathbf{int\ ref}_1 \vdash \mathbf{free}(x) \Rightarrow x : \mathbf{int\ ref}_1$ nor $\Theta; x : \mathbf{int\ ref}_{0.5} \vdash \mathbf{free}(x) \Rightarrow x : \mathbf{int\ ref}_0$ is.

One interesting feature of the type system is that ownerships can be transferred to aliases of the same pointer. For example, $\Theta; x : \mathbf{int\ ref}_1 \vdash \mathbf{let}\ y = x\ \mathbf{in\ free}(y) \Rightarrow x : \mathbf{int\ ref}_0$ is a valid type judgment. In fact, the pre type environment for $\mathbf{free}(y)$ is $x : \mathbf{int\ ref}_0, y : \mathbf{int\ ref}_1$ due to the transfer of the ownership from x to y at $\mathbf{let}\ \ x = y\ \mathbf{in}$. Although this example shows transfer of the whole ownership that x holds to y, the typing rule for $\mathbf{let}\ y = x\ \mathbf{in}\ s$ allows also *partial* transfer—that is, the original ownership of x can be "split" into two fractions for x and y in s. For example, $\mathbf{int\ ref}_1$ can be split into $\mathbf{int\ ref}_{0.5}$ and $\mathbf{int\ ref}_{0.5}$, which makes the following type judgment valid:

$$\Theta; x : \mathbf{int\ ref}_1 \vdash \begin{array}{l}\mathbf{let}\ y = x\ \mathbf{in\ let}\ z = *x\ \mathbf{in} \\ \mathbf{let}\ w = *y\ \mathbf{in}\ \ldots\end{array} \Rightarrow \Gamma'$$

Here, the pre type environment for $\mathbf{let}\ z = *x\ \mathbf{in}\ \ldots$ is $x : \mathbf{int\ ref}_{0.5}, y : \mathbf{int\ ref}_{0.5}$, making it possible to read from both x and y.[2]

For a program, which consists of a set of top-level functions and a command (which is considered as the body of the `main` function), the type system requires the main command to be typed under the *empty* pre and post type environments. The empty post type environment means that no ownerships are left—or, all obligations of deallocation are fulfilled—after the execution of the main command. This way, safe memory deallocation is guaranteed.

[2] In order to deallocate the memory cell pointed to by x, the split ownerships have to be "merged" to recover the full ownership. The type system uses must-alias information to merge ownerships. See Section 5 for more details.

3. Extension to Concurrency

The discussion in Section 1 and the program in Example 1 lead us to the following observations.

1. We need to guarantee safe deallocation not only of memory cells but also of locks and threads.
2. The type system should be able to allow different threads to access shared memory cells when locks are properly used to avoid racy accesses. For example, in the program in Example 1, two threads use the lock `l` to control write accesses to `*p`.

We address the first observation by *ownerships on locks and thread IDs*, and the second by *ownership transfer via procurable type environments*, as described below.

Ownerships on locks and thread IDs: In order to deal with locks and threads as resources that have to be deallocated, our type system introduces two new types: $\mathbf{lock}$ and $\mathbf{tid}$. As we have pointed out above, the type system also needs to ensure that they are safely deallocated. To this end, the type system assigns ownerships to lock types and thread types as well as pointer types. The type system will guarantee safe deallocation of these kinds of resources by the same idea as before: by ensuring that there is no ownership left at the end of a program.

In order to avoid a lock to be deallocated while it is held (see the quotation from [9] in Section 1), we also use ownerships to express obligation to release locks. Thus, a lock type comes with *two* ownerships. One is *lock ownership*, which expresses capability to access the lock and obligation to deallocate the lock. The other is *release ownership*, which expresses obligation to release the lock. On the one hand, a lock can be acquired when the ownership of the lock is non-zero and the release ownership is 0; after the acquire operation, the release ownership becomes 1, which means the lock has to be released afterwards. On the other hand, a lock can be released when the ownership for the lock is non-zero and the release ownership is 1; after the release operation, the release ownership becomes 0. The type system will use release ownership also to prevent a thread from acquiring (or releasing) a lock twice without releasing (or acquiring, respectively).

Ownership transfer via procurable type environments: In order to handle ownership transfer, we further extend lock types with what we call *procurable type environment*. A procurable type environment describes the ownerships granted to a thread while it holds the lock. As a result, a lock type is written $(\Gamma, f_1)\ \mathbf{lock}_{f_2}$, where Γ is a (procurable) type environment, f_1 is release ownership, and f_2 is the ownership of the lock itself.

For example, if a lock l is given type $((x{:}\mathbf{int\ ref}_1), 0)\ \mathbf{lock}_1$, then a thread gets the ownership $\mathbf{int\ ref}_1$ on x as it acquires l, allowing read from and write to x. After the acquire operation, the type of l becomes $((x : \mathbf{int\ ref}_1), 1)\ \mathbf{lock}_1$, where the release ownership is set to 1. When the release ownership is fulfilled by releasing l, the thread will lose the same amount of ownership as it obtained, thus accesses to x will be prohibited. After the release of l, the type of l becomes $((x : \mathbf{int\ ref}_1), 0)\ \mathbf{lock}_1$.

We also extend thread ID types with procurable type environments (written $\Gamma\ \mathbf{tid}_f$) to handle reclamation of ownerships that have been split for a spawned thread. Consider the program in Example 1 again. The two threads read from `q` without using locks, so the ownerships on `q` held by these threads are in $(0, 1)$ (more precisely, the two ownerships f_1 and f_2 satisfy $f_1 > 0, f_2 > 0$ and $f_1 + f_2 = 1$). In order to perform `free(q)`, however, the main thread, which has f_1 on `q`, has to reclaim the other ownership f_2 held by the spawned thread. Our type system considers this is done by `waiting child`—that is, when `child` is given type $(\mathtt{q} : \mathbf{int\ ref}_{f_2})\ \mathbf{tid}_1$ in the pre type environment of `wait(child)`, the typing rule for `wait` ensures $\mathtt{q} : \mathbf{int\ ref}_{f_1+f_2}$ in the post type environment. In general, the procurable type environment in a thread ID type describes the post type environment of the thread, that is, the ownerships left after the thread terminates.

Having these intuitions in mind, we proceed to the formal definitions of our target language and type system.

4. Target Language $\mathcal{L}$

4.1 Syntax

Let $\mathbf{Var}$ be a countably infinite set of variables. The syntax of $\mathcal{L}$ is given in Figure 2. A *value*, denoted by v, is either a variable or $\mathbf{null}$; the former represent memory addresses, while the latter the null pointer. A *thread ID*, denoted by t, is a variable or a special symbol $\star$, which is not a member of $\mathbf{Var}$. The former are IDs of dynamically created threads, while the latter is that of the main thread. *Commands*, ranged over by s, consist of

- sequential commands, including do-nothing ($\mathbf{skip}$), sequential composition ($s_1; s_2$), local binding ($\mathbf{let}\ x = v\ \mathbf{in}\ s$), procedure call ($f(x_1, \ldots, x_n)$), allocation of a new memory cell ($\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s$), deallocation of a memory cell ($\mathbf{free}(x)$), dereferencing a pointer ($\mathbf{let}\ x = *y\ \mathbf{in}\ s$), destructive update of a memory cell through a pointer ($*x_1 \leftarrow x_2$), testing whether a value is $\mathbf{null}$ ($\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$),
- must-alias annotations ($\mathbf{assert}(x_1 = x_2)$ and $\mathbf{assert}(x_1 = *x_2)$), which programmers can use as hints to the type system,
- lock-related primitives, including creation of a lock ($\mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s$), deallocation of a lock ($\mathbf{freelock}(x)$), acquiring a lock ($\mathbf{acq}(x)$), and releasing a lock ($\mathbf{rel}(x)$), and
- thread management primitives, including $\mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2$ for spawning a thread that executes s_1, binding x to the fresh ID of the new thread and executing s_2, and $\mathbf{wait}(x)$ for waiting for the thread with ID x to terminate and deallocating the thread ID.

A few words about the must-alias annotations above: they could be inserted manually or by using static must-alias analyzers. In the current implementation, programmers are required to insert them manually. The type system believes correctness of those annotations and uses them for ownership transfer. If an annotation turns out to be incorrect, the program terminates with an exception—see the operational semantics later.

A meta-variable d denotes a *function definition* and D a set of function definitions, where the names of functions are pairwise distinct. A *program* is a pair of the form (D, s). We write $[v/x]s$ for capture-avoiding substitution of v for x in s. The notion of bound/free variables are defined in a standard manner. We assume, without loss of generality, that each bound variable is different from each other. We also require that the actual arguments $\widetilde{y}$ of function-call command $f(\widetilde{y})$ to be pairwise distinct for a technical reason; if one would like to pass one value as different arguments, $\mathbf{let}$ has to be used beforehand to make aliases .

REMARK 3. *We do not incorporate primitive values such as integers in the current language because such extension is quite straightforward. One of the non-trivial extensions that are not discussed in this paper is C-like structures. Though extension with structures in the sequential setting has been discussed in the previous work [13], we leave it as future work to investigate whether the same extension is possible in the concurrent setting.*

EXAMPLE 4. *The following $\mathcal{L}$ command expresses the body of `main` function in Example 1. We assume that loop is defined elsewhere.*

$$\begin{array}{l}\mathbf{let}\ p = \mathbf{malloc}()\ \mathbf{in}\ \mathbf{let}\ q = \mathbf{malloc}()\ \mathbf{in} \\ \mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in}\ \mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p, q, l))\ \mathbf{in} \\ \quad \mathit{loop}(p, q, l); \mathbf{wait}(c); \mathbf{freelock}(l); \mathbf{free}(p); \mathbf{free}(q)\end{array}$$

In the rest of this section, we assume that D is fixed if not explicitly specified to make the notations less cumbersome.

4.2 Operational semantics

The operational semantics is defined by small-step reduction of configurations which represent execution states.

DEFINITION 5 (Configurations). *A* configuration *is a quadruple (P, H, L, R) where $P \in (\mathbf{Var} \cup \{\star\}) \overset{\mathbf{fin}}{\to} \mathbf{Cmd}$, $H \in \mathbf{Var} \overset{\mathbf{fin}}{\to} \mathbf{Val}$, $L \in \mathbf{Var} \overset{\mathbf{fin}}{\to} \{\top, \bot\}$, $R \in \mathbf{Var} \overset{\mathbf{fin}}{\to} \mathbf{Val}$, where $X \overset{\mathbf{fin}}{\to} Y$ is the set of partial functions from X to Y whose domain is finite, and the domains of P, H, L and R are pairwise disjoint. We use the meta-variable Conf for configurations. We write $P(\mathit{Conf})$, $R(\mathit{Conf})$, $H(\mathit{Conf})$ and $L(\mathit{Conf})$ for P, R, H and L in Conf, respectively. $P(\mathit{Conf})$ is called* thread pool *of Conf, $H(\mathit{Conf})$* heap *and $L(\mathit{Conf})$* lock environment.

$$\begin{array}{llll}
x, f, l, h, \ldots \text{ (Variables)} & \in \mathbf{Var} & & \\
v \text{ (Values)} & \in \mathbf{Val} & ::= & x \mid \mathbf{null} \\
t \text{ (Thread IDs)} & & ::= & x \mid \star \\
s \text{ (Commands)} & \in \mathbf{Cmd} & ::= & \mathbf{skip} \mid s_1; s_2 \mid \mathbf{let}\ x = v\ \mathbf{in}\ s \mid f(x_1, \ldots, x_n) \\
 & & \mid & \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \mid \mathbf{free}(x) \mid \mathbf{let}\ x = *y\ \mathbf{in}\ s \mid *x_1 \leftarrow x_2 \\
 & & \mid & \mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \mathbf{assert}(x_1 = x_2) \mid \mathbf{assert}(x_1 = *x_2) \\
 & & \mid & \mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s \mid \mathbf{freelock}(x) \mid \mathbf{acq}(x) \mid \mathbf{rel}(x) \\
 & & \mid & \mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2 \mid \mathbf{wait}(x) \\
d \text{ (Function definitions)} & \in \mathbf{Def} & ::= & f(x_1, \ldots, x_n) = s \\
E \text{ (Evaluation contexts)} & & ::= & [\,] \mid E; s
\end{array}$$

Figure 2. Syntax of $\mathcal{L}$.

A thread pool P records the thread IDs and the statements currently being executed. Recall that the symbol $\star$ stands for a special thread ID for the main thread. A heap H represents the current state of memory. It is essentially a directed graph whose vertices are memory locations. L records the states of locks: $L(x) = \top$ means that x is being held by a thread and $L(x) = \bot$ not being held. R represents a register file, which maps a register name to its value. Note that we use variables for thread IDs (other than $\star$), locations, lock IDs and register names.

Notations. Let X be a map. We write $dom(X)$ for the domain of X. By abuse of notation, we write $dom(D)$ for the set of the function names defined in D. We write $X[x \mapsto a]$ for the map defined by $dom(X[x \mapsto a]) = dom(X) \cup \{x\}$ and $X[x \mapsto a](x) = a$ and $X[x \mapsto a](y) = X(y)$ when $x \neq y$. We use a tilde $\widetilde{(\cdot)}$ to denote a sequence and write $X \setminus \{\widetilde{x}\}$ for the map whose domain is $dom(X) \setminus \{\widetilde{x}\}$ and defined by $(X \setminus \{\widetilde{x}\})(z) = X(z)$.

DEFINITION 6 (Small-Step Reduction). *The relations*

$$\begin{array}{ll}
(s, H, L, R) \rightsquigarrow (s', H', L', R') & (s, H, L, R) \rightsquigarrow \mathfrak{E} \\
(P, H, L, R) \rightsquigarrow (P', H', L', R') & (P, H, L, R) \rightsquigarrow \mathfrak{E},
\end{array}$$

where $\mathfrak{E} \in \{\mathbf{NullEx}, \mathbf{AssertFail}, \mathbf{Error}\}$*, are the least relations that satisfy the rules in Figures 3, 4 and 5. (We use a single symbol* $\rightsquigarrow$ *for all the relations by abuse of notation.) We write* $\rightsquigarrow^*$ *for the reflexive transitive closure of* $\rightsquigarrow$*.*

The relation $(s, H, L, R) \rightsquigarrow (s', H', L', R')$ represents small-step reduction inside a thread. The rules are mostly straightforward. In E-FREE, location $R(x)$ is removed from the heap if $R(x)$ is a valid location. If $R(x) = \mathbf{null}$, nothing happens; here, we follow the convention of the C language. E-MALLOC creates a memory cell allocated at a fresh location h and assigns it to a (fresh) register z and extends R with $z \mapsto h$ and H with $h \mapsto v$ where v is an arbitrary value[3]. E-ASSERTEQ and E-ASSERTDEREF represent the cases where the assertions actually hold—an assertion is no-op when it holds. E-NEWLOCK creates a new lock l and assigns it to a new register z and extends L with $l \mapsto \bot$ and R with $z \mapsto l$. E-FREELOCK removes the lock $R(x)$ from L if the lock is still alive and not held by any threads.

The relation $(P, H, L, R) \rightsquigarrow (P', H', L', R')$ represents reduction that pertains to the thread-related primitives. E-FORK generates a fresh thread ID z'' and assigns it to a fresh register z'; a thread pool is extended with $z'' \mapsto s_1$. E-WAIT removes $R(x)$ from P if $R(x)$ is a valid thread ID and $P(R(x))$ has already finished its execution. Note that $R(x)$ cannot be $\star$. E-ACQ confirms that $R(x)$ is a valid lock and no thread currently holds this lock, and turns its lockstate to $\top$. E-REL is the converse of E-ACQ. E-PROC arbitrarily chooses one thread in P and conducts one-step reduction.

The relation $(P, H, L, R) \rightsquigarrow \mathfrak{E}$ where $\mathfrak{E} \in \{\mathbf{NullEx}, \mathbf{AssertFail}, \mathbf{Error}\}$ represents reduction to an *exceptional state*. Here, $\mathbf{NullEx}$ represents null-pointer accesses, $\mathbf{AssertFail}$ assertion errors and $\mathbf{Error}$ accesses to dangling resources. As we see below, we *do not* treat $\mathbf{NullEx}$ nor $\mathbf{AssertFail}$ as *erroneous states*; our type system does not exclude the possibility of null-pointer accesses and assertion errors. It means that safe deallocation is guaranteed only when assertions, which could be inserted by must-alias analysis, are correct and the program does not perform null-pointer accesses.

4.3 Safety property

We formalize the safety property to be satisfied by a well-typed program below. Informally, a well-typed program does not cause $\mathbf{Error}$ or race-condition; or does not leak resources, either. A program is said to leak resources when its execution terminates but the heap, the lock environment, or the thread pool at termination is not empty (except for the main thread). We first define race condition.

DEFINITION 7 (Reading from / Writing to). *Thread t in configuration $Conf$ such that $t \in dom(P(Conf))$ is said to be* reading from h *if $P(Conf)(t)$ is syntactically equal to $E[\mathbf{let}\ y = *x\ \mathbf{in}\ s]$ and $R(Conf)(x) = h$ for some E, x, y and s. Thread t in configuration $Conf$ is said to be* writing

[3] Following the convention of the C language, we assume nothing on the value contained in the newly allocated memory cell.

$$\boxed{(s, H, L, R) \rightsquigarrow (s', H', L', R')}$$

$$(\mathbf{skip}; s, H, L, R) \rightsquigarrow (s, H, L, R) \quad \text{(E-Skip)}$$

$$\begin{array}{c}(*x \leftarrow y, H[R(x) \mapsto v], L, R) \rightsquigarrow \\ (\mathbf{skip}, H[R(x) \mapsto R(y)], L, R)\end{array} \quad \text{(E-Assign)}$$

$$\frac{R(x) \in dom(H) \cup \{\mathbf{null}\}}{(\mathbf{free}(x), H, L, R) \rightsquigarrow (\mathbf{skip}, H \backslash R(x), L, R)} \quad \text{(E-Free)}$$

$$\frac{z \text{ and } h \text{ are fresh} \qquad v \in \mathbf{Var} \cup \{\mathbf{null}\}}{\begin{array}{c}(\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, H, L, R) \rightsquigarrow \\ ([z/x]s, H[h \mapsto v], L, R[z \mapsto l])\end{array}} \quad \text{(E-Malloc)}$$

$$\frac{z \text{ is fresh}}{(\mathbf{let}\ x = v\ \mathbf{in}\ s, H, L, R) \rightsquigarrow ([z/x]s, H, L, R[z \mapsto v])} \quad \text{(E-Let)}$$

$$\frac{z \text{ is fresh}}{\begin{array}{c}(\mathbf{let}\ x = *y\ \mathbf{in}\ s, H[R(y) \mapsto v], L, R) \rightsquigarrow \\ ([z/x]s, H[R(y) \mapsto v], L, R[z \mapsto v])\end{array}} \quad \text{(E-Deref)}$$

$$\frac{R(x) = \mathbf{null}}{(\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, H, L, R) \rightsquigarrow (s_1, H, L, R)} \quad \text{(E-IfNullTrue)}$$

$$(\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, H, L, R[x \mapsto y]) \rightsquigarrow (s_2, H, L, R[x \mapsto y]) \quad \text{(E-IfNullFalse)}$$

$$\frac{f(\widetilde{x}) = s \in D}{(f(\widetilde{y}), H, L, R) \rightsquigarrow ([\widetilde{y}/\widetilde{x}]s, H, L, R)} \quad \text{(E-App)}$$

$$\frac{R(x_1) = R(x_2)}{(\mathbf{assert}(x_1 = x_2), H, L, R) \rightsquigarrow (\mathbf{skip}, H, L, R)} \quad \text{(E-AssertEq)}$$

$$\begin{array}{c}(\mathbf{assert}(x_1 = *x_2), H[R(x_2) \mapsto v], L, R[x_1 \mapsto v]) \rightsquigarrow \\ (\mathbf{skip}, R[x_1 \mapsto v], H[R(x_2) \mapsto v], L, R[x_1 \mapsto v])\end{array} \quad \text{(E-AssertDeref)}$$

$$\frac{z \text{ and } l \text{ are fresh}}{\begin{array}{c}(\mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s, H, L, R) \rightsquigarrow \\ ([z/x]s, H, L[l \mapsto \bot], R[z \mapsto l])\end{array}} \quad \text{(E-Newlock)}$$

$$\begin{array}{c}(\mathbf{freelock}(x), H, L[R(x) \mapsto \bot], R) \rightsquigarrow \\ (\mathbf{skip}, H, L \backslash R(x), R)\end{array} \quad \text{(E-Freelock)}$$

Figure 3. Semantics of $\mathcal{L}$ (1).

$$\boxed{(P, H, L, R) \rightsquigarrow (P', H', L', R')}$$

$$\frac{z' \text{ and } z'' \text{ are fresh}}{(P[t \mapsto E[\mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2]], H, L, R) \rightsquigarrow (P[t \mapsto E[[z'/x]s_2], z'' \mapsto s_1], H, L, R[z' \mapsto z''])} \quad \text{(E-Fork)}$$

$$(P[t \mapsto E[\mathbf{wait}(x)], x' \mapsto \mathbf{skip}], H, L, R[x \mapsto x']) \rightsquigarrow (P[t \mapsto E[\mathbf{skip}]] \backslash \{x'\}, H, L, R[x \mapsto x']) \quad \text{(E-Wait)}$$

$$(P[t \mapsto E[\mathbf{acq}(x)]], H, L[R(x) \mapsto \bot], R) \rightsquigarrow (P[t \mapsto E[\mathbf{skip}]], H, L[R(x) \mapsto \top], R) \quad \text{(E-Acq)}$$

$$(P[t \mapsto E[\mathbf{rel}(x)]], H, L[R(x) \mapsto \top], R) \rightsquigarrow (P[t \mapsto E[\mathbf{skip}]], H, L[R(x) \mapsto \bot], R) \quad \text{(E-Rel)}$$

$$\frac{(s, H, L, R) \rightsquigarrow (s', H', L', R')}{(P[t \mapsto E[s]], H, L, R) \rightsquigarrow (P[t \mapsto E[s']], H', L', R')} \quad \text{(E-Proc)}$$

Figure 4. Semantics of $\mathcal{L}$ (2).

to h *if $P(Conf)(t)$ is syntactically equal to $E[*x \leftarrow v]$ and $R(Conf)(x) = h$ for some E, v and x.*

DEFINITION 8 (Race Condition). *A configuration $Conf$ is in race if there are $h \in dom(H(Conf))$ and $t_1, t_2 \in dom(P(Conf))$ such that $t_1 \neq t_2$ and either (1) t_1 and t_2 are writing to h or (2) t_1 is writing to h and t_2 is reading from h.*

This definition intuitively means that $Conf$ is in race if there are two or more threads that are accessing the same location, and at least one of them is conducting write oper-

$\boxed{(s, H, L, R) \rightsquigarrow \mathfrak{E}, (P, H, L, R) \rightsquigarrow \mathfrak{E}}$

$$\frac{R(y) = \mathbf{null} \qquad s \text{ is } *y \leftarrow x, \text{ or } \mathbf{let}\ x = *y\ \mathbf{in}\ s', \text{ or } \mathbf{assert}(x = *y)}{(s, H, L, R) \rightsquigarrow \mathbf{NullEx}} \quad \text{(E-NULL)}$$

$$\frac{R(x_1) \neq R(x_2)}{(\mathbf{assert}(x_1 = x_2), H, L, R) \rightsquigarrow \mathbf{AssertFail}} \quad \text{(E-ASSERTEQERR)}$$

$$\frac{R(y) \notin \mathit{dom}(H) \cup \{\mathbf{null}\}}{(\mathbf{assert}(x = *y), H, L, R) \rightsquigarrow \mathbf{AssertFail}} \quad \text{(E-ASSERTDEREFERR1)}$$

$$\frac{R(x) \neq v}{(\mathbf{assert}(x = *y), H[R(y) \mapsto v], L, R) \rightsquigarrow \mathbf{AssertFail}} \quad \text{(E-ASSERTDEREFERR2)}$$

$$\frac{\begin{array}{c} s = *x \leftarrow y, \text{ or } \mathbf{let}\ x = *y\ \mathbf{in}\ s, \text{ or } \mathbf{free}(x), \text{ or } \mathbf{assert}(x = *y) \\ R(x) \notin \mathit{dom}(H) \qquad R(x) \neq \mathbf{null} \end{array}}{(s, H, L, R) \rightsquigarrow \mathbf{Error}} \quad \text{(E-DANGPTRACCERR)}$$

$$\frac{\begin{array}{c} s = \mathbf{acq}(x), \text{ or } \mathbf{rel}(x), \text{ or } \mathbf{freelock}(x) \\ R(x) \notin \mathit{dom}(L) \end{array}}{(s, H, L, R) \rightsquigarrow \mathbf{Error}} \quad \text{(E-DANGLOCKACCERR)}$$

$$\frac{R(x) \notin \mathit{dom}(P)}{(\mathbf{wait}(x), H, L, R) \rightsquigarrow \mathbf{Error}} \quad \text{(E-DANGPROCACCERR)}$$

$$\frac{(s, H, L, R) \rightsquigarrow \mathfrak{E}}{(P[t \mapsto E[s]], H, L, R) \rightsquigarrow \mathfrak{E}} \quad \text{(E-PROCERR)}$$

Figure 5. Semantics of $\mathcal{L}$ (3).

ation. This definition has been widely used in race-freedom analysis (e.g.,[1]).

