Task Types for Pervasive Atomicity

Yu David Liu

SUNY Binghamton
davidl@cs.binghamton.edu

Scott F. Smith

The Johns Hopkins University
scott@cs.jhu.edu

Abstract

Several languages now provide **atomic** block syntax to bracket code in which atomicity must be preserved. Outside of these blocks however, there are no atomicity guarantees. In this paper we define a language that takes the opposite approach: with no programmer declaration, atomicity holds pervasively in the code, and is broken only when an explicit need for sharing is declared. Furthermore, our principled way of sharing prevents a complete loss of atomicity; it only means a large region may need to be broken into smaller atomicity-enforcing regions.

At the core of our work is a novel constraint-based polymorphic type inference system, *Task Types*, to help enforce this flavor of pervasive atomicity statically, and to reduce the need for costly run-time support mechanisms such as locks or transactions. Task Types statically map objects onto *tasks* (threads). New program syntax is minimal; programmers need only declare the sharing points, as *subtasks*. We show the reasonableness of our type system by proving type soundness, isolation invariance, and atomicity enforcement properties hold at run time. Task Type programming requires more explicit focus on how objects are to be shared between threads; we illustrate the programming philosophy in several examples.

1. Introduction

In an era when multi-core programming is becoming the rule not the exception, it is now generally agreed that *atomicity* – the property that says an execution in the presence of interleavings always has the same effect as a sequential execution – is a crucial invariant. Several languages support a notion of **atomic** block, asking programmers to declare a code block as subject to atomicity enforcement; this is a large advance over lack of any atomicity, but *it leaves the rest of the program unattended*. The overall correctness of such a program then heavily depends on the judgment of the programmer, since any interleaved access on unattended shared memory is a violation of the sequential view, and is a spot where unexpected bugs may be introduced. Such designs are also known to lead to problems when the atomicity-enforcing code interleaves with the non-atomicity-enforcing code, a condition known as weak atomicity [8; 32; 1].

In this paper we take the approach that atomicity should be *pervasive* throughout the program – *every line of code is*

Figure 1. (1) Non-Shared Memory at Default (2) Subtasking (3) Capture

part of some atomic region. If atomicity is to be pervasive, there will be a much greater need for locking or software transactional memory (STM) control, to the point where it would seem that performance was severely impacted. A key insight that we build on – originally demonstrated in the Actor model [3] – is that if threads of a program *never share objects*, the threads can execute in *any interleaving manner* without violating atomicity; no additional run-time support is needed. We previously developed a dynamically typed language, Coqa, which enforces a non-shared-memory model by default [25]; in this paper we develop Task Types, which defines a primarily static, as opposed to completely dynamic, enforcement of this model.

Figure 1(a) illustrates how objects are statically localized in threads via Task Types. Here the two rectangular boxes represent two runtime threads, which we call *tasks*. Our type system statically guarantees this picture: each object is “owned by” only one task. The black *task objects* are special objects which can start a new task; the white *ordinary objects* are the vast majority of objects and are statically localized to tasks.

Models Support	Atomic Blocks/Sections	Task Types (This Work)	Message Passing De- signs (Erlang, MPI, <i>etc</i>)	Pure Actors
sharing strategy	shared memory	non-shared memory by default	non-shared memory	non-shared memory
atomicity property	weak or strong atomic- ity in atomic blocks	pervasive strong atom- icity	pervasive strong atom- icity	pervasive strong atomicity
atomicity span (code)	atomic block	entire task divided by subtask access	message handler di- vided by blocking operators	message handler
atomicity span (mem- ory)	multiple objects possi- ble	multiple objects possi- ble	(usually) single actor only	single actor only
programability (syn- chronous messaging)	Yes	Yes	Varies	No
programability (atomic- ity intention)	declare if you need it	implicit (divided at sub- tasking time)	implicit (divided at syn- chronization time)	implicit

Figure 2. The Spectrum of Atomicity-Oriented Programming Models

A pure non-shared memory model significantly impacts programmability: how can two threads communicate? One obvious solution is they only communicate when a new thread is launched – the pure Actor model adopts this strategy – but this would mean giving up synchronous messaging and dividing a piece of code that logically forms one thread into numerous sub-threads for each message reply continuation. We take an intermediate approach between standard multithreaded programming and Actors, where object sharing is supported, but by principled language abstractions only. Additionally, our design is flexible enough to naturally support both pure Actor-style as well as pure standard-style multithreaded programming in the cases where it is desired.

A key abstraction we make is the *subtasking* construct of Coqa. Tasks are allowed to interact, but only via special portal objects called *subtask objects*, as illustrated in Fig. 1(b). Each subtask object conceptually handles requests from tasks one at a time, spawning a *subtask*. Subtasks are in fact not threads, they are run on the caller thread which has blocked waiting for the reply – subtask messaging is synchronous. Like a task object, a subtask object can own a set of ordinary objects.

Subtask messaging gives tasks the sharing they critically need, but in a limited manner. The most important benefit of limited sharing is that some atomicity properties are preserved: the sending task has one region of atomicity from the start to the subtask invocation, the subtask itself is an atomic region, and the task execution following the subtask is a third region of atomicity. In general, any multithreaded application can be divided into a series of atomic regions of execution, with every line of code falling into *some* atomic region – atomicity is pervasive. A particular advantage of this methodology is it becomes more feasible to exhaustively enumerate all interleaving possibilities of concurrent applications, a boon for program analysis and testing, and ultimately for the deployment of more reliable software.

In the dynamic model of Coqa, ordinary objects are assigned to particular tasks at runtime, and when the task finishes the objects are freed for use by another task. Task Types statically assign objects to tasks. Rather than creating a complex static system to approximate the freeing of an object, our language adds a notion of *capture* in analogy to a type cast, to bridge the case where the runtime can be “smarter” than the compile time. The expression **capture** *o* allows a task to use an object *o* originally used by a now-completed task. Potentially captured objects must have their owner task tracked at runtime, and at capture time there must be a check to verify that the object is not currently used by any other task. This mechanism is illustrated in Fig. 1.

The design of subtasking and capture reflects a common philosophy: non-shared-memory is the default mode, and any programming need that does not observe this default must be explicitly specified. The two are complementary sides of the same coin: subtasking breaks atomicity but they can be used anytime, whereas capturing preserves atomicity but can only be effective when the object being captured is not used by another task any more.

Before we describe our system in more detail, we first summarize the state of the art for atomicity language design and static enforcement.

Other Approaches to Atomicity Flanagan and Qadeer [19] first explored type system support for atomicity. This type system assigns expressions with types indicating commutativity properties, such as *mover* to indicate the expression can be exchanged with an execution step adjacent to it without violating atomicity. Philosophically, such an approach is a bottom-up strategy whereas Task Types are top-down: the work by Flanagan and Qadeer assumes a shared-memory model, and supports reasoning about the composability of atomic blocks, whereas Task Types enforce a non-shared-memory model in which atomicity violations by default cannot happen in the first place. Since Flanagan and Qadeer’s pioneering work, many atomicity-oriented systems have been

designed. Researchers have realized the efficiency of such systems heavily rely on how much dynamic auditing on sharing is needed, and the correctness of such systems crucially rely on how memory is partitioned [32; 1]. We feel the time is now for us to have a foundational reflection on type system support for atomicity again – this time focusing on memory.

The Actor model and related message-passing languages [3; 5; 33] achieve atomicity by imposing much stronger limitations on object sharing: threads communicate only at thread launch. A primary appeal of this model is it trivially supports pervasive strong atomicity. See Fig. 2. The case in concern is the *span* of atomicity. In the pure Actor model, each message handler is atomic. To support Java-style synchronous messaging however, two explicit message handlers need to be involved, one on the receiver for processing the request, and one on the sender for processing the return value. The atomicity of a messaging round-trip is thus broken into halves. There is also a programmability concern with such a design – the flow of control has been broken into more pieces and is harder to piece together the program meaning. This latter aspect has been improved in implemented Actor-based languages, which include implicit CPS transformations for example, but that alone does not improve the span of an atomic region. Most Actor-based languages provide blocking constructs such as the **receive** in Erlang which further breaks the atomicity of a message handler.

Recently, several static inference techniques have been designed to automatically insert locks to enforce atomicity for atomic blocks [27; 23; 17; 9]. These techniques however fundamentally assume that atomic blocks are a “local” construct, and have made assumptions that are unrealistic for supporting larger atomic regions: some for example requires all invocations in an atomic block to be inlined [27], some would required a static bound on lock resources [17; 9; 23], and some would assume all objects accessed in the atomic block are not accessed elsewhere [23]. Such assumptions are reasonable in these experimental efforts – many of them C-based – but do not generalize to an object-oriented setting with pervasive atomicity: both the number of objects (and hence locks) and the number of threads are unknown statically; programs are fundamentally recursive; and atomicity enforcement is needed not for a block of code, but for all code.

Ownership types [12; 11; 7] and region types [34; 20; 10] are well-known for partitioning memory. Task Types were initially inspired by ownership type systems, but they turn out to be a significant departure from standard ownership type systems however. One key reason is our support for sharing. In addition, because task creations are dynamic, there may be an unbounded number of runtime tasks of a particular sort due to recursion; they must be modeled by only finitely many owners in the static type system, and so one “static owner” may represent multiple “dynamic owners”. Standard ownership type systems would not work

in this context because, to guarantee soundness, the single static owner cannot even share between “itself” – it may represent multiple runtime tasks that should not share with each other!

2. Informal Discussion

P	$::= \overrightarrow{Cls}$	<i>program</i>
Cls	$::= \mu \text{ class } a \text{ extends } a \{ F \ M \}$	<i>class</i>
μ	$::= \text{task} \mid \text{subtask} \mid \epsilon$	<i>modifier</i>
F	$::= \overrightarrow{Fe}$	<i>fields</i>
Fe	$::= a \ f = e$	<i>field entry</i>
M	$::= \overrightarrow{Me}$	<i>methods</i>
Me	$::= a \ m(a \ x) \{ e \}$	<i>method entry</i>
e	$::= x \mid \text{this} \mid \text{new}_L a$ $\mid f \mid f : e \mid \text{capture } e \mid (a)e$ $\mid e.m(e) \mid e \rightarrow m(e) \mid e \Rightarrow m(e)$	<i>expressions</i>

Figure 3. Abstract Syntax

In this section, we highlight a number of features of our language, focusing on its type system in particular. The abstract syntax of our language is given in Fig. 3, and is largely standard for a Java-like language: it includes classes and objects (Cls), methods (M), fields (F), inheritance, and private field access (read and write), and casting. Notation $\overrightarrow{X}$ represents a sequence of X ’s. We use metavariable a to represent class names, m for method names, f for field names, and x for variable names. The program point label L associated with the **new** expression is only used in the formal development and can be ignored by programmers. To avoid **null** pointers, our language supports field initializers and requires no forward referencing happens at initialization time, just like Java. The syntax omits several constructs that are useful (so much so that we will use some of them in our examples) but are easily encodable: integers, constructors, statement composition $e; e'$, local variable declarations and assignments (encodable by method invocations), and public field access (encodable by a pair of methods with getters and setters). The arity of method arguments is simplified to be 1.

New to our language are the *class modifiers* μ , signifying whether the class will create task objects ($\mu = \text{task}$), subtask objects ($\mu = \text{subtask}$), or ordinary objects ($\mu = \epsilon$). The messaging and capture syntax will be discussed below. Fig. 4 shows a code snippet.

2.1 Object Messaging

Expression $o \rightarrow m(v)$ creates a new task where o is a task object. Line [1] in Fig. 4 creates a task by sending a `start1` message to the task object instantiated inside the `create` method. Task creation messaging is asynchronous and has no interesting return value. Following Actors, each task object keeps a queue of all received messages and processes

```

class Main {
  void main() {
    TaskFactory f = new TaskFactory();
    MyShared s1 = new MyShared();
    MyShared s2 = new MyShared();
    MyShared s3 = new MyShared();
    Data d = new Data();
    (f.create()) ->start1(s1, s2, d); //[1]
    (f.create()) ->start2(s3);         //[2]
    (f.create()) ->start2(s3);         //[3]
    (f.create()) ->start1(s1, s2, d); //[4] (error)
  }
}
class TaskFactory {
  MyTask create() { return new MyTask(); }
}
task class MyTask {
  void start1(MyShared s1, MyShared s2, Data d) {
    s1 =>process(d); s2 =>process(d); //[*]
  }
  void start2(MyShared s) {
    s =>process(new Data());
  }
}
subtask class MyShared {
  void process(Data d) { d.access(); } }
}
class Data { void access() {...} }

```

Figure 4. A Code Snippet with (a) a snapshot right before `main` completion, where lines [1], [2], [3] are included (but not [4]), creating tasks `Ta1`, `Ta2`, `Ta3` respectively; (b) a snapshot right before `main` completion, when lines [1],[4] are included (but not [2],[3]), creating tasks `Tb1` and `Tb4`. Such a program would produce a type error so this run would never happen in reality.

them serially. This sequential processing constraint is to preserve atomicity; however, it is still easy to have the same code running multiple tasks in parallel by multiply instantiating the same task class as is done in lines [2] and [3].

Subtasks can be created if tasks need to share state. Expression $o \Rightarrow m(v)$ creates a subtask where o is a subtask object. Line [*] creates two subtasks by sending the process messages to `s1` and `s2` respectively. A subtask is similar to a task in that a subtask object also keeps a queue for received messages and executes them one by one. Unlike a task however, a subtask can return values, and $o \Rightarrow m(v)$ is a synchronous operation: the message sender blocks until the subtask is completed. For this reason, a subtask is not in fact a distinct thread – it can piggyback on its parent task. The subtask *object* however can be shared by multiple tasks/subtasks, the crucial rendezvous mechanism. In Fig. 4(a), the two tasks `Ta2` and `Ta3` share subtask object `s3`. Since subtasks are technically not threads, we do not illustrate subtasks as nested boundary boxes.

Expression $o.m(v)$ is the familiar synchronous messaging in Java-like languages where o is an ordinary, unshared, object (*i.e.* neither a task object nor a subtask object). Each such object must be owned by the same task that owns the sender object.

2.2 Dynamic and Static Enforcement of Atomicity

We previously showed how atomicity can be enforced dynamically [25]; the goal of our Task Type system is to statically approximate dynamic enforcement so that most of the dynamic enforcement mechanisms are unnecessary. We first give a simplified account of how atomicity can be achieved dynamically for the language defined in Fig. 3.

In our Coqa language [25] we implemented atomicity with locks, and proved that a running task is atomic if it locks every ordinary object it accesses, and if such an object has already been locked by another task, the task blocks until the object is unlocked – which happens when the lock-holding task completes. Alternatively, the Coqa locks could be implemented with STMs: atomicity is preserved if a transaction monitor records every object a task has accessed; upon completion, the task rolls back if any of the task’s objects were accessed by other tasks in the interim, and otherwise it commits.

Subtasking breaks whole-task atomicity, but as we explained in Sec. 1, it breaks it in the sense of partitioning the task into sequential regions that are still atomic, not in the sense of breaking atomicity completely; this fact is proved in [25]. Subtasking is partly related to the open nesting of STM transactions [13; 8; 29]. For a lock-based implementation, a subtask will lock all objects it has accessed, but will unlock all of them upon completion to claim partial victory. From a concurrency perspective, subtasking increases chances of parallelism by reducing the duration of contention. Another common issue in this context is whether a subtask can access objects locked by their ancestors. For STM-systems,

the answer is often negative [8] due to high cost, whereas the feature is supported relatively easily in Coqa [25].

2.3 Task Isolation

Task Types mimic the aforementioned dynamic enforcement behaviors at compile-time, so that much of the work for locking or transaction monitoring at run-time can go away. The type system enforces non-shared-memory model (called *task isolation* here), but not a strict one. Introducing sharing points, subtasking in particular, produces non-trivial issues the must be solved. This means that the static access graph – a static global view of how objects are accessed where object access is defined as being sent with a message – is not a strict hierarchy: the nodes representing subtask objects are sharing points. For instance, in Fig. 4(a), T2 and T3 share subtask object *s3*. Object *d1* is indeed being accessed by two tasks – and a DAG is formed – but this case is legal because sharing is encapsulated in the subtask. As another example, the access to object *d* in the same figure is also legal, but for a different reason: it is being accessed by two different subtasks, but such a sharing is not an atomicity violation because *s1* and *s2* originate from the same parent task. Recall that subtasking follows synchronous semantics, so the two accesses will not happen at the same time. When *s2* accesses *d*, *s1* must have already completed and *s2* should be able to access *d* without any problem.

Things are quite different if a snapshot like Fig. 4(b) could exist. Here, even though *d* is also accessed by *s1* and *s2* – just like in Fig. 4(a) – the two tasks Tb1 and Tb4 however both have access to *s1* and *s2*. It is possible that *s1* could access *d* for a request from Tb1 whereas *at the same time* *s2* would access *d* as well for a request from Tb4, which would violate our atomicity model.

In Sec. 3, we will define our task isolation model in a way that a run-time snapshot like Fig. 4(b) can never arise, and the program would typecheck only if line [4] were removed. For run-time snapshots that Task Types do allow, we show that the dynamic locking semantics outlined in Sec. 2.2 will be sound, and most importantly much of the need for run-time locking will be obviated.

2.4 Context-Sensitivity: The Demand for Instance Precision

The example above also shows the importance of correctly approximating the number of tasks being created. After all, the only difference between Fig. 4(a) and Fig. 4(b) from the perspective of object *d* is that the number of root tasks has increased from 1 to 2, and yet it makes the difference of correctness. The algorithm thus has to correctly identify that lines [1] and [4] will indeed create two different tasks. Here a factory method is used: each invocation of `create` will return a different `MyTask` instance, even though there is only one program point with the `new MyTask` expression. The algorithm must be able to treat different call sites of the same method differently: context-sensitivity is needed. Task

Type context sensitivity is related to polymorphic type-based constraint systems such as [36], and is also closely related to context-sensitive program analyses such as [16; 39].

2.5 Tunneling: Battling Against the Effect of Recursion

As is well-known, some imprecision must be introduced to analyze recursive programs yet keep the analysis terminating. For the specific problem we are tackling here, however, things are more serious in that a standard approach will *not even be sound*. Suppose we had a top-level loop (implemented via recursion in our primitive language) in the body of the `main` function.

```
...
x = 5;
while (x>0) {
    (f.create()) -> start1(s1, s2, d);
    x = x - 1;
}
```

Based on our analysis of Fig. 4(b), we know this program will cause problems. However, we obviously cannot reject all programs with recursion or looping. If we changed the asynchronous expression in the loop above to `(f.create()) ->start2(s3)`, the loop above should be legal since no ordinary objects are shared by the different tasks launched in the loop.

Our solution centers around a novel *tunnel detection analysis*. The type system first ignores recursion unsoundness and performs the non-shared-memory type inference as described in Sec. 2.3, and then it additionally explicitly outlaws the bad “tunnels”, the object-sharing between two tasks that cannot be differentiated by type variables. The intuition is that, if a task *x* and another task *y* both end up with access to some *o*, the “bad” case that cannot be detected by the non-shared-memory type checking algorithm are those cases where *y* is represented by the same type variable as *x*, so that the type system was tricked to believe *x* and *y* are the same task. It turns out only a few checks are needed to outlaw all such tunnels. This topic will be discussed in more detail in Sec. 3.4.

2.6 Capture

From a programmer’s perspective, capture allows more programs to typecheck without violating atomicity. As we explained earlier, a program with both line [1] and line [4] in Fig. 4 would not typecheck because the dangerous case of Fig. 4(b) might arise. The same program however would typecheck if line [*] was changed to

```
d0 = capture d;
s1 => process(d0); s2 => process(d0); //[*]
```

Semantically, capture is a typecast which allows an object to change its owner. As described in Sec. 1, the change only happens at runtime for an object already unlocked by

its original hosting task. If the object is still locked by a different task, capture is blocked.

2.7 Properties

We will show a type soundness property for Task Types. We also show Task Types preserve a non-shared-memory model modulo explicit sharing points such as subtask objects. As a result, we can prove a program with all locks removed – except those used for captured objects or for mutual exclusion on task and subtask objects – will behave the same as one subject to the purely dynamic semantics of Sec. 2.2. In other words, Task Types will preserve atomicity properties pervasively, and with a relatively low run-time cost.

3. The Formal Type System

$\mathcal{P}$	$::= \overrightarrow{a \mapsto \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle}$	<i>parameterized signatures</i>
$\mathcal{M}$	$::= \overrightarrow{m \mapsto \forall \alpha. (\tau \rightarrow \tau')}$	<i>method signatures</i>
$\mathcal{F}$	$::= \overrightarrow{f \mapsto \forall \alpha. \tau}$	<i>field signatures</i>
$\mathcal{U}$	$::= \overline{a}$	<i>superclass set</i>
τ	$::= a @ \alpha$	<i>types</i>
α		<i>type variable (incl. self)</i>
Σ	$::= \overline{cons}$	<i>constraint set</i>
$cons$	$::=$	<i>constraint</i>
	$\beta \leq \beta'$	<i>flow constraint</i>
	$\alpha \xrightarrow{hosts} \alpha'$	<i>hosting constraint</i>
	$lazy(\alpha, m, \sigma)$	<i>lazy closure constraint</i>
	$\langle \alpha_1; m_1 \rangle \xrightarrow{\alpha}_{ef} \langle \alpha_2; m_2 \rangle$	<i>data flow constraint</i>
β	$::=$	<i>flow constraint element</i>
	α	<i>object</i>
	$\langle \alpha; f \rangle$	<i>field access marker</i>
	$\langle a; L; \sigma \rangle$	<i>instantiation marker</i>
	$cap(\alpha, \alpha')$	<i>capture marker</i>
σ	$::= \overrightarrow{\alpha \mapsto \alpha'}$	<i>substitution</i>
G	$::= \langle a; m \rangle \mapsto \Sigma$	<i>labeled constraints</i>
Γ	$::= \overrightarrow{\gamma}$	<i>typing environment</i>
γ	$::= x \mapsto \tau$	
	$myC \mapsto a$	
	$myM \mapsto m$	
ef	$\in \mathbf{Bool}$	<i>effect label</i>

Figure 5. Auxiliary Definitions for the Type System

We now describe Task Types rigorously. First we review our basic notation. $\overline{x_n}$ denotes a set $\{x_1, \dots, x_n\}$. $\overline{x_n \mapsto y_n}$ is used to denote a mapping sequence (also called a mapping) $[x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$. Given $M = \overline{x_n \mapsto y_n}$, $\text{dom}(M)$ denotes the domain of M , and it is defined as $\{x_1, \dots, x_n\}$. We also write $M(x_1) = y_1, \dots, M(x_n) = y_n$. When no confusion arises, we also drop the subscript n for sets and mapping sequences and simply use $\overline{x}$ and $\overline{x \mapsto y}$.

Binary operator $\in$ is used both for set containment and sequence containment, and $\emptyset$ is used to represent both empty set and empty sequence. We write $M[x \mapsto y]$ as a mapping update: M and $M[x \mapsto y]$ are identical except that $M[x \mapsto y]$ maps x to y . Updateable mapping concatenation $\triangleright$ is defined as $M_1 \triangleright M \stackrel{\text{def}}{=} M_1[x_1 \mapsto y_1] \dots [x_n \mapsto y_n]$. We also write $M_1 \triangleright M$ as $M_1 \uplus M$, except the latter function requires the pre-condition of $\text{dom}(M_1) \cap \text{dom}(M) = \emptyset$.

