Task Types for Pervasive Atomicity

Aditya Kulkarni
SUNY Binghamton
akulkar1@cs.binghamton.edu

Yu David Liu
SUNY Binghamton
davidL@cs.binghamton.edu

Scott F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

Abstract

Atomic regions are an important concept in correct concurrent programming: since atomic regions can be viewed as having executed in a single step, atomicity greatly reduces the number of possible interleavings the programmer needs to consider. This paper describes a method for building atomicity into a programming language in an organic fashion. We take the view that atomicity holds for whole threads by default, and a division into smaller atomic regions occurs only at points where an explicit need for sharing is needed and declared. A corollary of this view is every line of code is part of some atomic region. We define a polymorphic type system, *Task Types*, to enforce most of the desired atomicity properties statically. We show the reasonableness of our type system by proving that type soundness, isolation invariance, and atomicity enforcement properties hold at run time. We also present initial results of a Task Types implementation built on Java.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Design, Languages, Theory

Keywords Pervasive Atomicity, Type Systems, Sharing-Aware Programming

1. Introduction

In an era when multi-core programming is becoming the rule not the exception, the property of *atomicity* – that program execution in the presence of interleavings has the same effect as a sequential execution – is a crucial invariant. Some programming languages now support a notion of **atomic** block, requiring the block to be viewable as executing atomically; this means there will not be any interleavings violating the sequential view of that block and program meaning is greatly clarified. One weakness of atomic blocks however is that guarantees of atomicity hold only in so-marked blocks, and code outside of the marked blocks may well have anomalous behaviour upon interleaving. When the atomicity-enforcing code interleaves with the non-atomicity-enforcing code, a weaker guarantee known as weak atomicity [CMC+06; SMAT+07; ABHI08] may happen.

This paper – built on top of our previous Coqa language [LLS08] – takes the opposite route to address atomicity. Instead of indicating which subparts of a thread should be atomic, a programmer of our language divides the thread into subzones of atomic execution, and every single line of code must be part of *some* atomic zone. This design principle, which we call *pervasive atomicity*, eliminates weak atomicity by design. The programming approach is the opposite of atomic zones: by default threads are completely atomic and uncommunicative, and specific atomicity break points are then inserted where the thread needs to communicate with other threads. In addition, since the number of zones is much smaller than the number of program instructions, the conceptual number of program interleavings is significantly reduced, a boon for program analysis and testing, and ultimately for the deployment of more reliable software.

Unfortunately, Coqa is not ideal: it requires dynamic monitoring to limit object sharing between threads. Dynamic monitoring mechanisms are known to incur heavy overhead, which can be particularly bad here since every object may need to be monitored. They can also suffer from problems of deadlock or livelock. Our initial Coqa compiler relied on *ad hoc* optimizations to achieve tolerable performance. What is needed is a principled means to keep the benefits of pervasive atomicity, but without the high cost.

This paper answers this need by developing a static, declarative method for dividing objects between threads, *Task Types*. It is common knowledge [CGS+99; WR99; Bla99] that if an object is accessed only by one thread, then dynamic atomicity enforcement on that object is unnecessary. Task Types lift this simple notion to the programming level, effectively enforcing a non-shared memory model by default at compile time.

Figure 1(a) illustrates how objects are statically localized in threads via Task Types. Here the two rectangular boxes represent two runtime threads, called *tasks* in our language. The solid black objects are special *task objects* which launch a new task every time they are invoked, while the white objects are the *non-shared ordinary objects*, the vast ma-

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

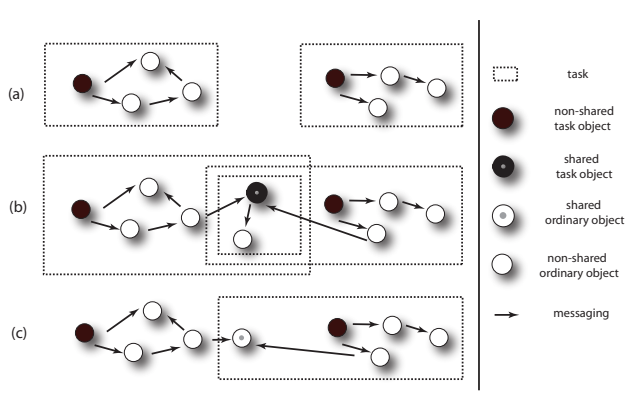

Figure 1. Isolation and Sharing at Run Time

jority of objects that are accessed by only one task. Our type system statically guarantees this picture of isolation. Making non-shared-memory the default case has parallels with Actor-based languages [Agh90; Arm96; SM08], MPI [GLS94] and DPJ [BAD+09] amongst others. Compared to these approaches, Task Types aim to get the benefits of these models while offering a more familiar setting for programmers: only minor changes need to be made to most Java programs to make them compilable by the Task Types compiler. One reason why fewer program changes are required is that we support limited inter-task sharing, unlike the above models. Fig. 1(b) and Fig. 1(c) are examples of forms of inter-task object sharing that we support.