Then, we define termination of a configuration and then the notion of program safety, which the type system will guarantee.

DEFINITION 9 (Termination). *A configuration Conf is in* termination *if and only if $P(\mathit{Conf})(t) = \mathbf{skip}$ for any $t \in \mathit{dom}(P(\mathit{Conf}))$. We write $\mathbf{End}(\mathit{Conf})$ if Conf is in termination.*

DEFINITION 10 (Safe Configuration). *A configuration Conf is* safe *(written $\mathbf{Safe}(\mathit{Conf})$) if and only if (1) $\mathit{Conf} \not\rightsquigarrow \mathbf{Error}$; (2) Conf is not in race; and (3) $\mathbf{End}(\mathit{Conf})$ implies $\mathit{Conf} = (\{\star \mapsto \mathbf{skip}\}, \emptyset, \emptyset, R)$ for some R. We say Conf* leaks resources *if (3) does not hold.*

DEFINITION 11 (Program Safety). *A program (D, s) is* safe *if $(\{\star \mapsto s\}, \emptyset, \emptyset, \emptyset) \rightsquigarrow^* \mathit{Conf}$ implies $\mathbf{Safe}(\mathit{Conf})$. We write $\mathbf{Safe}(D, s)$ if (D, s) is safe.*

REMARK 12. *A deadlocking configuration is not considered in termination. For example,*

$$\mathbf{End}\begin{pmatrix} \{\star \mapsto \mathbf{wait}(x), t_1 \mapsto \mathbf{acq}(y)\}, \\ \emptyset, \\ \{l \mapsto \top\}, \\ \{x \mapsto t_1, y \mapsto l\} \end{pmatrix}$$

does not hold, though the configuration is stuck.

5. Type System

5.1 Types and type environments

The set of ownerships, ranged over by f, is the subset $[0, 1]$ of rational numbers $\mathbb{Q}$. We define types and type environments as follows:

κ (reference types) $\in \{0\}^* \to [0, 1]$
τ (value types) $\in \mathbf{VTyp} ::= \kappa \mid (\Gamma, f_1)\ \mathbf{lock}_{f_2} \mid \Gamma\ \mathbf{tid}_f$
Γ (type environments) $\in \mathbf{Var} \overset{\mathbf{fin}}{\to} \mathbf{VTyp}$

A *reference type* κ is a map from $\{0\}^*$ to the set of ownerships. Here, $\{0\}^*$ is the set of finite sequences, ranged over by π, of the (only) alphabet 0. We assume that every reference type κ satisfies $\kappa(\pi) = 0$ implies $\kappa(\pi 0) = 0$ for any π. (This restriction is needed for soundness [13].) A sequence π intuitively represents the memory cell reached by dereferencing a pointer $|\pi|$ times; if x has type κ and $\kappa(\pi) = f$, it means that a programmer has to respect the ownership f of the value obtained by $|\pi|$-time dereferencing of x following the chain of the pointers, where $|\pi|$ is the length of π[4]. For example, the type of x in Example 2 is $\{\epsilon \mapsto 1, 0 \mapsto 0, 00 \mapsto 0, \ldots\}$.

We write $\mathbf{0}$ for the reference type that is defined by $\mathbf{0}(\pi) = 0$ for any π. Given a reference type κ, we write $\kappa\ \mathbf{ref}_f$ for the reference type defined by $(\kappa\ \mathbf{ref}_f)(\epsilon) = f$ and $(\kappa\ \mathbf{ref}_f)(0\pi) = \kappa(\pi)$.

REMARK 13. *In Suenaga and Kobayashi [13], they introduce a recursive type constructor $\mu\alpha.\kappa$,* interpret *a recursive type as an element in $\{0\}^* \to [0,1]$ (by infinite expansion), and define operations on types in terms of infinite expansions. Our choice of directly using $\{0\}^* \to [0,1]$ simplifies the presentation of the type system. The type inference algorithm in Section 6 uses the recursive type notation as a finite representation of reference types.*

A value type, often called type for short, is either a reference type, a lock type $(\Gamma, f_1)\ \mathbf{lock}_{f_2}$, or a thread ID type $\Gamma\ \mathbf{tid}_f$; we have already explained the syntax and intuition of lock and thread ID types in Section 3. We call f_1 in $(\Gamma, f_1)\ \mathbf{lock}_{f_2}$ *release ownership* and f_2 *lock ownership*.

REMARK 14. *We currently syntactically separate reference types from value types. However, this separation obviously prevents a lock and a thread ID from being stored in heap because this requires a type like $((\Gamma, 0)\ \mathbf{lock}_1)\ \mathbf{ref}_1$, which does not conform the grammar. Relaxing this restriction is left as future work.*

Before explaining typing rules, we need several auxiliary definitions and operations on types and type environments.

DEFINITION 15. *The sets $\mathbf{FV}(\tau)$ and $\mathbf{FV}(\Gamma)$ of* free variables *of a type τ and a type environment Γ, respectively, are defined by:*

$$\mathbf{FV}(\kappa) = \emptyset \quad \mathbf{FV}((\Gamma, f_1)\ \mathbf{lock}_{f_2}) = \mathbf{FV}(\Gamma)$$
$$\mathbf{FV}(\Gamma\ \mathbf{tid}_f) = \mathbf{FV}(\Gamma)$$
$$\mathbf{FV}(\Gamma) = \textstyle\bigcup_{x \in dom(\Gamma)} (\mathbf{FV}(\Gamma(x)) \cup \{x\}).$$

DEFINITION 16. *We write $\Gamma, x : \tau$ for the type environment $\Gamma[x \mapsto \tau]$ if $x \notin \mathbf{FV}(\Gamma)$ and $\mathbf{FV}(\tau) \subseteq dom(\Gamma)$. We say Γ is* well-formed *(written $\mathbf{wf}(\Gamma)$) if Γ can be written as $x_1 : \tau_1, \ldots, x_n : \tau_n$ for some $\widetilde{x}$ and $\widetilde{\tau}$.*

[4] Though using the set of natural numbers in place of $\{0\}^*$ could also work, we adopt the current definition for a future extension to multi-word cells. For example, when cells consist of two words, we can use $\{0, 1\}$ as the alphabet.

The condition "$x \notin \mathbf{FV}(\Gamma)$ and $\mathbf{FV}(\tau) \subseteq dom(\Gamma)$" means that, in order to add a new type declaration to a type environment, the variable has to be fresh and, if the newly added type is a lock/tid type, then its procurable type environment can mention only preceding variables. So, intuitively, well-formedness of a type environment means that bindings can be sorted so that the type of a variable mentions only preceding variables. We assume that every type environment is well formed in what follows.

We next define emptiness of types and type environments. Intuitively, a value of an empty type can be discarded immediately because there is no obligation to deallocate.

DEFINITION 17 (Empty Types/Type Environments). *A type τ is* empty, *written $\mathbf{empty}(\tau)$, if τ is either $(\Gamma, 0)\ \mathbf{lock}_0$ for some Γ, or $\Gamma\ \mathbf{tid}_0$ for some Γ, or a reference type $\mathbf{0}$. A type environment Γ is empty, written $\mathbf{empty}(\Gamma)$, if $\mathbf{empty}(\Gamma(x))$ for any $x \in dom(\Gamma)$.*

We also define summation on types and type environments to formalize splitting and merging ownerships.

DEFINITION 18 (Summation of Types/Type Environments). *Value type $\tau_1 + \tau_2$ is defined by:*

$$\begin{array}{l} \kappa_1 + \kappa_2 = \kappa \\ \quad \textit{where } \kappa(\pi) = \kappa_1(\pi) + \kappa_2(\pi) \textit{ for any } \pi \\ (\Gamma, f_1)\ \mathbf{lock}_{f_2} + (\Gamma, f_3)\ \mathbf{lock}_{f_4} = (\Gamma, f_1 + f_3)\ \mathbf{lock}_{f_2 + f_4} \\ \Gamma\ \mathbf{tid}_{f_1} + \Gamma\ \mathbf{tid}_{f_2} = \Gamma\ \mathbf{tid}_{f_1 + f_2} \end{array}$$

Type environment $\Gamma_1 + \Gamma_2$ is defined by:

$$\begin{array}{l} dom(\Gamma_1 + \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2) \\ (\Gamma_1 + \Gamma_2)(x) = \\ \quad \begin{cases} \Gamma_1(x) + \Gamma_2(x) & (x \in dom(\Gamma_1) \cap dom(\Gamma_2)) \\ \Gamma_1(x) & (x \in dom(\Gamma_1) \backslash dom(\Gamma_2)) \\ \Gamma_2(x) & (x \in dom(\Gamma_2) \backslash dom(\Gamma_1)) \, . \end{cases} \end{array}$$

Note that the sum $\tau_1 + \tau_2$ is defined only if τ_1 and τ_2 are the same kind of types; moreover, when a summand is a lock or thread ID type, their procurable type environments have to be identical. For example $(\{x \mapsto \mathbf{0}\}, 0)\ \mathbf{lock}_1 + (\{x \mapsto \mathbf{0}\}, 1)\ \mathbf{lock}_0 = (\{x \mapsto \mathbf{0}\}, 1)\ \mathbf{lock}_1$. However, none of $(\{x \mapsto \mathbf{0}\}, 0)\ \mathbf{lock}_1 + (\{x \mapsto \mathbf{1}\}, 0)\ \mathbf{lock}_1$, $(\{x \mapsto \mathbf{0}\}, 0)\ \mathbf{lock}_1 + (\{x \mapsto \mathbf{0}, y \mapsto \mathbf{1}\}, 0)\ \mathbf{lock}_1$, or $\mathbf{0} + (\{x \mapsto \mathbf{0}\}, 0)\ \mathbf{lock}_1$ is defined.

5.2 Typing

Let Θ be a function type environment, which is a finite mapping from variables to *function types* of the form $(x_1 : \tau_1, \ldots, x_n : \tau_n) \to (\tau'_1, \ldots, \tau'_n)$. A function type is dependent so that a parameter type can mention a preceding parameter in its procurable type environment. Although it returns no value in our target language, a function (implicitly) returns some ownership held by parameters back to the caller. For example, a function type $(x : \kappa\ \mathbf{ref}_0, y : ((x : \kappa\ \mathbf{ref}_1), 0)\ \mathbf{lock}_1) \to (\kappa\ \mathbf{ref}_0, ((x : \kappa\ \mathbf{ref}_1), 0)\ \mathbf{lock}_1)$

$\boxed{\Gamma \vdash v : \tau}$

$$\frac{\mathbf{empty}(\Gamma)}{\Gamma \vdash \mathbf{null} : \kappa} \quad \text{(T-NULL)} \qquad \frac{\mathbf{empty}(\Gamma)}{\Gamma, x : \tau \vdash x : \tau} \quad \text{(T-VAR)}$$

$\boxed{\Theta; \Gamma \vdash s \Rightarrow \Gamma'}$

$$\Theta; \Gamma \vdash \mathbf{skip} \Rightarrow \Gamma \quad \text{(T-SKIP)} \qquad \frac{\Theta; \Gamma_1 \vdash s_1 \Rightarrow \Gamma_3 \qquad \Theta; \Gamma_3 \vdash s_2 \Rightarrow \Gamma_2}{\Theta; \Gamma_1 \vdash s_1; s_2 \Rightarrow \Gamma_2} \quad \text{(T-SEQ)}$$

$$\frac{\Gamma_1 \vdash v : \tau_1 \qquad \Theta; \Gamma_2, x : \tau_1 \vdash s \Rightarrow \Gamma_3, x : \tau' \qquad \mathbf{empty}(\tau') \qquad x \notin \mathbf{FV}(\Gamma_1)}{\Theta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ x = v\ \mathbf{in}\ s \Rightarrow \Gamma_3} \quad \text{(T-LET)}$$

$$\frac{\begin{array}{c} dom(\Gamma) = dom(\Gamma') = \{y_1, \dots, y_n\} \\ \Gamma(y_i) = [y_1, \dots, y_{i-1}/x_1, \dots, x_{i-1}]\tau_i \text{ for each } i \qquad \Gamma'(y_i) = [y_1, \dots, y_{i-1}/x_1, \dots, x_{i-1}]\tau'_i \text{ for each } i \end{array}}{\Theta; f : (\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'}); \Gamma + \Gamma'' \vdash f(\widetilde{y}) \Rightarrow \Gamma' + \Gamma''} \quad \text{(T-APP)}$$

$$\frac{\Theta; \Gamma_1, x : \mathbf{0}\ \mathbf{ref}_1 \vdash s \Rightarrow \Gamma_2, x : \mathbf{0}}{\Theta; \Gamma_1 \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \Rightarrow \Gamma_2} \quad \text{(T-MALLOC)} \qquad \Theta; \Gamma, x : \mathbf{0}\ \mathbf{ref}_1 \vdash \mathbf{free}(x) \Rightarrow \Gamma, x : \mathbf{0} \quad \text{(T-FREE)}$$

$$\frac{\Theta; \Gamma_1, y : \kappa_1\ \mathbf{ref}_f, x : \kappa_2 \vdash s \Rightarrow \Gamma_2, x : \mathbf{0} \qquad f > 0}{\Theta; \Gamma_1, y : (\kappa_1 + \kappa_2)\ \mathbf{ref}_f \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s \Rightarrow \Gamma_2} \quad \text{(T-DEREF)}$$

$$\Theta; \Gamma, y : \mathbf{0}\ \mathbf{ref}_1, x : \kappa_1 + \kappa_2 \vdash *y \leftarrow x \Rightarrow \Gamma, y : \kappa_1\ \mathbf{ref}_1, x : \kappa_2 \quad \text{(T-ASSIGN)}$$

$$\frac{\Gamma_1(x) = \kappa\ \mathbf{ref}_f \qquad \Theta; \Gamma_1[x \mapsto \kappa_1] \vdash s_1 \Rightarrow \Gamma_2 \qquad \Theta; \Gamma_1 \vdash s_2 \Rightarrow \Gamma_2}{\Theta; \Gamma_1 \vdash \mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \Rightarrow \Gamma_2} \quad \text{(T-IFNULL)}$$

$$\frac{\tau_1 + \tau_2 = \tau'_1 + \tau'_2}{\Theta; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash \mathbf{assert}(x_1 = x_2) \Rightarrow \Gamma, x_1 : \tau'_1, x_2 : \tau'_2} \quad \text{(T-ASSERTEQ)}$$

$$\frac{\kappa_1 + \kappa_2 = \kappa'_1 + \kappa'_2 \qquad f > 0}{\Theta; \Gamma, x_1 : \kappa_1, x_2 : \kappa_2\ \mathbf{ref}_f \vdash \mathbf{assert}(x_1 = *x_2) \Rightarrow \Gamma, x_1 : \kappa'_1, x_2 : \kappa'_2\ \mathbf{ref}_f} \quad \text{(T-ASSERTEQDEREF)}$$

Figure 6. Typing rules for sequential commands.

describes a function that takes a pointer x and a lock to obtain ownership to access x; when it returns the same ownership will be returned to the caller.

As explained in Section 2, a type judgment is of the form $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$, whose intuition has already been given.

DEFINITION 19 (Typing for Commands). *The judgment $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$ is the least relation that is closed under the rules in Figures 6 and 7.*

Typing rules for sequential commands. The typing rules in Figure 6 are rules for values and sequential commands. Rule T-NULL means that a null pointer can be given any reference type. Rules T-NULL and T-VAR require Γ to be empty since variables in it are not used here.

Rules T-SKIP and T-SEQ are straightforward. In T-LET, the type of x in the post type environment has to be empty since ownerships on x should not be left after the scope of x ends here. Similar emptiness conditions are found in typing rules for commands involving variable binding. Rule T-APP expresses standard dependent function application.

In rule T-MALLOC, x is given type $\mathbf{0}\ \mathbf{ref}_1$ in the pre type environment of the body s. Recall the intuition of ownership 1 explained in Section 3 that, when a new cell is allocated, (1) exclusive access through x is granted and (2) obligation to deallocate the cell through x in future is imposed. Since the content of the newly-created cell should not be used until initialization has been done, the type of a value pointed to by x is $\mathbf{0}$. Rule T-FREE is basically the converse of T-MALLOC. The type of x becomes $\mathbf{0}$ in the post type environment to prevent accesses to the deallocated cell.

Rules T-DEREF and T-ASSIGN check if there is enough ownership on the type $\kappa\ \mathbf{ref}_f$ of y in the pre type environment of the conclusion, in order to perform reading ($f > 0$) and writing ($f = 1$), respectively. When a pointer

$\boxed{\Theta; \Gamma \vdash s \Rightarrow \Gamma' \text{ (cont'd.)}}$

$$\frac{\Theta; \Gamma_1, x : (\Gamma_2, 0)\ \mathbf{lock}_1 \vdash s \Rightarrow \Gamma_3, x : \tau \qquad \mathbf{empty}(\tau)}{\Theta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s \Rightarrow \Gamma_3} \quad \text{(T-NEWLOCK)}$$

$$\Theta; \Gamma_1, x : (\Gamma_2, 0)\ \mathbf{lock}_1 \vdash \mathbf{freelock}(x) \Rightarrow (\Gamma_1, x : (\Gamma_2, 0)\ \mathbf{lock}_0) + \Gamma_2 \quad \text{(T-FREELOCK)}$$

$$\frac{f > 0}{\Theta; \Gamma_1, x : (\Gamma_2, 0)\ \mathbf{lock}_f \vdash \mathbf{acq}(x) \Rightarrow (\Gamma_1, x \mapsto (\Gamma_2, 1)\ \mathbf{lock}_f) + \Gamma_2} \quad \text{(T-ACQ)}$$

$$\frac{f > 0}{\Theta; (\Gamma_1, x : (\Gamma_2, 1)\ \mathbf{lock}_f) + \Gamma_2 \vdash \mathbf{rel}(x) \Rightarrow \Gamma_1, x : (\Gamma_2, 0)\ \mathbf{lock}_f} \quad \text{(T-REL)}$$

$$\frac{\Theta; \Gamma_1 \vdash s_1 \Rightarrow \Gamma_1' \qquad \Theta; \Gamma_2, x : \Gamma_1'\ \mathbf{tid}_1 \vdash s_2 \Rightarrow \Gamma_2', x : \tau \qquad \mathbf{empty}(\tau) \qquad x \notin \mathbf{FV}(\Gamma_1)}{\Theta; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2 \Rightarrow \Gamma_2'} \quad \text{(T-FORK)}$$

$$\Theta; \Gamma_1, x : \Gamma_2\ \mathbf{tid}_1 \vdash \mathbf{wait}(x) \Rightarrow (\Gamma_1, x : \Gamma_2\ \mathbf{tid}_0) + \Gamma_2 \quad \text{(T-WAIT)}$$

Figure 7. Typing rules for concurrency-related commands.

$$\frac{\Theta; x_1 : \tau_1, \ldots, x_n : \tau_n \vdash s \Rightarrow x_1 : \tau_1', \ldots, x_n : \tau_n'}{\Theta \vdash f(\widetilde{x}) = s : (x_1 : \tau_1, \ldots, x_n : \tau_n) \to (\tau_1', \ldots, \tau_n')} \quad \text{(T-FUNDEF)}$$

$$\frac{\Theta \vdash f(\widetilde{x}) = s : \Theta(f) \text{ for each } f(\widetilde{x}) = s \in D \qquad \mathit{dom}(\Theta) = \mathit{dom}(D)}{\vdash D : \Theta} \quad \text{(T-FUNENV)}$$

$$\frac{\vdash D : \Theta \qquad \Theta; \emptyset \vdash s \Rightarrow \emptyset}{\vdash (D, s)\ \mathbf{ok}} \quad \text{(T-PROG)}$$

Figure 8. Typing rules for programs.

is derefeneced, the content of the memory cell pointed to by y is copied to x, and so the original ownership of the pointer stored in the memory cell is (partially) transferred to x. The type $\kappa_1 + \kappa_2$ expresses the original ownership and it is split for x. Conversely, in assignment, the assigned value x is copied to the memory cell that y points to, so the ownership on x is split. Moreover, the pointer that y points to should have the empty (pointer) type, because no obligation should be left for the old content before assignment. Ownership splittings at these rules are kept non-deterministic. The implementation in Section 6.4 chooses one of the possible splittings with an SMT solver.

Rule T-IFNULL ensures that the post type environment of the two clauses are the same. The rule also allows the **then** clause to use x as any type because x is **null** in this clause (c.f. T-NULL).

Rules T-ASSERTEQ and T-ASSERTEQDEREF describe how the type system exploits must-alias annotations. For example, in T-ASSERTEQ, the condition $\tau_1 + \tau_2 = \tau_1' + \tau_2'$ makes it possible to shuffle ownerships between x_1 and x_2, and thus enables the type system to transfer a part of ownership from one to the other. The following command (written in slightly changed syntax to save space) shows how this rule works:

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ \mathbf{let}\ y = x\ \mathbf{in} \\ \quad f(*y); f(*x); \mathbf{assert}(x = y); \mathbf{free}(x).\end{array}$$

Without the assertion, this example would not typecheck, because both x and y have a positive ownership and thus the ownership of x is not 1, required to **free**. However, thanks to $\mathbf{assert}(x = y)$, which lets the type system transfer the ownership of y to x, the type of x before $\mathbf{free}(x)$ is $\mathbf{0}\ \mathbf{ref}_1$, which allows **free**. Note that it is not a responsibility of this type system to ensure the correctness of must-alias assertions. If an incorrect assertion is inserted, a program may reduce to **AssertFail** and the type system guarantees nothing.

Typing rules for concurrency-related commands. Figure 7 shows the typing rules for concurrency-related commands.

The first four rules concern lock manipulation. Rule T-NEWLOCK splits the ownerships of the current thread into two (Γ_1 and Γ_2), and "deposit" one of them (Γ_2) in the newly-created lock type as its procurable type environment ($x : (\Gamma_2, 0)\ \mathbf{lock}_1$). The release ownership part of the lock type is 0 because the lock is initially in the state $\bot$. The lock ownership part is 1, which means that only this thread is currently allowed to access this lock and the lock has to be deallocated in future. Rule T-FREELOCK is similar to T-FREE; it also requires the release ownership of x to be 0 because, after executing $\mathbf{freelock}(x)$, the program cannot release the lock. As its post type environment shows, the current thread acquires the procurable type environment that has been deposited to the type of x. Rules T-ACQ and T-REL are straightforward once the meaning of procurable type environments and release ownership is understood. Acquiring a lock borrows ownerships deposited in the lock; the procurable type environment Γ_2 added to the post type environment expresses this. Releasing a lock returns the borrowed ownership back to the lock. To express this, the procurable type environment Γ_2 at the pre type environment is removed at the post type environment. Note also that the release ownerships at these rules are set according to the intuition explained in Section 3: The release ownership of x is 1 at the post type environment of T-ACQ, while it is 0 at T-REL.