We define function $\Xi(P) = \mathcal{P}$ to extract formalization-friendly signature information $\mathcal{P}$ from the more programmer-friendly P . We defer its verbose but obvious definition to Appx. A), and describe it here informally. The function takes a class and computes the signature $a \mapsto \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle$, including its name a , modifier μ , field signatures $\mathcal{F}$, method signatures $\mathcal{M}$, and a superclass set $\mathcal{U}$ containing the names of all superclasses (including a itself). For a class with superclasses, we “inline” all superclasses so that the signature of that class will include the signature information of all fields and methods on the hierarchy. To make parametric polymorphism more explicit and readable, the signatures computed by Ξ quantifies the object types appearing in the signatures. Thus, given a method $a \ m(a' \ x)\{e\}$, its signature in $\mathcal{M}$ is $m \mapsto \forall \alpha, \alpha'. (a @ \alpha \rightarrow a' @ \alpha')$ where α and α' are type variables. Similarly, given a field $a \ f = e$, its signature in $\mathcal{F}$ is $f \mapsto \forall \alpha''. a @ \alpha''$. For simplicity of the formal development, we require the $\forall$ -bound type variables for a particular class signature to be distinct for each other.

We define a constraint-based type system, with constraint set Σ containing constraints $cons$ inside. The meanings of specific constraints will be clarified later, but in general a constraint connected by $\leq$ is intuitively a flow constraint, and we use meta variable β to represent these flow elements. Typing environment Γ contains mappings from variables to types, including two special mappings denoting where the currently typed expression is lexically enclosed: $myC \mapsto a$ says the enclosing class is a and $myM \mapsto m$ says the enclosing method is m .

Function $FTV(\bullet)$ is defined as the set of free type variables in $\bullet$ where $\bullet$ is either Σ , $cons$, β , or τ . It includes all type variables (α) in these constructs, except those in $\text{dom}(\sigma)$ should $\bullet$ include a substitution σ as part of its construct, such as $lazy(\alpha, m, \sigma)$ and $\langle a; L; \sigma \rangle$. Substitution $\bullet[\sigma]$ is defined on the same domain as FTV , and it will substitute all free occurrences of type variables.

3.1 The Type Rules

Type-checking starts with the (T-Global) rule. Judgment $\vdash_{\text{global}} P \setminus G$ holds when program P typechecks. Its overall structure includes a per-class typechecking phase (T-Class), where type constraints Σ are collected, and recorded in a structure G in a per-class per-method manner, as it is indexed by $\langle a; m \rangle$. Constraints associated with field initializers do not belong to any method, so they are indexed by a built-in special method name m_* ; see (T-Fields). When a method/field

(T-Global)

$$\begin{array}{c}
\Xi(P) = \mathcal{P} \\
P = [Cls_1, Cls_2, \dots, Cls_n] \quad \text{dom}(\mathcal{P}) = \{a_1, \dots, a_n\} \\
i = 1, \dots, n \quad \text{name}(Cls_i) = a_i \quad \mathcal{P} \vdash_{cls} Cls_i : \mathcal{P}(a_i) \setminus G_i \\
G = G_1 \uplus G_2 \dots \uplus G_n \quad \text{isolatedTasks}(G, \mathcal{P}) \\
\text{tunnelFree}(G, \mathcal{P}) \quad \mathcal{P}(a_{\text{main}}) = \langle \text{task}; \emptyset; \mathcal{M}_{\text{main}}; \mathcal{U}_{\text{main}} \rangle \\
\text{dom}(\mathcal{M}_{\text{main}}) = \{m_{\text{main}}\} \\
\hline
\vdash_{\text{global}} P \setminus G
\end{array}$$

(T-Class)

$$\frac{[\text{myC} \mapsto a], \mathcal{P} \vdash_{m1} M : \mathcal{M} \setminus G_m \quad [\text{myC} \mapsto a], \mathcal{P} \vdash_{f1} F : \mathcal{F} \setminus G_f}{\mathcal{P} \vdash_{cls} \mu \text{ class } a \{ F \ M \} : \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \setminus (G_f \uplus G_m)}$$

(T-Methods)

$$\begin{array}{c}
M = [Me_1, Me_2, \dots, Me_n] \\
\text{dom}(\mathcal{M}) = \{m_1, m_n, \dots, m_n\} \quad i = 1, \dots, n \\
\text{name}(Me_i) = m_i \quad \Gamma, \mathcal{P} \vdash_{m1} Me_i : \mathcal{M}(m_i) \setminus G_i \\
\hline
\Gamma, \mathcal{P} \vdash_{m1} M : \mathcal{M} \setminus G_1 \uplus \dots \uplus G_n
\end{array}$$

(T-Method)

$$\frac{\Gamma(\text{myC}) = a_0 \quad \Gamma \triangleright [\text{myM} \mapsto m, x \mapsto \tau], \mathcal{P} \vdash e : \tau' \setminus \Sigma}{\Gamma, \mathcal{P} \vdash_m (a' \ m(a \ x)\{e\}) : \forall \alpha. (\tau \mapsto \tau') \setminus (\langle a_0; m \rangle \mapsto \Sigma)}$$

(T-Fields)

$$\begin{array}{c}
\Gamma(\text{myC}) = a_0 \\
F = [Fe_1, Fe_2, \dots, Fe_n] \quad \text{dom}(\mathcal{F}) = \{f_1, f_2, \dots, f_n\} \\
i = 1, \dots, n \quad \text{name}(Fe_i) = f_i \quad \Gamma, \mathcal{P} \vdash_{f1} Fe_i : \mathcal{F}(f_i) \setminus \Sigma_i \\
\hline
\Gamma, \mathcal{P} \vdash_{f1} F : \mathcal{F} \setminus (\langle a_0; m_* \rangle \mapsto \Sigma_1 \cup \dots \cup \Sigma_n)
\end{array}$$

(T-Field)

$$\frac{\Gamma, \mathcal{P} \vdash f := e : \tau \setminus \Sigma}{\Gamma, \mathcal{P} \vdash_f (a \ f = e) : \forall \alpha. \tau \setminus \Sigma}$$

$$\begin{aligned}
\text{name}(\mu \text{ class } a \text{ extends } a' \{ F \ M \}) &\stackrel{\text{def}}{=} a \\
\text{name}(a \ m(a' \ x)\{e\}) &\stackrel{\text{def}}{=} m \\
\text{name}(a \ f = e) &\stackrel{\text{def}}{=} f
\end{aligned}$$

Figure 6. Top-Level Typing Rule

is typed, the signature being used is the one we have computed in Ξ , as illustrated in (T-Method) and (T-Field). A program contains a bootstrapping **task** class named **Main** with a special method **main**. Since all types in our core system are object types, sugar **unit** $\stackrel{\text{def}}{=} a_{\text{object}} @ \alpha_{\text{object}}$ is used to declare methods with no arguments or return values.

Type constraint closure is defined in Fig. 8, as a system of $\vdash_{\text{ocl}}$ judgments. They are used by the definitions of $\text{isolatedTasks}(G, \mathcal{P})$ and $\text{tunnelFree}(G, \mathcal{P})$. The first enforces task isolation as we described in Sec. 2.3 and Sec. 2.4, while the second guarantees the model is correct with the presence of recursion, as we described in Sec. 2.5. We use this grayish color to represent formalism related to the definition of $\text{tunnelFree}(G, \mathcal{P})$ since the reader probably wants

(T-Var)

$$\Gamma, \mathcal{P} \vdash x : \Gamma(x) \setminus \emptyset$$

(T-New)

$$\frac{\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \{\alpha_1, \dots, \alpha_n\} = \{\alpha \mid \mathcal{F}(f) = \forall \alpha. a @ \alpha\} \\
\alpha', \alpha'_1, \dots, \alpha'_n \text{ fresh} \quad \sigma = [\text{self} \mapsto \alpha', \alpha_1 \mapsto \alpha'_1, \dots, \alpha_n \mapsto \alpha'_n]}{\Gamma, \mathcal{P} \vdash \text{new}_L a : (a @ \alpha') \setminus \{\langle a; L; \sigma \rangle \leq \alpha'\} \cup \text{lazy}(\alpha, m_*, \emptyset)}$$

(T-SyncM)

$$\frac{\Gamma, \mathcal{P} \vdash_{\text{inv}} \langle e; m; e' \rangle : \tau \setminus \Sigma \quad \Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma_0 \quad \mathcal{P}(a) = \langle \epsilon; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle}{\Gamma, \mathcal{P} \vdash e.m(e') : \tau \setminus \Sigma \cup \Sigma_0 \cup \{\text{self} \xrightarrow{\text{hosts}} \alpha\}}$$

(T-ASyncM)

$$\frac{\Gamma, \mathcal{P} \vdash_{\text{inv}} \langle e; m; e' \rangle : \text{unit} \setminus \Sigma \quad \Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma_0 \quad \mathcal{P}(a) = \langle \text{task}; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle}{\Gamma, \mathcal{P} \vdash e \rightarrow m(e') : \text{unit} \setminus \Sigma \cup \Sigma_0}$$

(T-Subtask)

$$\frac{\Gamma, \mathcal{P} \vdash_{\text{inv}} \langle e; m; e' \rangle : \tau \setminus \Sigma \quad \Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma_0 \quad \mathcal{P}(a) = \langle \text{subtask}; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle}{\Gamma, \mathcal{P} \vdash e \Rightarrow m(e') : \tau \setminus \Sigma \cup \Sigma_0 \cup \{\text{self} \xrightarrow{\text{hosts}} \alpha\}}$$

(T-Self)

$$\frac{\Gamma(\text{myC}) = a}{\Gamma, \mathcal{P} \vdash \text{this} : a @ \text{self} \setminus \emptyset}$$

(T-Read)

$$\frac{\mathcal{P}(\Gamma(\text{myC})) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \mathcal{F}(f) = \forall \alpha. a @ \alpha}{\Gamma, \mathcal{P} \vdash f : a @ \alpha \setminus \{\langle \text{self}; f \rangle \leq \alpha\}}$$

(T-Write)

$$\frac{\mathcal{P}(\Gamma(\text{myC})) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \Gamma, \mathcal{P} \vdash e : a @ \alpha' \setminus \Sigma \quad \mathcal{F}(f) = \forall \alpha. a @ \alpha}{\Gamma, \mathcal{P} \vdash f := e : a @ \alpha' \setminus \Sigma \cup \{\alpha' \leq \alpha\}}$$

(T-Capture)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma \quad \alpha' \text{ fresh} \quad \Sigma' = \{\text{cap}(\alpha, \text{self}) \leq \alpha', \text{self} \xrightarrow{\text{hosts}} \alpha'\} \quad \mathcal{P}(a) = \langle \epsilon; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle}{\Gamma, \mathcal{P} \vdash \text{capture } e : (a @ \alpha') \setminus \Sigma \cup \Sigma'}$$

(T-Cast)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a' @ \alpha') \setminus \Sigma \quad \mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \mathcal{P}(a') = \langle \mu'; \mathcal{F}'; \mathcal{M}'; \mathcal{U}' \rangle \quad a \in \mathcal{U}' \text{ or } a' \in \mathcal{U}}{\Gamma, \mathcal{P} \vdash (a) e : (a @ \alpha') \setminus \Sigma}$$

(T-Sub)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma \quad \mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad a' \in \mathcal{U}}{\Gamma, \mathcal{P} \vdash e : (a' @ \alpha) \setminus \Sigma}$$

(T-Call)

$$\begin{array}{c}
\Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma \\
\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \mathcal{M}(m) = \forall \alpha_1. \alpha_2. (a_1 @ \alpha_1 \mapsto a_2 @ \alpha_2) \\
\alpha'_1, \alpha'_2 \text{ fresh} \quad \Gamma, \mathcal{P} \vdash e' : a_1 @ \alpha_01 \setminus \Sigma' \\
\Sigma_h = \{\text{lazy}(\alpha, m, [\alpha_1 \mapsto \alpha'_1, \alpha_2 \mapsto \alpha'_2])\} \quad \Gamma(\text{myM}) = m_0 \\
\Sigma_d = \{\langle \text{self}; m_0 \rangle \xrightarrow{\alpha'_1}_{\text{false}} \langle \alpha; m \rangle, \langle \alpha; m \rangle \xrightarrow{\alpha'_2}_{\text{false}} \langle \text{self}; m_0 \rangle\} \\
\hline
\Gamma, \mathcal{P} \vdash_{\text{inv}} \langle e; m; e' \rangle : a_2 @ \alpha_02 \setminus \Sigma' \cup \Sigma_h \cup \Sigma_d \cup \{\alpha_01 \leq \alpha'_1, \alpha'_2 \leq \alpha_02\}
\end{array}$$

Figure 7. Type Rules

$$\begin{array}{c}
\text{(OCL-Main)} \\
\frac{}{G \vdash_{\text{ocl}} G(\langle a_{\text{main}}; m_{\text{main}} \rangle)}
\\
\text{(OCL-Subset)} \\
\frac{G \vdash_{\text{ocl}} \Sigma \quad \Sigma' \subseteq \Sigma}{G \vdash_{\text{ocl}} \Sigma'}
\\
\text{(OCL-Union)} \\
\frac{G \vdash_{\text{ocl}} \Sigma_1 \quad G \vdash_{\text{ocl}} \Sigma_2}{G \vdash_{\text{ocl}} \Sigma_1 \cup \Sigma_2}
\\
\text{(OCL-FlowHost)} \\
\frac{G \vdash_{\text{ocl}} \alpha'_1 \leq \alpha_1 \quad G \vdash_{\text{ocl}} \alpha'_2 \leq \alpha_2 \quad G \vdash_{\text{ocl}} \alpha_1 \xrightarrow{\text{hosts}} \alpha_2}{G \vdash_{\text{ocl}} \alpha'_1 \xrightarrow{\text{hosts}} \alpha'_2}
\\
\text{(OCL-CapClass)} \\
\frac{G \vdash_{\text{ocl}} \langle a; L; \sigma \rangle \leq \alpha \quad G \vdash_{\text{ocl}} \mathbf{cap}(\alpha, \alpha'') \leq \alpha'}{G \vdash_{\text{ocl}} \langle a; L; \sigma \rangle \leq \alpha'}
\\
\text{(OCL-Lazy-Intro)} \quad \text{(OCL-Lazy-Cancel)} \\
\frac{G \vdash_{\text{ocl}} \mathbf{lazy}(\alpha, m, \sigma)}{G, \emptyset \vdash_{\text{oclaux}} \mathbf{lazy}(\alpha, m, \sigma)} \quad \frac{G, \Pi \vdash_{\text{oclaux}} \Sigma}{G \vdash_{\text{ocl}} \Sigma}
\\
\text{(OCL-Lazy-NonRecursive)} \\
\frac{G, \Pi \vdash_{\text{oclaux}} \mathbf{lazy}(\alpha, m, \sigma_m) \quad G \vdash_{\text{ocl}} \langle a; L; \sigma_f \rangle \leq \alpha \quad \text{forall } \sigma'_m \langle a; \sigma'_m; m \rangle \notin \Pi \quad \Sigma = G(\langle a; m \rangle) \quad \sigma = \text{fresh}(FTV(\Sigma) - \text{dom}(\sigma_f) - \text{dom}(\sigma_m))}{G, \Pi \cup \{ \langle a; \sigma_m; m \rangle \} \vdash_{\text{oclaux}} \Sigma[\sigma][\sigma_f][\sigma_m]}
\\
\text{(OCL-Lazy-Recursive)} \\
\frac{G, \Pi \vdash_{\text{oclaux}} \mathbf{lazy}(\alpha, m, \sigma_m) \quad G \vdash_{\text{ocl}} \langle a; L; \sigma_f \rangle \leq \alpha \quad \langle a; \sigma'_m; m \rangle \in \Pi \quad \sigma_m = [\alpha_1 \mapsto \alpha'_1, \dots, \alpha_u \mapsto \alpha'_u] \quad \sigma'_m = [\alpha_1 \mapsto \alpha''_1, \dots, \alpha_u \mapsto \alpha''_u]}{G, \Pi \vdash_{\text{oclaux}} \bigcup_{i \in \{1, \dots, u\}} \{ \alpha'_i \leq \alpha''_i \} \cup \{ \alpha''_i \leq \alpha'_i \}}
\\
\text{(OCL-MutSource)} \\
\frac{G \vdash_{\text{ocl}} \langle \alpha_1; m_1 \rangle \xrightarrow{\alpha'}_{\text{ef}} \langle \alpha_2; m_2 \rangle \quad G \vdash_{\text{ocl}} \langle \alpha_1; f \rangle \leq \alpha}{G \vdash_{\text{ocl}} \langle \alpha_1; m_1 \rangle \xrightarrow{\alpha}_{\text{true}} \langle \alpha_2; m_2 \rangle}
\\
\text{(OCL-DataFlow)} \\
\frac{G \vdash_{\text{ocl}} \langle \alpha_1; m_1 \rangle \xrightarrow{\alpha'_1}_{\text{ef}_1} \langle \alpha_2; m_2 \rangle \quad G \vdash_{\text{ocl}} \langle \alpha_2; m_2 \rangle \xrightarrow{\alpha'_2}_{\text{ef}_2} \langle \alpha_3; m_3 \rangle \quad G \vdash_{\text{ocl}} \alpha'_1 \leq \alpha'_2}{G \vdash_{\text{ocl}} \langle \alpha_1; m_1 \rangle \xrightarrow{\alpha'_1}_{\text{ef}_1 \vee \text{ef}_2} \langle \alpha_3; m_3 \rangle}
\\
\text{Data Structure } \Pi ::= \overline{\langle a; \sigma; m \rangle} \\
\text{Function } \text{fresh}(\{\alpha_1, \dots, \alpha_n\}) \stackrel{\text{def}}{=} [\alpha_1 \mapsto \alpha'_1, \dots, \alpha_n \mapsto \alpha'_n] \\
\alpha'_1, \dots, \alpha'_n \text{ fresh}
\end{array}$$

Figure 8. Type Constraint Closure

to initially avoid that complex issue. Any discussion of this formalism will be deferred to Sec. 3.4.

Expressions are typed via the $\vdash$ rules. (T-New) generates a fresh type variable α , allowing objects of the same class created at different program points to be distinguished. Added constraint $\langle a; L; \sigma \rangle \leq \alpha$ associates the static information (class name a , lexical point L) with α . Constraint $\mathbf{lazy}(\alpha, m, \emptyset)$ leaves a marker in the constraint set, stating the constraints associated with field initializers of a should be included at closure time. Eager inclusion of such constraints are not always possible because of recursion. We will have more details discussion of this marker constraint shortly.

The most central constraints are the $\alpha \xrightarrow{\text{hosts}} \alpha'$ constraints, which piece together the static access graph that will be analyzed later for isolation preservation. They are generated by (T-SyncM) and (T-Subtask), where **self** is a type variable with a pre-defined name. Constraint $\mathbf{self} \xrightarrow{\text{hosts}} \alpha$ says that the object enclosing the $o.m(v)$ expression or $o \Rightarrow m(v)$ is accessing o (which has type variable α). Messaging claims access, and this builds an edge on the static access graph $\alpha \xrightarrow{\text{hosts}} \alpha'$ meaning α hosts α' . Our type system types each class only once. At this modular typechecking phase, the type system does not know yet what type variable is used to represented the enclosing object. Hence a place holder **self** is used, which will be instantiated with the precise type variable at type closure time. We also do not eagerly collect the constraints belonging to the method body of the receiver (due to recursion); instead, a marker $\mathbf{lazy}(\alpha, m, \sigma)$ is added, informing the type closure algorithm to include the constraints belonging to the m method of object α later. This is reflected by Σ_h in the (T-Call) rule, a rule used by all three messaging rules for the common part of the three rules.

The only other place a $\xrightarrow{\text{hosts}}$ constraint occurs is (T-Capture). Here a fresh type variable is generated to represent the captured object (intuitively, an object with a new identity). The $\xrightarrow{\text{hosts}}$ constraint says that the enclosing object now treats the captured object as one of its own. Constraint $\mathbf{cap}(\alpha, \mathbf{self}) \leq \alpha'$ records the flow, remembering that the captured object α' originally was α and was captured by the enclosing object of the expression. Asynchronous messaging (T-AsyncM) does not have a return value. Subtyping is captured by (T-Sub), which is standard nominal subtyping.

3.2 Type Closure, Recursion Bounding, and Context Sensitivity

Type closure is defined by $G \vdash_{\text{ocl}} \Sigma$ judgments. If Σ is a singleton set $\{\text{cons}\}$, we also write $G \vdash_{\text{ocl}} \text{cons}$ for brevity. The type closure process starts with (OCL-Main), which starts with the constraints collected from the `main` function. The most interesting rules here are perhaps the four rules related to the processing of lazy constraints. For every $\mathbf{lazy}(\alpha, m, \sigma)$ constraint in the constraint set of the

$$\begin{aligned}
\text{classn}(G, \alpha) &\stackrel{\text{def}}{=} \{\mathbf{a} \mid G \vdash_{\text{oc1}} \langle \mathbf{a}; \mathbf{L}; \sigma \rangle \leq \alpha\} \\
\text{mod}(G, \mathcal{P}, \alpha) &\stackrel{\text{def}}{=} \{\mu \mid \mathbf{a} \in \text{classn}(G, \alpha), \mathcal{P}(\mathbf{a}) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle\} \\
\text{accesses}(G, \alpha, \alpha') &\stackrel{\text{def}}{=} (G \vdash_{\text{oc1}} \alpha \leq \alpha') \vee (G \vdash_{\text{oc1}} \alpha \xrightarrow{\text{hosts}} \alpha') \vee (\text{accesses}(G, \alpha, \alpha'') \wedge (G \vdash_{\text{oc1}} \alpha'' \xrightarrow{\text{hosts}} \alpha')) \\
\text{isAccessed}(G, \mathcal{P}, \alpha) &\stackrel{\text{def}}{=} \text{mod}(G, \mathcal{P}, \alpha) = \{\epsilon\} \wedge (G \vdash_{\text{oc1}} \alpha' \xrightarrow{\text{hosts}} \alpha) \text{ for some } \alpha' \\
\text{roots}(G, \mathcal{P}, \alpha) &\stackrel{\text{def}}{=} \{\alpha' \mid \text{accesses}(G, \alpha', \alpha), \text{mod}(G, \mathcal{P}, \alpha') = \{\mathbf{task}\}\} \\
\text{cutVertices}(G, \mathcal{P}, \alpha) &\stackrel{\text{def}}{=} \{\alpha_{\text{cut}} \mid \text{mod}(G, \mathcal{P}, \alpha_{\text{cut}}) = \{\mathbf{task}\} \text{ or } \{\mathbf{subtask}\}, \bigwedge_{\alpha_{\mathbf{r}} \in \text{roots}(G, \mathcal{P}, \alpha)} \text{accesses}(G, \alpha_{\mathbf{r}}, \alpha_{\text{cut}})\} \\
\text{isolatedTasks}(G, \mathcal{P}) &\stackrel{\text{def}}{=} \left. \begin{aligned} &\forall \alpha. \text{isAccessed}(G, \mathcal{P}, \alpha) \\ &\forall \alpha_1. \alpha_2. \{\alpha_1, \alpha_2\} \in \text{cutVertices}(G, \mathcal{P}, \alpha) \end{aligned} \right\} \implies \text{accesses}(\alpha_1, \alpha_2) \vee \text{accesses}(\alpha_2, \alpha_1)
\end{aligned}$$

$$\begin{aligned}
\text{instHost}(G, \alpha) &\stackrel{\text{def}}{=} \{\sigma(\mathbf{self}) \mid G \vdash_{\text{oc1}} \langle \mathbf{a}; \mathbf{L}; \sigma \rangle \leq \alpha\} \\
\text{capHost}(G, \alpha) &\stackrel{\text{def}}{=} \{\alpha' \mid G \vdash_{\text{oc1}} \mathbf{cap}(\alpha'', \alpha') \leq \alpha\} \\
\text{extFree}(G, \alpha, \alpha_t) &\stackrel{\text{def}}{=} \forall \alpha' \in \text{instHost}(G, \alpha) \cup \text{capHost}(G, \alpha). \text{accesses}(G, \alpha_t, \alpha') \\
\text{dissemFree}(G, \mathcal{P}, \alpha, \alpha_t) &\stackrel{\text{def}}{=} \left\{ \begin{aligned} &G \vdash_{\text{oc1}} \langle \alpha_s; \mathbf{m}_s \rangle \xrightarrow{\alpha} \langle \alpha_d; \mathbf{m}_d \rangle \\ &\text{accesses}(G, \alpha_t, \alpha_d) \end{aligned} \right\} \implies \text{mod}(G, \mathcal{P}, \alpha_s) \neq \{\mathbf{subtask}\} \\
\text{cordFree}(G, \mathcal{P}, \alpha, \alpha_t) &\stackrel{\text{def}}{=} \left\{ \begin{aligned} &G \vdash_{\text{oc1}} \langle \alpha_s; \mathbf{m}_s \rangle \xrightarrow{\alpha_{\text{ef}}} \langle \alpha_d; \mathbf{m}_d \rangle \\ &\text{accesses}(G, \alpha_t, \alpha_s) \\ &\text{accesses}(G, \alpha_t, \alpha_d) \end{aligned} \right\} \implies \text{mod}(G, \mathcal{P}, \alpha_d) \neq \{\mathbf{task}\} \\
\text{tunnelFree}(G, \mathcal{P}) &\stackrel{\text{def}}{=} \forall \alpha. \forall \alpha_t \left\{ \begin{aligned} &\text{isAccessed}(G, \mathcal{P}, \alpha) \\ &\alpha_t \in \text{roots}(G, \mathcal{P}, \alpha) \end{aligned} \right\} \implies \left\{ \begin{aligned} &\text{extFree}(G, \alpha, \alpha_t) \\ &\text{dissemFree}(G, \mathcal{P}, \alpha, \alpha_t) \\ &\text{cordFree}(G, \mathcal{P}, \alpha, \alpha_t) \end{aligned} \right\}
\end{aligned}$$

Figure 9. Functions *isolatedTasks* and *tunnelFree*

main method, the type system uses (OCL-Lazy-Intro) to close in the constraints associated with method m of object α . To ensure termination, we mimic the call stack in data structure Π , recording the methods whose constraints have already been included on the call chain. If the constraints are new, (OCL-Lazy-Non-Recursive) is used to merge the new constraints, while if the same method has been closed before, (OCL-Lazy-Recursive) simply equates the type variables of the current invocation and its previous invocation. The remaining rules propagate constraints along flow paths $\leq$, which is implicitly reflexive and transitive.