In Fig. 1(b), two tasks communicate through *shared task objects*, special objects which may themselves hold a set of non-shared ordinary objects and serve as the rendezvous point for other tasks. An important benefit of limited sharing is the degree to which atomicity properties are preserved: the sending task has one zone of atomicity from the start to the shared task object invocation, the shared task itself is an atomic zone, and the task execution after the invocation is finished is a third zone of atomicity. So, our approach is a compromise between the extreme lack of sharing of Actor or Actor-like languages [Agh90; Arm96; SM08; GLS94; BAD+09] and the uncontrolled sharing of current multi-threaded languages; by aiming in the middle we can achieve a reasonable compromise between ease of programmability and the production of reliable code.

Fig. 1(c) shows an additional form of sharing that we support, the *shared ordinary objects*. These objects allow Coqa-style sharing to be used in limited cases within Task Types: only one task may use a shared ordinary object at a time, so no atomicity of tasks is ever violated due to access of these objects. As in Coqa, run-time support is needed to ensure two different tasks never access such an object at the same time. The programming choice between using a shared task object or a shared ordinary object reflects a clear choice between more parallelism or more atomicity.

Ownership types [CPN98; Cla01; BLR02] and region types [TT97; Gro03; CCQR04] are well-known type-based techniques for static partitioning of memory. Task Types are strongly related to such systems, but differ in two important aspects: explicit sharing exceptions are allowed in Task Types, and static type variables must invariably align with runtime tasks.

2. Informal Discussion

In this section, we highlight a number of features of our language, focusing on its type system. We use a simplified Map-Reduce algorithm [DG04] to illustrate basic language features; the code is in Fig. 2. Map-Reduce represents a common type of multi-core algorithm, of the "embarrassingly parallel" style. Later in this section, we will discuss a program of the opposite nature, a high-contention PuzzleSolver [LLS08]. Together, we aim to provide readers a real feel of programming in Task Types.

2.1 Sharing-Aware Programming

Task Types encourage programmers to make upfront sharing decisions, by associating *class modifiers* μ with classes. The default choice here ($\mu = \epsilon$) aligns precisely with the principle of having non-shared memory as the default. The objects instantiated from these classes are non-shared ordinary objects, and messaging to these objects uses the standard Java dot (.) invocation symbol. For instance, the code for MapReduce in Fig. 2 indicates `WorkUnit` is not shared. The `WorkUnit` object encapsulates data and a unit of work that needs to be done on the data, and each such instance is indeed exclusively used by each `Mapper`.

Here `Mapper` is declared as a **task**, meaning each `Mapper` object is a (non-shared) task object. Each such object spawns a new task (thread) when sent a message, in analogy to how an actor handles a message [Agh90]. Non-shared tasks are simply threads with a distinguished object representative, and the execution of the body of the invoked method constitutes the lifetime of a task. Note that the completion of the task does not end the lifetime of the task object or any state associated with it – the object may later receive and handle another message following the Actor model. Here the `main` method's expression `m -> map(ul, r)` creates a non-shared task by sending a `map` message to the entry object `m`. Such invocations are asynchronous and have no interesting return value. Here, twenty `Mapper` threads are executing in parallel. Individually, each task object keeps a queue of all received messages and processes them serially, following Actors. This sequential processing constraint is to preserve atomicity; if multiple tasks need to run in parallel, the programmer multiply instantiates the same task class multiple times, and this is illustrated by the twenty `Mapper` task objects in the example.

Any **shared** classes must be explicitly declared, and the exclamation mark (!) also must be used in the sym-

```
task class Mapper {
  void map(Loader ul,Reducer r) {
    WorkUnit wu = ul !->loadWorkUnit();
    r !->reduce(wu.work());
  }
}
shared task class Reducer {
  int sum = 0;
  Counter toReduce;
  void Reducer(CtrFactory cf)
    { toReduce = cf !.newCtr(); }
  void reduce(int rt) {
    sum = sum + rt;
    toReduce.dec();
    if(toReduce.val() == 0)
      { ...output sum ...}
  }
}
shared task class Loader {
  Counter toLoad;
  Loader(CtrFactory cf) {
    toLoad = cf !.newCtr();
  }
  WorkUnit loadWorkUnit() {
    toLoad.dec();
    return new WorkUnit(this);
  }
}
class WorkUnit {
  Loader l;
  WorkUnit(Loader l) { l = l; }
  int work()
    { ...return result ... }
}
shared class CtrFactory {
  int i;
  CtrFactory(int i) { i = i; }
  Counter newCtr()
    {return new Counter(i); }
}
class Counter {
  int v = 0;
  Counter(int v) { v = v; }
  void dec() { v--; }
  int val() {return v; }
}
```

```
task class Main {
  void main() {
    int NUM = 20;
    CtrFactory cf = new CtrFactory(NUM);
    r = new Reducer(cf);
    ul = new Loader(cf);
    for(i=1; i <= NUM; i++) {
      Mapper m = new Mapper();
      m ->map(ul,r);
} } }
```