The next two rules concern thread manipulation. Rule T-FORK means that the ownerships the current thread holds are split into two (Γ_1 and Γ_2) and one of them (Γ_1) is given to the spawned thread (as its pre type environment). The type of x in the pre type environment of s_2 is $\Gamma_1'\ \mathbf{tid}_1$, where Γ_1' is the post type environment of s_1. It expresses that the ID of the child thread has to be passed to **wait** afterwards, and that the thread that **wait**s the spawned thread will be granted Γ_1'—the ownerships left after the execution of the spawned thread s_1—as expressed by rule T-WAIT. Note that we do not force a spawned thread to use up or preserve the ownerships initially given.

Example. We show a typing example using the command in Example 4. First, let

$$\begin{aligned}\Theta = \mathit{loop}{:}(&p:\mathbf{0}, q:\mathbf{0}\ \mathbf{ref}_{0.5}, l:((p:\mathbf{0}\ \mathbf{ref}_1),0)\ \mathbf{lock}_{0.5})\\ &\to (\mathbf{0}, \mathbf{0}\ \mathbf{ref}_{0.5}, ((p:\mathbf{0}\ \mathbf{ref}_1),0)\ \mathbf{lock}_{0.5}).\end{aligned}$$

The type of *loop* means that the function *loop* takes two references p, which cannot be accessed at first, and q, which is read-only, and one lock l. This lock type means that (1) the lock is not acquired yet; and (2) it gives a full ownership on p, once the lock is acquired. At the end of *loop*, the ownership of these resources are returned unchanged.

The command (we call it s) is typed under the empty pre and post type environments:

$$\Theta;\emptyset \vdash s \Rightarrow \emptyset.$$

Now, let's take a look at main parts of its derivation tree. The first two steps (counted from the root) are as follows:

$$\dfrac{\dfrac{\begin{array}{l}\Theta; p:\mathbf{0}\ \mathbf{ref}_1, q:\mathbf{0}\ \mathbf{ref}_1 \vdash\\ \quad \mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in}\ \cdots \Rightarrow p:\mathbf{0}, q:\mathbf{0}\end{array}}{\begin{array}{l}\Theta; p:\mathbf{0}\ \mathbf{ref}_1 \vdash\\ \quad \mathbf{let}\ q = \mathbf{malloc}()\ \mathbf{in}\ \cdots \Rightarrow p:\mathbf{0}\ \mathbf{ref}_0\end{array}}\ \text{T-MALLOC}}{\Theta;\emptyset \vdash \begin{array}{l}\mathbf{let}\ p = \mathbf{malloc}()\ \mathbf{in}\\ \mathbf{let}\ q = \mathbf{malloc}()\ \mathbf{in}\\ \mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in}\ \cdots \Rightarrow \emptyset\end{array}}\ \text{T-MALLOC}$$

In the pre type environment for $\mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in}\ \cdots$, full ownership (i.e., 1) is assigned to p and q, while no ownership is assigned in the post.

The next rule is T-NEWLOCK. Here l is supposed to guard any access to p, the procurable type environment in the type of l should include $p:\mathbf{0}\ \mathbf{ref}_1$ whereas no ownership is left for p in the pre type environment (called Γ) for the continuation (that is, $\mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p,q,l))\ \mathbf{in}\ \cdots$):

$$\dfrac{\begin{array}{l}\Theta; \overbrace{p:\boxed{\mathbf{0}\ \mathbf{ref}_0}, q:\mathbf{0}\ \mathbf{ref}_1, l:((\boxed{p:\mathbf{0}\ \mathbf{ref}_1}),0)\ \mathbf{lock}_1}^{\Gamma} \vdash\\ \quad \mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p,q,l))\ \mathbf{in}\ \cdots\\ \quad \Rightarrow p:\mathbf{0}, q:\mathbf{0}, l:((p:\mathbf{0}\ \mathbf{ref}_1),0)\ \mathbf{lock}_0\end{array}}{\begin{array}{l}\Theta; p:\boxed{\mathbf{0}\ \mathbf{ref}_1}, q:\mathbf{0}\ \mathbf{ref}_1 \vdash\\ \quad \mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in}\\ \quad \mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p,q,l))\ \mathbf{in}\ \cdots \Rightarrow p:\mathbf{0}, q:\mathbf{0}\end{array}}\ \text{T-NEWLOCK}$$

Let $\Gamma_l = p:\mathbf{0}\ \mathbf{ref}_1$ in what follows.

The next rule is T-FORK. Here, the pre type environment Γ is split into two: one for the continuation of the main thread and one for the spawned thread. In both threads, q is used for reading (without using locks) and l is used for locking, so the ownerships of q and l in both type environments must be greater than 0. In the following derivation step we use $\Gamma_{\frac{1}{2}} = p{:}\mathbf{0}, q{:}\mathbf{0}\ \mathbf{ref}_{0.5}, l{:}(\Gamma_l,0)\ \mathbf{lock}_{0.5}$, which satisfies $\Gamma = \Gamma_{\frac{1}{2}} + \Gamma_{\frac{1}{2}}$:

$$\dfrac{\begin{array}{l}\Theta; \Gamma_{\frac{1}{2}}, c:\boxed{\Gamma_{\frac{1}{2}}}\ \mathbf{tid}_1 \vdash \mathit{loop}(p,q,l);\cdots\\ \qquad \Rightarrow p:\mathbf{0}, q:\mathbf{0}, l:(\Gamma_l,0)\ \mathbf{lock}_0, c:\Gamma_{\frac{1}{2}}\ \mathbf{tid}_0\\ \Theta; \Gamma_{\frac{1}{2}} \vdash \mathit{loop}(p,q,l) \Rightarrow \boxed{\Gamma_{\frac{1}{2}}}\end{array}}{\begin{array}{l}\Theta; \Gamma \vdash \mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p,q,l))\ \mathbf{in}\ \cdots\\ \qquad \Rightarrow p:\mathbf{0}, q:\mathbf{0}, l:(\Gamma_l,0)\ \mathbf{lock}_0\end{array}}\ \text{T-FORK}$$

Also it is important to notice that the post type environment for the new thread body and the procurable type environment for the thread ID c agree, as shaded boxes show.

Figure 9 shows the pre type environment for each subcommand as comments. We can observe that

1. after $\mathbf{wait}(c)$, the procurable type environment $\Gamma_{\frac{1}{2}}$ in c's type is merged and so q's ownership is 1 in the post type environment;
2. similarly, after $\mathbf{freelock}(l)$, the procurable type environment Γ_l in l's type is merged and so p's ownership is 1 in the post type environment.

(Note that when two commands s_1 and s_2 are combined with ';', the pre type environment for s_2 is the post for s_1.)

Typing rules for programs. A type judgment for a program is of the form $\vdash (D, s)$ $\mathbf{ok}$, which means that, if a program starts its execution from the main command s, the program respects types and ownerships in any scheduling. It is defined via typing for functions.

DEFINITION 20 (Typing for Programs). $\vdash (D, s)$ $\mathbf{ok}$ *is the least relation that satisfies the rules in Figure 8.*

Rule T-FUNDEF is straightforward. The condition that a parameter can depend on preceding parameters is guaranteed by the formation of type environments (recall that $\Gamma, x : \tau$ is defined only when $x \notin \mathbf{FV}(\Gamma) \cup \mathbf{FV}(\tau)$). Rule T-FUNENV is essentially a standard typing rule for mutually recursive functions.

Rule T-PROG checks that (1) the functions defined in D are well-typed and the types of those functions are described by some Θ, and (2) the main command s is well-typed under Θ and the empty pre and post type environments.

5.3 Soundness of the type system

The type soundness theorem is stated as expected:

THEOREM 21 (Type Soundness). *If* $\vdash$ (D, s) $\mathbf{ok}$*, then* $\mathbf{Safe}(D, s)$.

This theorem is proved in the standard manner: subject reduction and lack of immediate errors. For subject reduction, we define typing for configurations by using an auxiliary definition.

DEFINITION 22. *The function* $\mathbf{PTE}$ *takes a process type and returns its PTE part; it is defined by* $\mathbf{PTE}(\Gamma\ \mathbf{tid}_f) = \Gamma$.

DEFINITION 23 (Typing for Configurations). *The typing relation for configurations* $\Theta; \Gamma \vdash_D$ *Conf* $\mathbf{ok}$ *is defined by the rule in Figure 10.*

Rule T-CONFIG basically imposes that each thread respects the function type environment Θ and its pre type environment. The highlights of the rule are as follows.

- The post type environment of the main thread $\star$ has to be empty, which is crucial for resource-leak-freedom.
- The condition $\mathbf{ConOwn}(\mathit{Conf}, \Gamma)$, which essentially means that, for each thread ID, lock, and pointer, the sum of ownerships held by the threads is exactly 1, so that every resource is eventually deallocated before termination.
- The condition $\mathbf{acyclic}(\widetilde{t}, \widetilde{\Gamma'}, R)$ imposes that there is no "circular dependency" among the threads in Conf. We need this condition to ensure that the thread pool becomes $\{\star \mapsto \mathbf{skip}\}$ leaving no other threads when the program terminates.
- The last conditions means that, for each binding of the form $x : \Gamma'\ \mathbf{tid}_f$ in Γ, where the value of x is the ID of i-th thread, Γ' is equal to the post type environment Γ'_i of s_i modulo empty types. This condition is necessary for a thread not to die without returning all the non-empty ownerships.

The precise definitions of $\mathbf{ConOwn}$ and $\mathbf{acyclic}$ are given in Appendix .

Theorem 21 is proved by showing (1) the initial configuration is well-typed (Lemma 24), (2) well-typedness is preserved by reduction (Lemma 25) and (3) a well-typed configuration is safe (Lemma 26). Then, the theorem is easily proved.

LEMMA 24 (Initial Configuration is Well Typed). *If* $\vdash D : \Theta$ *and* $\Theta; \emptyset \vdash s \Rightarrow \emptyset$*, then* $\Theta; \emptyset \vdash_D (\{\star \mapsto s\}, \emptyset, \emptyset, \emptyset)$ $\mathbf{ok}$.

LEMMA 25 (Subject Reduction for Configurations). *If* $\Theta; \Gamma \vdash_D$ *Conf* $\mathbf{ok}$ *and Conf* $\rightsquigarrow$ *Conf'*, *then there exists* Γ' *such that* $\Theta; \Gamma' \vdash_D$ *Conf'* $\mathbf{ok}$.

LEMMA 26 (Immediate Safety). *If* $\Theta; \Gamma \vdash_D$ *Conf* $\mathbf{ok}$*, then* $\mathbf{Safe}(\mathit{Conf})$.

6. Type inference

We present a constraint-based type inference algorithm for the type system. The algorithm takes a simply typed program, generates a set of constraints for the program to be well-typed, and then tries to solve it.

The idea of the type inference algorithm is basically the same as the previous one presented in [13]; reducing constraints on types into a system of linear inequalities. However, in the current type system, a challenge consists in how to infer an appropriate PTE for each lock and thread ID type. To this end, we designate *type environment variables*, which represent unknowns on PTEs, and use them for building constraints on (procurable) type environments.

In the rest of this section, we assume that input to the type inference algorithm is a simply typed program, and each bound variable and $\mathbf{assert}$ command is annotated with its simple type; Figure 11 shows the syntax of commands, function definitions and simple types with those annotations. Those simple types can be inferred by straightforwardly adapting standard techniques [11] to our language. We also assume that bound variables are different from each other. We write $\mathit{stype}(x)$ for the simple type assigned to x.

6.1 Syntax and semantics of constraints

Figure 12 shows the syntax of constraint language. In order to distinguish the constructors of expressions in the con-

$$
\begin{array}{rl}
/*\,\emptyset\,*/ & \mathbf{let}\ p = \mathbf{malloc}()\ \mathbf{in} \\
/*\,p : \mathbf{0}\ \mathbf{ref}_1\,*/ & \mathbf{let}\ q = \mathbf{malloc}())\ \mathbf{in} \\
/*\,p : \mathbf{0}\ \mathbf{ref}_1, q : \mathbf{0}\ \mathbf{ref}_1\,*/ & \mathbf{let}\ l = \mathbf{newlock}()\ \mathbf{in} \\
/*\,p : \mathbf{0}, q : \mathbf{0}\ \mathbf{ref}_1, l : (\Gamma_l, 0)\ \mathbf{lock}_1\,*/ & \mathbf{let}\ c = \mathbf{fork}(\mathit{loop}(p, q, l))\ \mathbf{in} \\
/*\,p : \mathbf{0}, q : \mathbf{0}\ \mathbf{ref}_{0.5}, l : (\Gamma_l, 0)\ \mathbf{lock}_{0.5}, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_1\,*/ & \quad \mathit{loop}(p, q, l); \\
/*\,p : \mathbf{0}, q : \mathbf{0}\ \mathbf{ref}_{0.5}, l : (\Gamma_l, 0)\ \mathbf{lock}_{0.5}, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_1\,*/ & \quad \mathbf{wait}(c); \\
/*\,p : \mathbf{0}, q : \mathbf{0}\ \mathbf{ref}_1, l : (\Gamma_l, 0)\ \mathbf{lock}_1, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_0\,*/ & \quad \mathbf{freelock}(l); \\
/*\,p : \mathbf{0}\ \mathbf{ref}_1, q : \mathbf{0}\ \mathbf{ref}_1, l : (\Gamma_l, 0)\ \mathbf{lock}_0, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_0\,*/ & \quad \mathbf{free}(p); \\
/*\,p : \mathbf{0}, q : \mathbf{0}\ \mathbf{ref}_1, l : (\Gamma_l, 0)\ \mathbf{lock}_0, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_0\,*/ & \quad \mathbf{free}(q) \\
/*\,p : \mathbf{0}, q : \mathbf{0}, l : (\Gamma_l, 0)\ \mathbf{lock}_0, c : \Gamma_{\frac{1}{2}}\ \mathbf{tid}_0\,*/ &
\end{array}
$$

Figure 9. Typing example.

$$
\frac{\begin{array}{c}
\vdash D : \Theta \qquad P(\mathit{Conf}) = \{t_1 \mapsto s_1, \ldots, t_n \mapsto s_n\} \qquad \Theta; \Gamma_i \vdash s_i \Rightarrow \Gamma'_i \text{ for each } i \in \{1, \ldots, n\} \\
t_i = \star \text{ implies } \mathbf{empty}(\Gamma'_i) \qquad \Gamma = \Gamma_1 + \cdots + \Gamma_n \qquad \mathbf{ConOwn}(\mathit{Conf}, \Gamma) \qquad \mathbf{acyclic}(\widetilde{t}, \widetilde{\Gamma'}, R) \\
\text{For any } x \in \mathit{dom}(R(\mathit{Conf})) \text{ such that } R(\mathit{Conf})(x) = t_i, \\
\Gamma'_i = \mathbf{PTE}(\Gamma(x)) + \Gamma'' \text{ and } \mathbf{empty}(\Gamma'') \text{ for some } \Gamma''
\end{array}}{\Theta; \Gamma \vdash_D \mathit{Conf}\ \mathbf{ok}} \quad (\text{T-CONFIG})
$$

Figure 10. Typing rule for configurations.

$$
\begin{array}{rcl}
s & ::= & \ldots \mid \mathbf{let}\ x : a = y\ \mathbf{in}\ s \\
 & \mid & \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s \\
 & \mid & \mathbf{assert}_a(x = y) \\
d & ::= & f(\tilde{x} : \tilde{a}) = s \\
a\ (\text{Simple types}) & ::= & \mathbf{ref} \mid \mathbf{lock} \mid \mathbf{tid}
\end{array}
$$

Figure 11. Revised syntax for type inference. (Only extended or overridden parts are presented.)

straint language from the predicates and the operators introduced in the previous section, we put $\widehat{\cdot}$ on the constraint language constructors.

q ranges over ownership expressions. φ represents an unknown for an ownership, which is a member of a countably infinite set $\mathbf{OVar}$. $q_1 \mathbin{\widehat{+}} q_2$ represents summation of ownerships.

τ is the meta-variable for *type expressions* and represents a value type that may contain unknowns for ownerships or type environments. Expressions $(\mathbb{\Gamma}, q_1)\ \mathbf{lock}_{q_2}$, $\mathbb{\Gamma}\ \mathbf{tid}_q$ and $(\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1}$ represent lock types, thread ID types and reference types respectively. Here, $\mathbb{\Gamma}$ is a type environment expression (see below). In order to simplify the type inference algorithm, we fix the shape of reference types; we deal with only reference types that can be expressed by recursive type expression $(\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1}$. More concretely, the type expression $(\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1}$ represents a recursive type κ that satisfies $\kappa(\epsilon) = q_1$ and $\kappa(\pi) = q_2$ for $\pi \in \{0\}^+$. We can enhance the power of the type inference algorithm by using more expressive type expression

$$
\begin{array}{rcl}
q & ::= & f \mid \varphi \mid q_1 \mathbin{\widehat{+}} q_2 \\
\tau & ::= & (\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1} \mid (\mathbb{\Gamma}, q_1)\ \mathbf{lock}_{q_2} \\
 & \mid & \mathbb{\Gamma}\ \mathbf{tid}_q \\
\Gamma & ::= & \widetilde{x} : \widetilde{\tau} \\
\mathbb{\Gamma} & ::= & \Gamma \mid \gamma \mid \widehat{[\widetilde{y}/\widetilde{x}]}\,\gamma \\
c & ::= & \widehat{\mathbf{empty}}(\tau) \mid \widehat{\mathbf{empty}}(\Gamma \mathbin{\widehat{\setminus}} \widehat{\mathit{dom}}(\gamma)) \\
 & \mid & \gamma_1 \mathrel{\widehat{=}} \gamma_2 \mid q_1 \mathrel{\widehat{=}} q_2 \mid q_1 \mathrel{\widehat{>}} q_2 \\
 & \mid & \Gamma_1 \mathrel{\widehat{=}} \Gamma_2 \mathbin{\widehat{+}} \gamma \mid \widehat{\mathit{dom}}(\gamma) \mathrel{\widehat{\subseteq}} \{\widetilde{x}\} \\
 & \mid & \{\widetilde{x}\} \mathrel{\widehat{\subseteq}} \widehat{\mathit{dom}}(\gamma) \\
 & \mid & \widehat{\mathit{dom}}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{\mathit{dom}}(\gamma') \\
 & \mid & \widehat{\mathit{dom}}(\gamma) \mathrel{\utilde{\widehat{\subseteq}\,[}}\!\widetilde{y}/\widetilde{x}] \ \widehat{\mathit{dom}}(\gamma') \\
\Theta & ::= & \widetilde{f} : (\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'}) \\
\varphi & \in & \mathbf{OVar} \\
\gamma & \in & \mathbf{EVar}
\end{array}
$$

Figure 12. Syntax of constraint language.

such as $(\mu\alpha.\alpha\ \mathbf{ref}_{q_1})\mathbf{ref}_{q_2} \ldots \mathbf{ref}_{q_n}$ for a fixed n, and the type inference algorithm can be easily adapted for those expressions.

The meta-variable $\mathbb{\Gamma}$ ranges over the set of *type environment expressions*. γ ranges over the countably infinite set $\mathbf{EVar}$ of *type environment variables* and denotes an unknown (procurable) type environment. $\widehat{[\widetilde{y}/\widetilde{x}]}\,\gamma$ represents renaming of (procurable) type environment denoted by γ. We require that both $\widetilde{x}$ and $\widetilde{y}$ are pairwise distinct sequences.

The meta-variable c represents a constraint. The constraint syntax is designed carefully so that it is sufficient for a reasonably simple type inference. $\widehat{\mathbf{empty}}(\tau)$ expresses emptiness of τ. $\gamma_1 \mathrel{\widehat{=}} \gamma_2$ expresses equality of γ_1 and γ_2. $q_1 \mathrel{\widehat{=}} q_2$ and $q_1 \mathrel{\widehat{>}} q_2$ are constraints on ownerships. Note that the ownership constraints expressible in the constraint language consist of only linear inequalities on ownership variables. The constraint $\Gamma_1 \mathrel{\widehat{=}} \Gamma_2 \mathbin{\widehat{+}} \gamma$ represents equality between the type environment denoted by Γ_1 and the sum of Γ_2 and γ. $\widehat{\mathbf{empty}}(\Gamma \mathbin{\widehat{\setminus}} \widehat{dom}(\gamma))$ expresses that the type environment denoted by $\Gamma \mathbin{\widehat{\setminus}} \widehat{dom}(\gamma)$ is empty. $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \{\widetilde{x}\}$, $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma')$ and $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{y}/\widetilde{x}]}\, \widehat{dom}(\gamma')$ are set-inclusion constraints among the domains of type environments. We use a meta-variable C for sets of constraints.

θ, a *valuation*, ranges over $(\mathbf{OVar} \overset{\mathbf{fin}}{\to} \mathbb{Q}^{\geq 0}) \cup (\mathbf{EVar} \overset{\mathbf{fin}}{\to} \mathbf{TEnv})$. Here, $\mathbf{TEnv}$ is the set of type environments.

Denotational semantics of ownership expressions, type expressions and type environment expressions are defined in Figure 13. We write $\models C$ if there is a valuation θ such that $\theta \models C$.

6.2 Constraint generation

The first step of the type inference algorithm is constraint generation. Given a program, this phase generates a set of constraints that make the program well-typed. Before elaborating this phase, we introduce several auxiliary definitions.

DEFINITION 27. *rename_ovars is a function that takes an exprssion X and returns a pair X', C, where X' is an expression that is obtained by replacing every ownership in X with a fresh ownership variable and C is $\left\{\varphi_i \mathrel{\widehat{\geq}} \varphi'_i \mid (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi'_i})\ \mathbf{ref}_{\varphi_i} \text{ appears in } X'\right\}$.*

The constraint set that *rename_ovars* returns guarantees the restriction on reference types that $\kappa(\pi) = 0$ implies $\kappa(\pi 0) = 0$ mentioned in Section 5.1.

DEFINITION 28. *$(\widetilde{x}{:}\widetilde{\tau})|_{\{\widetilde{y}\}}$ is defined to be $x_{i_1}{:}\tau_{i_1}, \ldots, x_{i_n}{:} \tau_{i_n}$ where $\{x_{i_1}, \ldots, x_{i_n}\} = \{\widetilde{x}\} \cap \{\widetilde{y}\}$.*

DEFINITION 29. *The operator $\langle \widetilde{y}/\widetilde{x} \rangle$ is defined as follows.*

$$\begin{array}{l}
\langle \widetilde{y}/\widetilde{x} \rangle (\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1} = (\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1} \\
\langle \widetilde{y}/\widetilde{x} \rangle (\Gamma, q_1)\ \mathbf{lock}_{q_2} = ((\langle \widetilde{y}/\widetilde{x} \rangle \Gamma), q_1)\ \mathbf{lock}_{q_2} \\
\langle \widetilde{y}/\widetilde{x} \rangle \Gamma\ \mathbf{tid}_q = (\langle \widetilde{y}/\widetilde{x} \rangle \Gamma)\ \mathbf{tid}_q
\end{array}$$

$$\begin{array}{l}
\langle \widetilde{y}/\widetilde{x} \rangle (x_1 : \tau_1, \ldots, x_n : \tau_n) = \\
\quad (x_1 : \langle \widetilde{y}/\widetilde{x} \rangle \tau_1, \ldots, x_n : \langle \widetilde{y}/\widetilde{x} \rangle \tau_n)
\end{array}$$

$$\begin{array}{l}
\langle \widetilde{y}/\widetilde{x} \rangle (\tau_1, \ldots, \tau_n) = \\
\quad \left(\begin{array}{l}
y_1 : \tau_1, \\
y_2 : \langle y_1/x_1 \rangle \tau_2, \\
y_3 : \langle y_1, y_2/x_1, x_2 \rangle \tau_3, \\
\ldots, \\
y_n : \langle y_1, \ldots, y_{n-1}/x_1, \ldots, x_{n-1} \rangle \tau_n
\end{array} \right) \\
\langle \widetilde{y}/\widetilde{x} \rangle \gamma = \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma \\
\langle \widetilde{y}/\widetilde{x} \rangle\, \widehat{[\widetilde{y'}/\widetilde{x'}]}\, \gamma = (\widehat{[\widetilde{y}/\widetilde{x}]} \circ \widehat{[\widetilde{y'}/\widetilde{x'}]})\gamma.
\end{array}$$

Here, $\widehat{[\widetilde{y}/\widetilde{x}]} \circ \widehat{[\widetilde{y'}/\widetilde{x'}]}$ represents the renaming constructor that corresponds to the composition of $[\widetilde{y}/\widetilde{x}]$ and $[\widetilde{y'}/\widetilde{x'}]$.