Context sensitivity is usually calculated in a program analysis formalism, but here we use a type-constraint-based formalism for its ease of mathematical manipulation and proof of correctness. We are maximally context-sensitive in the sense that every call site will refresh the type variables of the constraints of the invoked method body; we only bound this process at recursion so termination is achieved. The key variable refreshing happens in (OCL-Lazy-Non-Recursive), where all type variables in Σ except those showing up in $\text{dom}(\sigma_m)$ and $\text{dom}(\sigma_f)$ are refreshed. $\text{dom}(\sigma_m)$ contains the type variables representing the formal argument and the return value of the enclosing

method. These type variables are refreshed by the caller, not the callee, as can be seen in (T-Call) – they end up in the caller constraint set and are refreshed there. $\text{dom}(\sigma_f)$ contains the type variables representing the fields of the object. They are refreshed by (T-New), so that different instances of the same class get different type variables for fields, but for different invocations on the same instance, a field keeps a consistent view of the object it holds. Each object creation point in turn lies in some method, and that method invocation will re-refresh these variables so different calls will yield different object types. This is the reason why different `MyTask` instances in Fig. 4 can be approximated statically even though they are all instantiated from one program point. Note that we are giving a high-level description of closure which is more a specification than an implementation; many efficiencies can be realized in an implementation which we are not covering here.

3.3 Isolation Preservation

If we define an access path to be a $\xrightarrow{\text{hosts}}$ path from a **task** object to an ordinary object, then the specific invariant Task Types enforce is: *a subgraph of the static access graph that includes exactly all access paths to a fixed ordinary object in the static access graph must have one articulation point (cut*

vertex) which is a subtask or a task. Intuitively, what this says is, if more than one task can access an object, then that object must “hide behind” a subtask which controls the sharing, *i.e.* always be accessed through a single subtask. Note that such a definition does not restrict a DAG case like d in Fig. 4(a). In this case, d “hides behind” the task object of T_{a1} itself, which is thread local as well. The rigorous definition for this notion is the $cutVertices(G, \mathcal{P}, \alpha)$ function in Fig. 9, intuitively meaning the set of cut vertex ancestors of α . Helper function $mod(G, \mathcal{P}, \alpha)$ computes the modifier information for an object. It is easy to see that according to our type system, it is always a singleton set. Predicate $accesses(G, \alpha, \alpha')$ defines whether an access path exists in the static access graph. Function $roots(G, \mathcal{P}, \alpha)$ computes all “root” task objects that can access α . The definition of $isolatedTasks$ then captures the intuition expressed above: all cut vertices must be in a total order; in other words, it is always possible to find a “least upper bound” cut vertex for α . Readers can verify why the case of $d1$ in Fig. 4(a) type-checks, and that d in Fig. 4(b) will lead to a type error.

According to this definition, both subtask objects and ordinary objects can be owned, and all three can be owners. Note that the static access graph we are defining is a rather relaxed one compared with ownership types. For instance, we are not interested in enforcing any hierarchical structures between a pair of ordinary objects: they can form cycles if a call chain leads back to an ordinary object later. What interests us are how they interact with tasks and subtasks. It can be easily verified that the definition here indeed allows a subtask (or the ordinary objects it owns) to access the objects belonging to its ancestors, as long as it will not violate the invariant we have just defined: in cases a subtask object is accessed by multiple tasks, such a sharing would lead to problems, and it will indeed be caught by the type system.

3.4 Tunnel Detection

As observed in Sec. 2.5, a tunnel detection analysis is needed to guarantee soundness. This analysis is defined by the $tunnelFree$ function in Fig. 9; there are only three cases that might lead to “smuggling” tunnels, each of which is illustrated in Fig. 10. In all three sub-figures, $T1$ and $T2$ are two tasks that cannot be differentiated statically, as was the case in the while loop example we presented in Sec. 2.5. Therefore, when $isolatedTasks$ is computed, only $T1$ and $T3$ are approximated, and $isolatedTasks$ holds. That however still leads to non-shared-memory violations at run time.

The first kind of violation is dubbed an “external tunnel”, as shown in Fig. 10(a). Here $T3$ might instantiate object o and pass it down to both $T1$ and $T2$; this is possible since $T1$ and $T2$ might both have been created by $T3$. Indeed, this case is why the **while** loop example of Sec. 2.5 fails to type-check, and this violation is caught by the $extFree$ function which is one requirement of $tunnelFree$. That function in turn relies on functions $instHost(G, \alpha)$ and $capHost(G, \alpha)$, which computes object α ’s “host” – the object that instanti-

Figure 10. Tunneling

ated or captured α . In both cases α is purely local to its host, and the $extFree$ function requires that the host is task-local, so α must be task-local as well.

We call the second violation “a disseminator tunnel”, and is illustrated in Fig. 10(b); it is prevented by the $dissemFree$ function. In the figure, suppose we have an object $o2$ that is instantiated inside $T3$, and a reference to $o2$ is stored in a field of subtask object $o3$. At different times, $o2$ can be passed into $T1$ via $sx1$, and into $T2$ via $sy2$. The $isolatedTasks$ definition would not be able to detect such a case for $o2$ and be tricked to believe $o2$ was in fact sent to the same task twice. Observe that for this “tunnel” to exist, a necessary condition is the subtask $o3$ has to store $o2$: the trick to be a “disseminator” is to send the *same* object ($o2$ here) to two different tasks that cannot be differentiated by a type variable. The $dissemFree$ function analyzes over the data flow chain, based on the constraints collected in the per-class typechecking phase, those that are in gray color that we have yet to explain. In (T-Call), notice that constraint $\{\langle \mathbf{self}; m_0 \rangle \xrightarrow{\alpha'_1 \text{ false}} \langle \alpha; m \rangle, \langle \alpha; m \rangle \xrightarrow{\alpha'_2 \text{ false}} \langle \mathbf{self}; m_0 \rangle\}$ is collected, meaning an object α'_1 flows from the m_0 method of **self** object into the m method of the α object, whereas the second constraint records the flow from the return value. The **false** here is an *effect label*, and a **true** value denotes the flow originates from an effectful source. With the help of the read constraints collected in (T-Read) and a type closure

rule for recording the mutable source (OCL-MutSource), and a propagation rule (OCL-DataFlow), the data flow can be combined with the effect. What the *dissemFree* definition states is that if a data flow exists between a pair of objects α_s and α_d and α_d is inside a task (α_t), it must not be the case that the flow path starts from a subtask object that has stored the object represented by α .

The third case is called a “birth cord tunnel”, illustrated in Fig. 10(c), and is prevented by the *cordFree* function. Again suppose we have an object o that is instantiated inside T1. A reference to o in T1 can in fact be passed from $sx1$, and then to $o3$. Suppose $o3$ in fact starts up T2 by sending an asynchronous message to $t1$ again and passes o as the argument. The *isolatedTasks* definition would also not be able to detect such an invalid case. What *cordFree* states is if a data flow exists between a pair of objects α_s and α_d inside a task (α_t), and one end smuggles the object α out and the other end smuggles it in, it is not allowed that the flow path ends with a task, irrelevant of whether the path is effectful or not. Here observe that since *accesses*(G, α_t, α_d) holds, if α_d were to be a task, then α_t and α_d must be equal according to the *accesses* definition – the task is sending an asynchronous message to a task object that statically cannot be differentiated from its own task object, and a smuggling tunnel is possible.

Note that the tunnel detection analysis is more relaxed than a typical escape analysis. The tunnel detection analysis believes it is perfectly fine for a locally instantiated object to leak out of the task boundary, as long as there are no “tunnels” for it to flow back to a task represented by the same type variable. It is sound because, if the aforementioned were to leak out the task boundary and were to be accessed by a task that *can* be differentiated by the type system via type variables, such a violation would have been caught by by *isolatedTasks*.

4. Dynamic Semantics

We now define a lock-based operational semantics for our language. Related definitions and selected reduction rules are given in Fig.11. The rest of rules are given in Appx. B. Small-step reductions $S \Rightarrow S'$ are defined over states $S = \langle H; T \rangle$ for H the object heap, and T a set of parallel tasks (and subtasks). H is a mapping from objects o to field stores (Fd) tagged with their class name a . Program point L and substitution σ are only used to facilitate proofs. A task (or subtask) is a tuple consisting of the task (or subtask) object o that initiates it, and an expression e to be evaluated. Expressions are extended with values v and runtime expression **wait** o that blocks the current execution and waits for the completion of subtask executed in o . For the convenience of computing closures, field read expression f is always treated as **this.f**, and similarly for the field write expression. Evaluation context E is standard. Let $\langle H; T \rangle \Rightarrow_* \langle H'; T' \rangle$ denote the multi-step reduction transitively defined over $\Rightarrow$.

$$\begin{aligned}
H &::= \overline{o \mapsto \langle a; L; \sigma; Acc; Fd \rangle} \\
Fd &::= \overline{f \mapsto v} \\
T &::= \langle o; e \rangle \mid T \parallel T' \\
Acc &::= \overline{o} \\
v &::= o \\
e &::= \dots \mid v \mid \mathbf{wait} \ o \mid e.f \mid e.f := e \\
o &::= \text{object ID} \\
E &::= \bullet \mid v.f := E \mid \mathbf{capture} \ E \mid (a)E \\
&\mid E.m(e) \mid v.m(E) \\
&\mid E \rightarrow m(e) \mid v \rightarrow m(E) \\
&\mid E \Rightarrow m(e) \mid v \Rightarrow m(E)
\end{aligned}$$

$$\begin{aligned}
\text{tryLock}(H, o, o_t) &\stackrel{\text{def}}{=} H' = H[o \mapsto \langle a; L; \sigma; Acc \cup \{o_t\}; Fd \rangle] \\
&\text{if } H(o) = \langle a; L; \sigma; Acc; Fd \rangle \\
&\quad Acc \subseteq \text{ancestors}(H, o_t) \\
\text{method}(P, H, o, m) &\stackrel{\text{def}}{=} Me \\
&\text{if } H(o) = \langle a; L; \sigma; Acc; Fd \rangle \\
&\quad \mu \text{ class } a \{ F \ M \} \in P \\
&\quad Me \in M, \text{name}(Me) = m
\end{aligned}$$

$$\begin{aligned}
&(\text{R-Capture}) \\
&\frac{H' = \text{tryLock}(H, o, o_t)}{H, \langle o_t; \mathbf{capture} \ o \rangle \Rightarrow H', \langle o_t; o \rangle} \\
&(\text{R-SyncM}) \\
&\frac{H' = \text{tryLock}(H, o, o_t) \quad \text{method}(P, H, o, m) = a' m(a \ x)\{e\}}{H, \langle o_t; o.m(v) \rangle \Rightarrow H', \langle o_t; e\{v/x\}\{o/\mathbf{this}\} \rangle} \\
&(\text{R-AsyncM}) \\
&\frac{H(o) = \langle a; L; \sigma; \emptyset; Fd \rangle \quad H' = H[o \mapsto \langle a; L; \sigma; \{o\}; Fd \rangle] \quad \text{method}(P, H, o, m) = a' m(a \ x)\{e\}}{H, \langle o_t; E[o \rightarrow m(v)] \rangle \Rightarrow H', \langle o_t; E[o_{\text{unit}}] \rangle \parallel \langle o; e\{v/x\}\{o/\mathbf{this}\} \rangle} \\
&(\text{R-SubTask}) \\
&\frac{H(o) = \langle a; L; \sigma; \emptyset; Fd \rangle \quad H' = H[o \mapsto \langle a; L; \sigma; \{o_t\}; Fd \rangle] \quad \text{method}(P, H, o, m) = a' m(a \ x)\{e\}}{H, \langle o_t; E[o \Rightarrow m(v)] \rangle \Rightarrow H', \langle o_t; E[\mathbf{wait} \ o] \rangle \parallel \langle o; e\{v/x\}\{o/\mathbf{this}\} \rangle}
\end{aligned}$$

Figure 11. Definitions and Selected Rules for Dynamic Semantics

Let $P \Rightarrow_{\text{init}} \langle H; T \rangle$ denote the first step of reduction that leads to the initial configuration. Let $P \Rightarrow_* \langle H; T \rangle$ denote $P \Rightarrow_{\text{init}} \langle H; T \rangle$ and $\langle H'; T' \rangle \Rightarrow_* \langle H; T \rangle$. We use $\langle H; T \rangle \uparrow_P$ to denote the computation diverges. Reductions are implicitly parameterized with code base P .

Each object o on the heap has a lock set Acc for recording tasks (or subtasks) that have accessed o . Lock-inducing object access can happen only in the reductions of the following four expressions: (R-Capture), (R-SyncM), (R-AsyncM), (R-SubTask). The latter two require that the task/subtask object is currently not accessed $Acc = \emptyset$, so that serial processing of requests is enforced. (R-Capture) and (R-SyncM) depend on pre-condition *tryLock*(H, o, o_t), which attempts to have task/subtask object o_t lock o if a subset condition, explained shortly, holds. If successful, the

Acc set is incremented, recording the fact that o_t currently locks the object. In fact, the combinational view of Acc is precisely the dynamic access graph:

Definition 1 (Dynamic Access Graph). *For heap H , dynamic access graph $accG(H)$ is defined as a graph $\langle V; E \rangle$ where $V = \text{dom}(H)$ and $E = \{o' \xrightarrow{\text{dhosts}} o \mid H(o) = \langle a; L; \sigma; Acc; Fd \rangle, o' \in Acc\}$.*

We are able to prove the dynamic access graph is always a forest.

Theorem 1 (Dynamic Task Isolation). *If $P \Rightarrow_* \langle H; T \rangle$, then graph $accG(H)$ is a forest of directed redundant trees (i.e. trees potentially with redundant edges). The direction is defined by $\xrightarrow{\text{dhosts}}$.*

For a forest with no redundant edges, every ordinary object would precisely have one parent – isolation is achieved. When redundant edges appear, it signifies the case that a subtask can access the objects accessed by its ancestors. Note that subtasking is synchronous as we explained in Sec. 2.1, and hence this is correct (see [25] for a more detailed discussion of this issue). An orphan node with no edges in the forest denotes an ordinary object not accessed by any task, or a task/subtask object with no running tasks/subtasks. A node representing a task object may also have cyclic edges, meaning the object itself has been accessed (i.e. it is currently running some task).

At first glance, it might first be unintuitive that the static access graph of Sec. 2.1 is not a forest, but the dynamic one is. The key issue here is that the static system is only aware that a subtask object is shared by multiple tasks, but dynamically, the subtask object is indeed processing the request from different tasks *one by one*, so for any snapshot of the run-time, each subtask can only be accessed by one task.

Knowing the dynamic access graph is a forest, we now turn back to the reduction rules. Function $ancestors(H, o_t)$ is the standard function to compute all ancestor nodes of o_t on the access graph of $accG(H)$, inclusive of o_t itself. Precondition $Acc \subseteq ancestors(H, o_t)$ intuitively means that to allow o_t to access o , it must be true that the tasks/subtasks that are currently accessing the object are the ancestors of o_t , the case we explained just above. When $Acc = \emptyset$, it can be trivially seen that o_t will be given access – o is currently not accessed by anyone. When o is currently accessed by some task and o_t is a task/subtask that are not from the same tree, the reduction is blocked. This is consistent with our intuition: **capture** o and $o.m(v)$ will be blocked if o is held by another task on a different tree according to (R-Capture). The rest of the formalism is relatively straightforward.

5. Formal Properties

We now establish the main properties of Task Types.

Theorem 2 (Type Soundness). *If judgment $\vdash_{\text{global}} P \setminus G$ and $\text{deadlock_free}(G, P)$, then either $S \uparrow_P$, or $S \Rightarrow_* \langle H'; \langle o; v \rangle \rangle$ for $P \Rightarrow_{\text{init}} S$, and some H', o, v .*

This Theorem states that the execution of a statically typed deadlock-free program either diverges or computes to a value. The Theorem is established by showing Lemmas of Subject Reduction, Progress, and the fact that the bootstrapping process leads to a well-typed initial state. The theorem depends on predicate deadlock_free , which we define now.

Definition 2 (Deadlock Freedom). *$\text{deadlock_free}(G, P)$ iff $\neg \text{cycleCap}(G) \wedge \neg \text{cycleSubtask}(G, P)$ where $\Xi(P) = \mathcal{P}$.*

Definition 3 (Cyclic Capture Chain). *$\text{cycleCap}(G)$ holds iff there exist $\alpha_{\text{host}}^0, \dots, \alpha_{\text{host}}^{n-1}$ and $\alpha_{\text{ord}}^0, \dots, \alpha_{\text{ord}}^{n-1}$ and for $i = 0, \dots, n-1$, predicate $\text{accesses}(G, \alpha_{\text{host}}^i, \alpha_{\text{ord}}^i)$ and judgment $G \vdash_{\text{oc1}} \text{cap}(\alpha_{\text{ord}}^i, \alpha_{\text{host}}^{(i+1) \bmod n}) \leq \alpha_{\text{ord}}^i$ holds.*

Definition 4 (Cyclic Subtask Invocation Chain). *Predicate $\text{cycleSubtask}(G, P)$ holds iff there exist distinct α and α' such that $\text{mod}(G, P, \alpha) = \text{mod}(G, P, \alpha') = \{\text{subtask}\}$, $\text{accesses}(G, \alpha, \alpha')$, $\text{accesses}(G, \alpha', \alpha)$, $G \vdash_{\text{oc1}} \langle a; L; \sigma \rangle \leq \alpha$, $G \vdash_{\text{oc1}} \langle a'; L'; \sigma' \rangle \leq \alpha'$, and $\sigma(\text{self}) \neq \sigma'(\text{self})$.*

These definitions constitute the standard decision procedure for statically detecting potential deadlocks: a program can deadlock if there is no partial ordering on object access. In our language, this may happen either when there is a cyclic chain of object capturing (Def. 3), or there is a cycle on subtask invocation (Def. 4). Note that in the latter definition, two subtasks are enough to express the invocation chain since accesses itself is transitive. The comparison $\sigma(\text{self}) \neq \sigma'(\text{self})$ is used to make sure self-deadlocking is not possible on subtasks.

Our language greatly reduces the likelihood of deadlocks. Potential deadlock can only occur with capture and subtask blocking, and those are in turn explicitly specified in the program and are used only to declare sharing. A precise type system will also maximally differentiate the type variables in the above, reducing the chance of forming false positive cycles in the static access graph.

If predicate deadlock_free was added as a pre-condition of the (T-Global) rule, Task Types would achieve deadlock-free programming; Autolocker [27] achieves deadlock freedom in a similar fashion. As a theoretical foundation however, we consciously avoid this route, and allow language implementors to decide on how to treat a program for which deadlock_free fails: reject it, provide warnings, or insert run-time checks.

We now describe some properties related to task isolation.

Definition 5 (Task-Local Objects). *Predicate $\text{local}(P, L)$ asserts that the objects created at lexical point L are local, and holds iff $\vdash_{\text{global}} P \setminus G$, $\Xi(P) = \mathcal{P}$, $G \vdash_{\text{oc1}} \langle a; L; \sigma \rangle \leq \alpha$, and $\mathcal{P}(a) = \langle \epsilon; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle$ implies there does not exist α', α'' such that $G \vdash_{\text{oc1}} \text{cap}(\alpha, \alpha'') \leq \alpha'$.*

Definition 6 (Lock Erasure Semantics). *Reduction in the presence of lock erasure $P \xRightarrow{*} S$ is defined as identical to $P \Rightarrow_* S$, except with function `tryLock` redefined as $\text{tryLock}(H, o, o_t) \stackrel{\text{def}}{=} H' = H[o \mapsto \langle a; L; \sigma; \text{Acc} \cup \{o_t\}; Fd \rangle]$ for $H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle$, if $\text{local}(P, L)$, and `tryLock` unchanged when $\text{local}(P, L)$ does not hold.*

In lock erasure semantics, the pre-condition $\text{Acc} \subseteq \text{ancestors}(H, o_t)$ of `tryLock` is removed for local objects. What this implies is the lock set is never used to determine local object reductions, so any real-world implementation would just ignore the book-keeping of this data structure.

Theorem 3 (Static Task Isolation). *If $P \xRightarrow{*} \langle H; T \rangle$, then graph $\text{accG}(H)$ is a forest of directed trees potentially with redundant edges. The direction is defined by $\xrightarrow{\text{dhosts}}$.*

This theorem says that even if the reduction always ignores the locks on local objects, the dynamic access graph is identical to the one achieved following $\Rightarrow_*$, as stated in Thm. 1. In other words, isolation is achieved without the need of run-time locking. Definition *local* thus defines a decision procedure that a compiler can use to find local objects that do not need runtime locks.

Last, we can prove static task isolation preserves atomicity as defined in [25].

Theorem 4 (Atomicity). $\vdash_{\text{global}} P \setminus G$, then the atomicity property defined in [25] is preserved for all executions of P with lock erasure semantics.