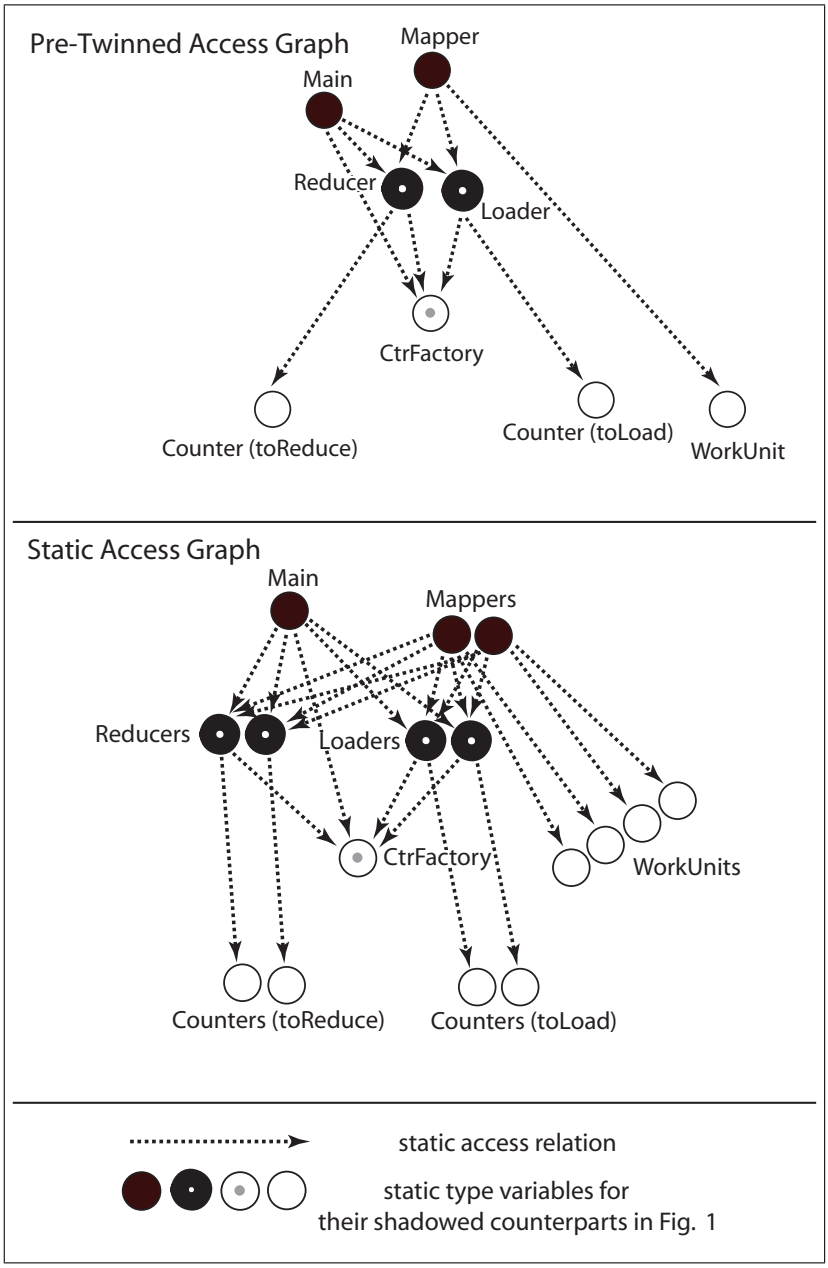

Figure 2. A Simplified Map-Reduce Example

Access \ Execution	atomicity zone entry (**task** $\in \mu$)	resource (**task** $\notin \mu$)
shared (**shared** $\in \mu$)	$\mu =$ **shared task** (shared task object `o3`)	$\mu =$ **shared** (shared ordinary object `o4`)
isolated (**shared** $\notin \mu$)	$\mu =$ **task** (non-shared task object `o2`)	$\mu = \epsilon$ (non-shared ordinary object `o1`)

messaging	what it is	why you should use it
`o1 .m(v)`	intra-task messaging	promotes mutual exclusion and atomicity
`o2 ->m(v)`	task creation	promotes parallelism by starting up a new thread
`o3 !->m(v)`	shared "service"	allows for atomicity-breaking sharing; promotes parallelism with early free
`o4 !.m(v)`	shared "data"	allows for atomicity-preserving sharing

Figure 3. Four Kinds of Objects, Four Kinds of Messaging

bol for sending messages to these objects so that sharing is highlighted in the source code. In the example, both `Reducer` and `Loader` are shared task objects: in particular, all twenty `Mapper` objects share a single `Reducer` object to sum up the numerical results, by the invocation `r !-> reduce(wu.work())` one by one. Shared task messaging does not run in parallel with the invoker – $o!{\to}\mathsf{m}(v)$ is synchronous and blocking and does not spawn a thread. This point can be made clear by looking at the `ul !-> loadWorkUnit()` expression – the `Mapper` object has to wait for the return of the `WorkUnit`. What makes a shared task a "task" is its ability to build its own atomicity zone: the shared task maintains its own objects and frees them all when the shared task ends, *i.e.* the method returns. This nature of shared tasks helps improve system performance, by not holding onto objects for too long, and makes rendezvous between two "live" tasks possible.