The operator $\langle \widetilde{y}/\widetilde{x} \rangle$ "pushes" the constructor $\widehat{[\widetilde{y}/\widetilde{x}]}$ as inward as possible. It also serves for creating pre/post type environments of a function call from a sequence of the types of function arguments; observe the correspondence between the definition of $\langle \widetilde{y}/\widetilde{x} \rangle (\tau_1, \ldots, \tau_n)$ above and the second and the third premises of T-APP in Figure 6.

$\tau_1 \doteq \tau_2$ (and $\mathbb{\Gamma}_1 \doteq \mathbb{\Gamma}_2$), a set of constraints that make τ_1 equal to τ_2 (and $\mathbb{\Gamma}_1$ equal to $\mathbb{\Gamma}_2$, respectively) are defined in Figure 14. The constraints also include those on domains of type environment expressions. $\Gamma_1 \doteq \Gamma_2 + \Gamma_3$ is a set of constraints that make Γ_1 equal to $\Gamma_2 + \Gamma_3$, which is defined below.

Note that, in the cases of $\gamma_1 \doteq \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma_2$, $\widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma_1 \doteq \mathbb{\Gamma}$ and $\Gamma \doteq \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma$, creating $\langle \widetilde{x}/\widetilde{y} \rangle$ is legitimate because $\widetilde{x}$ and $\widetilde{y}$ are both pairwise distinct sequences. Note also that the definition of $\gamma \doteq \Gamma$ and $\Gamma \doteq \gamma$ are a little complicated so that the generated conforms the syntax of the constraint language; we generate a type environment Γ' from Γ, generate constraints that guarantee $\mathbf{empty}(\Gamma')$ and generate $\Gamma \mathrel{\widehat{=}} \Gamma' + \gamma$ that effectively expresses $\Gamma = \gamma$ given $\mathbf{empty}(\Gamma')$.

$\tau_1 + \tau_2$ (N.B., no $\widehat{\ }$ on $+$) is a pair (τ, C) where τ is the sum of τ_1 and τ_2, and C is constraints for the sum to be well-defined. $\Gamma_1 + \Gamma_2$ is also defined accordingly.

DEFINITION 30. *$\tau_1 + \tau_2$ and $\Gamma_1 + \Gamma_2$ are defined as follows.*

$$\begin{array}{l}
(\mu\alpha.\alpha\ \mathbf{ref}_{q_2})\ \mathbf{ref}_{q_1} + (\mu\alpha.\alpha\ \mathbf{ref}_{q'_2})\ \mathbf{ref}_{q'_1} = \\
\quad ((\mu\alpha.\alpha\ \mathbf{ref}_{q'_1 \widehat{+} q'_2})\ \mathbf{ref}_{q_1 \widehat{+} q_2}, \emptyset) \\
(\mathbb{\Gamma}, q_1)\ \mathbf{lock}_{q_2} + (\mathbb{\Gamma}', q'_1)\ \mathbf{lock}_{q'_2} = \\
\quad ((\Gamma, q_1 \mathbin{\widehat{+}} q'_1)\ \mathbf{lock}_{q_2 \widehat{+} q'_2}, (\mathbb{\Gamma} \doteq \mathbb{\Gamma}')) \\
\mathbb{\Gamma}\ \mathbf{tid}_q + \mathbb{\Gamma}'\ \mathbf{tid}_{q'} = \\
\quad (\Gamma\ \mathbf{tid}_{q_1 \widehat{+} q'_1}, (\mathbb{\Gamma} \doteq \mathbb{\Gamma}'))
\end{array}$$

$$\begin{array}{l}
(\widetilde{x} : \widetilde{\tau^{(1)}}, \widetilde{y} : \widetilde{\tau^{(2)}}) + (\widetilde{y} : \widetilde{\tau^{(3)}}, \widetilde{z} : \widetilde{\tau^{(4)}}) = \\
\quad ((\widetilde{x} : \widetilde{\tau^{(1)}}, \widetilde{y} : \widetilde{\tau'}, \widetilde{z} : \widetilde{\tau^{(3)}}), \bigcup \widetilde{C})\ \mathbf{where} \\
\quad \widetilde{x} \cap \widetilde{y} = \widetilde{y} \cap \widetilde{z} = \widetilde{z} \cap \widetilde{x} = \emptyset \\
\quad \widetilde{\tau'}, \widetilde{C} = \widetilde{\tau^{(2)}} + \widetilde{\tau^{(3)}}.
\end{array}$$

$\mathit{template}_x(a)$ is a value type whose ownership and PTE part are left unknown.

DEFINITION 31. *Define $\mathit{template}_x(a)$ as follows:*

$$\begin{array}{l}
\mathit{template}_x(\mathbf{ref}) = (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_2})\ \mathbf{ref}_{\varphi_1} \\
\quad \textit{where } \varphi_1 \textit{ and } \varphi_2 \textit{ are fresh.} \\
\mathit{template}_x(\mathbf{lock}) = (\gamma_x, \varphi_1)\ \mathbf{lock}_{\varphi_2} \\
\quad \textit{where } \varphi_1 \textit{ and } \varphi_2 \textit{ are fresh.} \\
\mathit{template}_x(\mathbf{tid}) = \gamma_x\ \mathbf{tid}_{\varphi} \\
\quad \textit{where } \varphi \textit{ is fresh.}
\end{array}$$

In the definition above, γ_x is the type environment variable designated for the variable x. (See Remark 35 below.)

$\mathit{domtyp}(\tau)$ returns the expression that corresponds to the domain of PTE inside τ if exists.

$$[\![f]\!]_\theta = f \quad [\![\varphi]\!]_\theta = \theta(\varphi) \quad [\![q_1 \mathbin{\widehat{+}} q_2]\!]_\theta = [\![q_1]\!]_\theta + [\![q_2]\!]_\theta \quad [\![q_1 \mathbin{\widehat{-}} q_2]\!]_\theta = [\![q_1]\!]_\theta - [\![q_2]\!]_\theta$$

$$[\![(\mathbb{\Gamma}, q_1)\, \mathbf{lock}_{q_2}]\!]_\theta = ([\![\mathbb{\Gamma}]\!]_\theta, [\![q_1]\!]_\theta)\, \mathbf{lock}_{[\![q_2]\!]_\theta} \quad [\![\mathbb{\Gamma}\, \mathbf{tid}_q]\!]_\theta = [\![\mathbb{\Gamma}]\!]_\theta\, \mathbf{tid}_{[\![q]\!]_\theta}$$
$$[\![(\mu\alpha.\alpha\, \mathbf{ref}_{q_2})\, \mathbf{ref}_{q_1}]\!]_\theta = \kappa \quad \text{where } \kappa(\varepsilon) = [\![q_1]\!]_\theta \text{ and } \kappa(0\pi) = [\![q_2]\!]_\theta \text{ for any } \pi$$

$$[\![(\widetilde{x} : \widetilde{\tau})]\!]_\theta = (\widetilde{x} : \widetilde{[\![\tau]\!]_\theta}) \quad [\![\gamma]\!]_\theta = \theta(\gamma) \quad [\![\widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma]\!]_\theta = [\widetilde{y}/\widetilde{x}][\![\gamma]\!]_\theta$$

$$[\![(\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'})]\!]_\theta = (\widetilde{x} : [\![\widetilde{\tau}]\!]_\theta) \to ([\![\widetilde{\tau'}]\!]_\theta)$$
$$[\![\Theta]\!]_\theta = \{f \mapsto [\![\Theta(f)]\!]_\theta\}_{f \in dom(\Theta)}$$

$$\theta \models \widehat{\mathbf{empty}}(\tau) \text{ iff. } \mathbf{empty}([\![\tau]\!]_\theta)$$
$$\theta \models \widehat{\mathbf{empty}}(\Gamma \mathbin{\widehat{\setminus}} \widehat{dom}(\gamma)) \text{ iff. } \mathbf{empty}([\![\Gamma]\!]_\theta \setminus dom([\![\gamma]\!]_\theta))$$
$$\theta \models \Gamma_1 \mathrel{\widehat{=}} \Gamma_2 \mathbin{\widehat{+}} \gamma \text{ iff. } \begin{array}{l} [\![\Gamma_1]\!]_\theta = [\![\Gamma_2]\!]_\theta + [\![\gamma]\!]_\theta \text{ and} \\ dom([\![\Gamma_1]\!]_\theta) = dom([\![\Gamma_2]\!]_\theta) \end{array}$$
$$\theta \models \gamma_1 \mathrel{\widehat{=}} \gamma_2 \text{ iff. } [\![\gamma_1]\!]_\theta = [\![\gamma_2]\!]_\theta$$
$$\theta \models q_1 \mathrel{\widehat{=}} q_2 \text{ iff. } [\![q_1]\!]_\theta = [\![q_2]\!]_\theta$$
$$\theta \models q_1 \mathrel{\widehat{\geq}} q_2 \text{ iff. } [\![q_1]\!]_\theta > [\![q_2]\!]_\theta$$

$$\theta \models \widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \{\widetilde{x}\} \text{ iff. } dom([\![\gamma]\!]_\theta) \subseteq \{\widetilde{x}\}$$
$$\theta \models \{\widetilde{x}\} \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma) \text{ iff. } \{\widetilde{x}\} \subseteq dom([\![\gamma]\!]_\theta)$$
$$\theta \models \widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma') \text{ iff. } dom([\![\gamma]\!]_\theta) \subseteq dom([\![\gamma']\!]_\theta)$$
$$\theta \models \widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{y}/\widetilde{x}]}\, \widehat{dom}(\gamma') \text{ iff. } dom([\![\gamma]\!]_\theta) \subseteq dom([\![\widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma']\!]_\theta)$$
$$\theta \models C \text{ iff. } \theta \models c \text{ for every } c \in C$$

Figure 13. Denotation of constraint language expressions.

$$\begin{array}{rcl}
((\mu\alpha.\alpha\, \mathbf{ref}_{q_2})\, \mathbf{ref}_{q_1} \doteq (\mu\alpha.\alpha\, \mathbf{ref}_{q_2'})\, \mathbf{ref}_{q_1'}) & = & \{q_1 \mathrel{\widehat{=}} q_1', q_2 \mathrel{\widehat{=}} q_2'\} \\
((\mathbb{\Gamma}, q_1)\, \mathbf{lock}_{q_2} \doteq (\mathbb{\Gamma}', q_1')\, \mathbf{lock}_{q_2'}) & = & (\mathbb{\Gamma} \doteq \mathbb{\Gamma}') \cup \{q_1 \mathrel{\widehat{=}} q_1', q_2 \mathrel{\widehat{=}} q_2'\} \\
(\mathbb{\Gamma}\, \mathbf{tid}_q \doteq \mathbb{\Gamma}'\, \mathbf{tid}_{q'}) & = & (\mathbb{\Gamma} \doteq \mathbb{\Gamma}') \cup \{q \mathrel{\widehat{=}} q'\}
\end{array}$$

$$\begin{array}{rcl}
((\widetilde{x} : \widetilde{\tau}) \doteq (\widetilde{x} : \widetilde{\tau'})) & = & \bigcup_i (\tau_i \doteq \tau_i') \\
(\gamma_1 \doteq \gamma_2) & = & \left\{\gamma_1 \mathrel{\widehat{=}} \gamma_2, \widehat{dom}(\gamma_1) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma_2), \widehat{dom}(\gamma_2) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma_1)\right\} \\
(\gamma_1 \doteq \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma_2) & = & \left\{\gamma_1 \mathrel{\widehat{=}} \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma_2, \widehat{dom}(\gamma_1) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{y}/\widetilde{x}]}\, \widehat{dom}(\gamma_2), \widehat{dom}(\gamma_2) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{x}/\widetilde{y}]}\, \widehat{dom}(\gamma_1)\right\} \\
(\widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma_1 \doteq \mathbb{\Gamma}) & = & (\gamma_1 \doteq \langle \widetilde{x}/\widetilde{y} \rangle \mathbb{\Gamma}) \\
(\Gamma \doteq \widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma) & = & (\gamma \doteq \langle \widetilde{x}/\widetilde{y} \rangle \Gamma) \\
(\gamma \doteq \Gamma) = (\Gamma \doteq \gamma) & = & C_1 \cup C_2 \cup C_3 \text{ where } \begin{array}[t]{l} \Gamma', C_1 = rename\_ovars(\Gamma) \\ C_2 = \left\{\widehat{\mathbf{empty}}(\Gamma'(x)) \mid x \in dom(\Gamma')\right\} \\ C_3 = \left\{\Gamma \mathrel{\widehat{=}} \Gamma' + \gamma, \widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} dom(\Gamma), dom(\Gamma) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma)\right\} \end{array} \\
\Gamma_1 \doteq \Gamma_2 + \Gamma_3 & = & C_1 \cup C_2 \text{ where } \Gamma', C_1 = \Gamma_2 + \Gamma_3 \text{ and } C_2 = (\Gamma_1 \doteq \Gamma')
\end{array}$$

Figure 14. Definitions of $\tau_1 \doteq \tau_2$ and $\mathbb{\Gamma}_1 \doteq \mathbb{\Gamma}_2$ and $\Gamma_1 \doteq \Gamma_2 + \Gamma_3$.

DEFINITION 32. *$domtyp(\tau)$ is defined as follows.*

$$\begin{array}{l}
domtyp((\mu\alpha.\alpha\, \mathbf{ref}_{q_2})\, \mathbf{ref}_{q_1}) = \emptyset \\
domtyp((\mathbb{\Gamma}, q_1)\, \mathbf{lock}_{q_2}) = domtyp(\mathbb{\Gamma}) \\
domtyp(\mathbb{\Gamma}\, \mathbf{tid}_q) = domtyp(\mathbb{\Gamma})
\end{array}$$

$$\begin{array}{l}
domtyp((\widetilde{x} : \widetilde{\tau})) = \{\widetilde{x}\} \\
domtyp(\gamma) = \widehat{dom}(\gamma) \\
domtyp(\widehat{[\widetilde{y}/\widetilde{x}]}\, \gamma) = \widehat{[\widetilde{y}/\widetilde{x}]}\, \widehat{dom}(\gamma)
\end{array}$$

Constraint set $wfarg(x_1 : \tau_1, \ldots, x_n : \tau_n)$ is used for well-formedness of function argument types.

DEFINITION 33. *Let $wfarg(x_1 : \tau_1, \ldots, x_n : \tau_n)$ be the constraint set*

$$\left\{\begin{array}{l}
domtyp(\tau_1) \mathrel{\widehat{\subseteq}} \emptyset, \\
domtyp(\tau_2) \mathrel{\widehat{\subseteq}} \{x_1\}, \\
domtyp(\tau_3) \mathrel{\widehat{\subseteq}} \{x_1, x_2\}, \\
\ldots, \\
domtyp(\tau_n) \mathrel{\widehat{\subseteq}} \{x_1, \ldots, x_{n-1}\}.
\end{array}\right\}$$

We write $wfarg((\widetilde{x}{:}\widetilde{\tau}) \to (\widetilde{\tau'}))$ for $wfarg(\widetilde{x}{:}\widetilde{\tau}) \cup wfarg(\widetilde{x}{:}\widetilde{\tau'})$ and $wfarg(\Theta)$ for $\bigcup_{f \in dom(\Theta)} wfarg(\Theta(f))$.

The constraint generation algorithm $\mathcal{C}$ is defined in Figures 15 and 16. The core of the algorithm takes a function type environment Θ, a command s and a post type environment Γ_{post} as input, and returns a pre type environment Γ_{pre}

$$\boxed{\mathcal{C}(\Theta, s, \Gamma_{post}) = (\Gamma_{pre}, C)}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, \mathbf{skip}, \Gamma_{post}) = (\Gamma_{post}, \emptyset) \\
\mathcal{C}(\Theta, s_1; s_2, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2)\ \mathbf{where} \\
\quad \Gamma', C_1 = \mathcal{C}(\Theta, s_2, \Gamma_{post}) \\
\quad \Gamma_{pre}, C_2 = \mathcal{C}(\Theta, s_1, \Gamma')
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}(\Theta, \mathbf{let}\ x : a = y\ \mathbf{in}\ s, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where}} \\
\quad \tau = \mathit{template}_x(a) & \tau', C_2 = (\Gamma'(x) + \Gamma'(y)) \\
\quad \Gamma', C_1 = \mathcal{C}(\Theta, s, (\Gamma_{post}, x : \tau)) & C_3 = \left\{ \widehat{\mathbf{empty}(\tau)} \right\} \\
 & \Gamma_{pre} = (\Gamma' \setminus \{x, y\}, y : \tau')
\end{array}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s, \Gamma_{post}) = (\Gamma_{pre}, C)\ \mathbf{where} \\
\quad \Gamma', C = \mathcal{C}(\Theta, s, (\Gamma_{post}, x : (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_0)) \\
\quad \Gamma_{pre} = \Gamma' \setminus \{x\}
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}(\Theta, f(\widetilde{y}), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3 \cup C_4)\ \mathbf{where}} \\
\quad (\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'}) = \Theta(f) & C_3 = (\Gamma_{post} \doteq \langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau'} + \Gamma') \\
\quad \Gamma', C_1 = \mathit{rename\_ovars}(\Gamma_{post}) & C_4 = (\Gamma_{pre} \doteq \langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau} + \Gamma') \\
\quad \Gamma_{pre}, C_2 = \mathit{rename\_ovars}(\Gamma_{post}) &
\end{array}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2)\ \mathbf{where} \\
\quad \Gamma', C_1 = \mathcal{C}(\Theta, s, (\Gamma_{post}, x : (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_0)) \\
\quad C_2 = (\Gamma'(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_1) \\
\quad \Gamma_{pre} = \Gamma' \setminus \{x\} \\
\mathcal{C}(\Theta, \mathbf{free}(x), \Gamma_{post}) = (\Gamma_{pre}, C)\ \mathbf{where} \\
\quad \Gamma_{pre} = \Gamma_{post}[x \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_1] \\
\quad C = (\Gamma_{post}(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_0) \\
\mathcal{C}(\Theta, \mathbf{let}\ x = *y\ \mathbf{in}\ s, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2)\ \mathbf{where} \\
\quad \varphi, \varphi_1, \varphi_2 \text{ are fresh} \\
\quad \Gamma', C_1 = \mathcal{C}(\Theta, s, (\Gamma_{post}, x : (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_0)) \\
\quad C_2 = (\Gamma'(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_\varphi)\ \mathbf{ref}_\varphi) \cup (\Gamma'(y) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_2})\ \mathbf{ref}_{\varphi_1}) \cup \left\{ \varphi_1 \mathrel{\widehat{>}} 0, \varphi_1 \mathrel{\widehat{\geq}} \varphi_2 \right\} \\
\quad \Gamma_{pre} = (\Gamma' \setminus \{x\})[y \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi \mathbin{\widehat{+}} \varphi_2})\ \mathbf{ref}_{\varphi_1}] \\
\mathcal{C}(\Theta, *y \leftarrow x, \Gamma_{post}) = (\Gamma_{pre}, C)\ \mathbf{where} \\
\quad \varphi_1, \varphi_2, \varphi_3 \text{ are fresh} \\
\quad C = (\Gamma_{post}(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_3})\ \mathbf{ref}_{\varphi_2}) \cup (\Gamma_{post}(y) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_1})\ \mathbf{ref}_1) \cup \left\{ \varphi_2 \mathrel{\widehat{\geq}} \varphi_3 \right\} \\
\quad \Gamma_{pre} = \Gamma_{post}[y \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_0)\ \mathbf{ref}_1][x \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_1 \mathbin{\widehat{+}} \varphi_3})\ \mathbf{ref}_{\varphi_1 \mathbin{\widehat{+}} \varphi_2}] \\
\mathcal{C}(\Theta, \mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \Gamma_{post}) = (\Gamma_2, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \varphi_1 \text{ and } \varphi_2 \text{ are fresh} \\
\quad \Gamma_1, C_1 = \mathcal{C}(\Theta, s_1, \Gamma_{post}) \\
\quad \Gamma_2, C_2 = \mathcal{C}(\Theta, s_2, \Gamma_{post}) \\
\quad C_3 = (\Gamma_{post}(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_2})\ \mathbf{ref}_{\varphi_1}) \cup (\Gamma_1 \setminus \{x\} \doteq \Gamma_2 \setminus \{x\}) \cup \left\{ \varphi_1 \mathrel{\widehat{\geq}} \varphi_2 \right\}
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}(\Theta, \mathbf{assert}_a(x = y), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5)\ \mathbf{where}} \\
\quad \tau_1', \tau_2' = \Gamma_{post}(x), \Gamma_{post}(y) & \tau, C_4 = \tau_1 + \tau_2 \\
\quad \tau', C_1 = \tau_1' + \tau_2' & C_5 = (\tau \doteq \tau') \\
\quad \tau_1, C_2 = \mathit{rename\_ovars}(\tau_1') & \Gamma_{pre} = \Gamma[x \mapsto \tau_1, y \mapsto \tau_2] \\
\quad \tau_2, C_3 = \mathit{rename\_ovars}(\tau_2') &
\end{array}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, \mathbf{assert}(x = *y), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2)\ \mathbf{where} \\
\quad \varphi_1, \varphi_2, \varphi, \varphi_2', \varphi' \text{ are fresh} \\
\quad C_1 = (\Gamma_{post}(x) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_\varphi)\ \mathbf{ref}_\varphi) \cup (\Gamma_{post}(y) \doteq (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_2})\ \mathbf{ref}_{\varphi_1}) \cup \{ \varphi_1 \mathrel{\widehat{>}} 0, \varphi_1 \mathrel{\widehat{>}} \varphi_2 \} \\
\quad \Gamma_{pre} = \Gamma[x \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi'})\ \mathbf{ref}_{\varphi'}, y \mapsto (\mu\alpha.\alpha\ \mathbf{ref}_{\varphi_2'})\ \mathbf{ref}_{\varphi_1}] \\
\quad C_2 = \{ \varphi \mathbin{\widehat{+}} \varphi_2 \mathrel{\widehat{=}} \varphi' \mathbin{\widehat{+}} \varphi_2' \}
\end{array}$$

Figure 15. Constraint generation algorithm (cases for sequential commands).