We lack the space for a formal definition of the atomicity property; informally it is the pervasive, partitioned atomicity property described in Sec. 2.2. Since our dynamic system is very similar to [25] where this property is formally proved, the same proof technique applies to the context here.

6. Programming with Task Types

Task type programming brings object sharing between tasks to the fore, and constitutes a different programming philosophy; in this section we explore this philosophy by considering some common multithreaded programming paradigms. The vast majority of objects are ordinary objects not shared across tasks, and they can be programmed normally as well. Our focus here is on the more challenging case of object sharing and synchronization.

6.1 Pure Task Synchronization

Tasks often need to synchronize in ways that do not involve passing of objects. Examples include semaphores, barriers, latches, etc. All of these synchronizations obviously constitute break-points in the atomicity of the involved tasks, and this translates to the fact that their implementation interface will be a subtasking invocation. Since no significant object data is passed from one task to the other there will not be conflicts of which thread owns what object. These synchro-

```
// The semaphore itself
subtask class Semaphore {
    int counter = 0;
    void newProxy() {
        sh = new SemaphoreProxy();
        sh.setSem(this); return(sh);
    }
    boolean p() {
        if (counter > 0)
            { counter = counter - 1; return(true); }
        else { return(false); }
    }
    void v() {
        counter = counter + 1;
    }
}

// A local proxy for the semaphore which can busy wait
class SemaphoreProxy {
    Semaphore theSem;
    void setSem(Semaphore s) {
        theSem = s;
    }
    void p() {
        if (! theSem=>p()) { this.p(); } // busy wait
    }
    void v() {
        theSem =>v();
    }
}

// Test user code
task class SemaphoreUser {
    void testSem(Semaphore s) {
        sh = s =>newProxy(); // get a local proxy
        // ... arbitrary sh.p() and sh.v() messagings ...
    }
}

class Main {
    void main() {
        t1 = new SemaphoreUser();
        t2 = new SemaphoreUser();
        s = new Semaphore();
        t1 ->testSem(s);
        t2 ->testSem(s);
    }
}
```

Figure 12. A Simple Semaphore

nization primitives have similar implementation strategies; here we give the simplest, a semaphore.

A Semaphore Semaphores can be directly implemented with Task Types; however, the implementation requires busy waiting and so they would likely be built into a production language as a primitive. It is nonetheless useful to define a busy waiting semaphore as a case study. Figure 12 shows how a simple semaphore could be implemented as a shared subtask class `Semaphore`. The testing main program here creates two semaphore user tasks `t1` and `t2` which are launched and then repeatedly invoke semaphore `P` and `V` operations in some arbitrary order. The `P` operation on a semaphore may sometimes (correctly) block, but there would be a problem if method `p()` on the subtask `Semaphore` object blocked: subtasks can process only one message at a time, and so the blocking `p()` would unintentionally starve out any incoming `v()` and the system would deadlock. For this reason, each task asks for a local proxy, a

SemaphoreProxy, which it will interact with, and which can poll the actual Semaphore `p()` method, which is not blocking. Since access to subtask objects is mutually exclusive, there will be no race conditions on the semaphore operations.

The sharing of `s` between tasks `t1` and `t2` will typecheck because class Semaphore was declared as a **subtask class** and can be shared freely. Note that each `p()`/`v()` invocation on `theSem` will define a break point in the otherwise complete atomicity of `t1/t2`. The proxies `sh` local to each task can be polymorphically typed and instantiated uniquely for each `t1` and `t2`, meaning the type system will be smart enough to see two versions of `sh`'s type, one for each task, and not just one version. If there were only one `sh` type the two tasks could not share it since only one task can be owner, and there would be a type error. This shows the power of the polymorphic type system.

6.2 Synchronization with Data Sharing

The default case of task types is for no sharing of objects between tasks to occur; every such sharing is explicitly declared, and arises at a synchronization point or subtask when objects are passed out or in. Object sharing comes to the forefront in Task Type programming: the default behavior is that tasks cannot share data and so most of the “new thinking” needed for task type programming involves planning how objects will be shared for the cases that it needs to be.

Before studying a particular example let us analyze the different cases of object sharing. The following are all possible interface points where objects can pass from one task to another:

1. objects passed as arguments when a new task is started;
2. objects passed as arguments to subtasks; and,
3. objects returned by subtask invocations.

If in a program no objects are passed by any of these three mechanisms, then there can be no object sharing between tasks; in that sense these three cases define a complete interface for object sharing between tasks.

Objects passed by one of the three cases above can themselves be of several different forms. Here is a list of all possible interesting cases, including several that have not been defined in the system of this paper:

1. task and subtask objects, which can be freely shared since they cannot be owned by tasks;
2. captured objects, i.e. objects which the receiver captures to their task, which as long as the original owning task has completed will be successfully transferred to the receiving task;
3. black box objects, which the receiving task will not be able to touch since they are considered owned by a different task;

4. transferred objects (not in this paper), the case where the originating task has not yet terminated but it has released the object and will no longer be able to access it;
5. immutable objects (not in this paper), which can be shared freely; and,
6. objects passed by copy (not in this paper), another case where free sharing is supported.

For all but case 1. above there is in addition a “depth” dimension to the sharing – the fields of the objects themselves each fall into one of the six cases above. For example, an object could at the top level be immutable but could contain one field that is a subtask object and another field that is a captured object, *etc.*

For objects being shared, programmers need to decide which modality is the most appropriate. This work is rewarded by producing a program with a formally declared sharing policy which is fully enforced by the language, as opposed to the current practice of an informal policy which is spottily enforced by the language design.

Object Sharing in MapReduce Here we look at a common type of multi-core algorithm, the case where there is a large data structure with an “embarrassingly parallel” algorithm that works by partitioning the data structure between tasks. One such programming pattern is MapReduce [14], which is similar to the “join continuation” of actors [3]. Here we develop a simplified MapReduce which captures much of the behavior in a small amount of code. The code for our MapReduce is given in Figure 13. Each `WorkUnit` object encapsulates data and a unit of work that needs to be done on the data, performed as `aWorkUnit.work()` which in this simplified example just returns a single `int` result. We elide the details of the work units here, but an example of a work unit could be a single chromosome of DNA plus a pattern to search for in the chromosome; the result is the number of distinct occurrences of that pattern. The total work units is then the set of all 22 chromosomes and a pattern to search for. Each `Mapper` task loads in a unit of work, performs the work, and passes the result to a `Reducer` subtask which combines each of the numerical results via a commutative and associative operator `combine`. This example focuses on the shared objects; in practice there will be many objects within `WorkUnits` holding the actual data and doing the computation on that data, and since those objects are not shared there will be zero programmer burden at that point – only shared objects require programmer overhead.

The object sharing between tasks and subtasks is defined via Task Types. The subtask class `UnitLoader` loads the data of one work unit from somewhere, e.g. it may read it from a file. This activity takes place in a subtask because each `Mapper` needs to take its turn loading data and subtasks provide mutual exclusion. Observe that if `main()` were to read in the work units directly it would own all of them and then the individual `mapper` tasks would not be

```

task class Mapper {
    void map(UnitLoader ul, Reducer r) {
        wu = ul => loadWorkUnit();
        int result = wu.work();
        r => reduce(result);
    }
}
subtask class Reducer {
    int current = 0;
    int size;
    void setSize(int s) {
        size = s;
    }
}
void reduce(int r) {
    current = this.combine(current, r);
    size = size - 1;
    if (size == 0) { ...output final result ... };
}
int combine(int i, int j) {
    ... combine i and j, e.g. add them
}
}
subtask class UnitLoader {
    WorkUnit loadWorkUnit() {
        ... load in a new WorkUnit() ...
    }
}
class WorkUnit {
    ... declared local data here ...
    int work() { ... code which does work on data
}
}
task class Main {
    void main() {
        reducer = new Reducer();
        ul = new UnitLoader();
        reducer => setSize(this.getNumUnits());
        for (i=1; i == this.getNumUnits(); i++) do {
            mapper = new Mapper();
            mapper -> map(ul, reducer);
        }
    }
    int getNumUnits() {
        ... returns the number of work units
    }
}

```

Figure 13. A Simplified Map-Reduce Example

able to access them until `main()` completed. And, even at that point a **capture** expression would be needed in the `mapper` to dynamically transfer ownership from the `Main` to the `Mapper` task. In this example, the `Mapper` `map` method can freely access its work unit `wu`; it was created by the `UnitsLoader` but the only “root task” with access to it is the particular `mapper` task itself, so the `mapper` is the cut vertex and so the objects are appropriately partitioned. Also by inspection observe there are also no tunnels whereby `wu` could sneak out into some other `mapper` task: this is trivial because the `wu` are not passed anywhere except from `UnitsLoader` to the appropriate `mapper`, and the loader does not store it in a field. The `reducer`, `ul`, and all the `mappers` are either tasks or subtasks and so can be shared freely.

This example shows how programmers need to pay close attention to object sharing when programming with Task Types. The benefit is a guarantee of many fewer interleaving

scenarios and significantly eased debugging burden, resulting in faster deployment of more reliable software.

7. Related Work

Type Systems and Static Analyses STM systems primarily use dynamic means to enforce atomicity [21; 38; 8]. The problem of weak atomicity has attracted significant interest recently, and a number of static or hybrid solutions exist to ensure transactional code and non-transactional code do not interfere with each other. Such a property is enforced by Shpeisman et. al. [32] via a hybrid approach with dynamic escape analysis and a not-accessed-in-transaction static analysis. The latter has the good property of allowing “data hand-off”: a transaction can pass along an object without accessing it. Data handoff is useful for different threads sharing a data structure (say a queue) but not the elements in it. This property also holds for Task Types. Passing along an object reference or storing the reference is not considered access since atomicity is not violated. AME [1] describes a conceptually static procedure to guarantee violation-freedom of transactional code and non-transactional code. Harris et. al. [22] used the monads of Haskell to separate computations with effects and those without.

Constructing type systems to assure race condition freedom is a well-explored topic. These systems work under significantly different assumptions than we do: they typically assume a Java-like shared memory with explicit lock acquire/release expressions. Given the non-shared memory assumption and protected access to subtask objects, race conditions do not occur in our system. On a technical level, some of these race-free type systems share properties with Task Types. For instance, RaceFreeJava [18] allows programmers to declare certain classes to be **threadlocal** so that their instances can be excluded from consideration by their main algorithm. Some techniques are designed to remove unnecessary locks for synchronized blocks [4; 31]. These systems can be viewed as pre-cursors to lock inference for atomic sections [27; 17; 9; 23], which we reviewed in Sec. 1.

There are many static analysis algorithms for tracking the flow of objects. Closest to our work are several thread escape analyses [15; 40; 6] for object-oriented languages, which use reachability graphs to prevent or track alias escape from threads. Task Types share a focus on thread locality with this work, but differ in two important aspects: (1) the pervasive atomicity of our language requires subtask-style sharing, which is a “partial escape” that should be allowed but has no representation in these analyses; (2) Fundamentally, object references do not need to be confined to guarantee atomicity: it is perfectly fine for a task to create an object, pass it over to another task, which in turn stores it or passes it further to a third task. The key to atomicity is there is no conflicting object access. The effect of these differences is to produce unique issues that escape analyses do not encounter and we need to solve here. Two particular examples are our cut-

vertex definition of object isolation, and our tunnel detection methodology.

Language Designs for Atomicity The shortcomings of explicitly declared atomic blocks are summarized in [35]. In that work, a data-centric approach is taken: the fields of an object are partitioned into statically labelled *atomic sets*, and access to fields in the same set is guaranteed to be atomic, analogous to declaring different subtasks for different atomic sets in Task Types. Their data-centric approach is perhaps one step forward compared with atomic blocks, but the design philosophy is still no-atomicity-unless-you-declare-it, and hence fundamentally different from our notion of pervasive atomicity. Atomic sets are particularly well-suited for atomicity within one object; for atomicity spanning multiple objects, a keyword **unitfor** is provided to compose atomic sets, but this increases annotation overhead, and violates a property atomic blocks have that we believe are intuitive: “atomicity is deep”, *i.e.* all directly and indirectly invoked objects from an atomic block should be protected. In our work, task atomicity is deep by default.

A recent work that is closer to our spirit of pervasive atomicity is AME [24; 1]. The language constructs of AME are along the lines of Actors, where an **async** e expression starts up an atomicity-preserving thread. AME is different from our work in its support of an expression **unprotected** e , meaning atomicity is not preserved for e . Because of the interleaving of code inside the **unprotected** and the default protected code, strong atomicity cannot be achieved when the condition of violation freedom is not satisfied. AME does not support synchronous messaging, and does not overlap with the static type system aspect of our work.

In Sec. 1, we explained how the Actors model and actor-based message-passing languages such as Erlang [5] can achieve pervasive atomicity. Kilim [33] is a more recent actor-like language, with a focus on providing refined message passing mechanisms without sacrificing the isolation property of Actors. The Kilim type system relies on extra programmer declarations called *isolation modifiers* to denote how each parameter can be passed/used in the inter-actor context. Kilim and Task Types have the same focus on object isolation, but in orthogonal and complementary design spaces: Kilim on message passing, and Task Types on atomicity.

8. Towards a Realistic Language

This language proposal represents a fairly significant departure from the norm, and it produces many issues which we had to leave out of this initial design to keep it manageable. In this section we sketch some of the other features that may be needed in a full implementation. More programming experience will be needed to determine which features below are critically needed and which are not.

Type System Extensions Perhaps the most pressing area is to further increase the accuracy of Task Types, so more

programs that in fact do not violate atomicity properties are typeable. Some obvious extensions include the following. Immutable objects can be freely shared between tasks without violating atomicity, and so a mechanism for creating immutable objects is needed; this is not a difficult problem. Similarly, call-by-copy is a variation on immutability which also allows object data to be passed from one task to another without introducing any atomicity-violating back-channels of information flow. Both of these declarations are useful in a “shallow” form in which only the top-level object is copied/immutable, but deep versions may also be desirable, to allow a large immutable object structure to be shared or to copy a large object structure. In this paper, we have considered the most general case where every program point is recursive. A simple algorithm can be designed to label non-recursive program points (such as the `main` method), so that tunnel detection there would be unnecessary.

As a conceptual model, Task types define object access to occur at method invocation time. Of course, what really affects atomicity is when the object fields are read or written. This view can be supported by collecting the $\xrightarrow{\text{hosts}}$ constraints at (T-Read) and (T-Write), rather than at (T-SyncM). Non-exclusive read / exclusive write locks can also easily be added.

A less trivial extension is the support of object transfer. Currently, tasks hold on to objects even if they are in fact finished with them. In many cases it is statically determinable that once a task passes an ordinary object to another task it will no longer access it; in this case there is no atomicity violation, even though the object is shared – its owner was simply changed from one task to another. In cases where it cannot be statically determined that the host task is finished, it is still possible to dynamically transfer an object via an explicit declaration: the receiving task becomes the new owner, and the originating task will be forced to raise an exception if it touches the object. Implementing such a transfer is easily accomplished by the compiler by using a proxy pattern on the host side.

Language Construct Extensions Programmers may wish to force an ordinary object invocation, say $e.m(e')$, to be strictly atomic, *i.e.*, with no subtasking appearing inside the body of m or the methods directly or indirectly invoked by that method. Currently there is no direct way to achieve this, but we could easily extend our language with an expression, say $e..m(e')$, to indicate such strict atomicity. Since the type constraint set contains a conservative approximation of all methods m may invoke, and since the effect of dynamic dispatch can be mitigated with the precise concrete class type information for each object, it is easy to statically verify this property.

There are also different forms of object capture that could be considered beyond the current block-until-freed modality supported now. Both a non-blocking capture which raises an exception when an object is busy, as well as a capture with

timeout which will not block forever, are potential additions. It may similarly be useful to support these variations on subtask message blocking if the subtask object is currently busy: either raise an exception now, or support a timeout which could raise an exception later if the wait is too long. Supporting these variations can further help reduce deadlocks.

There are several language features not addressed here, including exceptions, native methods, interface types, packages, *etc.* As alluded to in the previous section, a production language would also need to include non-busy-waiting implementations of semaphores, synchronization barriers, latches, *etc.*

Towards an Efficient and Robust Implementation Effective implementation of context-sensitive/polymorphic inference algorithms have been investigated extensively [28; 16; 2; 36; 39]. We are currently implementing Task Types on top of the CoqJava compiler [25], which is in turn built on top of Polyglot [30]. The ongoing implementation of polymorphic inference is based on Wang and Smith’s work [36], and we are also actively looking at various optimizations such as BDDs [39]. For full-fledged deadlock avoidance, one strategy we are interested in pursuing is a Discrete Control Theory-based model [37], which re-organizes locks with maximum permissibility of parallelism.

9. Conclusion

We have presented a new approach to achieve atomicity in multithreaded object-oriented programs, taking a top-down, mostly static approach as opposed to the usual bottom-up and usually dynamic approach. While our language syntax requires only minor changes to a standard object model, the approach to data sharing between threads is significantly different than the norm. A consequence of the model is the object sharing between tasks is brought front and center for the programmer, where it should be.

Atomicity with Task Types is top-down in the sense that it is pervasive: we slice the whole into atomic parts instead of building islands of atomic blocks. Our notion of sharing between tasks is principled, via subtasking or capturing, both of which still preserve atomicity properties: strong atomicity is achieved for all code regardless of sharing.

Task Types incorporate a powerful inference algorithm with a precise polymorphic/context-sensitive analysis which statically verifies that ordinary objects are appropriately partitioned between tasks; Task Type programming thus requires minimum annotation overhead. Since the partitioning is verified statically, there is no need for dynamic partitioning of objects between tasks, meaning there is no additional runtime overhead. The one exception is when the type system is too weak – in this case we support a cast-like **capture** syntax which dynamically assigns an object to a particular task. We still need to use locks on captured objects since there may be real run-time contention; even though we use locks here, our approach is largely neutral as to whether

locks or STMs are used. Type soundness and isolation enforcement are formally proved. The “partial sharing” enabled by subtasks adds a significant subtlety to correctness.

A Technical Report containing the proofs may be found online [26].

References

- [1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL ’08*, pages 63–74, 2008.
- [2] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA, 1996.
- [3] Gul Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.
- [4] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *In SAS’99*, page pages, 1999.
- [5] J. Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP’96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, Hino, Tokyo, Japan, 1996.
- [6] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. *SIGPLAN Not.*, 34(10):20–34, 1999.
- [7] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA ’02*, pages 211–230, 2002.
- [8] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI’06*, June 2006.
- [9] Sigmund Cheren, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI’08*, pages 304–315, 2008.
- [10] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *PLDI ’04*, pages 243–254, 2004.
- [11] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.
- [12] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *In OOPSLA’98*, pages 48–64. ACM Press, 1998.
- [13] Cray Inc. Chapel Specification, 2005.
- [14] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI’04*, 2004.
- [15] Jong deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA’99*, pages 1–19, 1999.
- [16] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive inter-procedural points-to analysis in the presence of function pointers. In *PLDI’94*, pages 242–256, 1994.
- [17] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL ’07*, pages 291–296, 2007.
- [18] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *In PLDI’00*, pages 219–232. ACM Press, 2000.
- [19] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI’03*, pages 338–349, 2003.
- [20] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI ’03*, pages 13–25, 2003.
- [21] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA’03*, pages 388–402, 2003.
- [22] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP ’05*, pages 48–60, 2005.
- [23] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *TRANSACT’06*, June 2006.
- [24] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *HOTOS’07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.
- [25] Yu David Liu, Xiaoqi Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity. In *CC’08*, March 2008.
- [26] Yu David Liu and Scott F. Smith. Task Types for Pervasive Atomicity (Long Version), <http://www.cs.binghamton.edu/~davidl/tasktypes>. Technical report, March 2009.
- [27] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL’06*, pages 346–358, 2006.

- [28] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.
- [29] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP'07*, March 2007.
- [30] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC'03*, volume 2622, pages 138–152, NY, April 2003. Springer-Verlag.
- [31] Erik Ruf. Effective synchronization removal for java. In *PLDI'00*, pages 208–218, 2000.
- [32] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. In *PLDI'07*, pages 78–88, 2007.
- [33] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP'08*, 2008.
- [34] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.
- [35] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, pages 334–345, 2006.
- [36] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, pages 99–117, 2001.
- [37] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *POPL '09*, pages 252–263, 2009.
- [38] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP'04*, pages 519–542, 2004.
- [39] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*, pages 131–144, 2004.
- [40] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

A. The Ξ Function

$$\begin{aligned}
\Xi(P) &\stackrel{\text{def}}{=} \overrightarrow{a_n \mapsto \Xi_{\text{class}}(a_n, P)} \\
&\text{if } P = \overrightarrow{Cls_n} \\
&\quad \text{name}(Cls_i) = a_i, i = 1, \dots, n \\
\Xi_{\text{class}}(a, P) &\stackrel{\text{def}}{=} \langle \mu; \mathcal{F}' \uplus \mathcal{F}; \mathcal{M}' \triangleright \mathcal{M}; \{a\} \cup \mathcal{U}' \rangle \\
&\text{if } \mu \text{ class } a \text{ extends } a' \{ F M \} \in P \\
&\quad \Xi_{\text{class}}(a', P) = \langle \mu'; \mathcal{F}'; \mathcal{M}'; \mathcal{U}' \rangle \\
&\quad \Xi_{f1}(F, P) = \mathcal{F} \\
&\quad \Xi_{m1}(M, P) = \mathcal{M} \\
&\quad \text{overrideOK}(\mathcal{M}', \mathcal{M}) \\
&\quad \mu = \mu' \text{ unless } \mu' = a_{\text{object}} \\
\Xi_{\text{class}}(a_{\text{object}}, P) &\stackrel{\text{def}}{=} \langle \epsilon; \emptyset; \emptyset; \{a_{\text{object}}\} \rangle \\
\Xi_{m1}(a \text{ m}(a' x)\{e\}, P) &\stackrel{\text{def}}{=} \overrightarrow{m \mapsto \forall \alpha, \alpha'. (a @ \alpha \rightarrow a' @ \alpha')} \\
&\quad \alpha, \alpha' \text{ fresh} \\
\Xi_{f1}(\overrightarrow{a f = e}, P) &\stackrel{\text{def}}{=} \overrightarrow{f \mapsto \forall \alpha. a @ \alpha} \\
&\quad \alpha \text{ fresh}
\end{aligned}$$

Figure 14. $\Xi(P) = \mathcal{P}$: Computing Signatures

Fig. 14 defines the Ξ function. $\text{overrideOK}(\mathcal{M}_1, \mathcal{M}_2)$ is defined as for any m such that $\mathcal{M}_1(m) = \forall \alpha_1, \alpha'_1. (a_1 @ \alpha_1 \rightarrow a'_1 @ \alpha'_1)$ and $\mathcal{M}_2(m) = \forall \alpha_2, \alpha'_2. (a_2 @ \alpha_2 \rightarrow a'_2 @ \alpha'_2)$, it holds that $a_1 = a_2$ and $a'_1 = a'_2$. We also do not consider shadowing of fields on the hierarchy (see the $\uplus$ operator in the definition). The two restrictions are consistent with Featherweight Java. Furthermore, we require all class modifiers (such as **task**) are the same for all classes on the same

inheritance hierarchy, unless the class in concern is the special root a_{object} .

B. Additional Reduction Rules

All reduction rules other than in the main text are presented in Fig. 15, except the standard rule for evaluation context reduction and reducing $e; e'$. $\text{droots}(H)$ is defined as the set of roots (represented by object IDs) in forest $\text{accG}(H)$. Function $\text{removeLocks}(H, o)$ is a reductive operation on the forest $\text{accG}(H)$, so that all edges (in-edges and out-edges) for node o are removed. Anything related to σ in the rules are purely used for facilitating the proof.