Shared ordinary objects, such as `CtrFactory`, can only be accessed over the lifetime of one task at a time, and as such will never break the atomicity of a task. However, they may be accessed by one task first, and then accessed by another when the first task has completed. As a result, shared ordinary objects must be dynamically monitored. Shared task objects and shared ordinary objects are related, but their effects on atomicity and parallelism are clearly different.

Fig. 3 summarizes the four kinds of objects and their messaging. To represent four possibilities, we use the presence or absence of two modifiers, **task** and **shared**; the first differentiates the execution policy (a task or not), and the second distinguishes the access policy (shared or non-shared). For convenience, we will equivalently view μ as consisting of set of 0-2 keywords. Thus for instance, **task** $\notin \mu$ matches either case in the right column of the first table.

2.2 Static Enforcement of Atomicity

Our previous Coqa compiler [LLS08] implemented atomicity solely with locks, and we proved that a running task is atomic when it locks every (ordinary) object it accesses; if a task attempts to access an object which is already locked by another task, it blocks until the locking task execution is completed.

The main property of Task Types is that non-shared ordinary objects are provably isolated inside at most one runtime task *throughout their lifetime*, and thus need no mechanism to support mutually exclusive access at runtime. The main goal of the type system is to prove such isolation indeed holds at compile-time, and so the run-time lock monitoring can be removed. We now give a high-level description of how limited, safe sharing of ordinary objects is supported in the type system. A key structure involved in typechecking is the *static access graph*, a static directed graph with edges for object access via field read/write or being sent a message. Since some data sharing needs to be supported, the static access graph is not a strict hierarchy; in particular, nodes representing shared task objects are sharing points. In the MapReduce example, `Loader` and `Reducer` for example share a `CtrFactory`.

The MapReduce static access graph is illustrated in a box inside Fig. 2. The graph generated by the type system is the bottom one labeled "Static Access Graph." For the purpose of presenting basic ideas however, let us first focus on a simplified version of that graph, labeled "Pre-Twinned Access Graph" in the Figure. Over this graph we define an *access path* to be a path on this graph from a non-shared task object to a non-shared ordinary object; then we say a *cut vertex exists* in the graph for some non-shared ordinary object `o` iff all distinct access paths to `o` in fact go through a single "cut" node in the static access graph. The name of this property comes from a graph-theoretic property of this invariant, and is formalized as predicate $cutExists$ in Sec. 3.5. To see a potential violation of isolation, suppose we removed the **shared task** modifier from class `Loader` (and changed all `!->` symbols to . for messaging to its instances). The resulting program is obviously troublesome at runtime – different `Mapper` instances would race to mutate field `toLoad` in class `Loader`, violating its mutual exclusive access and invalidating our atomicity model. Such a bug would be caught by our type system because there is no cut

vertex for access paths of `Loader`, in this case from `Main` and from `Mapper`.

Note that the typechecking described above subsumes aggregate locking, *i.e.* if a large data structure needs to be lock protected, there should not be a need to lock each and every element. To see how this can be supported, observe that when the cut vertex above is a shared ordinary object, all non-shared ordinary objects can "hide" behind it, and the type system still typechecks.

The example above also shows the need for a polymorphic type system. Two different `Counter`'s, `toLoad` and `toReduce`, are created there by two invocations of the factory method `newCtr`. Even though there is only one **new** `Counter` statement in the program, two instances are created at runtime. Task Types aim to be maximally expressive in this case, and use context-sensitivity to give unique typings to each counter, as is reflected in the diagram in Fig. 2. The form of context-sensitivity employed in Task Types follows [WS01; EGH94; MRR05; WL04].

We now summarize why the MapReduce example typechecks. The `Mapper` `map` method can freely access its work unit `wu`; it was created by the `UnitsLoader` and thus the object is passed across a task boundary, but the only task with access to it is the particular `mapper` task itself; this means the `mapper` is the cut vertex, *i.e.* the *cutExists* typing predicate holds for `wu`. The `reducer`, `ul`, and all the `mapper`s are tasks and so can be shared freely.