$$\boxed{\mathcal{C}(\Theta, s, \Gamma_{post}) = (\Gamma_{pre}, C)}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, \mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \Gamma', C_1 = \mathcal{C}(\Theta, s, (\Gamma_{post}, x : (\gamma_x, 0)\ \mathbf{lock}_0)) \\
\quad \Gamma_{pre}, C_2 = \mathit{rename\_ovars}(\Gamma_{post}) \\
\quad C_3 = \left\{ \Gamma_{pre} \mathrel{\widehat{=}} (\Gamma' \setminus \{x\}) \mathbin{\widehat{+}} \gamma_x, \widehat{\mathit{dom}}(\gamma_x) \mathrel{\widehat{\subseteq}} \mathit{dom}(\Gamma_{post}) \right\} \cup (\Gamma'(x) \doteq (\gamma_x, 0)\ \mathbf{lock}_1) \\
\mathcal{C}(\Theta, \mathbf{freelock}(x), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \Gamma_{pre}, C_1 = \mathit{rename\_ovars}(\Gamma_{post}) \\
\quad C_2 = (\Gamma_{pre}(x) \doteq (\gamma_x, 0)\ \mathbf{lock}_1) \\
\quad C_3 = \left\{ \Gamma_{post} \mathrel{\widehat{=}} \Gamma_{pre}[x \mapsto (\gamma_x, 0)\ \mathbf{lock}_0] \mathbin{\widehat{+}} \gamma_x \right\} \\
\mathcal{C}(\Theta, \mathbf{acq}(x), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \Gamma_{pre}, C_1 = \mathit{rename\_ovars}(\Gamma_{post}) \\
\quad \varphi \text{ is fresh} \\
\quad C_2 = (\Gamma_{pre}(x) \doteq (\gamma_x, 0)\ \mathbf{lock}_\varphi) \\
\quad C_3 = \left\{ \Gamma_{post} \mathrel{\widehat{=}} \Gamma_{pre}[x \mapsto (\gamma_x, 1)\ \mathbf{lock}_\varphi] \mathbin{\widehat{+}} \gamma_x, \varphi \mathrel{\widehat{>}} 0 \right\} \\
\mathcal{C}(\Theta, \mathbf{rel}(x), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \Gamma_{pre}, C_1 = \mathit{rename\_ovars}(\Gamma_{post}) \\
\quad \varphi \text{ is fresh} \\
\quad C_2 = (\Gamma_{post}(x) \doteq (\gamma_x, 0)\ \mathbf{lock}_\varphi) \\
\quad C_3 = \left\{ \Gamma_{pre} \mathrel{\widehat{=}} \Gamma_{post}[x \mapsto (\gamma_x, 1)\ \mathbf{lock}_\varphi] \mathbin{\widehat{+}} \gamma_x, \varphi \mathrel{\widehat{>}} 0 \right\} \\
\mathcal{C}(\Theta, \mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2, \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup \cdots \cup C_5)\ \mathbf{where} \\
\quad \Gamma'_{post}, C_1 = \mathit{rename\_ovars}(\Gamma_{post}|_{\mathbf{FV}(s_1)}) \\
\quad \Gamma_1, C_2 = \mathcal{C}(\Theta, s_1, \Gamma'_{post}) \\
\quad \Gamma_2, C_3 = \mathcal{C}(\Theta, s_2, (\Gamma_{post}, x : \gamma_x\ \mathbf{tid}_0)) \\
\quad \Gamma_{pre}, C_4 = (\Gamma_1 + \Gamma_2 \setminus \{x\}) \\
\quad C_5 = \left\{ \mathbf{FV}(s_1) \mathrel{\widehat{\subseteq}} \widehat{\mathit{dom}}(\gamma_x), \widehat{\mathit{dom}}(\gamma_x) \mathrel{\widehat{\subseteq}} \widehat{\mathit{dom}}(\Gamma_{post}), \widehat{\mathbf{empty}}(\Gamma'_{post} \mathbin{\widehat{\setminus}} \widehat{\mathit{dom}}(\gamma_x)) \right\} \\
\mathcal{C}(\Theta, \mathbf{wait}(x), \Gamma_{post}) = (\Gamma_{pre}, C_1 \cup C_2 \cup C_3)\ \mathbf{where} \\
\quad \Gamma_{pre}, C_1 = \mathit{rename\_ovars}(\Gamma_{post}) \\
\quad C_2 = (\Gamma_{pre}(x) \doteq \gamma_x\ \mathbf{tid}_1) \\
\quad C_3 = \left\{ \Gamma_{post} \mathrel{\widehat{=}} (\Gamma_{pre} \setminus \{x\}, x : \gamma_x\ \mathbf{tid}_0) \mathbin{\widehat{+}} \gamma_x \right\}
\end{array}$$

$$\boxed{\mathcal{C}(D) = (\Theta, C)}$$

$$\begin{array}{l}
\mathcal{C}(\{f_i(x_{i1} : a_{i1}, \ldots, x_{im_i} : a_{im_i}) = s_i\}_i) = (\Theta, \bigcup_i C_i)\ \mathbf{where} \\
\quad \tau_{ij} = \mathit{template}_{x_{ij}}(a_{ij}) \text{ for each } i \text{ and for each } j \in \{1, \ldots, m_i\} \\
\quad \tau'_{ij} = \mathit{template}_{x_{ij}}(a_{ij}) \text{ for each } i \text{ and for each } j \in \{1, \ldots, m_i\} \\
\quad \Theta = (f_i : (\widetilde{x_i} : \widetilde{\tau_i}) \to (\widetilde{\tau'_i}))_i \\
\quad C_i = \mathcal{C}(\Theta, f_i(\widetilde{x_i} : a_i) = s_i) \cup \mathit{wfarg}(\Theta(f_i)) \text{ for each } i
\end{array}$$

$$\boxed{\mathcal{C}(\Theta, f(\widetilde{x}) = s) = C}$$

$$\begin{array}{l}
\mathcal{C}(\Theta, f(\widetilde{x}) = s) = C\ \mathbf{where} \\
\quad (\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'}) = \Theta(f) \\
\quad \widetilde{\tau''}, \widetilde{C} = \mathcal{C}(\Theta, s, \widetilde{x} : \widetilde{\tau'}) \\
\quad \widetilde{C'} = \bigcup_i (\tau''_i \doteq \tau_i) \\
\quad C = \bigcup C_i \cup \bigcup C'_i
\end{array}$$

$$\boxed{\mathcal{C}(D, s) = C}$$

$$\begin{array}{l}
\mathcal{C}(D, s) = C_1 \cup C_2\ \mathbf{where} \\
\quad \Theta, C_1 = \mathcal{C}(D) \\
\quad \emptyset, C_2 = \mathcal{C}(\Theta, s, \emptyset)
\end{array}$$

Figure 16. Constraint generation algorithm (cases for concurrency-related commands).

and a set of constraints C. The way of deriving $\mathcal{C}$ from the typing rules in Section 5 is standard. We add explanation to several non-trivial cases.

Case $s = \mathbf{let}\ x : a = y\ \mathbf{in}\ s_0$: The algorithm creates a fresh type expression τ from the simple type annotation a, extends Γ_{post} with $x : \tau$ and recursively passes it to $\mathcal{C}$ with the subcommand. From the returned pre type environment Γ', the algorithm produces the type of y, $\Gamma'(x) + \Gamma'(y)$, in the pre type environment of the whole input Γ_{pre}. Note that, from the syntax of the constraint language, Γ' in $\Gamma'(x), \Gamma'(y)$ and $\Gamma' \setminus \{x, y\}$ is neither γ nor $\widehat{[\widetilde{y}/\widetilde{x}]}\,\gamma$.

Case $s = f(\widetilde{y})$: The algorithm creates two type environments Γ' and Γ_{pre} from Γ_{post} by *rename_ovars*. Operator $\langle \widetilde{x}/\widetilde{y} \rangle$ is used to generate the constraints on pre/post type environments from the types of function arguments.

Case $s = \mathbf{let}\ x = *y\ \mathbf{in}\ s_0$: The idea is the same as the previous case; create a fresh type expression, extend Γ_{post}, pass it to the recursive call and calculate the type of x in Γ_{pre} from the output of the recursive call. However, in the cases in which reference types are involved, we have to be aware that the type of x in the returned Γ' is of the shape $(\mu\alpha.\alpha\ \mathbf{ref}_\varphi)\ \mathbf{ref}_\varphi$ due to our assumption on the shape of reference types.

Case $s = \mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s_0$: We need to express the conditions on type environments in T-NEWLOCK in this case. To do that, we generate the constraint $\Gamma_{pre} \mathrel{\widehat{=}} (\Gamma' \setminus \{x\}) \mathbin{\widehat{+}} \gamma_x$. γ_x is the type environment variable designated for the variable x.

Case $s = \mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2$: The algorithm restricts the domain of Γ_{post} with $\mathbf{FV}(s_1)$, renames its ownership part and passes it to the recursive call for s_1. The passed type environment Γ' subjects to $\widehat{\mathbf{empty}}(\Gamma'_{post} \mathbin{\widehat{\setminus}} \widehat{dom}(\gamma_x))$; with this constraint and the following lemma, it is guaranteed that the conditions in T-FORK are met.

LEMMA 34.
- *If* $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$, *then* $dom(\Gamma_1) = dom(\Gamma_2)$.
- *If* $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$ *and* $\mathbf{empty}(\tau)$ *and* $x \notin \mathbf{FV}(\Theta) \cup \mathbf{FV}(\Gamma_1) \cup \mathbf{FV}(s) \cup \mathbf{FV}(\Gamma_2)$, *then* $\Theta; \Gamma_1, x : \tau \vdash s \Rightarrow \Gamma_2, x : \tau$.

Proof Sketch Both follow from induction on the derivation of $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$. □

REMARK 35. *The type inference algorithm makes use of the property of the type system that, under the assumption that a bound variable is different from each other, the PTE assigned to each lock/pid-typed variable is globally unique (i.e., no flow-sensitivity on PTE). Thanks to this property, the definition of* $template_x(\mathbf{lock})$ *and* $template_x(\mathbf{tid})$ *can assign the unique type environment variable for each x. This design decision simplifies the type inference algorithm.*

The following lemma guarantees soundness of this step; it says that if a constraint set C generated by the algorithm $\mathcal{C}$ satisfies $\models C$, then the input program is indeed well-typed.

LEMMA 36 (Soundness of $\mathcal{C}$). *If* $\mathcal{C}(\Theta, \Gamma_{post}, s) = (\Gamma_{pre}, C)$ *and* $\theta \models C$, *then* $[\![\Theta]\!]_\theta; [\![\Gamma_{pre}]\!]_\theta \vdash s \Rightarrow [\![\Gamma_{post}]\!]_\theta$.

6.3 Constraint reduction

The next step of the type inference is to find a valuation θ such that $\theta \models C$ holds for the constraint set C returned by $\mathcal{C}$. This constraint reduction phase first determines the domain of each type environment variable and decomposes the constraints into ones on ownership variables. Then, the resulting constraints are solved by a linear inequality solver.

Step 1: Deciding the domain of type environment variables The constraint reduction phase first reduces those set-inclusion constraints ($\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \{\widetilde{x}\}$, $\{\widetilde{x}\} \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma)$, $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma')$ and $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{y}/\widetilde{x}]}\,\widehat{dom}(\gamma')$) by calculating the domain of each type environment variable and instantiating it with a type environment template with fresh ownership variables. In calculating the domain, we calculate particularly the greatest domain allowed to each type environment variable. This is justified by the observation that, intuitively, taking the larger domain for a type environment variable is "safe" estimation; even if the domain turns out to be larger than necessary, the types of the redundant variables can be made empty in the following phases.

It is in fact possible to calculate the greatest domain by a standard fixed-point iteration because each $\widehat{dom}(\gamma)$ is either bound from above by a monotone expression, or, from below by a set of program variables. Concretely, such an algorithm starts from the assignment $\left\{\widehat{dom}(\gamma_i) \mapsto V\right\}_i$, where V is the (finite) set of all the variable names that appear in the input program, and update the assignment according to the constraints of the form $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \{\widetilde{x}\}$, $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma')$ and $\widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} \widehat{[\widetilde{y}/\widetilde{x}]}\,\widehat{dom}(\gamma')$ until it reaches the fixed-point. Then, the algorithm checks whether the obtained assignment satisfies $\{\widetilde{x}\} \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma)$ and returns the assignment.

Figure 17 presents the definition of the algorithm $\mathcal{R}$. In the definition, $solve\_dom(C)$ returns the map

$$\left\{\widehat{dom}(\gamma_i) \mapsto \{\widetilde{x_i}\}\right\}_i$$

that gives the greatest solution of the set-inclusion constraints C as mentioned above. In addition to applying substitution, θC simplifies the constraints if possible. For example, if

$$\begin{aligned}
C &= \left\{ \begin{array}{l} \widehat{\mathbf{empty}}((\gamma_x, q_1)\ \mathbf{lock}_{q_2}), \\ (x : (\gamma_x, q_3)\ \mathbf{lock}_{q_4}) \mathrel{\widehat{=}} (x : (\gamma_x, q_5)\ \mathbf{lock}_{q_6}) \mathbin{\widehat{+}} \gamma_y \end{array} \right\} \\
\theta &= \left\{ \begin{array}{l} \gamma_x \mapsto (y : (\mu\alpha.\alpha\ \mathbf{ref}_{q_8})\ \mathbf{ref}_{q_7}), \\ \gamma_y \mapsto (x : (\gamma_x, q_9)\ \mathbf{lock}_{q_{10}}) \end{array} \right\},
\end{aligned}$$

$$
\begin{aligned}
&\mathcal{R}(C) = C' \textbf{ where}\\
&\quad C_1 = \left\{ c \in C \mid c \text{ is of the form } \widehat{dom}(\gamma) \mathrel{\widehat{\subseteq}} X \right\}\\
&\quad C_2 = \left\{ c \in C \mid c \text{ is of the form } \{\widetilde{x}\} \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma) \right\}\\
&\quad F = solve\_dom(C_1 \cup C_2)\\
&\quad \theta = makeval(F)\\
&\quad C' = iter(\theta, C \backslash (C_1 \cup C_2))\\
&makeval(\left\{ \widehat{dom}(\gamma_i) \mapsto \{\widetilde{x_i}\} \right\}_i) =\\
&\quad \left\{ \gamma_i \mapsto (\widetilde{x_i} : template_{\widetilde{x_i}}(stype(\widetilde{x_i}))) \right\}_i\\
&iter(\theta, C) = C' \textbf{ where}\\
&\quad \text{If } \theta C = C \text{ then } C' = C\\
&\quad \text{else } C' = iter(\theta, \theta C)
\end{aligned}
$$

Figure 17. Definition of $\mathcal{R}$.

then

$$\theta C = \left\{ \; q_1 \mathrel{\widehat{=}} 0, q_2 \mathrel{\widehat{=}} 0, q_3 \mathrel{\widehat{=}} q_5 \mathrel{\widehat{+}} q_9, q_4 \mathrel{\widehat{=}} q_6 \mathrel{\widehat{+}} q_{10} \; \right\}.$$

Lemma 37 guarantees soundness of this phase: the algorithm $\mathcal{R}$ does not turn an unsatisfiable constraint set into a satisfiable one. Completeness is left as future work.

LEMMA 37. *If* $\models \mathcal{R}(C)$*, then* $\models C$.

Step 2: Solving linear inequalities After completion of Step 1, γ does not appear in the constraint set. Moreover, constraints of the form $\widehat{\mathbf{empty}}(\cdot)$ have been simplified to ownership constraints. Thus, the residual constraints consist of ones of the form $q_1 \mathrel{\widehat{=}} q_2$ or $q_1 \mathrel{\widehat{>}} q_2$. From the syntax of the constraint language, these constraints form a system of linear inequalities over rational numbers. Thus, we can decide whether the constraints are satisfiable or not. Note that, thanks to the use of rational numbers as ownerships, this step is done in polynomial time on the number of ownership variables and inequalities.

6.4 Implementation

We have implemented an automated verifier based on the algorithm described in this section. The frontend of the verifier that is in charge of $\mathcal{C}$ and $\mathcal{R}$ is implemented with OCaml. As a linear-inequality solver, we use SMT solver Z3 [5]. The Web interface is available at `http://www.fos.kuis.kyoto-u.ac.jp/~rfukuda/freesafety-con/`.

7. Related Work

Terauchi [15] has proposed a type-based race-freedom analysis based on fractional capabilities. His type system assigns a fractional permission to each abstract location. Fractional permissions describe read/write permission to each abstract location. In order to deal with lock-based synchronization and fork/join concurrency, his type system associates what we could call "procurable capabilities" with lock types and thread ID types; each lock type comes with capabilities granted by acquiring/releasing the locks and each thread ID type comes with ones granted by waiting the threads. He also reported the result of experiment.

Our idea of procurable type environments is inspired by his type system. An important difference, however, is that our ownerships also represent *obligation* to be fulfilled (e.g., every lock to be deallocated exactly once before termination), not only *capability*. This leads to the difference in the typing rules for deallocation of resources; his type system does not exclude programs that deallocate locks twice or more.

Besides Terauchi's work [15], the idea of using rational numbers for representing ownerships has been used in the area of program analysis [3, 8, 13, 17]. Among them, the work by Suenaga and Kobayashi [13] and that by Heine and Lam [8] deal with safe memory deallocation. However, they only deal with sequential programs. Extension of the previous techniques, especially one proposed in [13], to concurrency is not trivial: it requires, for example, the acyclicity condition in T-CONFIG, which is not necessary in the sequential setting. Type inference, as we have shown, is nontrivial either due to the presence of procurable type environments.

Gotsman et al. [6] have proposed an extension of the concurrent separation logic [10] with dynamic creation/disposal of locks and threads. Though their work does not deal with safe resource deallocation, it seems that it is not difficult to adapt their framework to do so. They can treat programs that store locks and thread IDs to heap, which is currently not allowed in our framework. They also support structures which, for example, enable a program to create a pair of a memory cell and a lock guarding the cell. We would need to incorporate dependent pairs to the type system in order to deal with such structures. As far as we know, they have not proposed an automated verification based on their framework yet. It seems that their framework requires for spawned threads to terminate with empty ownerships to locks, while ours allow threads to leave release/lock ownerships.

Calcagno et al. [4] have proposed an automated verification technique for concurrent programs. They use biabduction to infer resource invariant, a separation logic formula that describes which part of memory is protected by conditional variables. Our procurable type environments correspond to their resource invariant. The expressiveness of concurrent separation logic allows resource invariant to express more properties than ours (e.g., a pointer not being null). Their work has not been extended with dynamic creation of threads nor conditional variables.

Haack et al. [7] have designed a variant of the concurrent separation logic for multithreaded Java programs. Their framework supports fork/join parallelism and re-entrant monitors. They use fractional permissions to express sharing of a location among several threads. However, it is not clear whether their logic can be easily adapted also for safe resource deallocation because it is based on the *intuitionistic* version of separation logic. Because intuitionistic separation logic admits weakening, their logic would allow facts about the existence of an allocated memory cell to be freely discarded.

Bornat et al. [2] have extended separation logic to concurrency. They use fractional permissions in their assertion language to express sharing of locations among threads. They also support synchronization with conditional variables by which the logic grants access right to critical regions. This feature is comparable to our procurable type environments in the sense that a resource for synchronization (conditional variables, in their case) is associated with some (fixed) permissions to access guarded resources. However, they do not support dynamic creation of threads.

8. Conclusion

We have proposed a type system based on fractional ownerships to guarantee safe resource deallocation and race-freedom in a low-level concurrent language. The type system is a non-trivial extension of the previous type system by Suenaga and Kobayashi [13]. The key ideas of the extension are (1) to assign ownerships not only to reference types but also to lock and thread ID types, and (2) to assign *procurable type environments* to lock types and thread ID types to describe the amount of ownerships granted by or required for operating values of those types. We have also proposed a type inference algorithm for the type system. The algorithm reduces a type inference problem into satisfiability of a system of linear inequalities, which is checked by the SMT solver Z3.

Incorporating more "real-world" features to our framework is another important future direction. For example, we need to add C-like structures to our framework so that programmers can use data structures. This extension could be done as that in the first author's previous work [13]. Extension with pointers to locks is also necessary for dealing with real-world programs. After completing these extensions, we plan to extend FreeSafeTy, the prototype verifier implemented in the previous work [13], and conduct feasibility study by experiment to observe the scalability of our technique and how much manual insertion of must-alias annotations is needed in reality.

The following program, which is considered to be safe in our semantics and also typechecks in our type system, suggests another direction to be pursued:

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\\ \quad \mathbf{acq}(x); \mathbf{let}\ y = \mathbf{fork}(\mathbf{rel}(x))\ \mathbf{in}\ \mathbf{wait}(y); \mathbf{freelock}(x).\end{array}$$

This program is defined to be erroneous in many thread libraries including pthreads because the thread that acquires a lock is different from one that releases the lock. Extending our framework to exclude more of such bad behaviors that consist in real-world software is an important task.

In the current paper, the type system excludes all the racy programs. However, in reality, it is sometimes convenient to allow some memory cells to be accessed in a racy manner. Allowing such partially racy programs would be another interesting future work.

Acknowledgments We are grateful for valuable comments by the anonymous reviewers. Suenaga is partially supported by KAKENHI 23·571 and Hakubi Project at Kyoto University. Igarashi is partially supported by KAKENHI 23220001.

References

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

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

[3] J. Boyland. Checking interference with fractional permissions. In *Proceedings of SAS 2003*, volume 2694 of *LNCS*, pages 55–72. Springer-Verlag, 2003.

[4] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *Proceedings of APLAS 2009*, pages 259–274, 2009.

[5] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[6] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. Technical Report MSR-TR-2007-39, Microsoft Research, 2007.

[7] C. Haack, M. Huisman, and C. Hurlinc. Permission-based separation logic for multithreaded Java programs. `http://fmt.cs.utwente.nl/files/projects/VerCors.p1.pdf`.

[8] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proc. of PLDI*, pages 168–181, 2003.

[9] IEEE. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition*, 2004. `http://pubs.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html`.

[10] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

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

[12] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India*, volume 5536 of *LNCS*, pages 155–170. Springer, Dec. 2008.

[13] K. Suenaga and N. Kobayashi. Fractional ownerships for safe memory deallocation. In Z. Hu, editor, *Programming*

Languages and Systems, 7th Asian Symposium, APLAS 2009, volume 5904 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, Dec. 2009.

[14] N. Swamy, M. W. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in Cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

[15] T. Terauchi. Checking race freedom via linear programming. In *Proc. of PLDI*, pages 1–10, 2008.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Info. Comput.*, 132(2):109–176, 1997.

[17] K. Ueda. Resource-passing concurrent programs. In *Proceedings of 4th International Symposium on Theoretical Aspects of Computer Science (TACS2001)*, volume 2215 of *LNCS*, pages 95–126. Springer-Verlag, 2001.

[18] D. Walker, K. Crary, and J. G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, 2000.

Appendix

A. Proof of Type Soundness (Theorem 21)

To prove the soundness theorem, we first define invariants for well-typed configuration.

DEFINITION 38. *We write $\mathcal{A}$ for the set $\{\mathtt{R}\} \times \mathbb{Q}^{\geq 0} \cup \{\mathtt{L}\} \times \mathbb{Q}^{\geq 0} \times \mathbb{Q}^{\geq 0} \cup \{\mathtt{P}\} \times \mathbb{Q}^{\geq 0}$. Sums and multiplication by a rational number on $\mathcal{A}$ are defined as follows.*

$$\begin{array}{rcl} (\mathtt{R}, f_1) + (\mathtt{R}, f_2) & = & (\mathtt{R}, f_1 + f_2) \\ (\mathtt{L}, f_{11}, f_{12}) + (\mathtt{L}, f_{21}, f_{22}) & = & (\mathtt{L}, f_{11} + f_{21}, f_{12} + f_{22}) \\ (\mathtt{P}, f_1) + (\mathtt{P}, f_2) & = & (\mathtt{P}, f_1 + f_2) \\ \\ f \cdot (\mathtt{R}, f') & = & (\mathtt{R}, f \cdot f') \\ f \cdot (\mathtt{L}, f_1, f_2) & = & (\mathtt{L}, f \cdot f_1, f \cdot f_2) \\ f \cdot (\mathtt{P}, f') & = & (\mathtt{P}, f \cdot f'). \end{array}$$

These operations on $\mathcal{A}$ are pointwise to those on $\mathbf{Var} \overset{\mathbf{fin}}{\to} \mathcal{A}$. Let $F_1, F_2 \in \mathbf{Var} \overset{\mathbf{fin}}{\to} \mathcal{A}$. Then,

$$\begin{array}{rcl} (F_1 + F_2)(x) & = & \begin{cases} F_1(x) & (x \in dom(F_1) \backslash dom(F_2)) \\ F_2(x) & (x \in dom(F_2) \backslash dom(F_1)) \\ F_1(x) + F_2(x) & \\ \quad (x \in dom(F_1) \cap dom(F_2)) & \end{cases} \\ (f \cdot F)(x) & = & f \cdot F(x). \end{array}$$

DEFINITION 39. *$\mathbf{OwnH}(H, v, \kappa)$ is defined as follows.*

$$\begin{array}{ll} \mathbf{OwnH}(H, v, \kappa)(\pi) = \emptyset & (v \notin dom(H)) \\ \mathbf{OwnH}(H, x, \kappa\, \mathbf{ref}_0)(\varepsilon) = \emptyset & \\ \mathbf{OwnH}(H, x, \kappa\, \mathbf{ref}_f)(\varepsilon) = \{x \mapsto (\mathtt{R}, f)\} \quad (f > 0) & \\ \mathbf{OwnH}(H, x, \kappa\, \mathbf{ref}_f)(0\pi) = \mathbf{OwnH}(H, H(x), \kappa)(\pi) & \end{array}$$

DEFINITION 40. *$\mathbf{Own}(P, H, L, v, \tau)$ is defined as follows.*

$$\begin{array}{l} \mathbf{Own}(P, H, L, v, \kappa) = \\ \quad \sum_{\pi \in \{0\}^*} \mathbf{OwnH}(H, v, \kappa)(\pi) \quad \text{if } v \in \{\mathbf{null}\} \cup dom(H) \\ \mathbf{Own}(P, H, L, x, (\Gamma, f_r)\, \mathbf{lock}_{f_o}) = \\ \quad \{x \mapsto (\mathtt{L}, f_r, f_o)\} + f_o \cdot \mathbf{Own}(P, H, L, R, \Gamma) \\ \quad \text{if } x \in dom(L) \text{ and } L(x) = \bot \\ \mathbf{Own}(P, H, L, x, (\Gamma, f_r)\, \mathbf{lock}_{f_o}) = \{x \mapsto (\mathtt{L}, f_r, f_o)\} \\ \quad \text{if } x \in dom(L) \text{ and } L(x) = \top \\ \mathbf{Own}(P, H, L, x, \Gamma\, \mathbf{tid}_f) = \{x \mapsto (\mathtt{P}, f)\} \quad \text{if } x \in dom(P) \\ \mathbf{Own}(P, H, L, v, \tau) = \emptyset \\ \quad \text{if } v \notin dom(H) \cup dom(L) \cup dom(P) \cup \{\mathbf{null}\} \\ \\ \mathbf{Own}(P, H, L, R, \emptyset) = \emptyset \\ \mathbf{Own}(P, H, L, R, (\Gamma, x : \tau)) = \\ \quad \mathbf{Own}(P, H, L, R, \Gamma) + \mathbf{Own}(P, H, L, R(x), \tau) \end{array}$$

If $\sum_{\pi \in \{0\}^*} \mathbf{OwnH}(H, v, \kappa)(\pi)$ in the definition above does not converge, $\mathbf{Own}(P, H, L, v, \kappa)$ is not defined.