(R-New)

$$\begin{array}{c}
\Xi(P) = \mathcal{P} \\
\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad o \text{ fresh} \quad H' = H[o \mapsto \langle a; L; \sigma; \emptyset; \emptyset \rangle] \\
\alpha' \text{ fresh} \quad \{\alpha_1, \dots, \alpha_n\} = \{\alpha \mid \mathcal{F}(f) = \forall \alpha. a @ \alpha\} \\
\alpha', \alpha'_1, \dots, \alpha'_n \text{ fresh} \quad \sigma = [\text{self} \mapsto \alpha', \alpha_1 \mapsto \alpha'_1, \dots, \alpha_n \mapsto \alpha'_n] \\
\mu \text{ class } a \{ F M \} \in P \quad F = a_n f_n = e_n \\
\hline
H, \langle o_t; \text{new}_L a \rangle \Rightarrow H', \langle o_t; (o.f := e_1; \dots o.f := e_n; o) \rangle
\end{array}$$

(R-Write)

$$\begin{array}{c}
H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle \\
H' = H[o \mapsto \langle a; L; \sigma; \text{Acc}; Fd[f \mapsto v] \rangle] \\
\hline
H, \langle o_t; o.f := v \rangle \Rightarrow H', \langle o_t; v \rangle
\end{array}$$

(R-Read)

$$\begin{array}{c}
H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle \quad Fd(f) = v \\
\hline
H, \langle o_t; o.f \rangle \Rightarrow H, \langle o_t; v \rangle
\end{array}$$

(R-Cast)

$$\begin{array}{c}
H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle \\
\Xi(P) = \mathcal{P} \quad \mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad a' \in \mathcal{U} \\
\hline
H, \langle o_t; (a')o \rangle \Rightarrow H, \langle o_t; o \rangle
\end{array}$$

(R-TaskEnd)

$$\begin{array}{c}
o_t \in \text{droots}(H) \\
\hline
H, \langle o_t; v \rangle \parallel T \Rightarrow \text{removeLocks}(H, o_t), T
\end{array}$$

(R-SubtaskEnd)

$$\begin{array}{c}
\text{accG}(H) = \langle V; E \rangle \quad (o'_t \xrightarrow{\text{dhosts}} o_t) \in E \\
\hline
H, \langle o_t; v \rangle \parallel \langle o'_t; \mathbf{E}[\text{wait } o_t] \rangle \Rightarrow \text{removeLocks}(H, o_t), \langle o'_t; \mathbf{E}[v] \rangle
\end{array}$$

(R-Commute)

$$H, T \parallel T' \Rightarrow H, T' \parallel T$$

(R-Compose)

$$\begin{array}{c}
H, T \Rightarrow H', T' \\
\hline
H, T \parallel T'' \Rightarrow H', T' \parallel T''
\end{array}$$

(R-Init)

$$\begin{array}{c}
\text{task class } a_{\text{main}} \{ F M \} \in P \quad a_{\text{unit}} m_{\text{main}}(a_{\text{unit}} x)\{e\} \in M \\
H = o_{\text{main}} \mapsto \langle a_{\text{main}}; L_{\text{main}}; [\text{self} \mapsto a_{\text{main}}]; \{o_{\text{main}}\}; \emptyset \rangle \\
H' = o_{\text{unit}} \mapsto \langle a_{\text{unit}}; L_{\text{unit}}; \emptyset; \emptyset; \emptyset \rangle \\
\hline
P \Rightarrow_{\text{init}} \langle H \uplus H'; \langle o_{\text{main}}; e \rangle \rangle
\end{array}$$

Figure 15. Additional Operational Semantics Rules

$$\begin{array}{lll}
ancestors(H, o) & \stackrel{\text{def}}{=} & \{o\} \cup anch(H, o) \\
anch(H, o) & \stackrel{\text{def}}{=} & \emptyset \quad \text{if } accG(H) = \langle V; E \rangle, (o' \xrightarrow{\text{dhosts}} o) \notin E \\
anch(H, o) & \stackrel{\text{def}}{=} & \{o'\} \cup anch(H, o') \quad \text{if } accG(H) = \langle V; E \rangle, (o' \xrightarrow{\text{dhosts}} o) \in E \\
removeLocks(H, o) & \stackrel{\text{def}}{=} & H'[o \mapsto \langle a; L; \sigma; \emptyset; Fd \rangle] \quad \text{if } H(o) = \langle a; L; \sigma; Acc; Fd \rangle \\
& & H' = \biguplus_{H(o_0) = \langle a_0; L_0; \sigma_0; Acc_0; Fd_0 \rangle} (o_0 \mapsto \langle a_0; L_0; \sigma_0; Acc_0 - \{o\}; Fd_0 \rangle)
\end{array}$$

$$\begin{array}{c}
\text{(R-Context)} \\
\frac{H, \langle o_t; e \rangle \Rightarrow H', \langle o_t; e' \rangle}{H, \langle o_t; \mathbf{E}[e] \rangle \Rightarrow H', \langle o_t; \mathbf{E}[e'] \rangle}
\end{array}$$

Figure 16. Auxiliary Definitions for Reduction Rules

S	$::=$	$\langle \mathcal{H}; T \rangle$	<i>configuration signature</i>
$\mathcal{H}$	$::=$	$o \mapsto \langle a; L; \sigma; Acc \rangle$	<i>heap signature</i>
$\mathcal{T}$	$::=$	$\bar{o}$	<i>task signature</i>
γ	$::=$	$\dots \mid \mathbf{iconf} \mapsto S$	<i>extended environment element</i>

$$\begin{array}{ll}
structure(\langle H; T \rangle) & \stackrel{\text{def}}{=} \langle structure(H); structure(T) \rangle \\
structure(H) & \stackrel{\text{def}}{=} \mathcal{H} \\
& \text{if } \mathbf{dom}(H) = \mathbf{dom}(\mathcal{H}) = \{o_1, \dots, o_n\} \\
& \quad \forall i \in \{1, \dots, n\} \ structure(H(o_i)) = \mathcal{H}(o_i) \\
structure(\langle a; L; \sigma; Acc; Fd \rangle) & \stackrel{\text{def}}{=} \langle a; L; \sigma; Acc \rangle \\
structure(T \parallel T') & \stackrel{\text{def}}{=} structure(T) \cup structure(T') \\
structure(\langle o_t; e \rangle) & \stackrel{\text{def}}{=} \{o_t\}
\end{array}$$

Figure 17. Auxiliary Definitions for Proof

$$\begin{array}{c}
\text{(WFT-Config)} \\
\frac{\mathcal{T} \vdash_{\text{wftH}} \mathcal{H} \quad \mathcal{H} \vdash_{\text{wftT}} \mathcal{T}}{\mathcal{P} \vdash_{\text{wft}} \langle \mathcal{H}; T \rangle} \\
\\
\text{(WFT-Heap)} \\
\frac{\mathcal{H}(o) = \langle a; L; \sigma; Acc \rangle \quad \forall o \in \mathbf{dom}(\mathcal{H}) \quad Acc \subseteq \{o_t \mid \langle o_t; o \rangle \text{ in } \mathcal{T}\} \quad \forall \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \in \text{range}(\mathcal{P}). FTV(\mathcal{M}) \cap \mathbf{dom}(\sigma) = \emptyset}{\mathcal{P}, \mathcal{T} \vdash_{\text{wftH}} \mathcal{H}} \\
\\
\text{(WFT-TaskPool)} \\
\frac{\forall o \text{ in } \mathcal{T} \quad \mathcal{H}(o) = \langle a; L; \sigma; Acc \rangle \quad o \in Acc}{\mathcal{H} \vdash_{\text{wftT}} \mathcal{T}}
\end{array}$$

Figure 18. Well-Formedness Definitions

(T-Config)

$$\frac{\Xi(P) = \mathcal{P} \quad \text{structure}(S) = S \quad \mathcal{P} \vdash_{\text{wft}} S \quad \vdash_{\text{global}} P \setminus G \quad S, \mathcal{P} \vdash_{\text{h}} H \setminus \Sigma_{\text{h}} \quad S, \mathcal{P} \vdash_{\text{tp}} T \setminus \Sigma_{\text{tp}} \quad \text{accG}(H) \text{ forest}}{P \vdash_{\text{conf}} \langle H; T \rangle \setminus (\Sigma_{\text{h}} \cup \Sigma_{\text{tp}})}$$

(T-Heap)

$$\frac{\text{dom}(H) = \text{dom}(\mathcal{H}) = \{o_1, \dots, o_n\} \quad \forall i \in \{1, \dots, n\} \quad S, \mathcal{P} \vdash_{\text{hc}} H(o_i) \setminus \Sigma_i}{S, \mathcal{P} \vdash_{\text{h}} H \setminus (\Sigma_1 \cup \dots, \Sigma_n)}$$

(T-HeapCell)

$$\frac{\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \text{dom}(Fd) = \{f_1, \dots, f_n\} \quad [\mathbf{iconf} : S, \mathbf{myC} : a], \mathcal{P} \vdash Fd(f_i) : a_i @ \alpha'_i \setminus \emptyset \quad \mathcal{F}(f_i) = \forall \alpha_i. a_i @ \alpha_i \quad \sigma(\alpha_i) = \alpha'_i \quad \text{dom}(\sigma) \cap \text{range}(\sigma) = \emptyset \quad \sigma(\mathbf{self}) = \alpha}{S, \mathcal{P} \vdash_{\text{hc}} \langle a; L; \sigma; \text{Acc}; Fd \rangle \setminus \{ \langle a; L; \sigma \rangle \leq \alpha \}}$$

(T-TaskPool)

$$\frac{S, \mathcal{P} \vdash_{\text{tp}} T : T \setminus \Sigma \quad S, \mathcal{P} \vdash_{\text{tp}} T' : T' \setminus \Sigma'}{S, \mathcal{P} \vdash_{\text{tp}} T \parallel T' \setminus \Sigma \cup \Sigma'}$$

(T-Task)

$$\frac{[\mathbf{iconf} : S], \mathcal{P} \vdash e : \tau \setminus \Sigma \quad S = \langle \mathcal{H}; T \rangle \quad \mathcal{H}(o) = \langle a; L; \sigma; \text{Acc} \rangle}{S, \mathcal{P} \vdash_{\text{tp}} \langle o; e \rangle \setminus \Sigma[\sigma]}$$

(T-RID)

$$\frac{\Gamma(\mathbf{iconf}) = \langle \mathcal{H}; T \rangle \quad \mathcal{H}(o) = \langle a; L; \sigma; \text{Acc} \rangle \quad \alpha = \sigma(\mathbf{self})}{\Gamma, \mathcal{P} \vdash o : (a @ \alpha) \setminus \emptyset}$$

(T-AuxRead)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a_0 @ \alpha_0) \setminus \Sigma_0 \quad \mathcal{P}(a_0) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \mathcal{F}(f) = \forall \alpha. a @ \alpha}{\Gamma, \mathcal{P} \vdash e.f : a @ \alpha \setminus \Sigma_0 \cup \{ \langle \mathbf{self}; f \rangle \leq \alpha \}}$$

(T-AuxWrite)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a_0 @ \alpha_0) \setminus \Sigma_0 \quad \mathcal{P}(a_0) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad \Gamma, \mathcal{P} \vdash e' : a' @ \alpha' \setminus \Sigma \quad \mathcal{F}(f) = \forall \alpha. a @ \alpha}{\Gamma, \mathcal{P} \vdash e.f : e' : a' @ \alpha' \setminus \Sigma_0 \cup \Sigma \cup \{ \alpha' \leq \alpha \}}$$

(T-Wait)

$$\frac{\Gamma, \mathcal{P} \vdash e : (a @ \alpha) \setminus \Sigma}{\Gamma, \mathcal{P} \vdash \mathbf{wait} e : \mathbf{unit} \setminus \Sigma}$$

Figure 19. Auxiliary Typing Rules

C. Proof

C.1 Auxiliary Definitions for Reduction Rules

Fig. 16 gives rigorous definitions that have been used by reduction rules by omitted due to page limit.

C.2 Auxiliary Definitions for Proof

Fig. 17 gives definitions to data structures that are used to type dynamic data structures. Some auxiliary functions are also defined. They are solely used by the proof. In addition, $\text{droot}(H, o)$ is defined as the root (represented by an object ID) of the tree o is in on forest $\text{accG}(H)$. If the node is an isolated node, then the function returns the ID of itself.

C.3 Auxiliary Typing Rules

Fig. 18 and Fig. 19 define auxiliary rules to be used for the proof.

C.4 Properties About Values

Lemma 1 (Empty Constraints). *If $\Gamma, \mathcal{P} \vdash v : \tau \setminus \Sigma$, then $\Sigma = \emptyset$.*

Proof. Induction on the type derivation and case analysis on the last step. There are only two cases, (T-RID), and (T-Sub). $\square$

Lemma 2 (Unique Type Variable for Values). *If $\Gamma, \mathcal{P} \vdash v : a_1 @ \alpha_1 \setminus \Sigma_1$ and $\Gamma, \mathcal{P} \vdash v : a_2 @ \alpha_2 \setminus \Sigma_2$, then $\alpha_1 = \alpha_2$.*

Proof. Case analysis on v . There are only three cases for the last step leading to the two derivations, (T-RID), and (T-Sub). $\square$

Lemma 3 (Value Substitution). *For any Γ such that*

$$\Gamma, \mathcal{P} \vdash v : \tau' \setminus \Sigma \quad (3.1)$$

$$[\mathbf{myC} \mapsto a, \mathbf{myM} \mapsto m, x \mapsto \tau'], \mathcal{P} \vdash e : \tau \setminus \Sigma' \quad (3.2)$$

then $\Gamma \triangleright [\mathbf{myC} \mapsto a, \mathbf{myM} \mapsto m], \mathcal{P} \vdash e\{v/x\} : \tau \setminus \Sigma \cup \Sigma'$.

Proof. Standard substitution lemma. $\square$

C.5 Properties About σ

Lemma 4 (Context-Sensitive Signature Type Refreshment).

$$\Gamma \triangleright [x \mapsto a_d @ \alpha_d], \mathcal{P} \vdash e : a_r @ \alpha_r \setminus \Sigma \quad (4.1)$$

$$\alpha'_d \notin FTV(\Sigma) \quad (4.2)$$

$$\alpha'_r \notin FTV(\Sigma) \quad (4.3)$$

$\Gamma \triangleright [x \mapsto a_d @ \alpha'_d], \mathcal{P} \vdash e : a_r @ \alpha'_r \setminus \Sigma[\sigma]$ where $\sigma = [\alpha_d \mapsto \alpha'_d, \alpha_r \mapsto \alpha'_r]$.

Proof. Induction on the judgment and case analysis on the last step of the derivation. Here the only interesting rule is (T-Var) as this is where the type variable is freshed. $\square$

Lemma 5 (Type Variable Substitution Over Subsetting). *If $\Sigma_1 \subseteq \Sigma_2$, $\text{range}(\sigma) \cap FTV(\Sigma_2) = \emptyset$, then $\Sigma_1[\sigma] \subseteq \Sigma_2[\sigma]$.*

Proof. Basic property of substitution. $\square$

Lemma 6 (Context-Sensitive Substitution).

$$\vdash_{\text{global}} P \setminus G \quad (6.1)$$

$$\Xi(P) = \mathcal{P} \quad (6.2)$$

$$\mu \text{ class } a \{F \ M\} \in P \quad (6.3)$$

$$t \ m(t \ x)\{e\} \in M \quad (6.4)$$

$$\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad (6.5)$$

$$\mathcal{M}(m) = a_d @ \alpha_d \mapsto a_r @ \alpha_r \quad (6.6)$$

$$\Gamma, \mathcal{P} \vdash v : a_d @ \alpha_{d0} \setminus \Sigma_v \quad (6.7)$$

$$\alpha'_d, \alpha'_r \text{ fresh} \quad (6.8)$$

then $\Gamma \triangleright [\mathbf{myC} : a, \mathbf{myM} : m], \mathcal{P} \vdash e\{v/x\} : a_r @ \alpha_{r0} \setminus \Sigma \cup \Sigma_v$ and $\Sigma = G(\langle a; m \rangle)[\alpha_d \mapsto \alpha'_d, \alpha_r \mapsto \alpha'_r] \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\}$

Proof. Given (6.1), (6.2), (6.3), (6.4), (6.5), (6.6), (T-Global), (T-Class), we know

$$[\mathbf{myC} \mapsto a, \mathbf{myM} \mapsto m, x \mapsto a_d @ \alpha_d], \mathcal{P} \vdash e : a_r @ \alpha_r \setminus \Sigma_m \quad (6.9)$$

$$\Sigma_m = G(\langle a; m \rangle) \quad (6.10)$$

According to Lem. 4, (6.9), (6.8), we know

$$[\mathbf{myC} \mapsto a, \mathbf{myM} \mapsto m, x \mapsto a_d @ \alpha'_d], \mathcal{P} \vdash e : a_r @ \alpha'_r \setminus \Sigma'_m \quad (6.11)$$

$$\Sigma'_m = \Sigma_m[\alpha_d \mapsto \alpha'_d, \alpha_r \mapsto \alpha'_r] \quad (6.12)$$

According to Lem. 3, (6.7), (6.11), we know

$$\Gamma \triangleright [\mathbf{myC} \mapsto a, \mathbf{myM} \mapsto m], \mathcal{P} \vdash e\{v/x\} : a_r @ \alpha'_r \setminus \Sigma'_m \cup \Sigma_v \quad (6.13)$$

Thus let $\Sigma = \Sigma'_m$. According to Lem. 5, (6.10), (6.12), (6.8), we know

$$\Sigma'_m = G(\langle a; m \rangle)[\alpha_d \mapsto \alpha'_d, \alpha_r \mapsto \alpha'_r] \quad (6.14)$$

$\square$

C.6 Definition and Properties of $\xrightarrow{G}$

Definition 7. $\Sigma \xrightarrow{G} \Sigma'$ holds iff $G \triangleright [\langle a_{\text{main}}; m_{\text{main}} \rangle \mapsto \Sigma] \vdash_{\text{ocl}} \Sigma'$.

Lemma 7. *If $\Sigma \xrightarrow{G} \Sigma'$, then $FTV(\Sigma') = \{\alpha \mid \alpha \in FTV(\Sigma) \text{ or } \alpha \text{ fresh}\}$.*

Proof. Case analysis on the (OCL-) rules. $\square$

Lemma 8 ($\xrightarrow{G}$ Subsetting). *If $\Sigma_1 \xrightarrow{G} \Sigma_2$, $\Sigma_3 \subseteq \Sigma_2$, then $\Sigma_1 \xrightarrow{G} \Sigma_3$.*

Proof. Directly follows the definition of $\xrightarrow{G}$, (OCL-Main), and (OCL-Subset). $\square$

Lemma 9 ($\xrightarrow{G}$ Padding). *If $\Sigma_1 \xrightarrow{G} \Sigma_2$, then $\Sigma_1 \cup \Sigma \xrightarrow{G} \Sigma_2 \cup \Sigma$.*

Proof. Directly follows the definition of $\xrightarrow{G}$, (OCL-Main), (OCL-Subset), and (OCL-Union). $\square$

C.7 Properties of Γ

Definition 8. $\Gamma \leq_{\text{acc}} \Gamma'$ is defined as Γ' is the same as in Γ , except for the H part, the Acc for Γ' is superset of its counterpart in Γ .

Lemma 10 (Acc Strengthening). If $\Gamma, \mathcal{P} \vdash e : \tau \setminus \Sigma$, and $\Gamma \leq_{\text{acc}} \Gamma'$, then $\Gamma', \mathcal{P} \vdash e : \tau \setminus \Sigma$

Proof. Induction on the type derivation and case analysis on the last step. The only interesting rules are (T-SyncM), (T-Capture), (T-AsyncM), (T-Subtask). $\square$

C.8 Dynamic Isolation

Definition 9 (Stack Access inside Tasks/Subtasks). Predicate $\text{stack}(H, T)$ iff for any $\langle o; e \rangle$ in T and $\text{acc}G(H) = \langle V; E \rangle$, $(o \xrightarrow{\text{dhosts}} o') \in E$ and o' must be an ordinary object. In addition, for all ancestors of o , the same holds as well.

Lemma 11 (Stack Access Preservation Over Reduction). Given $S_i = \langle H_i; T_i \rangle$ for $i = 1, 2$,

$$S_1 \Rightarrow S_2 \quad (11.1)$$

$$\text{stack}(H_1, T_1) \quad (11.2)$$

then $\text{stack}(H_2, T_2)$.

Proof. Simple case analysis on reduction rules. $\square$

Lemma 12 (Stack Access). If $P \Rightarrow_* \langle H; T \rangle$, then $\text{stack}(H, T)$.

Proof. Directly follows Lem. 11, together with the initialization holds trivially. $\square$

Definition 10 (Zero In-Edge Corresponds Zero Out-Edge). $\text{zeroInzeroOut}(H)$ iff $\text{acc}G(H) = \langle V; E \rangle$, $\{o_1 \mid (o_1 \xrightarrow{\text{dhosts}} o) \in E\} = \emptyset$ implies $\{o_2 \mid (o \xrightarrow{\text{dhosts}} o_2) \in E\} = \emptyset$.

Lemma 13 (Zero Edge Correspondence Over Reduction). Given $S_1 \Rightarrow S_2$, where $S_i = \langle H_i; T_i \rangle$ for $i = 1, 2$, $\text{zeroInzeroOut}(H_1)$ and $\text{stack}(H_1, T_1)$, then it must hold that $\text{zeroInzeroOut}(H_2)$.

Proof. Induction on the length of the execution sequence, and case analysis over the reduction rule being applied to the last step of reduction $S_1 \Rightarrow S_2$.

For **Cases** (R-Self), (R-Read), (R-Cast), (R-AsyncM), (R-Subtask), (R-Commute), note that $H_2 = H_1$ so the conclusion holds trivially. For **Case** (R-New), note that the only change $\text{acc}G(H_2)$ has relative to $\text{acc}G(H_1)$ is to add one extra isolated node to the forest, hence the result is also a forest. For **Case** (R-Write), note that the Acc component of the heap does not change, and hence the graph in concern remain unchanged, so the conclusion holds trivially.

For **Cases** (R-Context), (R-Compose), the conclusion immediately follows induction hypothesis.

For **Case** (R-Capture) and **Case** (R-SyncM), the reduction in both cases attempts to add one more edge to forest $\text{acc}G(H_1)$. Graph-theoretically, adding edges to a forest only leads to fewer nodes with zero in-edges, and thus the lemma holds trivially.

For **Cases** (R-TaskEnd), (R-SubtaskEnd), note that the change from graph $\text{acc}G(H_1)$ to graph $\text{acc}G(H_2)$ is to eliminate all the in-edges and out-edges of o_t where $T_1 = \langle o_t; o_t; v \rangle \parallel T'_1$ for some T'_1 . Thus, for the node o_t , the property of zeroInzeroOut holds trivially. Such an operation would have also reduced the degree of in-edges that o_t is a parent of. Not to lose generality, let us pick one of such node, and let it be o'_t . Hence we know given $\text{acc}G(H_1) = \langle V_1; E_1 \rangle$, $(o_t \xrightarrow{\text{dhosts}} o'_t) \in E_1$. Note that the form of T_1 qualifies the pre-condition of Lem. 11, and we also know that $\text{stack}(H_1, T_1)$ holds. Thus by that lemma we know if there exists some o''_t such that $(o'_t \xrightarrow{\text{dhosts}} o''_t) \in E_1$ then $o'_t = o''_t$. What this says is all the out-edges of o'_t , if any, will be cyclic self-edges. So suppose by performing the reduction we have indeed reduced the degree of in-edges of o'_t to zero, that would mean the self-edges are removed as well, and hence all the out-edges of o'_t would be reduced to zero as well. Hence predicate $\text{zeroInzeroOut}(H_2)$ holds. $\square$

Lemma 14 (Isolation Preservation Over Reduction). Given $S_i = \langle H_i; T_i \rangle$ for $i = 1, 2$,

$$S_1 \Rightarrow S_2 \quad (14.1)$$

$$\text{acc}G(H_1) \text{ is a forest} \quad (14.2)$$

$$\text{zeroInzeroOut}(H_1) \quad (14.3)$$

then $\text{acc}G(H_2)$ is a forest.