Task Twinning We now explain why the "Pre-Twinned Access Graph" is not good enough for preserving isolation of non-shared ordinary objects. Observe in that graph that only one node is created for `Mapper`, even though their might be 20 `Mapper`'s at run time. The fundamental problem here is static approaches have to finitely approximate the in principle unbounded tasks that arise in the presence of recursion. Since different runtime contexts must share static representations for the analysis to remain finite, we need to make sure this approximation will not introduce errors, and it does in fact introduce some very subtle complications. Suppose we added an additional field called `secretShare` of type `WorkUnit` to class `Loader`, and we changed the **return** statement at the end of that method to **return** `secretShare`. It is not hard to see this is a problem program as all `Mapper` instances would be sharing the same `WorkUnit` stored in `secretShare`; furthermore, the cut vertex predicate above would be unable to detect this problem if we chose to use the "Pre-Twinned Access Graph" for analysis. The root of this problem is that the type system as described up to now created only one instantiation of `Mapper` to model the many created at runtime, and that approximation does not soundly model sharing between the different `Mapper` instances at runtime.

To address this case we invent a technique called *task twinning*: we make *two* static instances instead of a single task instance, which here means two `Mapper`'s, two `Loader`'s and two `Reducer`'s. The type system in fact produces the graph at the bottom of Fig. 2. This technique intuitively captures the fact that every program point for task instantiation potentially may lead to more than one task instantiation at run time, so it directly uses two distinct static type variables to split all possible runtime objects into two subsets. Combining this with our polymorphic treatment of method invocations, each object instantiated inside the scope of twinned task objects is also multiplied in the graph – there are two `Counter` instances for `toReduce` (one for each twinned `Reducer`), two `Counter` instances for `toLoad` (one for each twinned `Loader`), and four `WorkUnit` instances (from each twinned `Mapper` invoking `loadWorkUnit` of each twinned `Loader`).

Our formal system is constructed to handle the case where every line of code may potentially be involved in recursion, and hence every task object needs to be twinned. In practice, mechanical application of twinning is not always necessary. For instance, the `Reducer` and `Loader` objects are in fact instantiated in the bootstrapping `main` method, so the system would still be sound even if our static access graph had not twinned them. We treat simplifications in this flavor as implementation-level optimizations, and do not model them in the theory, except that the bootstrapping task `Main` is trivially singular, so we do not twin it.

Now, looking at the actual "Static Access Graph" the type system generates in Fig. 2 we see that the cut vertex property still holds for every non-shared ordinary object in the graph. If, however, the program incorporated the `secretShare` modification above, there would be two `WorkUnit` instances in the full static graph, and each `WorkUnit` would be accessed by both `Loader`'s in addition to both `Mapper`'s. Such a graph would violate the cut vertex condition since there is an access path to each `WorkUnit` from each `Mapper` and there is no cut vertex dominating both `Mapper`'s.

Properties For the simplicity of the presentation, the static access graph drawn in Fig. 2 does not distinguish between read and write access. Our formal type system is more refined, and is constructed using a non-exclusive-read-exclusive-write principle. Additionally, we can prove there is no atomicity violation when a field write in constructors are treated as non-exclusive. Immutable objects consequently can be freely shared.

In Section 4 we prove type soundness and decidability of type inference for Task Types. We also will show how Task Types preserve a non-shared memory model for all objects declared as non-shared and ordinary. Since these objects do not need lock protection to preserve atomicity, Task Types will have the same pervasive atomicity properties as did Coqa but with a lower run-time cost. Pervasive atomicity also provably subsumes race condition freedom [LLS08], so programmers also will be spared from race conditions –

for example, in the `reduce` method, `sum = sum + rt` is guaranteed to compute predictable results.

2.3 Programmability

Effective Task Types programming requires the programmer to develop a careful plan for object sharing. Declaring as many objects as possible to be non-shared ordinary objects, $\mu = \epsilon$, will increase run-time performance, and at the same time increase the size of atomic zones, so the goal of the programmer is simply to maximize the non-shared ordinary classes but still allow the program to typecheck. Objects that cannot typecheck as unshared ordinary should be typechecked as shared ordinary ($\mu =$ **shared**) or shared task ($\mu =$ **shared** ->). Fortunately, problems with typing a non-shared ordinary object are always solved by hoisting to a shared one since the type system imposes no additional constraints on shared objects. So, this is always available as a last resort; the key is to hoist up as few objects as possible, to obtain the largest zones of atomicity and highest performance.

Both shared ordinary and shared task objects are locked for mutual exclusion, but shared ordinary objects do not add new atomicity break points into tasks and are also more appropriate when a task makes continual use of an object. So, **shared** is the preference over **shared task** if the object really "should be" local to the task, but the type system is too weak to realize that. A good situation to use shared tasks is when the class wraps up a relatively independent "service," so that when the service is completed, "partial victory" can be declared. Examples include `Reducer` and `Loader` in the MapReduce example.

We believe this additional programmer focus on object sharing is time well-spent if the final goal is production of reliable software. It is our belief that the vast majority of objects in a vast range of applications are ordinary objects not shared across tasks, and they can be programmed normally, so additional planning is required only on the shared object portions of the code.

We don't expect this paradigm will extend to every line of every single application – just as there is a rare need to escape to C in Java there will be rare cases where for efficiency the Task Type framework needs to be bypassed. One such example is data structure implementations that use hand-over-hand locking.