Intuitively, $\mathbf{Own}(P, H, L, x, \tau)(y) = f$ represents that the ownership f on y is held by x of type τ.

DEFINITION 41. *We write $t_i \prec_{\widetilde{t}, \widetilde{\Gamma}, R} t_j$ if there exists $x \in dom(\Gamma_i)$ such that*

- $\neg\mathbf{empty}(\Gamma_i(x))$ *and*
- $R(x) = t_j$.

We often omit "$\widetilde{t}, \widetilde{\Gamma}, R$" from $\prec_{\widetilde{t},\widetilde{\Gamma},R}$ if they are obvious from contexts.

DEFINITION 42 (Acyclicity). *A sequence $(t_1, \Gamma_1), \ldots, (t_n, \Gamma_n)$ of pairs of a variable and a ype environment is* acyclic with respect to R, *written* $\mathbf{acyclic}(\widetilde{t}, \widetilde{\Gamma}, R)$, *if $t \not\prec^{+}_{\widetilde{t},\widetilde{\Gamma'},R} t$ for any $t \in \widetilde{t}$.*

DEFINITION 43. *A tuple (P, H, L, R, Γ) is* ownership-consistent, *written* $\mathbf{ConOwn}(P, H, L, R, \Gamma)$, *if the following conditions hold for any $x \in dom(F) \cup dom(H) \cup dom(L) \cup dom(P)$ where $F = \mathbf{Own}(P, H, L, R, \Gamma)$:*

- $F(x) = (\mathtt{R}, 1)$ *if* $x \in dom(H)$
- $F(x) = (\mathtt{L}, 0, 1)$ *if* $x \in dom(L)$ *and* $L(x) = \bot$
- $F(x) = (\mathtt{L}, 1, 1)$ *if* $x \in dom(L)$ *and* $L(x) = \top$
- $F(x) = (\mathtt{P}, 1)$ *if* $x \in dom(P)$
- $F(x)$ *is* $(\mathtt{R}, 0)$, $(\mathtt{L}, 0, 0)$ *or* $(\mathtt{P}, 0)$ *if* $x \in dom(F) \backslash (dom(H) \cup dom(L) \cup dom(P))$.

Proof of Lemma 24. Straightforward. To see $\mathbf{ConOwn}(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$, notice that $\sum_{x \in dom(\emptyset)} \mathbf{Own}(\ldots)$ is $\emptyset$. □

LEMMA 44. *If $\Theta; \Gamma, x : \tau \vdash s \Rightarrow \Gamma, x : \tau'$ and $\mathbf{empty}(\tau)$, then $\mathbf{empty}(\tau')$.*

PROOF. Induction on the derivation of $\Theta; \Gamma, x : \tau \vdash s \Rightarrow \Gamma, x : \tau'$. □

LEMMA 45. *If $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$, then $dom(\Gamma_1) = dom(\Gamma_2)$.*

PROOF. Induction on the derivation of $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$. □

LEMMA 46 (Weakening). *If $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$ and $\mathbf{wf}(\Gamma_1, x{:}\tau)$, then $\Theta; \Gamma_1, x : \tau \vdash s \Rightarrow \Gamma_2, x : \tau$.*

PROOF. Induction on the derivation of $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$. □

LEMMA 47. $\Gamma[x \mapsto \tau] = \Gamma + (x : \tau)$ *if $x \notin dom(\Gamma)$. Specifically, $\Gamma, x : \tau = \Gamma + \{x \mapsto \tau\}$.*

PROOF. Case analysis on $x \in \mathbf{FV}(\Gamma)$. If $x \notin \mathbf{FV}(\Gamma)$, then $\Gamma[x \mapsto \tau] = \Gamma + \{x \mapsto \tau\}$ from Definition 18. Then, $\Gamma[x \mapsto \tau] = \Gamma \backslash \{x\} + (x : \tau)$ holds because $\Gamma \backslash \{x\} = \Gamma$.

If $x \in \mathbf{FV}(\Gamma)$, then $\Gamma[x \mapsto \tau] = \Gamma \backslash \{x\} \uplus \{x \mapsto \tau\}$. This is equal to $\Gamma \backslash \{x\} + \{x \mapsto \tau\}$ from Definition 18. □

LEMMA 48. $\emptyset + \Gamma = \Gamma$

PROOF. Obvious from the definition of the sum of type environments. □

LEMMA 49. $\mathbf{OwnH}(H, v, \kappa)(\pi) = \emptyset$ *for any π if* $\mathbf{empty}(\kappa)$.

LEMMA 50. $\mathbf{Own}(P, H, L, R, \emptyset) = \emptyset$.

PROOF. Obvious from the definition of $\mathbf{Own}$. □

LEMMA 51. $\mathbf{Own}(P, H, L, R, \Gamma_1 + \Gamma_2) = \mathbf{Own}(P, H, L, R, \Gamma_1) + \mathbf{Own}(P, H, L, R, \Gamma_2)$.

PROOF. Induction on the size of $dom(\Gamma_1) + dom(\Gamma_2)$.

- Case $|dom(\Gamma_1) + dom(\Gamma_2)| = 0$: In this case, $\Gamma_1 = \Gamma_2 = \emptyset$. Then,

$$\begin{aligned}&\mathbf{Own}(P, H, L, R, \emptyset + \emptyset) = \emptyset + \mathbf{Own}(P, H, L, R, \emptyset)\\ &\quad (\because \text{from Lemma 48 and Definition 38})\\ &= \mathbf{Own}(P, H, L, R, \emptyset) + \mathbf{Own}(P, H, L, R, \emptyset) \quad (\because \text{Lemma 50}).\end{aligned}$$

- Case $|dom(\Gamma_1) + dom(\Gamma_2)| > 0$:
  - $|dom(\Gamma_1)| = 0$: In this case $\Gamma_1 = \emptyset$. Thus,

$$\begin{aligned}&\mathbf{Own}(P, H, L, R, \Gamma_1 + \Gamma_2)\\ &= \emptyset + \mathbf{Own}(P, H, L, R, \Gamma_2)\\ &\quad (\because \text{from Lemma 48 and Definition 38})\\ &= \mathbf{Own}(P, H, L, R, \Gamma_1 + \Gamma_1) + \mathbf{Own}(P, H, L, R, \Gamma_2)\\ &\quad (\because \text{from Lemma 50}).\end{aligned}$$

  - $|dom(\Gamma_1)| > 0$: In this case, there exists Γ'_1, x and τ such that $\Gamma_1 = \Gamma'_1[x : \tau]$ and $x \notin dom(\Gamma'_1)$. Then,

$$\begin{aligned}&\mathbf{Own}(P, H, L, R, \Gamma_1 + \Gamma_2)\\ &= \mathbf{Own}(P, H, L, R, \Gamma'_1[x : \tau] + \mathbf{Own}(P, H, L, R, \Gamma_2)\\ &= \mathbf{Own}(P, H, L, R, (\Gamma'_1 + (x : \tau)) + \mathbf{Own}(P, H, L, R, \Gamma_2)\\ &\quad (\because \text{from Lemma 47})\\ &= \mathbf{Own}(P, H, L, R, \Gamma_1 + \Gamma_1) + \mathbf{Own}(P, H, L, R, \Gamma_2)\\ &\quad (\because \text{from Lemma 50}).\end{aligned}$$

DEFINITION 52. *Type environments $\Gamma\downarrow_S$ and $\Gamma\uparrow_S$, where $S \subseteq \mathbf{Var}$, are defined as follows.*

$$\begin{array}{rcll}\emptyset\downarrow_S &=& \emptyset & \\ (\Gamma, x : \tau)\downarrow_S &=& \Gamma\downarrow_S & (\{x\} \cup \mathbf{FV}(\tau)) \cap S = \emptyset\\ (\Gamma, x : \tau)\downarrow_S &=& \Gamma\downarrow_S, x : \tau & (\{x\} \cup \mathbf{FV}(\tau)) \cap S \neq \emptyset\end{array}$$

$$\begin{array}{rcll}\emptyset\uparrow_S &=& \emptyset & \\ (\Gamma, x : \tau)\uparrow_S &=& \Gamma\uparrow_S & (\{x\} \cup \mathbf{FV}(\tau)) \cap S \neq \emptyset\\ (\Gamma, x : \tau)\uparrow_S &=& \Gamma\uparrow_S, x : \tau & (\{x\} \cup \mathbf{FV}(\tau)) \cap S = \emptyset\end{array}$$

(Note that $\Gamma\downarrow_S$ and $\Gamma\uparrow_S$ are not necessarily well formed.)

LEMMA 53. $\Gamma = \Gamma\downarrow_S + \Gamma\uparrow_S$.

PROOF. Induction on $|\Gamma|$. □

LEMMA 54. *Suppose $\mathbf{ConOwn}(P, H, L, R, \Gamma)$ and let S_y be the set $\{x \in dom(R) \mid R(x) = R(y)\}$ for $y \in dom(R) \cap dom(\Gamma)$ and $\Gamma = \Gamma_1 + \cdots + \Gamma_n$. Then,*

- *If $R(y) \in dom(P)$ then $\mathbf{Own}(P, H, L, R, \Gamma\downarrow_{S_y}) = \{R(y) \mapsto (\mathtt{P}, 1)\}$.*
- *If $R(y) \in dom(L)$ and $L(R(y)) = \bot$ then $\mathbf{Own}(P, H, L, R, \Gamma\downarrow_{S_y}) = \{R(y) \mapsto (\mathtt{L}, 0, 1)\}$.*
- *If $R(y) \in dom(L)$ and $L(R(y)) = \top$ then $\mathbf{Own}(P, H, L, R, \Gamma\downarrow_{S_y}) = \{R(y) \mapsto (\mathtt{L}, 1, 1)\}$.*

PROOF. We only prove the first item. The proofs of the other items are similar.

We have $\mathbf{Own}(P, H, L, R, \Gamma) = \mathbf{Own}(P, H, L, R, \Gamma{\downarrow_{S_y}}) + \mathbf{Own}(P, H, L, R, \Gamma{\uparrow_{S_y}})$ from Lemma 53. We claim that $R(y) \notin \mathit{dom}(\mathbf{Own}(P, H, L, R, \Gamma{\uparrow_{S_y}}))$. Then, $\mathbf{Own}(P, H, L, R, \Gamma{\downarrow_{S_y}})(R(y)) = \mathbf{Own}(P, H, L, R, \Gamma)(R(y)) = (\mathtt{P}, 1)$ as required. To observe $R(y) \notin \mathit{dom}(\mathbf{Own}(P, H, L, R, \Gamma{\uparrow_{S_y}})$, note that, for any Γ', v and f, $\mathbf{Own}(P, H, L, R, \Gamma')(v) = (\mathtt{P}, f)$ only if there exists $x \in \mathit{dom}(R) \cap \mathit{dom}(\Gamma')$ such that $R(x) = v$ and $\Gamma'(x) = \Gamma''\ \mathbf{tid}_f$ for some Γ'' from the definition of $\mathbf{Own}$. Then, if $R(y)$ belonged $\mathit{dom}(\mathbf{Own}(P, H, L, R, \Gamma{\uparrow_{S_y}}))$, then there should be x such that $x \in \mathit{dom}(R) \cap (\mathit{dom}(\Gamma)\backslash S_y)$ and $R(x) = R(y)$, which is in fact impossible from the definition of S_y.
□

LEMMA 55. *The following statements are equivalent:*

1. $\Theta; \Gamma_1 \vdash E[s] \Rightarrow \Gamma_2$.
2. $\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_3$ *and* $\Theta; \Gamma_3 \vdash E[\mathbf{skip}] \Rightarrow \Gamma_2$ *for some* Γ_3.

PROOF. Induction on the structure of E. □

DEFINITION 56 (κ-reachability). $\mathbf{Paths}(h_1, H, \kappa, h_2)$ *is the set*

$$\left\{ \pi \in \{0\}^* \;\middle|\; \begin{array}{l} H^{|\pi|}(h_1) = h_2 \textit{ and } \kappa(\pi') > 0 \\ \textit{for any prefix } \pi' \textit{ of } \pi. \end{array} \right\}$$

If $\mathbf{Paths}(h_1, H, \kappa, h_2) \neq \emptyset$*, then we say* h_2 is κ-reachable from h_1 on H.

LEMMA 57. *If* $h_1 \neq h_2$*, then* h_2 *is* κ*-reachable from* $H(h_1)$ *on* H *if* h_2 *is* $\kappa\ \mathbf{ref}_f$*-reachable from* h_1 *for some* $f > 0$.

PROOF. Suppose h_2 is $\kappa\ \mathbf{ref}_f$-reachable from h_1 for some $f > 0$. Then, there exists π_1 such that $H^{|\pi_1|}(h_1) = h_2$ and $\kappa(\pi_2) > 0$ for any prefix π_2 of π_1. Conduct case-analysis on π_1. From $h_1 \neq h_2$, π_1 cannot be an empty sequence. Suppose $\pi_1 = \pi_3 0$. Then, $H^{|\pi_3 0|}(h_1) = h_2$ implies $H^{|\pi_3|}(H(h_1)) = h_2$. Thus, it suffices to show that, for any prefix π_4 of π_3, $\kappa(\pi_4) > 0$, which easily follows from the fact that any prefix π_3 is a prefix of π_1.

Suppose h_2 is not $\kappa\ \mathbf{ref}_f$-reachable from h_1 for some $f > 0$ but h_2 is κ-reachable from $H(h_1)$. Then, there exists π_1 such that $H^{|\pi_1|}(H(h_1)) = h_2$ and $\kappa(\pi_2) > 0$ for any prefix π_2 of π_1. We show that $0\pi_1 \in \mathbf{Paths}(h_1, H, \kappa\ \mathbf{ref}_f, h_2)$. This follows from the following reasoning:

- $H^{|0\pi_1|}(h_1) = h_2$, which follows from $H^{|0\pi_1|}(h_1) = H^{|\pi_1|}(H(h_1))$, and
- $(\kappa\ \mathbf{ref}_f)(\pi_3) > 0$ for any prefix π_3 of $0\pi_1$, which follows from the following case analysis.
  - If π_3 is an empty sequence, then $(\kappa\ \mathbf{ref}_f(\pi_3) = f > 0$.
  - Otherwise, $\pi_3 = 0\pi_4$ for some π_4, thus $(\kappa\ \mathbf{ref}_f)(\pi_3) = (\kappa\ \mathbf{ref}_f)(0\pi_4) = \kappa(\pi_4$. Here, because π_4 is a prefix of π_1, we have $\kappa(\pi_4) > 0$.

□

LEMMA 58 (Obliviousness). *1.* $\mathbf{Own}(P, H, L, R, \Gamma) = \mathbf{Own}(P', H, L, R', \Gamma)$ *if* $R \subseteq R'$ *and either (1) for any* $x \in \mathit{dom}(\Gamma)$*,* $R(x) \in \mathit{dom}(P) \cup \mathit{dom}(P')$ *implies* $R(x) \in \mathit{dom}(P) \cap \mathit{dom}(P')$ *or (2)* $\mathbf{empty}(\Gamma)$.

2. $\mathbf{Own}(P, H, L, R, \Gamma) = \mathbf{Own}(P', H, L', R', \Gamma)$ *if* $\mathit{dom}(P) = \mathit{dom}(P')$ *and* $R \subseteq R'$ *and either (1) for any* $x \in \mathit{dom}(\Gamma)$*,* $R(x) \in (\mathit{dom}(L) \cup \mathit{dom}(L'))$ *and* $z \notin \mathbf{FV}(\Gamma(x))$ *imply* $L(R(z)) = L'(R(z))$ *or (2)* $\mathbf{empty}(\Gamma)$.

3. $\mathbf{OwnH}(H[h \mapsto v'], h', \kappa) = \mathbf{OwnH}(H[h \mapsto v], h', \kappa)$ *if* h *is not* κ*-reachable from* h' *and* $v \neq v'$.

4. $\mathbf{Own}(P, H[h \mapsto v'], L, R, \Gamma) = \mathbf{Own}(P', H[h \mapsto v], L, R', \Gamma)$ *if* $\mathit{dom}(P) = \mathit{dom}(P')$ *and* $R \subseteq R'$ *and* $v \neq v'$ *and, either (1) for any* $x \in \mathit{dom}(\Gamma)$*,* $R(x) \in \mathit{dom}(H)\backslash\{h\}$ *and* h *is not* $\Gamma(x)$*-reachable from* $R(x)$ *on* H*, or (2)* $\mathbf{empty}(\Gamma)$.

PROOF.

1. We show

   $$\mathbf{Own}(P, H, L, R(x), \Gamma(x)) = \mathbf{Own}(P', H, L, R'(x), \Gamma(x))$$

   for any $x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(R)$. The only non-trivial case is $R(x) \in \mathit{dom}(P)$. In this case, we have $\Gamma(x) = \Gamma'\ \mathbf{tid}_f$ and $\mathbf{Own}(P, H, L, R(x), \Gamma(x)) = \{R(x) \mapsto (\mathtt{P}, f)\}$ and, from $R(x) \in \mathit{dom}(P) \cap \mathit{dom}(P')$,

   $$\mathbf{Own}(P', H, L, R'(x), \Gamma(x)) = \{R'(x) \mapsto (\mathtt{P}, f)\}$$

   for some f. The conclusion follows from $R(x) = R'(x)$ because $x \in \mathit{dom}(R)$ and $R \subseteq R'$.

2. Let $\Gamma_1 \prec \Gamma_2$ be the least relation that satisfies the following rules:

   $$\frac{\Gamma_2(x) = (\Gamma_1, f_1)\ \mathbf{lock}_{f_2}}{\Gamma_1 \prec \Gamma_2}$$

   $$\frac{\Gamma_2(x) = (\Gamma_1', f_1)\ \mathbf{lock}_{f_2} \qquad \Gamma_1 \prec \Gamma_1'}{\Gamma_1 \prec \Gamma_2}$$

   Intuitively, $\Gamma_1 \prec \Gamma_2$ represents that Γ_1 appears in Γ_2 as a procurable environment of a lock type. It should be easy to observe that $\prec$ is well-founded. We show $\mathbf{Own}(P, H, L, R(x), \Gamma(x)) = \mathbf{Own}(P', H, L', R'(x), \Gamma(x))$ for any $x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(R)$ by well-founded induction on $\prec$. Suppose $R(x) \in \mathit{dom}(L)$. (Otherwise, the conclusion easily follows.) Then, $\Gamma(x) = (\Gamma', f_1)\ \mathbf{lock}_{f_2}$ for some Γ', f_1 and f_2, and,

   $$\mathbf{Own}(P, H, L, R(x), \Gamma(x)) = \begin{cases} \{R(x) \mapsto (\mathtt{L}, f_1, f_2)\} + f_2 \cdot \mathbf{Own}(P, H, L, R, \Gamma') \\ \quad \text{if } L(R(x)) = \bot \\ \{R(x) \mapsto (\mathtt{L}, f_1, f_2)\} \\ \quad \text{if } L(R(x)) = \top, \end{cases}$$

and

$$\mathbf{Own}(P', H, L', R'(x), \Gamma(x)) = \begin{cases} \{R'(x) \mapsto (\mathtt{L}, f_1, f_2)\} + f_2 \cdot \mathbf{Own}(P', H, L', R', \Gamma') \\ \quad \text{if } L'(R'(x)) = \bot \\ \{R'(x) \mapsto (\mathtt{L}, f_1, f_2)\} \\ \quad \text{if } L'(R'(x)) = \top. \end{cases}$$

We have $R(x) = R'(x)$ from $x \in \mathit{dom}(R)$ and $R \subseteq R'$, and $L(R(x)) = L'(R'(x))$ from assumption. Conduct case-analysis on $L(R(x))$. The case $L(R(x)) = L'(R'(x)) = \top$ is easy. Otherwise, from $\Gamma' \prec \Gamma$ and I.H., $\mathbf{Own}(P, H, L, R, \Gamma) = \mathbf{Own}(P', H, L', R', \Gamma')$ follows.

3. We show $\mathbf{OwnH}(H[h \mapsto v'], h', \kappa)(\pi) = \mathbf{OwnH}(H[h \mapsto v], h', \kappa)(\pi)$ if h is not κ-reachable from h' on $H[h \mapsto v']$ by induction on π. The base case is easy. Suppose $\pi = 0\pi'$ for some π'. Then, from the definition of $\mathbf{OwnH}$,

$$\mathbf{OwnH}(H[h \mapsto v'], h', \kappa)(\pi) = \begin{cases} \emptyset \quad \text{if } h' \notin \mathit{dom}(H) \cup \{h\} \\ \mathbf{OwnH}(H[h \mapsto v'], (H[h \mapsto v'])(h'), \kappa')(\pi') \\ \quad \text{if } h' \in \mathit{dom}(H) \cup \{h\} \text{ and } \kappa = \kappa' \ \mathbf{ref}_f \text{ for some } f \end{cases}$$

and

$$\mathbf{OwnH}(H[h \mapsto v], h', \kappa)(\pi) = \begin{cases} \emptyset \quad \text{if } h' \notin \mathit{dom}(H) \cup \{h\} \\ \mathbf{OwnH}(H[h \mapsto v], (H[h \mapsto v])(h'), \kappa')(\pi') \\ \quad \text{if } h' \in \mathit{dom}(H) \cup \{h\} \text{ and } \kappa = \kappa' \ \mathbf{ref}_f \text{ for some } f. \end{cases}$$

If $h' \notin \mathit{dom}(H) \cup \{h\}$, then the conclusion is immediate. Suppose $h' \in \mathit{dom}(H) \cup \{h\}$ and $\kappa = \kappa' \ \mathbf{ref}_f$ for some f. Then,

$$\mathbf{OwnH}(H[h \mapsto v'], h', \kappa)(\pi) = \mathbf{OwnH}(H[h \mapsto v'], (H[h \mapsto v'])(h'), \kappa')(\pi').$$

Then, from Lemma 57, h is not κ'-reachable from $(H[h \mapsto v'])(h')$ on $H[h \mapsto v']$. Thus, from I.H.,

$$\mathbf{OwnH}(H[h \mapsto v'], (H[h \mapsto v'])(h'), \kappa')(\pi') = \mathbf{OwnH}(H[h \mapsto v], (H[h \mapsto v'])(h'), \kappa')(\pi').$$

Here, note that we have either (1) $h \neq h'$ or (2) $\mathbf{empty}(\kappa)$. Indeed, if $h = h'$, then, because h is not κ-reachable from h', $\kappa(\epsilon) = 0$. This implies $\mathbf{empty}(\kappa)$. (Recall that, for any π', $\kappa(\pi\pi') = 0$ if $\kappa(\pi) = 0$.) If $h \neq h'$, then $H[h \mapsto v'])(h') = (H[h \mapsto v'])(h')$, thus $\mathbf{OwnH}(H[h \mapsto v], (H[h \mapsto v'])(h'), \kappa')(\pi') = \mathbf{OwnH}(H[h \mapsto v], (H[h \mapsto v])(h'), \kappa')(\pi')$, which concludes the case. If $\mathbf{empty}(\kappa)$, then $\mathbf{empty}(\kappa')$, thus the conclusion follows from Lemma 49.