Proof. Induction on the length of the execution sequence, and case analysis over the reduction rule being applied to the last step of reduction $S_1 \Rightarrow S_2$.

For **Cases** (R-Self), (R-Read), (R-Cast), (R-AsyncM), (R-Subtask), (R-Commute), note that $H_2 = H_1$ so the conclusion holds trivially. For **Case** (R-New), note that the only change $\text{acc}G(H_2)$ has relative to $\text{acc}G(H_1)$ is to add one extra isolated node to the forest, hence the result is also a forest. For **Case** (R-Write), note that the Acc component of the heap does not change, and hence the graph in concern remain unchanged, so the conclusion holds trivially.

For **Cases** (R-Context), (R-Compose), the conclusion immediately follows induction hypothesis.

For **Case (R-Capture)** and **Case (R-SyncM)**, the reduction in both cases attempts to add one more edge to forest $accG(H_1)$, namely $o_t \xrightarrow{dhosts} o$, where $T_1 = \langle o_t; o'; e_1 \rangle$ and $e_1 = o.m(v)$ for **Case (R-SyncM)**, and $e_1 = \text{capture } o$ for **Case (R-Capture)**. Starting from now, we only discuss **Case (R-SyncM)**, since **Case (R-Capture)** is the same. In this case, there are only several possibilities where $H_1(o) = \langle a; L; \sigma; Acc; Fd \rangle$:

- If $Acc = \emptyset$, we know by the definition of $accG$ that an edge $o' \xrightarrow{dhosts} o$ does not exist for any o' in forest $accG(H_1)$. By (14.3) and the definition of $zeroInzeroOut$, we know an edge $o \xrightarrow{dhosts} o'_0$ does not exist for any o' in forest $accG(H_1)$ either. Thus, o is an isolated node. Adding edge $o_t \xrightarrow{dhosts} o$ will still result in a forest by the basic property of forests.
- If $Acc \neq \emptyset$ and $o_t \in Acc$, then the reduction does not make any change to the heap according to (R-SyncM), so the conclusion holds trivially.
- If $Acc \neq \emptyset$ and $o_t \notin Acc$, in this case by (R-SyncM) we know pre-condition $Acc \subseteq anc(H_1, o_t)$ holds. What it says for any object ID in Acc , it is an ancestor of o_t . Let any such ancestor to be o'_t . We know by the definition of $accG$ that an edge $o'_t \xrightarrow{dhosts} o$ is already in the graph of $accG(H_1)$. Since o'_t is ancestor of o_t , we know there exists a $o_{t1}, \dots, o_{tn}$ such that edges $o'_t \xrightarrow{dhosts} o_{t1}$, $o_{t1} \xrightarrow{dhosts} o_{t2}, \dots, o_{tn} \xrightarrow{dhosts} o_t$ exist. Thus, we know that even if edge $o_t \xrightarrow{dhosts} o$ is added to the original graph, the graph is still a forest since it at most will be a redundant edge.

For **Cases (R-TaskEnd)**, **(R-SubtaskEnd)**, note that the change from graph $accG(H_1)$ to graph $accG(H_2)$ is reductive in terms of edges. Graph-theoretically, removing edges from a forest results in a (more sparse) forest. $\square$

Theorem 5 (Isolation Preservation). *See Thm. 1.*

Proof. It can be trivially seen that (R-Init) leads to a forest. The rest of the proof is trivial by Lem. 14. $\square$

C.9 Static Isolation

Lemma 15 (Locality Preservation Reduction). *Given*

$$\vdash_{\text{global}} P \setminus G \quad (15.1)$$

$$S_1 \xrightarrow{e} S_2 \quad (15.2)$$

$$S_i = \langle H_i; T_i \rangle \text{ for } i = 1, 2 \quad (15.3)$$

then

- either $droot(H_2, o) = droot(H_1, o)$
- or $droot(H_2, o) = o$

for all $o \in \text{dom}(H_1)$ such that $H_1(o) = \langle a; L; \sigma; Acc_1; Fd_1 \rangle$ and $local(P, L)$.

Proof. Induction on the reduction sequence defined in (15.2) and case analysis on the last step. If the reduction is an instance of (R-TaskEnd) or (R-SubtaskEnd), note that the reduction is reductive in terms of the Acc component for all objects on the heap. In other words, the edges in $accG(H_1)$ are reductive. By Thm. 1, we know $accG(H_1)$ forms a forest. Graph theoretically, we know there are only two possibilities for any node in the graph when some edges are removed: either they are still connected (and thus have the same root), or they are isolated. These two cases are $droot(H_2, o) = droot(H_1, o)$ and $droot(H_2, o) = o$ respectively. The conclusion thus holds.

If the reduction is an instance of (R-Capture), By (T-Capture), the only possibility of change on the graph of $accG(H_1)$ is on o_0 where the redex is $\text{capture } o_0$. If this is the case, by (15.1), we know $\text{cap}(\alpha, \text{self}) \leq \alpha'$ constraint would be put into the constraint set, which contradicts the definition of $local(P, L)$, since we know given $\Xi(P) = \mathcal{P}$, for all $G \vdash_{\text{oc1}} \langle a_0; L; \sigma_0 \rangle \leq \alpha_0$, and $\mathcal{P}(a_0) = \langle \epsilon; \mathcal{F}_0; \mathcal{M}_0; \mathcal{U}_0 \rangle$, it never exists α', α'' such that $G \vdash_{\text{oc1}} \text{cap}(\alpha_0, \alpha'') \leq \alpha'$.

If the reduction is an instance of (R-SyncM), thus we know the redex $e_1 = o_0.m(v)$. If this is the case, by (15.1), and by (T-SyncM), we know $\alpha'_0 \xrightarrow{hosts} \alpha_0$ is part of the constraint set for some α'_0 and α_0 where α_0 represents o_0 . By (15.1), we also know $isolatedTasks(G, \mathcal{P})$. Hence, we know if $isAccessed(G, \mathcal{P}, \alpha_0)$, then $\forall \alpha_1. \alpha_2. \{\alpha_1, \alpha_2\} \in \text{cutVertices}(G, \mathcal{P}, \alpha_0)$ implies either $accesses(\alpha_1, \alpha_2)$ or $accesses(\alpha_2, \alpha_1)$. It is easy to verify $isAccessed(G, \mathcal{P}, \alpha_0)$ does hold in this case, so we know $\forall \alpha_1. \alpha_2. \{\alpha_1, \alpha_2\} \in \text{cutVertices}(G, \mathcal{P}, \alpha_0)$ implies either $accesses(\alpha_1, \alpha_2)$ or $accesses(\alpha_2, \alpha_1)$. What this suggests is that all the cut vertices of α_0 will be on the same path going to the same static variable root, say α_{root} . This would have given us $droot(H_2, o) = droot(H_1, o)$ except that when the static graph is instantiated to the dynamic one, α_{root} might represent more than one objects. Should that happen, the equation above would not hold. This luckily is not true because given (15.1), we also know $tunnelFree(G, \mathcal{P})$. The latter can be expanded to three cases. Not to lose generality, let us assume there are indeed two o_{root1} and o_{root2} that both map to α_{root} , and $droot(H_2, o) = o_{\text{root2}}$ and $droot(H_1, o) = o_{\text{root1}}$. We now show the three cases: 1) o_{root1} and o_{root2} acquire a reference of o when both of them as tasks are instantiated. Note that in this case, the only possibility is for them to pass o in as an argument when (R-AsyncM) happened to create o_{root1} and o_{root2} . We know in this context, a member in either $instHost(G, \alpha_0)$ or $capHost(G, \alpha_0)$ cannot be a descendent of root α_{root} . Contradiction with the definition of $extFree$. 2) o_{root1} and o_{root2} acquire a reference of o when both of them are in the middle of their execution. We now need to consider how could two tasks receive a data back in the middle of their

execution. The **new** expression certainly would not work, since in that case what is returned is not the shared data o . The asynchronous invocation expression does not have a return value. The only interesting cases are both acquiring o through a path that involves a number of synchronous invocations, and end at a subtasking operation, where o is stored. This is precisely what is outlawed by *dissemFree*. 3) o_{root1} and o_{root2} acquire a reference of o when o_{root1} is in the middle of their execution, and o_{root2} as a task is being created. Or o_{root2} and o_{root1} acquire a reference of o when o_{root2} is in the middle of their execution, and o_{root1} as a task is being created. Not to lose generality, we only need to prove one case (the former). Note that in this case, it must have been that o_{root1} , through a chain of calls, eventually sends an asynchronous message to o_{root2} to start a task, passing o as an argument. This in reality wouldn't happen as it will be captured by *cordFree*.

All the other reductions either do not change the heap, or only change the parts that are not related to the concern here (such as (R-Write) updates the *Fd* component of the heap entry, or (R-New) adds a new isolated node in the graph). The conclusion holds trivially for those cases. $\square$

Lemma 16 (Hierarchical Elements in *Acc*). *If $P \Rightarrow^* \langle H; T \rangle$ and for any o such that $H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle$, then either $\text{Acc} = \emptyset$, or all elements in Acc are on a tree path in $\text{accG}(H)$. (In other words, if we consider the partial order on each tree in $\text{accG}(H)$ as transitive, then all elements in Acc form a total order as a sub-graph of a specific tree in $\text{accG}(H)$.)*

Proof. A simple induction on length of execution sequences, and the only interesting cases are (R-SyncM) and (R-Capture). Note that the pre-conditions on both rules $\text{Acc} \subseteq \text{anc}(H, o_t)$ successfully enforces this property. It says that Acc must be the ancestors of o_t . Not to lose generality, let us assume the tree path from the root to o_t is $[o_1, \dots, o_n, o_t]$. Thus the set Acc must be a subset of $\{o_1, \dots, o_n, o_t\}$. Now it is trivial to see $\text{Acc} \cup \{o_t\}$ also forms a tree path, which is trivially o_t is $[o_1, \dots, o_n, o_t]$. Hence the invariant holds over the reductions of (R-SyncM) and (R-Capture). (R-TaskEnd) and (R-SubTaskEnd) only reduces Acc for all nodes involved (or set it to $\emptyset$). Reducing nodes on a tree path does not change the property: the rest of the nodes will trivially be on the same tree path (only sparser apart from each other). $\square$

Theorem 6 (Static Isolation). *See Thm. 3 in Sec. 4.*

Proof. For any heap cell with a label L where $\text{local}(P, L)$ does not hold, we know according to the definition of $\Rightarrow^*$, the definition of *tryLock* remains the same for any reduction related to these cells, so the proof of isolation is identical to Thm. 5. The only interesting case to prove then

is the heap cells where $\text{local}(P, L)$ holds. In this case, we know if we can prove for such an o such that $H(o) = \langle a; L; \sigma; \text{Acc}; Fd \rangle$, the reduction would be the same as the case of Thm. 5 iff we can prove $\text{Acc} \subseteq \text{ancestors}(H, o_t)$. If $\text{Acc} = \emptyset$, then the condition holds trivially. Otherwise, let $\text{Acc} = \{o_1, o_2, \dots, o_n\}$. According to Lem. 16, we know $o_1, o_2, \dots, o_n$ in fact forms a path on the same tree in the forest $\text{accG}(H)$. Thus, if o_t is a descendant of o_n (or being o_n itself), then we know the pre-condition holds trivially according to the definition of *ancestor*—the ancestors of o_t will subsume Acc . If not, not to lose generality we know there are only several cases:

- **Case 1:** o_t is on the tree path of $\{o_1, o_2, \dots, o_n\}$, but it is not o_n . Not to lose generality, let us assume it is o_k , where $1 \leq k \leq n-1$, or it is between o_k and o_{k+1} . By Lem. 12, we know o_t can at most have children nodes on $\text{acc}(H)$, but will never have grandchildren. Thus the only possibility would be to have o_{n-1} being o_t , and o_n being its direct child. Even so wouldn't work, since the same lemma says all the children nodes of o_t must be ordinary objects, and by the basic properties of Acc we know they are always task objects and subtask objects. Contradiction.
- **Case 2:** o_t is not on the tree path of $\{o_1, o_2, \dots, o_n\}$, but in the same tree, such as o_t is higher up on the tree, or being a part of a sub-tree where its ancestors do not completely cover $\{o_1, o_2, \dots, o_n\}$. It is not possible for o_t to be higher up on the tree, since that way, it would have children which are not ordinary objects. Contradiction with Lem. 12. It is not possible for o_t to be part of a sub-tree where its ancestors do not completely cover $\{o_1, o_2, \dots, o_n\}$. This is because, according to Lem. 12, all ancestors of o_t can not have non-ordinary children either. There is no place on the graph to place those nodes in $\{o_1, o_2, \dots, o_n\}$ but are not ancestors of o_t .
- **Case 3:** o_t does not belong to the same tree at all. This would not happen directly follows Lem. 15.

Hence the conclusion holds after the proof-by-contradiction of three cases. $\square$

C.10 Subject Reduction

Lemma 17 (Structural Invariant over Reduction). *If $P \vdash S_1 \setminus \Sigma_1, S_1 \Rightarrow S_2$, then*

$$\text{structure}(S_2) = S_2 \quad (17.1)$$

$$\mathcal{P} \vdash_{\text{wft}} S_2 \quad (17.2)$$

Proof. Straightforward case analysis. $\square$

Lemma 18 (Well-Typed Heap over Reduction). *If*

$$\vdash_{\text{global}} P \setminus G \quad (18.1)$$

$$\Xi(P) = \mathcal{P} \quad (18.2)$$

$$S_1 \Rightarrow S_2 \quad (18.3)$$

$$S_i = \langle H_i; T_i \rangle, \text{ for } i = 1, 2 \quad (18.4)$$

$$\text{structure}(S_i) = \mathbf{S}_i, \text{ for } i = 1, 2 \quad (18.5)$$

$$\text{structure}(H_i) = \mathcal{H}_i, \text{ for } i = 1, 2 \quad (18.6)$$

$$\mathbf{S}_1, \mathcal{P} \vdash_{\text{h}} H_1 \setminus \Sigma_{\text{h1}} \quad (18.7)$$

$$o \in \text{dom}(\mathcal{H}_1) \quad (18.8)$$

then

$$\mathbf{S}_2, \mathcal{P} \vdash_{\text{h}} H_2 \setminus \Sigma_{\text{h2}} \quad (18.9)$$

$$\Sigma_{\text{h1}} \xrightarrow{G} \Sigma_{\text{h2}} \quad (18.10)$$

$$\mathcal{H}_i(o) = \langle \mathbf{a}; \mathbf{L}; \sigma; \text{Acc}_i \rangle, \text{ for } i = 1, 2 \quad (18.11)$$

Proof. Straightforward case analysis. $\square$

Lemma 19 (Well-Typed Single Task Reduction). *If*

$$\vdash_{\text{global}} P \setminus G \quad (19.\text{Known1})$$

$$\Xi(P) = \mathcal{P} \quad (19.\text{Known2})$$

$$S_1 \Rightarrow S_2 \quad (19.\text{Known3})$$

$$S_i = \langle H_i; T_i \rangle, \text{ for } i = 1, 2 \quad (19.\text{Known4})$$

$$T_i = \langle o_{\text{t}}; o; e_i \rangle, \text{ for } i = 1, 2 \quad (19.\text{Known5})$$

$$\text{structure}(S_i) = \mathbf{S}_i, \text{ for } i = 1, 2 \quad (19.\text{Known6})$$

$$\mathbf{S}_i, \mathcal{P} \vdash_{\text{h}} H_i \setminus \Sigma_{\text{h1}}, \text{ for } i = 1, 2 \quad (19.\text{Known7})$$

$$\text{structure}(H_i) = \mathcal{H}_i, \text{ for } i = 1, 2 \quad (19.\text{Known8})$$

$$\mathcal{H}_i(o) = \langle \mathbf{a}; \mathbf{L}; \sigma; \text{Acc}_i \rangle, \text{ for } i = 1, 2 \quad (19.\text{Known9})$$

$$[\mathbf{iconf} : \mathbf{S}_1, \mathbf{myC} : \mathbf{a}], \mathcal{P} \vdash e_1 : \tau_1 \setminus \Sigma_1 \quad (19.\text{Known10})$$

then

$$[\mathbf{iconf} : \mathbf{S}_2, \mathbf{myC} : \mathbf{a}], \mathcal{P} \vdash e_2 : \tau_2 \setminus \Sigma_2 \quad (19.\text{Goal1})$$

$$\Sigma_{\text{h1}} \cup \Sigma_1 \xrightarrow{G} \Sigma_{02} \quad (19.\text{Goal2})$$

$$\text{range}(\sigma_{\text{fresh}}) \cap FTV(\Sigma_{02} \cup \Sigma_1) = \emptyset \quad (19.\text{Goal3})$$

$$\text{for all } \alpha_0 \in (\text{dom}(\sigma_{\text{fresh}}) \cap \text{dom}(\sigma)). \sigma_{\text{fresh}}(\alpha_0) = \sigma(\alpha_0) \quad (19.\text{Goal4})$$

$$\Sigma_2 = \Sigma_{02}[\sigma_{\text{fresh}}] \quad (19.\text{Goal5})$$

$$\tau_2 = \tau_1[\sigma_{\text{fresh}}] \quad (19.\text{Goal6})$$

Proof. Induction on the length of the reduction in (19.Known3), and case analysis on the last step of the reduction rule being used:

Case (R-New): By (R-New) and (19.Known2), we know

$$e_1 = \mathbf{new}_{\mathbf{L}'} \mathbf{a}'$$

$$\mathcal{P}(\mathbf{a}') = \langle \mu'; \mathcal{F}'; \mathcal{M}'; \mathcal{U}' \rangle$$

$$o' \text{ fresh}$$

$$Fd' = \biguplus_{f \in \text{dom}(\mathcal{F}')} (f \mapsto \mathbf{null})$$

$$H_2 = H_1[o' \mapsto \langle \mathbf{a}'; \mathbf{L}'; \sigma'; \emptyset; Fd' \rangle]$$

$$\alpha' \text{ fresh}$$

$$\{\alpha_1, \dots, \alpha_n\} = \{\alpha_0 \mid \mathcal{F}'(f) = \forall \alpha_0. \mathbf{a}_0 @ \alpha_0\}$$

$$\alpha', \alpha'_1, \dots, \alpha'_n \text{ fresh}$$

$$\sigma' = [\mathbf{self} \mapsto \alpha', \alpha_1 \mapsto \alpha'_1, \dots, \alpha_n \mapsto \alpha'_n]$$

$$N_2 = N_1$$

$$e_2 = o'$$

By (T-New), (19.Known10), and the definition of e_1 above, we know

$$\tau_1 = \mathbf{a}' @ \alpha'$$

$$\Sigma_1 = \{\langle \mathbf{a}'; \mathbf{L}'; \sigma' \rangle \leq \alpha'\}$$

Thus by (19.Known4), (19.Known5), (19.Known6), and the definition of *structure*, we know

$$\mathbf{S}_1 = \langle \mathcal{H}_1; N_1; \mathcal{T}_1 \rangle$$

$$\mathbf{S}_2 = \mathbf{S}_1[o' \mapsto \langle \mathbf{a}'; \mathbf{L}'; \sigma'; \emptyset \rangle]$$

With the definition of $\mathbf{S}_2$ above, and (T-RID), we know

$$[\mathbf{iconf} : \mathbf{S}_2, \mathbf{myC} : \mathbf{a}], \mathcal{P} \vdash o' : \mathbf{a}' @ \alpha' \setminus \emptyset \quad (19.1)$$

Let $\Sigma_2 = \emptyset$ and $\tau_2 = \mathbf{a}' @ \alpha'$. Hence (19.1) is identical to (19.Goal1). Let $\sigma_{\text{fresh}} = \emptyset$ and let $\Sigma_{02} = \emptyset$. Thus (19.Goal3), (19.Goal4), (19.Goal5), (19.Goal6) hold trivially. (19.Goal2) also holds trivially because of Lem. 8.

Case (R-Cast): By (R-Cast), we know $e_1 = (\mathbf{a}')v$ and $e_2 = v$. We also know $H_2 = H_1$, $N_2 = N_1$. Thus by (19.Known4), (19.Known5), (19.Known6), and the definition of *structure*, we know $\mathbf{S}_2 = \mathbf{S}_1$. By the same reduction rule, together with (19.Known2), we also know

$$H_1(v) = \langle \mathbf{a}_v; \mathbf{L}_v; \sigma_v; \text{Acc}_v; Fd_v \rangle \quad (19.2)$$

$$\mathcal{P}(\mathbf{a}_v) = \langle \mu_v; \mathcal{F}_v; \mathcal{M}_v; \mathcal{U}_v \rangle \quad (19.3)$$

$$\mathbf{a}' \in \mathcal{U}_v \quad (19.4)$$

By (19.Known1), (T-Heap), (T-HeapCell), we know

$$\sigma_v(\mathbf{self}) = \alpha_v \quad (19.5)$$

By the definition of *structure* and (19.2), we know

$$\mathcal{H}_1(v) = \langle \mathbf{a}_v; \mathbf{L}_v; \sigma_v; \text{Acc}_v \rangle \quad (19.6)$$

By (T-RID), (19.6), (19.5), we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a_v @ \alpha_v) \setminus \emptyset \quad (19.7)$$

By (T-Cast), (19.Known10), and the definition of e_1 above, we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a'_v @ \alpha'_v) \setminus \Sigma'_v \quad (19.8)$$

$$\mathcal{P}(a') = \langle \mu_v; \mathcal{F}'; \mathcal{M}'; \mathcal{U}' \rangle \quad (19.9)$$

$$a'_v \in \mathcal{U}' \text{ or } a' \in \mathcal{U}_v \quad (19.10)$$

$$\Sigma_1 = \Sigma'_v$$

$$\tau_1 = a' @ \alpha'_v$$

By Lem. 1 and (19.8), we know $\Sigma'_v = \emptyset$. By Lem. 2, (19.7), (19.8), we know $\alpha_v = \alpha'_v$. Thus (19.7) can be rewritten as

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a_v @ \alpha'_v) \setminus \emptyset \quad (19.11)$$

By (T-Sub), (19.11), (19.3), (19.4), we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a' @ \alpha'_v) \setminus \emptyset \quad (19.12)$$

Let $\Sigma_2 = \emptyset$ and $\tau_2 = a' @ \alpha'_v$. Earlier we have shown $S_2 = S_1$, hence (19.12) is identical to (19.Goal1). Let $\sigma_{\text{fresh}} = \emptyset$ and let $\Sigma_{02} = \emptyset$. (19.Goal2) also holds trivially because of Lem. 8. (19.Goal3), (19.Goal4), (19.Goal5), (19.Goal6) hold trivially.