We next describe a benchmark program with significant contention. Intuitively, this is precisely the category of programs one would expect Task Types to be uncomfortable with, so it will be more of a stress test of our language.

2.3.1 The PuzzleSolver Benchmark

In Section 5 we discuss the implementation and a few benchmarks of its performance. One benchmark, PuzzleSolver, solves a generalized version of the 15-puzzle, the famous 4x4 sliding puzzle where numbers 1-15 must be slid to arrive at numerical order. The primary worker tasks of the program are n `SolverTask`s, each of which in parallel takes existing legal gameplay move sequences (`PuzzleMove`'s) from a central work queue (`PuzzleTaskQueue`) and puts back on the queue all gameplay moves extending the play sequence grabbed by one game step, if any exist. All worker tasks loop on this activity until there are no more move sequences to extend on the shared queue. In order to avoid repeating play, all visited board states (`PuzzlePositions`) are centrally logged in a `PositionSeen` object so no worker `SolverTask` will add an already investigated position to the `PuzzleTaskQueue`. Lastly there are classes `Block` and `Puzzle`, the former containing the ID and size of a block (the generalization supports $n \times m$ blocks), and the latter which checks for legal puzzle moves.

The primary interest for Task Types is what sharing declarations are placed on the classes. `SolverTask` is obviously a task and therefore declared a **task**. Classes `PuzzleTaskQueue` and `PositionSeen` are data structures that must be shared by all tasks and so logically are declared as **shared task** which also implicitly guarantees their mutually exclusive access. Classes `Block` and `Puzzle` are not changed after they are constructed, and even though they are shared across worker tasks it is still possible to declare them as ordinary unshared classes since they are known to be immutable – this is an example of the practical usefulness of the non-exclusive read we support in the type system, as was discussed in the previous subsection.

The only class which the typechecker is less than optimal on is `PuzzlePosition` which must be declared **shared task** in order to typecheck. The `PuzzlePosition` objects are used by `SolverTask`'s to replay `PuzzleMove` sequences on the grid, and each worker task has its own private `PuzzlePosition` which logically is unshared. So, it sounds like `PuzzlePosition` could be declared unshared ordinary, but there is a subtle problem: there is also a distinguished `PuzzlePosition` called `initial_position` which holds the initial board configuration and is set up by the main task that launches the worker tasks. This `initial_position` is passed to each worker task so they can set up their own `PuzzlePosition` (by copying `initial_position`), but even though they are only reading it, the main task had written to `initial_position` when the data was read from a file and so it is considered owned by the main task and so cannot be read by the worker tasks. If Task Types were to support a per-instance declaration of sharing policy, the private per-worker-task `PuzzlePosition` objects could be declared unshared ordinary, and only the `initial_position` would need to be a **shared task**. A flow-sensitive typing would also support this since `initial_position` is not mutated after it is read from the file. And, a call-by-copy syntax as discussed in Sec. 7 would solve this problem as well. For simplicity we elected not to include any of these three extensions in the

current design, but conceptually one will likely be needed in a future extension to increase expressiveness.

3. The Formal System

3.1 Abstract Syntax

The core syntax of our language is formalized in Fig. 4, where notation $\overline{X}$ is used to represent a sequence of X's. As a convention, metavariable c is used for class names, m for method names, f for field names, and x for variable names. Special class names include $\texttt{Object}_{\mu}$, the root classes for the inheritance hierarchy, one for each class modifier μ. The bootstrapping code is located inside the body of class `Main`'s no-argument method `main`. The class directly inherits from class $\texttt{Object}_{\textbf{task}}$ with no additional fields. We encode unit/void type via class name $\texttt{Unit} \stackrel{\text{def}}{=} \texttt{Object}_{\epsilon}$. Default constructors are supported: a constructor for class c is syntactically a method in c with method name also c. To make the formalism more uniform, we assume all constructors always return **this** as the return value. In this formalism-friendly syntax, field read/write expressions are annotated with a *scope modifier* ζ, denoting whether the expression is lexically scoped in a constructor ($\zeta = \mathsf{cons}$) or not ($\zeta = \mathsf{reg}$).