4. It suffices to show that, for any $x \in \mathit{dom}(\Gamma) \cap \mathit{dom}(R)$, $\mathbf{Own}(P, H[h \mapsto v'], L, R(x), \Gamma(x)) = \mathbf{Own}(P, H[h \mapsto v], L, R(x), \Gamma(x))$. The only non-trivial case is $R(x) \in \{\mathbf{null}\} \cup \mathit{dom}(H)$. In this case,

$$\mathbf{Own}(P, H[h \mapsto v'], L, R(x), \Gamma(x)) = \textstyle\sum_{\pi \in \{0\}^*} \mathbf{OwnH}(H[h \mapsto v'], R(x), \Gamma(x)),$$

and

$$\mathbf{Own}(P, H[h \mapsto v], L, R(x), \Gamma(x)) = \textstyle\sum_{\pi \in \{0\}^*} \mathbf{OwnH}(H[h \mapsto v], R(x), \Gamma(x)).$$

If $R(x) = \mathbf{null}$, then the conclusion immediately follows. Otherwise, the conclusion follows from the previous item of this Lemma and h being not $\Gamma(x)$-reachable from $R(x)$.

LEMMA 59 (Subject reduction for commands). *If* $\Theta; \Gamma \vdash_D (P[t_1 \mapsto E[s_1]], H, L, R)$ **ok** *and* $(s_1, H, L, R) \rightsquigarrow (s_1', H', L', R')$, *then* $\Theta; \Gamma \vdash_D (P[t_1 \mapsto E[s_1']], H', L', R')$ **ok**.

PROOF. By T-CONFIG, we have, in particular,

$$\Theta; \Gamma_1 \vdash E[s_1] \Rightarrow \Gamma_1'' \tag{1}$$
$$\mathbf{ConOwn}(P[t_1 \mapsto E[s_1]], H, L, R, \Gamma_1 + \Gamma_r). \tag{2}$$

for some $\Gamma_1, \Gamma_r, \widetilde{\Gamma''}$ such that $\Gamma = \Gamma_1 + \Gamma_r$. Then, it suffices to show that $\Theta; \Delta_1 \vdash E[s_1'] \Rightarrow \Gamma_1'''$ and $\mathbf{ConOwn}(P[t_1 \mapsto E[s_1']], H', L', R', \Delta_1 + \Gamma_r)$ for some Δ_1 and Γ_1'''.

By Lemma 55, there exists Γ_1' such that $\Theta; \Gamma_1 \vdash s_1 \Rightarrow \Gamma_1'$ and $\Theta; \Gamma_1' \vdash E[\mathbf{skip}] \Rightarrow \Gamma_1''$.

The proof proceeds by case analysis on the rule used to derive $(s_1, H, L, R) \rightsquigarrow (s_1', H', L', R')$. In all cases, it is easy to show $\mathbf{acyclic}(\widetilde{t}, (\Gamma_1''', \Gamma_2'', \ldots, \Gamma_n''), R')$.

We write $F(\cdot)$ for $\mathbf{Own}(P[t_1 \mapsto s_1], H, L, R, \cdot)$ and $F'(\cdot)$ for $\mathbf{Own}(P[t_1 \mapsto s_1'], H', L', R', \cdot)$.

Case E-NEWLOCK***:*** We have

$$\begin{aligned} s_1 &= \mathbf{let}\ x = \mathbf{newlock}()\ \mathbf{in}\ s_1'' && (3) \\ s_1' &= [z/x]s_1'' && (4) \\ H' &= H && (5) \\ L' &= L[l \mapsto \bot] && (6) \\ R' &= R[z \mapsto l] && (7) \end{aligned}$$

for some fresh z and l. Inversion on $\Theta; \Gamma_1 \vdash s_1 \Rightarrow \Gamma_1'$ gives

$$\Gamma_1 = \Gamma_{11} + \Gamma_{12} \tag{8}$$
$$\Theta; \Gamma_{11}, x : (\Gamma_{12}, 0)\ \mathbf{lock}_1 \vdash s_1'' \Rightarrow \Gamma_1', x : \tau \tag{9}$$
$$\mathbf{empty}(\tau) \tag{10}$$

for some Γ_{11}, Γ_{12} and τ. From (9) and freshness of z, we have

$$\Theta; \Gamma_{11}, z : (\Gamma_{12}, 0)\ \mathbf{lock}_1 \vdash [z/x]s_1'' \Rightarrow \Gamma_1', z : \tau \tag{11}$$

By Lemmas 46 and 55,

$$\Theta; \Gamma_{11}, z : (\Gamma_{12}, 0)\ \mathbf{lock}_1 \vdash E[[z/x]s_1''] \Rightarrow \Gamma_1'', z : \tau$$

We claim that it suffices to take $\Gamma_{11}, z : (\Gamma_{12}, 0)\ \mathbf{lock}_1$ as Δ_1 and $\Gamma_1'', z : \tau$ as Γ_1'''. To observe that this choice works, it suffices to show $\mathbf{ConOwn}(P[t_1 \mapsto E[s_1']], H, L', R', \Delta_1 + \Gamma_r)$.

$$F(\Gamma) = F(\Gamma_1 + \Gamma_r)$$

and

$$\begin{array}{rl} & F'((\Gamma_{11}, z : (\Gamma_{12}, 0)\ \mathbf{lock}_1) + \Gamma_r) \\ = & F'(\Gamma_{11}) + \{l \mapsto (\mathtt{L}, 0, 1)\} + 1 \cdot F'(\Gamma_{12}) + F'(\Gamma_r) \\ = & F'(\Gamma_{11} + \Gamma_{12}) + F'(\Gamma_r) + \{l \mapsto (\mathtt{L}, 0, 1)\} \\ = & F'(\Gamma_1 + \Gamma_r) + \{l \mapsto (\mathtt{L}, 0, 1)\}\,. \end{array}$$

Since l is fresh, $l \notin dom(F'(\Gamma_1 + \Gamma_r))$. So, it suffices to show $F(\Gamma_1 + \Gamma_r) = F'(\Gamma_1 + \Gamma_r)$, which follows from Lemma 58. In fact, it is easy to show that, for any $y \in \{R(x) \mid x \in dom(\Gamma_1 + \Gamma_r)\} \cap (dom(L) \cup dom(L'))$ implies $L(y) = L'(y)$ since $\{R(x) \mid x \in dom(\Gamma_1 + \Gamma_r)\} \cap (dom(L) \cup dom(L')) = dom(L)$. Thus, we have $\mathbf{ConOwn}(P[t_1 \mapsto E[s_1']], H, L', R', \Delta_1 + \Gamma_r)$.

Case E-FREELOCK***:*** We have

$$\begin{align} s_1 &= \mathbf{freelock}(x) \tag{12}\\ R(x) &= l \tag{13}\\ L(l) &= \bot \tag{14}\\ s_1' &= \mathbf{skip} \tag{15}\\ H' &= H \tag{16}\\ L' &= L \setminus \{l\} \tag{17}\\ R' &= R. \tag{18} \end{align}$$

Inversion of T-FREELOCK gives

$$\begin{align} \Gamma_1 &= \Gamma_{11}, (\Gamma_{12}, 0)\ \mathbf{lock}_1 \tag{19}\\ \Gamma_1' &= (\Gamma_{11}, x \mapsto (\Gamma_{12}, 0)\ \mathbf{lock}_0) + \Gamma_{12} \tag{20} \end{align}$$

for some Γ_{11} and Γ_{12}. We show that it suffices to take $(\Gamma_{11}, x \mapsto (\Gamma_{12}, 0)\ \mathbf{lock}_0) + \Gamma_{12}$ as Δ_1 and Γ_1'' as Γ_1'''.

We show $\mathbf{ConOwn}(P[t_1 \mapsto E[s_1']], H, L', R, \Delta_1)$. Let us write $F(\cdot)$ for $\mathbf{Own}(P, H, L, R, \cdot)$ and $F'(\cdot)$ for $\mathbf{Own}(P', H, L', R, \cdot)$ and S for $\{x \in dom(R) \mid R(x) = l\}$. Then,

$$\begin{array}{rl} & F(\Gamma_1 + \Gamma_r) \\ = & F(\Gamma_{11}) + F(x : (\Gamma_{12}, 0)\ \mathbf{lock}_1) + F(\Gamma_r) \\ = & F(\Gamma_{11}) + \{l \mapsto (\mathtt{L}, 0, 1)\} + 1 \cdot F(\Gamma_{12}) + F(\Gamma_r) \\ = & F(\Gamma_{11} + \Gamma_{12} + \Gamma_r) + \{l \mapsto (\mathtt{L}, 0, 1)\}\,. \\ = & F((\Gamma_{11} + \Gamma_{12} + \Gamma_r){\downarrow_S}) + \\ & F((\Gamma_{11} + \Gamma_{12} + \Gamma_r) \uparrow_S) + \\ & \{l \mapsto (\mathtt{L}, 0, 1)\} \end{array}$$

$$\begin{array}{rl} & F'(\Delta_1 + \Gamma_r) \\ = & F'((\Gamma_{11}, x \mapsto (\Gamma_{12}, 0)\ \mathbf{lock}_0) + \Gamma_{12} + \Gamma_r) \\ = & F'(\Gamma_{11}) + 0 \cdot F'(\Gamma_{12}) + F'(\Gamma_{12}) + F'(\Gamma_r) \\ = & F'(\Gamma_{11} + \Gamma_{12} + \Gamma_r) \\ = & F'((\Gamma_{11} + \Gamma_{12} + \Gamma_r){\downarrow_S}) + F'((\Gamma_{11} + \Gamma_{12} + \Gamma_r) \uparrow_S) \end{array}$$

On the one hand, by ownership consistency, it must be the case that $F((\Gamma_{11} + \Gamma_{12} + \Gamma_r){\downarrow_S}) \subseteq \{l \mapsto (\mathtt{L}, 0, 0)\}$ and so $\mathbf{empty}(\Gamma_{11} + \Gamma_{12} + \Gamma_r)$. By Lemma 58,

$$F((\Gamma_{11} + \Gamma_{12} + \Gamma_r){\downarrow_S}) = F'((\Gamma_{11} + \Gamma_{12} + \Gamma_r){\downarrow_S}).$$

On the other hand, $l \notin dom(F((\Gamma_{11} + \Gamma_{12} + \Gamma_r) \uparrow_S)$, and so, by Lemma 58(2),

$$F((\Gamma_{11} + \Gamma_{12} + \Gamma_r) \uparrow_S) = F'((\Gamma_{11} + \Gamma_{12} + \Gamma_r) \uparrow_S).$$

Case E-ASSIGN***:*** We have

$$\begin{align} s_1 &= *x \leftarrow y \tag{21}\\ R(x) &= h \tag{22}\\ R(y) &= v \tag{23}\\ H(h) &= v' \tag{24}\\ s_1' &= \mathbf{skip} \tag{25}\\ H' &= H[h \mapsto v] \tag{26}\\ L' &= L \tag{27}\\ R' &= R. \tag{28} \end{align}$$

Inversion of T-ASSIGN gives

$$\begin{align} &\mathbf{empty}(\kappa) \tag{29}\\ &\Gamma_1(x) = \kappa\ \mathbf{ref}_1 \tag{30}\\ &\Gamma_1(y) = \kappa_1 + \kappa_2 \tag{31}\\ &\Gamma_1'(x) = \kappa_1\ \mathbf{ref}_1 \tag{32}\\ &\Gamma_1'(y) = \kappa_2 \tag{33}\\ &\Gamma_1 \setminus \{x, y\} = \Gamma_1' \setminus \{x, y\} \tag{34} \end{align}$$

for some κ, κ_1 and κ_2. We claim that it suffices to take Γ_1' as Δ_1 and Γ_1'' as Γ_1'''. Then, we have

$$\begin{array}{rl} & F(\Gamma_1 + \Gamma_r) \\ = & F(\Gamma_1 \setminus \{x, y\}) + \{h \mapsto (\mathtt{R}, 1)\} + F(\Gamma_r) + \mathbf{OwnH}(H, v, \kappa_1 + \kappa_2) \\ & \quad \text{(Note } \mathbf{empty}(\kappa)\text{)} \\ = & F(\Gamma_1 \setminus \{x, y\} + \Gamma_r) + \{h \mapsto (\mathtt{R}, 1)\} + \mathbf{OwnH}(H, v, \kappa_1 + \kappa_2), \end{array}$$

and

$$\begin{array}{rl} & F'(\Delta_1 + \Gamma_r) \\ = & F'(\Delta_1 \setminus \{x, y\} + \Gamma_r) + \{h \mapsto (\mathtt{R}, 1)\} + \\ & \mathbf{OwnH}(H', v, \kappa_1 + \kappa_2) \\ = & F'(\Delta_1 \setminus \{x, y\} + \Gamma_r) + \{h \mapsto (\mathtt{R}, 1)\} + \\ & \mathbf{OwnH}(H', v, \kappa_1 + \kappa_2) \end{array}$$

It suffices to show

$$\begin{align} F(\Gamma_1 \setminus \{x, y\} + \Gamma_r) = F'(\Gamma_1 \setminus \{x, y\} + \Gamma_r) \text{ and} \tag{35}\\ \mathbf{OwnH}(H, v, \kappa_1 + \kappa_2) = \mathbf{OwnH}(H', v, \kappa_1 + \kappa_2). \tag{36} \end{align}$$

Let us write Γ'' for $\Gamma_1 \setminus \{x, y\} + \Gamma_r$. To show (35), we show that, for any $x \in dom(\Gamma'')$, h is not κ-reachable from$R(x)$ on H if $R(x) \in dom(H) \setminus \{h\}$ where $\kappa =$

$\Gamma''(x)$. Then, (35) follows from Lemma 58. To show this, suppose h is κ-reachable from $R(x)$ on H. Then, there exists π such that $H^{|\pi|}(R(x)) = h$ and $\kappa(\pi') > 0$ for any prefix π' of π, especially $\kappa(\pi) > 0$. Then, $F(\Gamma'')(h) = (\mathtt{R}, \kappa(\pi))$ from the definition of $\mathbf{Own}$, which contradicts with (2).

To show (36), we show h is not $\kappa_1 + \kappa_2$-reachable from v on H. The proof of this fact is almost as the same as the previous one in the previous paragraph.

DEFINITION 60. *We write* $\Gamma_1 \leq \Gamma_2$ *if* $\Gamma_2 = \Gamma_1 + \Gamma_3$ *and* $\mathbf{empty}(\Gamma_3)$ *for some* Γ_3.

Proof of Lemma 25 Assume $\Theta; \Gamma \vdash_D (P, H, L, R)$ **ok** and $(P, H, L, R) \rightsquigarrow (P', H', L', R')$. By T-CONFIG, we have

$$\mathbf{ConOwn}(P, H, L, R, \Gamma) \tag{37}$$
$$\Theta; \Gamma_i \vdash s_i \Rightarrow \Gamma_i' \text{ for each } i \in \{1, \dots, n\} \tag{38}$$
$$\Gamma = \Gamma_1 + \cdots + \Gamma_n \tag{39}$$
$$P = \{t_1 \mapsto s_1, \dots, t_n \mapsto s_n\} \tag{40}$$
$$\vdash D : \Theta \tag{41}$$
$$\text{for any } x,\ R(x) = t_i \text{ implies } \mathbf{PTE}(\Gamma(x)) \leq \Gamma_i' \tag{42}$$
$$t_i = \star \text{ implies } \mathbf{empty}(\Gamma_i') \tag{43}$$

for some $\Gamma_1, \dots, \Gamma_n, \Gamma_1', \dots, \Gamma_n'$. Conduct case-analysis on the last rule to derive $(P, H, L, R) \rightsquigarrow (P', H', L', R')$. We show only that **ConOwn** and **acyclic** are preserved because the other conditions are easy. We deal with **acyclic** only in the case of E-FORK because the other cases are easy.

Case E-ACQ*:* Without loss of generality, we can assume $s_1 = E[\mathbf{acq}(x)]$. Then, $L(R(x)) = \bot$, $P' = P[t_1 \mapsto E[\mathbf{skip}]]$ and $L' = L[R(x) \mapsto \top]$ and $H' = H$ and $R' = R$. Let l be $R(x)$. From Lemma 55 and inversion of T-ACQ, we have

$$\Gamma_1 = \Gamma_1'', x : (\Gamma_2'', 0)\ \mathbf{lock}_f \tag{44}$$
$$\Theta; \Gamma_1'', x : (\Gamma_2'', 1)\ \mathbf{lock}_f + \Gamma_2'' \vdash E[\mathbf{skip}] \Rightarrow \Gamma_1' \tag{45}$$
$$f > 0 \tag{46}$$

for some Γ_1'', Γ_2'' and f. Let Γ_r be $\Gamma_2 + \cdots + \Gamma_n$ and Δ be $\Gamma_1'', x : (\Gamma_2'', 1)\ \mathbf{lock}_f + \Gamma_2'' + \Gamma_r$. We show $\mathbf{ConOwn}(P', H, L', R, \Delta)$. Let $F(\cdot)$ be $\mathbf{Own}(P, H, L, R, \cdot)$ and $F'(\cdot)$ be $\mathbf{Own}(P', H, L', R, \cdot)$. Then,

$$\begin{array}{rl} & F(\Gamma_1 + \Gamma_r) \\ = & F(\Gamma_1'') + F(x : (\Gamma_2'', 0)\ \mathbf{lock}_f) + F(\Gamma_r) \\ & \quad (\because \text{Lemmas 47 and 51}) \\ = & F_1(\Gamma_1'' + \Gamma_r) + \{l \mapsto (\mathtt{L}, 0, f)\} + f \cdot F_1(\Gamma_2'') \\ & \quad (\because (\text{Lemma 51 and Definition of } \mathbf{Own})). \end{array}$$

Let S_l be $\{z \mid R(z) = l\}$. Then, by Lemma 53,

$$\begin{array}{l} F(\Gamma_1 + \Gamma_r) = \\ \quad F((\Gamma_1'' + \Gamma_r){\downarrow_{S_l}}) + F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + \\ \quad \{l \mapsto (\mathtt{L}, 0, f)\} + f \cdot F(\Gamma_2'') \end{array}$$

We have $l \notin dom(F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}))$ by the definition of **Own**. We also have $l \notin dom(f \cdot F(\Gamma_2''))$, since, otherwise, $x \in dom(\Gamma_2'')$, which contradicts the well-formedness of Γ_1. We claim $F((\Gamma_1'' + \Gamma_r){\downarrow_{S_l}}) + \{l \mapsto (\mathtt{L}, 0, f)\} + f \cdot F(\Gamma_2'') = \{l \mapsto (\mathtt{L}, 0, 1)\} + F(\Gamma_2'')$. In fact,

- if $(\Gamma_1'' + \Gamma_r) {\downarrow_{S_l}} \neq \emptyset$, then from Lemma 54, we have $F((\Gamma_1'' + \Gamma_r){\downarrow_{S_l}}) = \{l \mapsto (\mathtt{L}, f_1, f_1')\} + f_1' \cdot F(\Gamma_2'')$ for some $0 \leq f_1' \leq 1$. Then, from (37), we have $f_1 = 0$ and $f_1' = 1 - f$, and
- if $(\Gamma_1'' + \Gamma_r){\downarrow_{S_l}} = \emptyset$, then

$$\begin{array}{rl} & F((\Gamma_1'' + \Gamma_r){\downarrow_{S_l}}) + F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + \\ & \{l \mapsto (\mathtt{L}, 0, f)\} + f \cdot F(\Gamma_2'') \\ = & F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + \{l \mapsto (\mathtt{L}, 0, f)\} + f \cdot F(\Gamma_2''), \end{array}$$

  which is followed by $f = 1$, thanks to (37).

Thus, in both cases, we have $F(\Gamma_1 + \Gamma_r) = F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + \{l \mapsto (\mathtt{L}, 0, 1)\} + F(\Gamma_2'')$.

We next calculate $F'(\Delta)$, which goes as follows:

$$\begin{array}{rl} & F'(\Delta) \\ = & F'(\Gamma_1'', x : (\Gamma_2'', 1)\ \mathbf{lock}_f + \Gamma_2'' + \Gamma_r) \\ = & F'(\Gamma_1'' + \Gamma_r) + F'(\Gamma_2'') + \{l \mapsto (\mathtt{L}, 1, f)\} \\ = & F'((\Gamma_1'' + \Gamma_r){\downarrow_{S_l}}) + F'((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + \\ & F'(\Gamma_2'') + \{l \mapsto (\mathtt{L}, 1, f)\}\,. \end{array}$$

Then, by a similar reasoning, we have $F'((\Gamma_1'' + \Gamma_r) {\downarrow_{S_l}}) + \{l \mapsto (\mathtt{L}, 1, f)\} = \{l \mapsto (\mathtt{L}, 1, 1)\}$. Thus,

$$F'(\Delta) = F'((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + F'(\Gamma_2'') + \{l \mapsto (\mathtt{L}, 1, 1)\}$$

Note that

- $F((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) = F'((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}})$ from Lemma 58;
- $F(\Gamma_2'') = F'(\Gamma_2'')$ from Lemma 58; and
- $l \notin F'((\Gamma_1'' + \Gamma_r) {\uparrow_{S_l}}) + F'(\Gamma_2'')$.

which is followed by $\mathbf{ConOwn}(P', H, L', R, \Delta)$ as required.

Case E-REL*:* Similar to the case for E-ACQ.

Case E-FORK*:* We assume that $s_1 = E[\mathbf{let}\ x = \mathbf{fork}(s_{n+1})\ \mathbf{in}\ s_1']$. Then, $P' = P[t_1 \mapsto [y/x]s_1', t_{n+1} \mapsto s_{n+1}]$ and $H' = H$ and $L' = L$ and $R' = R[y \mapsto t_{n+1}]$ for fresh y and t_{n+1}. From Lemma 55 and inversion of T-FORK, we have

$$\Gamma_1 = \Gamma_{11} + \Gamma_{12} \tag{47}$$
$$\Theta; \Gamma_{12} \vdash s_{n+1} \Rightarrow \Gamma_{12}' \tag{48}$$
$$\Theta; \Gamma_{11}, x : \Gamma_{12}'\ \mathbf{tid}_1 \vdash s_1' \Rightarrow \Gamma_{11}', x : \tau \tag{49}$$
$$\mathbf{empty}(\tau) \tag{50}$$
$$\Theta; \Gamma_{11}' \vdash E[\mathbf{skip}] \Rightarrow \Gamma_1' \tag{51}$$

for some $\Gamma_{11}, \Gamma_{12}, \Gamma_{11}'$, and Γ_{12}'. Let Γ_r be $\Gamma_2 + \cdots + \Gamma_n$ and Δ be $(\Gamma_{11}, y : \Gamma_{12}'\ \mathbf{tid}_1) + \Gamma_{12} + \Gamma_r$. From (49) and freshness of y, it easily follows that $\Theta; \Gamma_{11}, y : \Gamma_{12} \vdash$

$[y/x]s'_1 \Rightarrow \Gamma'_{11}, y{:}\tau$. By Lemmas 46 and 55, $\Theta; \Gamma_{11}, y{:}\Gamma_{12} \vdash E[[y/x]s'_1] \Rightarrow \Gamma'_1, y : \tau$.