Case (R-Capture): In this case we know $e_1 = \mathbf{capture } v$ and $e_2 = v$. We also know

$$H_1(v) = \langle a_v; L_v; \sigma_v; Acc_v; Fd_v \rangle \quad (19.13)$$

$$Acc_v \subseteq anc(o_t) \quad (19.14)$$

$$H_2 = H_1[v \mapsto \langle a_v; L_v; \sigma_v; Acc_v \cup \{o_t\}; Fd_v \rangle] \quad (19.15)$$

$$N_2 = N_1 \quad (19.16)$$

Thus by (19.Known4), (19.Known5), (19.Known6), the definition of *structure*, and Def. 8, we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a] \leq_{\text{acc}} [\mathbf{iconf} : S_2, \mathbf{myC} : a] \quad (19.17)$$

By (19.Known1), (T-Heap), (T-HeapCell), we know

$$\sigma_v(\mathbf{self}) = \alpha_v \quad (19.18)$$

By the definition of *structure* and (19.17), we know

$$\mathcal{H}_1(v) = \langle a_v; L_v; \sigma_v; Acc_v \rangle \quad (19.19)$$

By (T-RID), (19.19), (19.18), we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a_v @ \alpha_v) \setminus \emptyset \quad (19.20)$$

By (T-Capture), (19.Known10), we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : (a'_v @ \alpha'_v) \setminus \Sigma'_v \quad (19.21)$$

$$\alpha' \text{ fresh} \quad (19.22)$$

$$\mathcal{P}(a'_v) = \langle \epsilon; \mathcal{F}'_v; \mathcal{M}'_v; \mathcal{U}'_v \rangle \quad (19.23)$$

$$\tau_1 = a'_v @ \alpha' \quad (19.24)$$

$$\Sigma_1 = \Sigma'_v \cup \{\mathbf{cap}(\alpha'_v, \mathbf{self}) \leq \alpha', \mathbf{self} \xrightarrow{\text{hosts}} \alpha'\} \quad (19.25)$$

By Lem. 1 and (19.21), we know $\Sigma'_v = \emptyset$. By Lem. 2, (19.20), (19.21), we know $\alpha_v = \alpha'_v$. With all these, together with Lem. 10, (19.17), (19.21), we know

$$[\mathbf{iconf} : S_2, \mathbf{myC} : a], \mathcal{P} \vdash v : (a'_v @ \alpha_v) \setminus \emptyset \quad (19.26)$$

Let $\Sigma_2 = \emptyset$ and $\tau_2 = a'_v @ \alpha_v$. Hence (19.26) is identical to (19.Goal1). Let $\sigma_{\text{fresh}} = [\alpha' \mapsto \alpha_v]$ and let $\Sigma_{02} = \emptyset$. (19.Goal2) also holds trivially because of Lem. 8. (19.Goal5), (19.Goal6) hold trivially. To prove (19.Goal3), note that $\text{range}(\sigma_{\text{fresh}}) = \{\alpha_v\}$. $FTV(\Sigma_{02} \cup \Sigma_1) = FTV(\Sigma_1) = \{\alpha_v, \mathbf{self}, \alpha'\}$. (19.Goal4) holds trivially since α' is fresh and $\text{dom}(\sigma_{\text{fresh}}) \cap \text{dom}(\sigma) = \emptyset$.

Case (R-Read): given (19.Known8) and (19.Known9), and the definition of *structure*, $H_1(o) = \langle a; L; \sigma; Acc; Fd_1 \rangle$ for some Fd_1 . Thus by (R-Read), we know $e_1 = f$ and $e_2 = Fd_1(f)$. We also know $H_2 = H_1$, $N_2 = N_1$. Thus by (19.Known4), (19.Known5), (19.Known6), and the definition of *structure*, we know $S_2 = S_1$. By (T-Read), (19.Known10), and the definition of e_1 above, we know

$$\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad (19.27)$$

$$\mathcal{F}(f) = \forall \alpha_f. a @ \alpha_f \quad (19.28)$$

$$\tau_1 = a @ \alpha_f \quad (19.29)$$

$$\Sigma_1 = \{\langle \mathbf{self}; f \rangle \leq \alpha_f\} \quad (19.30)$$

By (19.Known1), (T-Heap), (T-HeapCell), (19.Known7), (19.27), (19.28), we know

$$\sigma(\mathbf{self}) = \alpha \quad (19.31)$$

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash Fd(f) : a @ \alpha'_f \setminus \emptyset \quad (19.32)$$

$$\sigma(\alpha_f) = \alpha'_f \quad (19.33)$$

Earlier we have proven $S_2 = S_1$. Let $\Sigma_2 = \emptyset$ and $\tau_2 = a @ \alpha'_f$. Hence (19.32) is identical to (19.Goal1). Let $\sigma_{\text{fresh}} = [\alpha_f \mapsto \alpha'_f]$ and let $\Sigma_{02} = \emptyset$. Thus (19.Goal5), (19.Goal6) hold trivially. (19.Goal2) also holds trivially because of Lem. 8. By (19.Known1), (T-Heap), (T-HeapCell), we know $\{\alpha_f\} \cap \{\alpha'_f\} = \emptyset$. Thus also with the definition of $\Sigma_{02} = \emptyset$, we know (19.Goal3) holds. To prove (19.Goal4), note that $\text{dom}(\sigma_{\text{fresh}}) \cap \text{dom}(\sigma) = \{\alpha_f\}$ according to (19.33) and the definition of σ_{fresh} . Thus (19.Goal4) holds because $\sigma_{\text{fresh}}(\alpha_f) = \sigma'_f = \sigma(\alpha_f)$ again according to (19.33).

Case (R-Write): given (19.Known8) and (19.Known9), and the definition of *structure*, $H_1(o) = \langle a; L; \sigma; Acc; Fd_1 \rangle$ for some Fd_1 . Thus by (R-Write), we know $e_1 = f : = v$ and $e_2 = v$. We also know $H_2 = H_1[o \mapsto \langle a; L; \sigma; Acc; Fd_1 \{f \mapsto v\} \rangle]$, $N_2 = N_1$. Thus by (19.Known4), (19.Known5), (19.Known6), and the definition of *structure*, we know $S_2 = S_1$. By (T-Write), (19.Known10), and the definition

of e_1 above, we know

$$\mathcal{P}(a) = \langle \mu; \mathcal{F}; \mathcal{M}; \mathcal{U} \rangle \quad (19.34)$$

$$\mathcal{F}(f) = \forall \alpha_f. a @ \alpha_f \quad (19.35)$$

$$\tau_1 = a @ \alpha_v \quad (19.36)$$

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash v : a @ \alpha_v \setminus \Sigma_v \quad (19.37)$$

$$\Sigma_1 = \Sigma_v \cup \{ \alpha_v \leq \alpha_f, \alpha_f \leq \langle \mathbf{self}; f \rangle \} \quad (19.38)$$

By Lem. 1 and (19.37), we know $\Sigma_v = \emptyset$. Earlier we have proven $S_2 = S_1$. Let $\Sigma_2 = \Sigma_v = \emptyset$ and $\tau_2 = a @ \alpha_v$. Hence (19.37) is identical to (19.Goal1). Let $\sigma'_{\text{fresh}} = \emptyset$ and let $\Sigma_{02} = \emptyset$. Thus (19.Goal4), (19.Goal5), (19.Goal6) hold trivially. (19.Goal2) also holds trivially because of Lem. 8.

Case (R-SyncM): thus

$$e_1 = \mathbf{in}(o', e'_1)$$

$$e_2 = \mathbf{in}(o', e'_2)$$

$$H_1, \langle o_t; o'; e'_1 \rangle \Rightarrow H_2, N_2, \langle o_t; o'; e'_2 \rangle \quad (19.39)$$

By (T-In), (19.Known10), and the definition of e_1 above, we know

$$\mathcal{H}_1(o') = \langle a'; L'; \sigma'_f; Acc' \rangle \quad (19.40)$$

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash e'_1 : \tau' \setminus \Sigma'_1 \quad (19.41)$$

$$\sigma' = \text{fresh}(FTV(\Sigma'_1) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)) \quad (19.42)$$

$$\tau_1 = \tau'_1[\sigma'][\sigma'_f][\sigma'_m] \quad (19.43)$$

$$\Sigma_1 = \Sigma'_1[\sigma'][\sigma'_f][\sigma'_m] \quad (19.44)$$

Given (19.66), (19.Known1), (19.Known2), (19.Known3), (19.Known4), (19.Known5), (19.Known6), (19.Known7), (19.Known8), (19.Known9), we have for some Acc'' ,

$$\mathcal{H}_2(o') = \langle a'; L'; \sigma'_f; Acc'' \rangle \quad (19.45)$$

By induction hypothesis, (19.41), (19.Known1), (19.Known2), (19.39), (19.Known6), (19.Known7), (19.Known8), (19.Known9), (19.66), (19.45), we know

$$[\mathbf{iconf} : S_2, \mathbf{myC} : a], \mathcal{P} \vdash e'_2 : \tau'_2 \setminus \Sigma'_2 \quad (19.46)$$

$$\Sigma_{h1} \cup \Sigma'_1 \xrightarrow{G} \Sigma'_{02} \quad (19.47)$$

$$\text{range}(\sigma'_{\text{fresh}}) \cap FTV(\Sigma'_{02} \cup \Sigma_{h1} \cup \Sigma'_1) = \emptyset \quad (19.48)$$

$$\Sigma'_2 = \Sigma'_{02}[\sigma'_{\text{fresh}}] \quad (19.49)$$

$$\tau'_2 = \tau'_1[\sigma'_{\text{fresh}}] \quad (19.50)$$

Let

$$\sigma'' = \text{fresh}(FTV(\Sigma'_2) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)) \quad (19.51)$$

By (T-In), and (19.45), (19.46), (19.51), and the definition of e_2 , we know

$$[\mathbf{iconf} : S_2, \mathbf{myC} : a], \mathcal{P} \vdash e_2 : \tau_2 \setminus \Sigma_2 \quad (19.52)$$

$$\tau_2 = \tau'_2[\sigma''][\sigma'_f][\sigma'_m] \quad (19.53)$$

$$\Sigma_2 = \Sigma'_2[\sigma''][\sigma'_f][\sigma'_m] \quad (19.54)$$

Given (19.47) and , we can easily know

$$\Sigma_{h1}[\sigma'][\sigma'_f][\sigma'_m] \cup \Sigma'_1[\sigma'][\sigma'_f][\sigma'_m] \xrightarrow{G} \Sigma'_{02}[\sigma'][\sigma'_f][\sigma'_m]$$

which according to and (19.44), we know it can be rewritten as

$$\Sigma_{h1} \cup \Sigma_1 \xrightarrow{G} \Sigma'_{02}[\sigma'][\sigma'_f][\sigma'_m] \quad (19.55)$$

Now according to Lem 7, and (19.47), we know the type variables in Σ'_{02} is either fresh or belongs to $FTV(\Sigma_{h1} \cup \Sigma'_1)$. Let the former variables are $\alpha_1, \dots, \alpha_k$, and the latter variables are $\alpha_{k+1}, \dots, \alpha_m$. By the definition of Σ'_2 in (19.49), we know the type variables in Σ'_2 are either in $\Sigma_{02'}$ or they are in $\text{range}(\sigma'_{\text{fresh}})$. According to (19.48), the two aforementioned sets are disjoint. We let the type variables in $\text{range}(\sigma'_{\text{fresh}})$ to be $\alpha_{m+1}, \dots, \alpha_n$. Thus we know $FTV(\Sigma'_2) = \{\alpha_1, \dots, \alpha_n\}$. Rewriting the definition of σ'' in (19.51), we know

$$\sigma'' = \text{fresh}(\{\alpha_1, \dots, \alpha_n\} - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m))$$

Note that $\{\alpha_1, \dots, \alpha_k\}$ are fresh, and $\alpha_{m+1}, \dots, \alpha_n$ are fresh as well, so The above equation can be written as

$$\sigma'' = \sigma''_x \triangleright \sigma''_y \triangleright \sigma''_z \quad (19.56)$$

$$\sigma''_x = \text{fresh}(\{\alpha_{k+1}, \dots, \alpha_m\} - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)) \quad (19.57)$$

$$\sigma''_y = \text{fresh}(\{\alpha_1, \dots, \alpha_k\}) \quad (19.58)$$

$$\sigma''_z = \text{fresh}(\{\alpha_{m+1}, \dots, \alpha_n\}) \quad (19.59)$$

Since $\{\alpha_{k+1}, \dots, \alpha_m\}$ contains all the type variables that are part of $FTV(\Sigma'_2)$ that belong to $FTV(\Sigma_{h1} \cup \Sigma'_1)$, we know

$$\Sigma'_2[\sigma''_{\text{new}}] = \Sigma'_2[\sigma''_x] \quad (19.60)$$

$$\tau'_2[\sigma''_{\text{new}}] = \tau'_2[\sigma''_x] \quad (19.61)$$

$$\sigma''_{\text{new}} = \text{fresh}(FTV(\Sigma_{h1} \cup \Sigma'_1) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)) \quad (19.62)$$

By the definition of set theory, we know $FTV(\Sigma_{h1} \cup \Sigma'_1) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m) = FTV(\Sigma_{h1}) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m) \cup FTV(\Sigma'_1) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)$. According to the definition of σ' in (19.42), we know

$$\sigma''_{\text{new}} = \sigma' \triangleright \sigma_h \quad (19.63)$$

$$\sigma_h = \text{fresh}(FTV(\Sigma_{h1}) - \text{dom}(\sigma'_f) - \text{dom}(\sigma'_m)) \quad (19.64)$$

We know (19.Goal1) holds because of (19.52). (19.Goal2) holds because of (19.55) when $\Sigma_{02} = \Sigma'_{02}[\sigma'][\sigma'_f][\sigma'_m]$. To

prove (19.Goal5), we know

$$\Sigma_2 = \Sigma'_2[\sigma''_f][\sigma'_f][\sigma'_m] \quad \text{By (19.54)}$$

$$= \Sigma'_2[\sigma''_x][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.56)}$$

$$= \Sigma'_2[\sigma''_{\text{xnew}}][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.60)}$$

$$= \Sigma'_2[\sigma'][\sigma_h][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.63)}$$

$$= \Sigma'_{02}[\sigma'_{\text{fresh}}][\sigma'][\sigma_h][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.49)}$$

$$= \Sigma'_{02}[\sigma'][\sigma'_f][\sigma'_m][\sigma'_{\text{fresh}}][\sigma_h][\sigma''_y][\sigma''_z]$$

where $\sigma_{\text{fresh}} = [\sigma'_{\text{fresh}}][\sigma_h][\sigma''_y][\sigma''_z]$. To prove (19.Goal6), note that

$$\tau_2 = \tau'_2[\sigma''_f][\sigma'_f][\sigma'_m] \quad \text{By (19.53)}$$

$$= \tau'_2[\sigma''_x][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.56)}$$

$$= \tau'_2[\sigma''_{\text{xnew}}][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.61)}$$

$$= \tau'_2[\sigma'][\sigma_h][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.63)}$$

$$= \tau'_1[\sigma'_{\text{fresh}}][\sigma'][\sigma_h][\sigma''_y][\sigma''_z][\sigma'_f][\sigma'_m] \quad \text{By (19.50)}$$

$$= \tau'_1[\sigma'][\sigma'_f][\sigma'_m][\sigma'_{\text{fresh}}][\sigma_h][\sigma''_y][\sigma''_z] \quad \text{By}$$

$$= \tau_1[\sigma'_{\text{fresh}}][\sigma_h][\sigma''_y][\sigma''_z] \quad \text{By (19.43)}$$

thus (19.Goal3) holds. The last part is to prove **self** substitution. Thus we know $e_1 = \mathbf{this}$ and $e_2 = o$. We also know $H_2 = H_1$, $N_2 = N_1$. Thus by (19.Known4), (19.Known5), (19.Known6), and the definition of *structure*, we know $S_2 = S_1$. By (T-Self), (19.Known10), and the definition of e_1 above, we know $\tau_1 = \mathbf{a}@\mathbf{self}$ and $\Sigma_1 = \emptyset$. By (19.Known1), (T-Heap), (T-HeapCell), (19.Known7), we know for some α , $\sigma(\mathbf{self}) = \alpha$. With this, by (T-RID) and (19.Known9), we know

$$[\mathbf{iconf} : S_1, \mathbf{myC} : a], \mathcal{P} \vdash o : \mathbf{a}@\alpha \setminus \emptyset \quad (19.65)$$

Earlier we have proven $S_2 = S_1$. Let $\Sigma_2 = \emptyset$ and $\tau_2 = \mathbf{a}@\alpha$. Hence (19.65) is identical to (19.Goal1). Let $\sigma_{\text{fresh}} = [\mathbf{self} \mapsto \alpha]$ and let $\Sigma_{02} = \emptyset$. Thus (19.Goal5), (19.Goal6) hold trivially. (19.Goal2) also holds trivially because of Lem. 8. (19.Goal3) also holds trivially since $\Sigma_{02} \cup \Sigma_1 = \emptyset$. To prove (19.Goal4), note that $\text{dom}(\sigma_{\text{fresh}}) \cap \text{dom}(\sigma) = \{\mathbf{self}\}$ according to the definition of σ_{fresh} and our earlier conclusion that $\sigma(\mathbf{self}) = \alpha$. Thus (19.Goal4) trivially holds as a result of the latter.

Case (R-ASyncM): With (19.Known9), and (19.Known3), (19.Known4), (T-RID) we know

$$\Sigma_o = \{\langle a; L; \sigma \rangle \leq \alpha\} \quad (19.66)$$

Given (19.Known2), (T-Call), and (19.39), we know

$$\Gamma, \mathcal{P} \vdash v : \mathbf{a}_d@\alpha_{d0} \setminus \Sigma_v \quad (19.67)$$

$$\Sigma_{\text{lazy}} = \{\mathbf{lazy}(\alpha, m, \sigma_m)\} \quad (19.68)$$

$$\Sigma_d = \left\{ \begin{array}{l} \langle \mathbf{self}; \Gamma(\mathbf{myM}) \rangle \xrightarrow{\alpha'_d}_{\text{false}} \langle \alpha; m \rangle \\ \langle \alpha; m \rangle \xrightarrow{\alpha'_r}_{\text{false}} \langle \mathbf{self}; \Gamma(\mathbf{myM}) \rangle \end{array} \right\} \quad (19.69)$$

$$\tau_1 = \mathbf{a}_r@\alpha_{r0} \quad (19.70)$$

$$\Sigma_1 = \Sigma_v \cup \Sigma_{\text{lazy}} \cup \Sigma_d \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\} \quad (19.71)$$

According to Lem. 6, (19.Known1), (19.Known5), (19.Known6), (19.Known7), (19.Known9), (19.52), (19.53), (??), (19.67), we know

$$\Gamma \triangleright [\mathbf{myC} : a, \mathbf{myM} : m], \mathcal{P} \vdash e\{v/x\} : \mathbf{a}_r@\alpha_{r0} \setminus \Sigma'' \quad (19.72)$$

$$\Sigma'' = G(\langle a; m \rangle)[\sigma_m] \cup \Sigma_v \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\} \quad (19.73)$$

According to Def. 8, (19.Known8), we know

$$\Gamma'(\mathbf{iconf}) = \langle \mathcal{H}'; \mathcal{T} \rangle \quad (19.74)$$

$$\mathcal{H}'(o) = \langle a; L; \sigma; Acc' \rangle \quad (19.75)$$

According to Lem. 10, (19.72), we know

$$\Gamma' \triangleright [\mathbf{myC} : a, \mathbf{myM} : m], \mathcal{P} \vdash e\{v/x\} : \mathbf{a}_r@\alpha_{r0} \setminus \Sigma'' \quad (19.76)$$

With (T-In), (19.74), (19.75), (19.Known9), (19.76), we know

$$\Gamma', \mathcal{P} \vdash \mathbf{in}(o, e\{v/x\}) : \tau_2 \setminus \Sigma_2 \quad (19.77)$$

$$\tau_2 = \mathbf{a}_r@\alpha_{r0}[\sigma_{\text{in}}][\sigma][\sigma_m] \quad (19.78)$$

$$\Sigma_2 = \Sigma''[\sigma_{\text{in}}][\sigma][\sigma_m] \quad (19.79)$$

$$\sigma_{\text{in}} = \text{fresh}(FTV(\Sigma'') - \text{dom}(\sigma) - \{\alpha_d, \alpha_r\}) \quad (19.80)$$

Let $\sigma_{\text{fresh}} = [\sigma_{\text{in}}][\sigma][\sigma_m]$. We know (19.Goal6). The tricky thing is to show (19.Goal3). The rest of the proof is to show (19.Goal2) holds. We know that if $\Sigma_{02} = \Sigma''$, then (19.Goal5) holds according to (19.79). Now according to (OCL-Main), we know trivially

$$G \triangleright [\langle a_{\text{main}}; m_{\text{main}} \rangle \mapsto \Sigma_o \cup \Sigma_1] \vdash_{\text{oc1}} \Sigma_o \cup \Sigma_1 \quad (19.81)$$

Let $G' = G \triangleright [\langle a_{\text{main}}; m_{\text{main}} \rangle \mapsto \Sigma_o \cup \Sigma_1]$. By (19.66), (19.41), (19.68), (19.71), and the definition of $\vdash_{\text{oc1}}$, we know

$$G' \vdash_{\text{oc1}} \langle a; L; \sigma \rangle \leq \alpha$$

$$G' \vdash_{\text{oc1}} \mathbf{lazy}(\alpha, m, \sigma_m)$$

As a result of (??), we know $G'(\langle a; m \rangle) = G(\langle a; m \rangle)$. By (OCL-Lazy-Intro), (OCL-Lazy-NonReursive), (OCL-Lazy-Cancel), and the rules above, we know

$$\begin{aligned} G' \vdash_{\text{oc1}} G(\langle a; m \rangle)[\sigma_{\text{fitg}}][\sigma][\sigma_m] \\ \sigma_{\text{fitg}} = \text{fresh}(FTV(G(\langle a; m \rangle)) - \text{dom}(\sigma) - \text{dom}(\sigma_m)) \end{aligned} \quad (19.82)$$

It is obvious σ_{fitg} , σ , and σ_m have disjoint domains, so the above can be rewritten as:

$$G' \vdash_{\text{ocl}} G(\langle a; m \rangle)[\sigma_m][\sigma_{\text{fitg}}][\sigma] \quad (19.83)$$

Now according to the definition of σ_{fitg} and σ_{in} , and the definition of Σ'' in (19.73), we know $G(\langle a; m \rangle)[\sigma_m][\sigma_{\text{fitg}}] = G(\langle a; m \rangle)[\sigma_m][\sigma_{\text{in}}]$. Plus the disjointness of the three substitutions, we know

$$G' \vdash_{\text{ocl}} G(\langle a; m \rangle)[\sigma_{\text{in}}][\sigma][\sigma_m] \quad (19.84)$$

which is

$$\Sigma_o \cup \Sigma_{\text{lazy}} \cup \Sigma_d \xrightarrow{G} G(\langle a; m \rangle)[\sigma_m] \quad (19.85)$$

Now according to Lem. 9, we know

$$\Sigma_o \cup \Sigma_v \cup \Sigma_{\text{lazy}} \cup \Sigma_d \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\} \xrightarrow{G} G(\langle a; m \rangle)[\sigma_m] \cup \Sigma_v \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\}$$

That is according to the definition of Σ_1 in (19.71):

$$\Sigma_o \cup \Sigma_1 \xrightarrow{G} G(\langle a; m \rangle)[\sigma_m] \cup \Sigma_v \cup \{\alpha_{d0} \leq \alpha'_d, \alpha'_r \leq \alpha_{r0}\}$$

which is (19.Goal2).