Parametric Polymorphism Support The formal syntax differs from the programmer syntax in that several program annotations are included to help streamline the formal presentation. Programmer syntax **new** $\mathsf{c}(e)$ is formally expressed as $\textbf{new}_{\mathcal{A}}\ \mathsf{c}(e)$ when class c has modifier $\mu \notin$ **task** (*i.e.* "resource" object instantiation), and as $\textbf{new}_{\mathcal{A}_1,\mathcal{A}_2}\ \mathsf{c}(e)$ otherwise (*i.e.* "concurrency unit" object instantiation). The associated subscript $\mathcal{A}$ serves to distinguish different instantiation sites. Structurally, each $\mathcal{A}$ represents a list of type variables α (of set $\mathbb{STV}$). The first type variable in $\mathcal{A}$ represents the object instantiated at the specific program point where the expression occurs, and each of the rest represents an object that can be stored in a field of the instantiated object. For any two distinct **new** expressions in the source code, we require their respective $\mathcal{A}$'s have distinct elements. The $\textbf{new}_{\mathcal{A}_1,\mathcal{A}_2}\ \mathsf{c}(e)$ form for concurrency unit object instantiation is present to realize our need for task twinning to model self-sharing, as was outlined in Sec. 2.2. To support context sensitivity, each call site is differentiated by associating a singleton list $[\alpha]$ with each messaging expression, $e*_{[\alpha]}\mathsf{m}(e')$. For any two distinct messaging expressions in the source code, we require their respective $[\alpha]$'s to be distinct. As examples, the several expressions in the `Loader` class of Fig. 2 is automatically annotated as follows:

$$\texttt{cf!.newCtr}() \text{ as } \texttt{cf!.}_{[\alpha_{100}]}\texttt{newCtr}(\textbf{null})$$
$$\texttt{toLoad.dec}() \text{ as } \texttt{toLoad.}_{[\alpha_{101}]}\texttt{dec}(\textbf{null})$$
$$\textbf{new}\ \texttt{WorkUnit}(\textbf{this}) \text{ as } \textbf{new}_{[\alpha_{102},\alpha_{103}]}\ \texttt{WorkUnit}(\textbf{this})$$

For each class definition μ **class** c **extends** c' $\{F\ M\}$ inside codebase C and each method definition $\mathsf{c}'\ \mathsf{m}(\mathsf{c}''\ x)\{e\}$ inside M, we define function $mbody(\pi) = x.e$ to return the method body and deterministic function $mtype(\pi) = \forall\mathcal{A}.(\mathsf{c}'' \to \mathsf{c}')$ to return the signature for method index $\pi = \langle \mathsf{c}; \mathsf{m}\rangle$. Both functions are implicitly parameterized by the fixed codebase C. The type variable list above $\mathcal{A} = [\alpha'', \alpha']$ includes two distinct type variables representing the argument and the return value respectively. To illustrate, selected methods of Fig. 2 have the following signatures:

$$mtype(\langle \texttt{Loader}; \texttt{Loader}\rangle) = \forall[\alpha_{200}, \alpha_{201}].\ (\texttt{CtrFactory} \to \texttt{Loader})$$
$$mtype(\langle \texttt{Loader}; \texttt{loadWorkUnit}\rangle) = \forall[\alpha_{202}, \alpha_{203}].\ (\texttt{Unit} \to \texttt{WorkUnit})$$
$$mtype(\langle \texttt{CtrFactor}; \texttt{newCtr}\rangle) = \forall[\alpha_{204}, \alpha_{205}].\ (\texttt{Unit} \to \texttt{Counter})$$
$$mtype(\langle \texttt{Counter}; \texttt{dec}\rangle) = \forall[\alpha_{206}, \alpha_{207}].\ (\texttt{Unit} \to \texttt{Unit})$$

To model inheritance, we further define $mtype(\langle \mathsf{c}_0; \mathsf{m}\rangle) = mtype(\langle \mathsf{c}; \mathsf{m}\rangle)$ for any c_0 a subclass of c. This is the only case where *mtype* may compute overlapping $\mathcal{A}$'s for different π's, *i.e.* for all other cases, if $mtype(\pi_1)$ and $mtype(\pi_2)$ compute $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively and $\pi_1 \neq \pi_2$, $\mathcal{A}_1$ and $\mathcal{A}_2$ have disjoint elements. $mtype(\langle \texttt{Main}; \texttt{main}\rangle) = \forall\mathcal{A}.(\texttt{Unit} \to \texttt{Unit})$ for some (uninteresting) $\mathcal{A}$.

Auxiliary Definitions For class μ **class** c **extends** c' $\{F\ M\}$ in C, we further define $modifier(\mathsf{c}) = \mu$ and $supers(\mathsf{c}) = \{\mathsf{c}\} \cup supers(\mathsf{c}')$. Here we assume $\texttt{Object}_{\mu} \neq \mathsf{c}$ and there is no cycle on the inheritance chain induced by C. In addition, we define $modifier(\texttt{Object}_{\mu}) = \mu$ and $supers(\texttt{Object}_{\mu}) = \texttt{Object}_{\mu}$. These functions are also implicitly parameterized by C.