Let us write $F(\cdot)$ for $\mathbf{Own}(P, H, L, R, \cdot)$ and $F'(\cdot)$ for $\mathbf{Own}(P', H, L, R', \cdot)$. Then,

$$\begin{array}{ll} & F(\Gamma) \\ = & F(\Gamma_1 + \Gamma_r) \\ = & F(\Gamma_{11} + \Gamma_{12} + \Gamma_r), \end{array}$$

and

$$\begin{array}{ll} & F'(\Delta) \\ = & F'(\Gamma_{11} + \Gamma_{12} + \Gamma_r) + \{t_{n+1} \mapsto (\mathtt{P}, 1)\}\,. \end{array}$$

From the freshness condition of y and t_{n+1}, it follows that $F(\Gamma_{11} + \Gamma_{12} + \Gamma_r) = F'(\Gamma_{11} + \Gamma_{12} + \Gamma_r)$ from Lemma 58.

We show

$$\mathbf{acyclic}((t_1, \dots, t_{n+1}), ((\Gamma'_1, y : \tau), \Gamma'_2, \dots, \Gamma'_n, \Gamma'_{12}), R')$$

by contradiction. Suppose there exists a sequence $t_{j_1} \prec t_{j_2} \prec \dots \prec t_{j_m} = t_{j_1}$.

- If $\{j_1, \dots, j_m\} \subseteq \{2, \dots, n\}$, then it contradicts with $\mathbf{acyclic}((t_1, \dots, t_n), (\Gamma'_1, \dots, \Gamma'_n), R)$.
- Suppose $j_i = n + 1$ for some $i \in \{2, \dots, m\}$. (Considering $\{2, \dots, m\}$ is enough because $j_1 = j_m$.) Then, for some $z \in \mathit{dom}(R)$, $R(z) = t_{n+1}$ and, from the freshness of t_{n+1}, it should be the case of $y = z$ and thus, from freshness of y, j_{i-1} should be 1. However, this implies $\neg\mathbf{empty}((\Gamma'_1, y : \tau)(y))$ and leads to contradiction with $\mathbf{empty}(\tau)$.
- Suppose $\{j_1, \dots, j_m\}$ contains 1 and does not contain $n + 1$. Let $j_i = 1$. Then, there exists $z \in \mathit{dom}(R)$ such that $\neg\mathbf{empty}((\Gamma'_1, y : \tau)(z))$ and $R(z) = t_{j_{i+1}}$. Since $\mathbf{empty}(\tau)$, z must be different from y and so $\neg\mathbf{empty}(\Gamma'_1(z))$. Then, it is easy to show that

  $$\neg\mathbf{acyclic}((t_1, \dots, t_n), (\Gamma'_1, \dots, \Gamma'_n), R),$$

  which is a contradiction.

Case E-WAIT***:*** We assume that $s_1 = E[\mathbf{wait}(x)]$ and $s_n = \mathbf{skip}$ and $R(x) = t_n$. Note that $t_n \neq \star$ from $\star \notin \mathbf{Val}$. Then, $P' = P[t_1 \mapsto E[\mathbf{skip}]] \backslash \{t_n\}$ and $H' = H$ and $L' = L$ and $R' = R$. By Lemma 55 and inversion of T-WAIT and T-SKIP, we have, for some Γ_{11} and Γ_{12},

$$\Gamma_1 = \Gamma_{11}, x : \Gamma_{12}\ \mathbf{tid}_1 \tag{52}$$
$$\Theta; (\Gamma_{11}, x : \Gamma_{12}\ \mathbf{tid}_0) + \Gamma_{12} \vdash E[\mathbf{skip}] \Rightarrow \Gamma'_1 \tag{53}$$
$$\Gamma_n = \Gamma'_n. \tag{54}$$

Let Γ_r be $\Gamma_2 + \dots + \Gamma_{n-1}$ and Δ be $(\Gamma_{11}, x : \Gamma_{12}\ \mathbf{tid}_0) + \Gamma_{12} + \Gamma_r$. Then, the only non-trivial part is the condition on ownership consistency, which is shown below.

Let us write $F(\cdot)$ for $\mathbf{Own}(P, H, L, R, \cdot)$ and $F'(\cdot)$ for $\mathbf{Own}(P', H, L, R, \cdot)$ and S for $\{x \in \mathit{dom}(R) \mid R(x) = t_n\}$.

Then,

$$\begin{array}{ll} & F(\Gamma) \\ = & F(\Gamma_1 + \Gamma_r) + F(\Gamma_n) \\ = & F(\Gamma_{11} + \Gamma_r) + F(\Gamma'_n) + \{t_n \mapsto (\mathtt{P}, 1)\} \\ = & F((\Gamma_{11} + \Gamma_r){\downarrow_S}) + F((\Gamma_{11} + \Gamma_r){\uparrow_S}) + \\ & F(\Gamma'_n) + \{t_n \mapsto (\mathtt{P}, 1)\}\,, \end{array}$$

and

$$\begin{array}{ll} & F'(\Delta) \\ = & F'(\Gamma_{11} + \Gamma_r) + F'(\Gamma_{12}) \\ = & F'((\Gamma_{11} + \Gamma_r){\downarrow_S}) + F'((\Gamma_{11} + \Gamma_r){\uparrow_S}) + F'(\Gamma_{12}). \end{array}$$

We have $F((\Gamma_{11} + \Gamma_r){\uparrow_S}) = F'((\Gamma_{11} + \Gamma_r){\uparrow_S})$ from Lemma 58. We claim that

(a) $\mathbf{empty}((\Gamma_{11} + \Gamma_r){\downarrow_S})$, and thus $F((\Gamma_{11} + \Gamma_r){\downarrow_S}) = F'((\Gamma_{11} + \Gamma_r){\downarrow_S})$ from Lemma 58, and

(b) $\Gamma_{12} \leq \Gamma'_n$, which follows from (42).

In fact, from the definition of $\mathbf{Own}$, $\mathit{dom}(F((\Gamma_{11} + \Gamma_r){\downarrow_S})) \subseteq \{t_n\}$. Then, from $\mathbf{ConOwn}(P, H, L, R, \Gamma)$, $\mathit{dom}(F((\Gamma_{11} + \Gamma_r){\downarrow_S})) = \emptyset$ or $F((\Gamma_{11} + \Gamma_r){\downarrow_S})(t_n) = (\mathtt{P}, 0)$. Both cases are followed by $\mathbf{empty}((\Gamma_{11} + \Gamma_r){\downarrow_S})$. Then, we have

- $F(\Gamma'_n) = F(\Gamma_{12}) = F'(\Gamma_{12})$ from (b) and Lemma 58

which is followed by $\mathbf{ConOwn}(P', H', L', R', \Gamma')$.

Case E-PROC***:*** Follows from Lemma 59. □

Proof of Lemma 26 By T-CONFIG, we have

$$\mathbf{ConOwn}(P, H, L, R, \Gamma) \tag{55}$$
$$\vdash D : \Theta \tag{56}$$
$$\Gamma = \Gamma_1 + \dots + \Gamma_n \tag{57}$$
$$P = \{t_1 \mapsto s_1, \dots, t_n \mapsto s_n\} \tag{58}$$
$$\Theta; \Gamma_i \vdash s_i \Rightarrow \Gamma'_i \text{ for each } i \in \{1, \dots, n\} \tag{59}$$
$$R(x) = t_i \text{ implies } \mathbf{PTE}(\Gamma(x)) \leq \Gamma'_i \tag{60}$$
$$t_i = \star \text{ implies } \mathbf{empty}(\Gamma'_i) \tag{61}$$

for some $\Gamma_1, \dots, \Gamma_n, \Gamma'_1, \dots, \Gamma'_n$. We have three things to prove: $(P, H, L, R) \not\rightsquigarrow \mathbf{Error}$ and (P, H, L, R) is not in race, and if $\mathbf{End}(P, H, L, R)$, then (P, H, L, R) does not leak resource.

$(P, H, L, R) \not\rightsquigarrow$ **Error***:* Proof by contradiction. Suppose $(P, H, L, R) \rightsquigarrow \mathbf{Error}$. Then, it should be derived by E-PROCERR. Thus, there exists $t \in \mathit{dom}(P)$ such that $P(t) = s$ and $(s, H, L, R) \rightsquigarrow \mathbf{Error}$. Then, there exist E and s_0 such that $s = E[s_0]$. From (59) and Lemma 55, we have

$$\Theta; \Gamma_1 \vdash s_0 \Rightarrow \Gamma' \tag{62}$$
$$\Theta; \Gamma' \vdash E[\mathbf{skip}] \Rightarrow \Gamma'_1 \tag{63}$$

for some Γ'. Conduct case-analysis on $(s_0, H, L, R) \rightsquigarrow \mathbf{Error}$. We show only the case of E-DANGPTRACCERR (in particular, the case where $s_0 = *x \leftarrow y$ for some x and y) below. The other cases are similar.

In this case, $R(x) \notin dom(H)$ and $R(x) \neq \mathbf{null}$. Inversion of T-ASSIGN gives $\Gamma_1(x) = \kappa\ \mathbf{ref}_1$ for some κ. From (55), we have $\mathbf{Own}(P,H,L,R,\Gamma)(R(x)) = (\mathtt{R},1)$. Let us write Γ_r for $\Gamma_2 + \cdots + \Gamma_n$. Then,

$$\begin{array}{ll} & \mathbf{Own}(P,H,L,R,\Gamma)(R(x)) = (\mathtt{R},1) \\ \iff & \mathbf{Own}(P,H,L,R,\Gamma_1)(R(x)) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_r)(R(x)) \\ & = (\mathtt{R},1) \\ \iff & \mathbf{Own}(P,H,L,R(x),\Gamma_1(x))(R(x)) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_1\backslash\{x\})(R(x)) + \cdots = (\mathtt{R},1) \end{array}$$

For the left-hand side of the last equation to be defined, $\mathbf{Own}(P,H,L,R(x),\Gamma_1(x))(R(x))$ should be defined, thus $R(x)$ should be in $dom(H)$ from the definition of $\mathbf{Own}$, which contradicts with $R(x) \notin dom(H)$.

(P,H,L,R) ***is not in race:*** Proof by contradiction. Suppose (P,H,L,R) *is* in race. Then, without loss of generality, we can assume that there are $x_1, x_2 \in dom(R)$ such that $R(x_1) = R(x_2) \in dom(H)$ and,

- t_1 is writing to $R(x_1)$ in (P,H,L,R) and t_2 is writing to $R(x_2)$ in (P,H,L,R); or
- t_1 is writing to $R(x_1)$ in (P,H,L,R) and t_2 is reading from $R(x_2)$ in (P,H,L,R).

In the first case, $P(t_1) = E_1[*x_1 \leftarrow y_1]$ and $P(t_2) = E_2[*x_2 \leftarrow y_2]$ for some E_1, E_2, y_1 and y_2. Let us write h for $R(x_1)$ and $R(x_2)$, and Γ_r for $\Gamma_3 + \cdots + \Gamma_n$. Then, by T-ASSIGN, we have $\Gamma_1(x_1) = \kappa_1'\ \mathbf{ref}_1$ and $\Gamma_2(x_2) = \kappa_2'\ \mathbf{ref}_1$ for some κ_1' and κ_2'. Then,

$$\begin{array}{ll} & \mathbf{Own}(P,H,L,R,\Gamma) \\ = & \mathbf{Own}(P,H,L,R,\Gamma_1) + \mathbf{Own}(P,H,L,R,\Gamma_2) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_r) \\ = & \mathbf{Own}(P,H,L,h,\kappa_1'\ \mathbf{ref}_1) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_1\backslash\{x_1\}) + \\ & \mathbf{Own}(P,H,L,h,\kappa_2'\ \mathbf{ref}_1) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_2\backslash\{x_2\}) + \mathbf{Own}(P,H,L,R,\Gamma_r) \\ = & \{h \mapsto (\mathtt{R},1)\} + \mathbf{Own}(P,H,L,H(h),\kappa_1') + \\ & \mathbf{Own}(P,H,L,R,\Gamma_1\backslash\{x_1\}) \\ & + \{h \mapsto (\mathtt{R},1)\} + \mathbf{Own}(P,H,L,H(h),\kappa_2') + \\ & \mathbf{Own}(P,H,L,R,\Gamma_2\backslash\{x_2\}) \\ & + \mathbf{Own}(P,H,L,R,\Gamma_r) \\ = & \{h \mapsto (\mathtt{R},2)\} + \mathbf{Own}(P,H,L,H(h),\kappa_1') + \\ & \mathbf{Own}(P,H,L,R,\Gamma_1\backslash\{x_1\}) \\ & + \mathbf{Own}(P,H,L,H(h),\kappa_2') + \mathbf{Own}(P,H,L,R,\Gamma_2\backslash\{x_2\}) \\ & + \mathbf{Own}(P,H,L,R,\Gamma_r), \end{array}$$

which contradicts with $\mathbf{ConOwn}(P,H,L,R,\Gamma)$.

The second case is similar.

Conf ***does not leak resources:*** We first assume $\mathbf{End}(P,H,L,R)$, i.e., $s_i = \mathbf{skip}$ for every $i = 1,\ldots,n$. Then, by T-SKIP,

$$\Gamma_i = \Gamma_i' \text{ for } i \in \{1,\ldots,n\}.$$

Then, we prove two auxiliary propositions below.

PROPOSITION 61. *If $\Gamma_i(x) = \Gamma''\ \mathbf{tid}_f$ for some Γ'', and if $f > 0$, then there exist y and j such that $R(y) = t_i$ and $\Gamma_j(y) = \Gamma'''\ \mathbf{tid}_{f'}$ for some Γ''' and $f' > 0$.*

PROOF. Fix x, i and f such that $\Gamma_i(x) = \Gamma''\ \mathbf{tid}_f$ for some Γ'' and f, and assume $f > 0$. Note that there is at least one y such that $R(y) = t_i$ from the definition of $\mathbf{Own}$ and $\mathbf{ConOwn}(P,H,L,R,\Gamma)$. Suppose that, for any y and j, if $\Gamma_j(y) = \Gamma'''\ \mathbf{tid}_{f'}$ for some Γ''' and f', then $f' = 0$. Then,

$$\begin{array}{ll} & \mathbf{Own}(P,H,L,R,\Gamma)(t_i) \\ = & \mathbf{Own}(P,H,L,R,\Gamma_\star + \Gamma_1 + \cdots + \Gamma_n)(t_i) \\ & \quad \text{(where } \Gamma_\star \text{ is the pre type environment for thread } \star) \\ = & \mathbf{Own}(P,H,L,R,\Gamma_\star)(t_i) + \\ & \mathbf{Own}(P,H,L,R,\Gamma_1)(t_i) + \\ & \cdots + \\ & \mathbf{Own}(P,H,L,R,\Gamma_n)(t_i). \end{array}$$

However, from the assumption that $R(y) = t_i$ and $\Gamma_j(y) = \Gamma'''\ \mathbf{tid}_{f'}$ imply $f' = 0$, the last line of the equation above is equal to $(\mathtt{P},0)$, which contradicts with $\mathbf{ConOwn}(P,H,L,R,\Gamma)$ and $t_i \in dom(P)$. Thus, there are y and j such that $R(y) = t_i$ and $\Gamma_j(y) = \Gamma'''\ \mathbf{tid}_{f'}$ and $f' > 0$. □

Then, to prove that the configuration does not leak resources, it suffices to show that $P = \{\star \mapsto \mathbf{skip}\}$, since it implies $n = 1$ and $t_1 = \star$ and, by (61), $\mathbf{empty}(\Gamma_1)$ and so, by ownership-consistency, $H = \emptyset$ and $L = \emptyset$.

Suppose $P = \{\star \mapsto \mathbf{skip}, t_2 \mapsto \mathbf{skip}, \ldots, t_n \mapsto \mathbf{skip}\}$. (Note $\mathbf{End}(P,H,L,R)$.) We show that $n < 2$. We use the following propositions in the proof.

Suppose $n \geq 2$ and let $m > n+1$. By $\mathbf{ConOwn}(P,H,L,R,\Gamma)$, we have $\mathbf{Own}(P,H,L,R,\Gamma)(t_n) = (\mathtt{P},1)$. By Proposition 61, there are y_m and j_m such that $R(y_m) = t_n$ and $\Gamma_{j_m}(y_m) = \Gamma''\ \mathbf{tid}_f$ for some Γ'' and $f > 0$. Then, $t_{j_m} \prec_{(\star,t_2,\ldots,t_n),(\Gamma_1',\ldots,\Gamma_n'),R} t_n$. (Notice that $\Gamma_i = \Gamma_i'$ for any i.) By repeatedly using Proposition 61, we can obtain a sequence $t_{j_1} \prec \cdots \prec t_{j_m} \prec t_n$ for some $j_1,\ldots,j_m$.

Because $m > n+1$, there exist i and k such that $i < k$ and $t_{j_i} = t_{j_k}$, which violates $t \not\prec^+ t$ for any $t \in dom(P)$. Thus, $P = \{\star \mapsto \mathbf{skip}\}$. □

Proof of Theorem 21 An easy consequence of Lemmas 24, 25, and 26. □

B. Proof of Lemma 36

LEMMA 62. *1.* $[\![\langle\widetilde{y}/\widetilde{x}\rangle\tau]\!]_\theta = [\![[\widetilde{y}/\widetilde{x}]\tau]\!]_\theta$.

2. $[\![\langle\widetilde{y}/\widetilde{x}\rangle\mathbf{\Gamma}]\!]_\theta = [\![[\widetilde{y}/\widetilde{x}]\mathbf{\Gamma}]\!]_\theta$.

3. If $\Gamma = \langle\widetilde{y}/\widetilde{x}\rangle\widetilde{\tau}$ and $\Gamma' = \langle\widetilde{y}/\widetilde{x}\rangle\widetilde{\tau'}$, then

- $dom(\Gamma) = dom(\Gamma') = \{\widetilde{y}\}$
- $[\![\Gamma(y_i)]\!]_\theta = [\![[x_1,\ldots,x_{i-1}/y_1,\ldots,y_{i-1}]\tau_i]\!]_\theta$ *and*
- $[\![\Gamma'(y_i)]\!]_\theta = [\![[x_1,\ldots,x_{i-1}/y_1,\ldots,y_{i-1}]\tau_i']\!]_\theta$.

PROOF. Induction on the structure of $\tau, \mathbf{\Gamma}$. □

LEMMA 63. *1. If $\tau, C \doteq \tau_1 + \tau_2$ and $\theta \models C$, then $[\![\tau]\!]_\theta = [\![\tau_1]\!]_\theta + [\![\tau_2]\!]_\theta$.*

2. *If $C = (\tau_1 \doteq \tau_2)$ and $\theta \models C$, then $[\![\tau_1]\!]_\theta = [\![\tau_2]\!]_\theta$.*
3. *If $C = (\mathbf{\Gamma}_1 \doteq \mathbf{\Gamma}_2)$ and $\theta \models C$, then $[\![\mathbf{\Gamma}_1]\!]_\theta = [\![\mathbf{\Gamma}_2]\!]_\theta$.*

PROOF.

1. Induction on the structure of τ_1.
2. Induction on the structure of τ_1.
3. Induction on the structure of $\mathbf{\Gamma}_1$.

Proof of Lemma 36 Assume $\mathcal{C}(\Theta, \Gamma_{post}, s) = (\Gamma_{pre}, C)$ and $\theta \models C$. We prove $[\![\Theta]\!]_\theta; [\![\Gamma_{pre}]\!]_\theta \vdash s \Rightarrow [\![\Gamma_{post}]\!]_\theta$ by induction on the structure of s. We present only non-trivial cases.

Case $s = \mathbf{let}\ x : a = y\ \mathbf{in}\ s_1$ We have

$$\tau = \mathit{template}_x(a) \tag{64}$$
$$\Gamma', C_1 = \mathcal{C}(\Theta, s_1, (\Gamma_{post}, x : \tau)) \tag{65}$$
$$\tau', C_2 = (\Gamma'(x) + \Gamma'(y)) \tag{66}$$
$$C_3 = \left\{ \widehat{\mathbf{empty}}(\tau) \right\} \tag{67}$$
$$\Gamma_{pre} = (\Gamma' \setminus \{x, y\}, y : \tau'). \tag{68}$$

From Lemma 63 and the definition of $[\![\cdot]\!]_\theta$, we have

$$[\![\tau']\!]_\theta = [\![\Gamma'(x)]\!]_\theta + [\![\Gamma'(y)]\!]_\theta \tag{69}$$
$$[\![\Gamma_{pre}]\!]_\theta = [\![\Gamma']\!]_\theta \setminus \{x, y\}, y : [\![\tau']\!]_\theta \tag{70}$$
$$\mathbf{empty}[\![\tau]\!]_\theta. \tag{71}$$

From I.H., we have

$$[\![\Theta]\!]_\theta; [\![\Gamma']\!]_\theta \vdash s_1 \Rightarrow [\![\Gamma_{post}]\!]_\theta, x : [\![\tau]\!]_\theta. \tag{72}$$

Thus, (69)–(72) and T-LET completes this case.

Case $s = f(\widetilde{x})$: We have

$$(\widetilde{x} : \widetilde{\tau}) \to (\widetilde{\tau'}) = \Theta(f) \tag{73}$$
$$C_1 = (\Gamma_{post} \doteq \langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau'} + \Gamma') \tag{74}$$
$$C_2 = (\Gamma_{pre} \doteq \langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau} + \Gamma') \tag{75}$$
$$C = C_1 \cup C_2. \tag{76}$$

Then, from Lemma 63, we have

$$[\![\Gamma_{post}]\!]_\theta = [\![\langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau'}]\!]_\theta + [\![\Gamma']\!]_\theta \tag{77}$$
$$[\![\Gamma_{pre}]\!]_\theta = [\![\langle \widetilde{y}/\widetilde{x} \rangle \widetilde{\tau}]\!]_\theta + [\![\Gamma']\!]_\theta. \tag{78}$$

Thus, Lemma 62.3 and T-APP complete this case.

Case $s = (\mathbf{let}\ x = \mathbf{fork}(s_1)\ \mathbf{in}\ s_2)$: We have

$$\Gamma_1, C_1 = \mathcal{C}(\Theta, s_1, \Gamma'_{post}) \tag{79}$$
$$\Gamma_2, C_2 = \mathcal{C}(\Theta, s_2, (\Gamma_{post}, x : \gamma_x\ \mathbf{tid}_0)) \tag{80}$$
$$\Gamma_{pre}, C_3 = (\Gamma_1 + \Gamma_2 \setminus \{x\}) \tag{81}$$
$$C_4 = \left\{ \begin{array}{l} \mathbf{FV}(s_1) \mathrel{\widehat{\subseteq}} \widehat{dom}(\gamma_x), \\ \widehat{dom}(\gamma_x) \mathrel{\widehat{\subseteq}} \widehat{dom}(\Gamma_{post}), \\ \widehat{\mathbf{empty}}(\Gamma'_{post} \mathbin{\widehat{\setminus}} \widehat{dom}(\gamma_x)) \end{array} \right\} \tag{82}$$
$$C = C_1 \cup \cdots \cup C_4. \tag{83}$$

From Lemma 63, we have

$$[\![\Gamma_{pre}]\!]_\theta = [\![\Gamma_1]\!]_\theta + [\![\Gamma_2]\!]_\theta \setminus \{x\} \tag{84}$$
$$\mathbf{FV}(s_1) \subseteq dom([\![\gamma_x]\!]_\theta) \tag{85}$$
$$dom([\![\gamma_x]\!]_\theta) \subseteq dom([\![\Gamma_{post}]\!]_\theta) \tag{86}$$
$$\mathbf{empty}([\![\Gamma'_{post}]\!]_\theta \setminus dom([\![\gamma_x]\!]_\theta)) \tag{87}$$

From I.H., we have

$$[\![\Theta]\!]_\theta; [\![\Gamma_1]\!]_\theta \vdash s_1 \Rightarrow [\![\Gamma'_{post}]\!]_\theta \tag{88}$$
$$[\![\Theta]\!]_\theta; [\![\Gamma_2]\!]_\theta \vdash s_2 \Rightarrow [\![\Gamma_{post}]\!]_\theta, x : [\![\gamma_x\ \mathbf{tid}_0]\!]_\theta \tag{89}$$

From T-FORK, it suffices to show that $[\![\Theta]\!]_\theta; [\![\Gamma_1]\!]_\theta \vdash s_1 \Rightarrow [\![\gamma_x]\!]_\theta$, which follows from (86) and (87) and Lemma 34.

□