Case (R-Context): thus

$$\begin{aligned} e_1 &= \mathbf{E}[e'_1] \\ e_2 &= \mathbf{E}[e'_2] \\ H_1, \langle o_t; o; e'_1 \rangle &\Rightarrow H_2, N_2, \langle o_t; o; e'_2 \rangle \end{aligned} \quad (19.86)$$

By (T-Context), (19.Known10), the conclusion holds by induction. $\square$

Lemma 20 (Subject Reduction). *Given $S_i = \langle H_i; T_i \rangle$ for $i = 1, 2$,*

$$P \vdash S_1 \setminus \Sigma_1 \quad (20.1)$$

$$S_1 \Rightarrow S_2 \quad (20.2)$$

$$\vdash_{\text{global}} P \setminus G \quad (20.3)$$

then $P \vdash S_2 \setminus \Sigma_2$ and $\Sigma_1 \xrightarrow{G} \Sigma_2$.

Proof. Given (20.1) and (T-Config), we know

$$\Xi(P) = \mathcal{P} \quad (20.4)$$

$$\text{structure}(S_1) = \mathbf{S}_1 \quad (20.5)$$

$$\mathcal{P} \vdash_{\text{wft}} \mathbf{S}_1 \quad (20.6)$$

$$\mathbf{S}_1, \mathcal{P} \vdash_{\text{h}} H_1 \setminus \Sigma_{\text{h1}} \quad (20.7)$$

$$\mathbf{S}_1, \mathcal{P} \vdash_{\text{tp}} T_1 \setminus \Sigma_{\text{tp1}} \quad (20.8)$$

$$\Sigma_1 = \Sigma_{\text{h1}} \cup \Sigma_{\text{tp1}} \quad (20.9)$$

Given Lem. 17, (20.1), (20.2), we know

$$\text{structure}(S_2) = \mathbf{S}_2 \quad (20.10)$$

$$\mathcal{P} \vdash_{\text{wft}} \mathbf{S}_2 \quad (20.11)$$

Given Lem. 18, (20.3), (20.4), (20.2), (20.5), (20.10), (20.7), we know

$$\mathbf{S}_2, \mathcal{P} \vdash_{\text{h}} H_2 \setminus \Sigma_{\text{h2}} \quad (20.12)$$

$$\Sigma_{\text{h1}} \xrightarrow{G} \Sigma_{\text{h2}} \quad (20.13)$$

With these preconditions, the non-trivial cases are proved in Lem. ??, Lem. 19 and Lem. 18. $\square$

C.11 Progress

Definition 11. *deadlockable($\langle H; T \rangle$) holds iff we know daccesses(H, o_i, o'_i) for $i = 0 \dots n-1$, $\langle o_i, \mathbf{E}[e_i] \rangle$ belongs to T , and $e_i = \text{capture } o'_{i+1} \text{ mod } n$ or $o'_{i+1} \text{ mod } n \cdot m_i(v_i)$.*

Lemma 21 (Progress). *Given $S_1 = \langle H_1; T_1 \rangle$, and $P \vdash S_1 \setminus \Sigma_1$, then either $T_1 = \langle o; v \rangle$, or deadlockable(S_1), or there exists S_2 such that $S_1 \Rightarrow S_2$.*

Proof. Induction on the execution sequence and case analysis on the last step of the reduction. (R-Context) progresses by induction hypothesis. (R-New), (R-Write), (R-Read), (R-Commute), (R-Compose) always progress. (R-Cast) progresses because the pre-condition of $a' \in \mathcal{U}$ can be immediately satisfied by (T-Cast), which in turn is a subderviation of $P \vdash S_1 \setminus \Sigma_1$. The non-trivial cases are (R-SyncM), (R-Capture), (R-AsyncM), (R-Subtask), which we now prove.

To prove the case of (R-SyncM), note that the pre-condition to satisfy is to have $\text{Acc} \subseteq \text{anc}(H, o_t)$. We know the elements in Acc do belong to the same tree in the forest of $\text{accG}(H)$, and they form a tree path. In plain words, the pre-condition means that the executing task/subtask o_t must be lower on the tree path where all elements in Acc belongs to. If *deadlockable*(S) does not hold, we know a cycle does not exist in T_1 , so we know there must be a chain in T_1 that can reduce. (R-Capture), (R-AsyncM), (R-Subtask) can be proved in the similar manner. $\square$

C.12 Atomicity

We write step $st_r = (S, r, S')$ to denote a transition $S \Rightarrow S'$ by reduction rule r . We let $\text{change}(st_r) = (s_o, r, s'_o)$ denote the fact that the begin and end heaps of step st_r differ at most on their state of o , taking it from s_o to s'_o . Similarly, the change in two consecutive steps which changes at most two objects o_1 and o_2 , $o_1 \neq o_2$, is represented as $\text{change}(st_{r_1} st_{r_2}) = ((s_{o_1}, s_{o_2}), r_1 r_2, (s'_{o_1}, s'_{o_2}))$. If $o_1 = o_2$, then $\text{change}(st_{r_1} st_{r_2}) = (s_{o_1}, r_1 r_2, s'_{o_1})$.

Definition 12 (Local and Nonlocal Step). *A step $st_r = (S, r, S')$ is a local step if r is one of the local rules: either (R-Read), (R-Write), (R-New) or (R-SyncM). st_r is a*

nonlocal step if r is one of nonlocal rules: either (R-Task), (R-SubTask), (R-Tend) or (R-STEnd).

Every nonlocal rule has a label given in Fig 11. For example, the (R-Task) rule has label (R-Task)(t, γ, mn, v, o, t') meaning asynchronous message mn was sent from object γ in task t to another object o in a new task t' , and the argument passed was v . These labels are used as the *observable*; the local rules also carry labels but since they are local steps internal to a t so that they are *not* observable steps.

Intuitively, observable steps are places where tasks interact by making results of local steps possibly visible to other tasks. For instance, in a (R-Task) step, a task t creates a new task t' by sending an asynchronous message mn . Along with the message mn , t sends a parameter v to t' . If v is a value calculated by t 's local steps, then upon the execution of the (R-Task) step, t makes its internal computation visible to the newly created t' .

Lemma 22. *In any given local step st_r , at most one object o 's state can be changed from s_o to s'_o (s_o is **null** if st_r creates o).*

Proof. A local step is step of applying one of the local rules defined in Definition 12. Among all these rules, only (R-New), (R-Read) and (R-Write) can possibly change object state. (R-New) creates a single object. The (R-Read) and (R-Write) rules each operates on exactly one object and may change its state. No other rules change object state. $\square$

Definition 13 (Computation Path). *A computation path p is a finite sequence of steps $st_{r_1}st_{r_2} \dots st_{r_{i-1}}st_{r_i}$ such that we know $st_{r_1}, st_{r_2}, \dots, st_{r_{i-1}}, st_{r_i} = (S_0, r_1, S_1), (S_1, r_2, S_2), \dots, (S_{i-2}, r_{i-1}, S_{i-1}), (S_{i-1}, r_i, S_i)$.*

Here we only consider finite paths as is common in process algebra, which simplifies our presentation. Infinite paths can be interpreted as a set of ever-increasing finite paths. For brevity purpose, we use computation path and path in an interchangeable way if no confusion would arise.

The initial run time configuration of a computation path, S_0 , always has $H = \emptyset$, and $T = \{o_{\text{main}}\}$. o_{main} is the default bootstrapping task that starts the execution of a program from the entry `main` method of a class in the program.

Definition 14 (Observable Behavior). *The observable behavior of a computation path p , $ob(p)$, is the label sequence of all nonlocal steps occurring in p .*

Definition 15 (Observable Equivalence). *Two paths p_1 and p_2 are observably equivalent, written as $p_1 \equiv p_2$, iff $ob(p_1) = ob(p_2)$.*

Definition 16 (Object-blocked). *A task t is in an object-blocked state S at some point in a path p if it would be enabled for a next step $st_r = (S, r, S')$ for which r is a (R-SyncM) or (R-AsyncM) or (R-Subtask) or (R-Capture) step on object o , except for the fact that there is a capture violation on o : one of the $Acc \subseteq$ preconditions fails to hold*

in S and so the step cannot in fact be the next step at that point.

Object-blocked state is a state a task cannot make process because it has to wait for an object becoming available to it.

Definition 17 (Sub-path and Maximal Sub-path). *Given a fixed p , for some task t a sub-path sp_t of p is a sequence of steps in p which are all local steps of task t . A maximal sub-path is a sp_t in p which is longest: no local t steps in p can be added to the beginning or the end of sp_t to obtain a longer sub-path.*

Note that the steps in sp_t need not be consecutive in p , they can be interleaved with steps belonging to other tasks.

Definition 18 (Pointed Maximal Sub-path). *For a given path, a pointed maximal sub-path for a task t ($pmsp_t$) is a maximal sub-path sp_t with either 1) it has one nonlocal step appended to its end or 2) there are no more t steps ever in the path.*

The second case is the technical case of when the (finite) path has ended but the task t is still running. The last step of a $pmsp_t$ is called its *point*. We omit the t subscript on $pmsp_t$ when we do not care which task a $pmsp$ belongs to.

Since we have extended the $pmsp$ maximally and have allowed inclusion of one nonlocal step at the end, we have captured all the steps of any path in some $pmsp$:

Lemma 23. *For a given path p , all the steps of p can be partitioned into a set of $pmsp$'s where each step st_r of p occurs in precisely one $pmsp$, written as $st_r \in pmsp$.*

Proof. This immediately follows the definition of $pmsp$. $\square$

Given this fact, we can make the following unambiguous definition.

Definition 19 (Indexed $pmsp$). *For some fixed path p , define $pmsp(i)$ to be the i^{th} pointed maximal sub-path in p , where all the steps of the $pmsp(i)$ occur after any of $pmsp(i+1)$ and before any of $pmsp(i-1)$.*

The $pmsp$'s are the units which we need to serialize: they are all spread out in the initial path p , and we need to show there is an equivalent path where each $pmsp$ runs in turn as an atomic unit.

Definition 20 (Path around a $pmsp(i)$). *The path around a $pmsp(i)$ is a finite sequence of all of the steps in p from the first step after $pmsp(i-1)$ to the end of $pmsp(i)$ inclusive. It includes all steps of $pmsp(i)$ and also all the interleaved steps of other tasks.*

Definition 19 defines a global ordering on all $pmsp$'s in a path p without concerning which task a $pmsp$ belongs to. The following Definition 21 defines a task scope index of a $pmsp$ which is used to indicate the local ordering of $pmsp$'s of a task within the scope of the task.

Definition 21 (Task Indexed pmsp). For some fixed path p , define $pmsp_{t,i}$ to be the i^{th} pointed maximal sub-path of task t in p , where all the steps of the $pmsp_{t,i}$ occur after any of $pmsp_{t,i+1}$ and before any of $pmsp_{t,i-1}$.

For a $pmsp_{t,i}$, if we do not care its task scope ordering, we omit the index i and simply use $pmsp_t$.

Definition 22 (Waits-for and Deadlocking Path). For some path p , $pmsp_{t_1,i}$ waits-for $pmsp_{t_2,j}$ if t_1 goes into a object-blocked state in $pmsp_{t_1,i}$ on an object captured by t_2 in the blocked state. A deadlocking path p is a path where this waits-for relation has a cycle: $pmsp_{t_1,i}$ waits-for $pmsp_{t_2,j}$ while $pmsp_{t_2,i'}$ waits-for $pmsp_{t_1,j'}$.

Hereafter we assume in this theoretical development that there are no such cycles.

Definition 23 (Quantized Sub-path and Quantized Path). A quantized sub-path contained in p is a $pmsp_t$ of p where all steps of $pmsp_t$ are consecutive in p . A quantized path p is a path consisting of a sequence of quantized sub-paths.

The main technical Lemma is the following Bubble-Down Lemma, which shows how local steps can be pushed down in the computation. Use of such a Lemma is the standard technique to show atomicity properties. In this approach, all state transition steps of a potentially interleaving execution are categorized based on their commutativity with consecutive steps: a right mover, a left mover, a both mover or a non-mover. The reduction is defined as moving the transition steps in the allowed direction. We show the local steps are right movers; in fact they are both-movers but that stronger result is not needed.

Definition 24 (Step Swap). For any two consecutive steps $st_{r_1}st_{r_2}$ in a computation path p , a step swap of $st_{r_1}st_{r_2}$ is defined as swapping the order of application of rules in the two steps, i.e., apply r_2 first then r_1 . We let $st'_{r_2}st'_{r_1}$ denote a step swap of $st_{r_1}st_{r_2}$.

Definition 25 (Equivalent Step Swap). For two consecutive steps $st_{r_1}st_{r_2}$ in a computation path p , where $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$, $t_1 \neq t_2$ and $st_{r_1}st_{r_2} = (S, r_1, S')(S', r_2, S'')$, if the step swap of $st_{r_1}st_{r_2}$, written as $st'_{r_2}st'_{r_1}$, gives a new path p' such that $p \equiv p'$ and $st'_{r_2}st'_{r_1} = (S, r_2, S^*)(S^*, r_1, S'')$, then it is an equivalent step swap.

Lemma 24 (Bubble-down Lemma). For any path p with any two consecutive steps $st_{r_1}st_{r_2}$ where $st_{r_1} \in pmsp_{t_1}$, $st_{r_2} \in pmsp_{t_2}$ and $t_1 \neq t_2$, if st_{r_1} is a local step, then a step swap of $st_{r_1}st_{r_2}$ is an equivalent step swap.

Proof. First, observe that if t_2 is a subtask of t_1 , then it is impossible for st_{r_1} to be a local step while st_{r_2} is a step of t_2 . Because according to semantics defined in Figure 11 a task and its subtask never have their local steps consecutive to each other. Figure. 20 illustrates all possible cases of how steps of a task and steps of its subtask may layout in a path. It clearly shows that local steps of the task t_1 and its subtask t_2

always have their local steps demarcated by some nonlocal steps.

According to Definition 14 and 15, $\equiv$ is defined by the sequence of labels of nonlocal steps occurring in path p , so a step swap of $st_{r_1}st_{r_2}$ always gives a new path p' , $p' \equiv p$, since st_{r_1} is a local step by the lemma so that swap st_{r_1} with any st_{r_2} never changes the ordering of labels. Therefore, to show that the step swap of $st_{r_1}st_{r_2}$ is an equivalent step swap we only need to prove that if $st_{r_1}st_{r_2} = (S, r_1, S')(S', r_2, S'')$, then the step swap of $st_{r_1}st_{r_2}$ is $st'_{r_2}st'_{r_1} = (S, r_2, S^*)(S^*, r_1, S'')$.

Because st_{r_1} is a local step, it can at most change one object's state on the heap H and a local step does not change N according to the operational semantics. So we can represent st_{r_1} as $st_{r_1} = ((H_1, T_1), r_1, (H_2, T_2))$ where H_1 and H_2 differ at most on one object o .

(Case I) st_{r_2} is also a local step.

In this case, $st_{r_2} = ((H_2, T_2), r_2, (H_3, T_3))$. Because st_{r_1} and st_{r_2} are steps of different tasks t_1 and t_2 , st_{r_1} and st_{r_2} must change different elements of T (t_1 and t_2 respectively) by inspection of the rules. This means any change made by st_{r_1} and st_{r_2} to T always commute. So in the section of **Case I**, we focus only on changes st_{r_1} and st_{r_2} make on H , and omit N and T for concision.

Suppose $change(st_{r_1}) = (s_{o_1}, r_1, s'_{o_1})$ and $change(st_{r_2}) = (s_{o_2}, r_2, s'_{o_2})$.

If $o_1 \neq o_2$, then it must hold that $change(st_{r_1}st_{r_2}) = ((s_{o_1}, s_{o_2}), r_1r_2, (s'_{o_1}, s'_{o_2}))$. Swapping the order of st_{r_1}, st_{r_2} by applying r_2 first then we know r_1 results in the change $((s_{o_1}, s_{o_2}), r_2r_1, (s'_{o_1}, s'_{o_2}))$, which has the same start and end states as $change(st_{r_1}st_{r_2})$, regardless of what r_1 and r_2 are.

If $o_1 = o_2 = o$, then $change(st_{r_1}) = (s_o, r_1, s'_o)$, $change(st_{r_2}) = (s'_o, r_2, s''_o)$, $change(st_{r_1}st_{r_2}) = (s_o, r_1r_2, s''_o)$. Let $s_o = \langle a; L; Acc; Fd \rangle$.

By inspection of the rules this case only arises if both of the rules are amongst (R-SyncM), (R-Capture), (R-AsyncM), (R-Subtask) and (R-New).

Subcase a: $r_1 = (\text{R-New})$.

$change(st_{r_1}) = (\text{null}, r_1, s_o)$ so r_2 cannot be (R-New) since o cannot be created twice. And, r_2 also cannot be any other local rule: o is just created in st_{r_1} by t_1 . For t_2 to be able to access o , it must obtain a reference to o first. Only t_1 can pass a reference of o to t_2 directly or indirectly. As a result, if st_{r_2} is a local step of t_2 that operates on o , it cannot be consecutive to st_{r_1} , and vice visa. Therefore, r_1 (R-New) is not possible.

Subcase b: $r_1 = (\text{R-SyncM}), (\text{R-Capture}), (\text{R-AsyncM}), (\text{R-Subtask})$.

Trivially, r_2 cannot be (R-New) that creates o because no steps can operate on an object before it is created. In this case, $change(st_{r_1}) = (\langle a; L; Acc; Fd \rangle, r_1, \langle a; L; Acc \cup \{o_t\}; Fd \rangle)$, if o_t locks o in st_{r_1} or $Acc' = Acc$ if Acc is a subset of the ancestors of t_1 . Either way, there does not

Figure 20. Task and Subtask

exist an consecutive st_{r_2} of t_2 where r_2 is either (R-SyncM) or (R-AsyncM) or (R-Capture) or (R-Subtask) because if so, firing up st_{r_2} would violate the preconditions of the aforementioned rule.

(Case II) st_{r_2} is a nonlocal step.

Let $st_{r_1} = ((H_1, T_1), r_1, (H_2, T_2))$ and H_2 differs from H_1 at most on object o since st_{r_1} is a local step. Because st_{r_1} and st_{r_2} are steps of t_1 and t_2 respectively, they must change different elements of T , i.e., t_1 and t_2 . st_{r_2} as a nonlocal step may add a new fresh element t to T . But st_{r_1} and st_{r_2} still obviously commute in terms of changes to T . So in the following proof, we do not need to concern about T commutativity.

Subcase a: $r_2 = (\text{R-Task})$.

Let o_t be the new task created in st_{r_2} , then it must hold that $st_{r_2} = ((H_2, T_2), r_2, (H_2, T_2 \parallel \langle o_t; e \rangle))$. The final state change of taking st_{r_1} and st_{r_2} in this order is $((H_1, T_1), r_1 r_2, (H_2, T_2 \parallel \langle o_t; e \rangle))$. Swapping st_{r_1} and

st_{r_2} results consecutive steps $st'_1 = ((H_1, T_1), r_2, (H_1, T_1 \parallel \langle o_t; e \rangle))$ and $st'_2 = ((H_1, T_1 \cup t), r_1, (H_2, T_2 \parallel \langle o_t; e \rangle))$, which makes the combined state change of $st'_1 st'_2$ to be $((H_1, T_1), r_2 r_1, (H_2, T_2 \parallel \langle o_t; e \rangle))$, the same state change as that of $st_{r_1} st_{r_2}$.

Subcase b: $r_2 = (\text{R-SubTask})$.

Let t be the new task created in st_{r_2} , then we know it must have $st_{r_2} = ((H_2, T_2), r_2, (H_2, T_3))$. $st_{r_1} st_{r_2} = ((H_1, o_{t1}), r_1 r_2, (H_2, T_3))$. If we apply r_2 first, we get $st'_1 = ((H_1, o_{t1}), r_2, (H_1, T_2))$. Then r_1 is applied to get $st'_2 = ((H_1, T_2), r_1, (H_2, T_3))$. Therefore, $st'_1 st'_2$ results in a transition $((H_1, o_{t1}), r_2 r_1, (H_2, T_3))$ which has the same start and final states as $st_{r_1} st_{r_2}$.

Subcase c: $r_2 = (\text{R-Tend})$.

Let $st_{r_2} = ((H_2, o_{t2}), r_2, (H_3, T_3))$, where $H_3 = \biguplus_{H_2(o)=\langle a; L; Acc; Fd \rangle} (o \mapsto \langle a; L; Acc \setminus o_{t2}; Fd \rangle)$. Namely,

st_{r_2} removes o_{t_2} from R and W sets of all objects in H_2 . Consider the case when st_{r_1} changes an object o 's F but not its R or W : taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} also changes R or W of o , then it can only add o_{t_1} to Acc of o , while st_{r_2} may only remove o_{t_2} from R or W of o . Consequently, swapping the two steps we still get the same result because the set operations commute since $o_{t_1} \neq o_{t_2}$.

Subcase d: $r_2 = (R-STEnd)$.

In this case $st_{r_2} = ((H_2, o_{t_2}), r_2, (H_3, T_3))$, where $H_3 = \biguplus_{H_2(o)=\langle a; L; Acc \setminus o_{t_2}; Fd \rangle} (o \mapsto \langle a; L; Acc \setminus o_{t_2}; Fd \rangle)$. Namely, st_{r_2} removes o_{t_2} from Acc sets of all objects in H_2 . Consider the case when st_{r_1} changes an object o 's F but not its Acc : taking st_{r_1} then st_{r_2} has the same state change as taking the two steps in reversed order because the two steps work on different regions of the heap. If st_{r_1} adds o_{t_1} to R or W of o , swapping the two steps we still get the same result because the set operations commute since $o_{t_1} \neq o_{t_2}$. $\square$

Given this Lemma we can now directly prove the Quantized Atomicity Theorem.

Theorem 7 (Quantized Atomicity). *Atomicity in Thm. 4 holds for all paths p there exists a quantized path p' such that $p' \equiv p$.*

Proof. Proceed by first sorting all $pmsp$'s of p into a well ordering induced by the ordering of their points in p . Write $pmsp(i)$ for the i -th indexed $pmsp$ in this ordering. Suppose that there are n $pmsp$'s in total in p for some n . We proceed by induction on n to show for all $i \leq n$, p is equivalent to a path p_i where the 1st to i th indexed $pmsp$'s in this ordering have been bubbled to be quantized subpaths in a prefix of p_i : $p \equiv p_i = pmsp(1) \dots pmsp(i) \dots$ where $pmsp(k)$ is quantized with $k = 1 \dots i$. With this fact, for $i = n$ we have $p \equiv p_n = pmsp(1) \dots pmsp(n)$ where $pmsp(k)$ is quantized with $k = 1 \dots n$, proving the result.

The base case $n = 0$ is trivial since the path is empty. Assume by induction that all $pmsp(i)$ for $i < n$ have been bubbled to be quantized subpaths and the bubbled path $p_i = pmsp(1) \dots pmsp(i) \dots$ where $pmsp(k)$ is quantized with $k = 1 \dots i$, has the property $p_i \equiv p$. Then, the path around $pmsp(i+1)$ includes steps of $pmsp(i+1)$ or $pmsp$'s with bigger indices. By repeated applications of the Bubble-Down Lemma, all these local steps that do not belong to $pmsp(i+1)$ can be pushed down past its point, defining a new path p_{i+1} . In this path $pmsp(i+1)$ is also now a quantized subpath, and $p_{i+1} \equiv p$ because $p_i \equiv p$ and the Bubble-Down lemma which turns p_i to p_{i+1} does not shuffle any nonlocal steps so $p_i \equiv p_{i+1}$. $\square$