We now define standard mathematical notation used in this paper. $[\,]$ is used to represent an empty sequence, and $x : [x_1, \ldots, x_n] \stackrel{\text{def}}{=} [x, x_1, \ldots, x_n]$, and $|[x_1, \ldots, x_n]| = n$. When ordering of a sequence does not matter, we liberally consider them convertible to their unordered counterpart (a set) and standard set operators such as $\in$ and $\subseteq$ will be used on it. A special kind of sequence, a mapping sequence, is denoted as $\overline{x \mapsto y}$ and defined as $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ for some unspecified length n. Given ι being the mapping sequence above, $\mathrm{dom}(\iota) \stackrel{\text{def}}{=} \{x_1, \ldots x_n\}$ and $\mathrm{range}(\iota) \stackrel{\text{def}}{=} \{y_1, \ldots y_n\}$. We write $\iota[x \mapsto y]$ for mapping update: ι and $\iota[x \mapsto y]$ are identical except that $\iota[x \mapsto y]$ maps x to y. Updateable mapping concatenation $\rhd$ is defined as $\iota_1 \rhd \iota \stackrel{\text{def}}{=} \iota_1[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$. We also write $\iota_1 \rhd \iota$ as $\iota_1 \uplus \iota$, except the latter function requires the precondition of $\mathrm{dom}(\iota_1) \cap \mathrm{dom}(\iota) = \emptyset$. Given two sequences, a "zip" like operator $\longmapsto$ produces a mapping sequence: $[x_1, \ldots, x_n] \longmapsto [y_1, \ldots, y_n] \stackrel{\text{def}}{=} [x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$.

3.2 A Bird's Eye View of the Type System

Task Types are a constraint-based type system with (T-Program) as the top-level typing rule:

$$
\begin{array}{rcll}
C & ::= & \overline{\mathsf{c} \mapsto \mu\ \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}'\ \{F\ M\}} & \textit{classes} \\
\mu & ::= & \epsilon \mid \textbf{shared} \mid \textbf{task} \mid \textbf{shared task} & \textit{class modifier} \\
F & ::= & \overline{\mathsf{c}\ \mathsf{f}} & \textit{fields} \\
M & ::= & \overline{\mathsf{m} \mapsto \mathsf{c}\ \mathsf{m}(\mathsf{c}'\ x)\{e\}} & \textit{methods} \\
e & ::= & x \mid \textbf{null} \mid \textbf{this} \mid \mathsf{f}^{\zeta} \mid \mathsf{f}^{\zeta}\textbf{:=}e \mid (\mathsf{c})e \mid e *_{[\alpha]} \mathsf{m}(e) \mid \textbf{new}_{\mathcal{A}}\ \mathsf{c}(e) \mid \textbf{new}_{\mathcal{A},\mathcal{A}'}\ \mathsf{c}(e) & \textit{expression} \\
* & ::= & \texttt{!->} \mid \texttt{!.} \mid \texttt{->} \mid \texttt{.} & \textit{method invocation symbol} \\
\alpha & \in & \mathbb{STV} & \textit{annotated type variable} \\
\mathcal{A} & ::= & \overline{\alpha} & \textit{α sequence} \\
\zeta & ::= & \mathsf{reg} \mid \mathsf{cons} & \textit{scope modifier} \\
\pi & ::= & \langle \mathsf{c}; \mathsf{m} \rangle & \textit{method index}
\end{array}
$$

Figure 4. Abstract Syntax

$$
\text{(T-Program)}\ \frac{\vdash_{\text{cls}} C(\mathsf{c}_i) \backslash \mathcal{C}(\mathsf{c}_i) \text{ for all } \mathsf{c}_i \in \text{dom}(C) \qquad WF(\circlearrowleft \mathcal{C})}{\vdash_{\text{p}} C : \mathcal{C}}
$$

This rule defines typechecking in two phases:

- First, constraints are collected for each method of each class modularly, and inconsistencies are detected as early as possible. The per-class typing rule $\vdash_{\text{cls}}$ and the related expression typing rules are defined in Fig. 5. At the end of this phase, constraints are stored in a per-class, per-method fashion, in constraint store $\mathcal{C}$.
- Second, a closure phase propagates inter-procedural information. The resulting constraint closure is computed by the $\circlearrowleft$ function, as defined in Sec. 3.4. Afterwards the static access graph represented in the closure is checked by a simple $WF()$ function in Sec. 3.5, determining whether static isolation for non-shared ordinary objects holds.

We favor a phased definition to be more realistic with object-oriented languages. Today's OO languages often rely on a modular phase to find as many bugs as possible (either via source code compilation or bytecode verification), and delays as few as possible non-modular constraint solving to dynamic class loading time (such as those related to Java subtyping [LB98]). Presenting Task Types in phases *de facto* describes how it can be constructed in a language with dynamic class loading. Notably, extracting a modular phase out of an inter-procedural algorithm is no trivial task for object-oriented programs, as the latter are fundamentally mutually recursive: class c_1 might contain expression **new** $\mathsf{c}_2(e_1)$ whereas class c_2 might contain expression **new** $\mathsf{c}_1(e_2)$; or class c_1's method m_1 might invoke c_2's m_2 which in turn invokes c_1's m_1. By constructing an explicit formal definition of the modular phase here, we gain confidence in the ability to construct a practical and decidable typechecking process.

3.3 Modular Class Typing

Expressions are typed via the $\vdash$ rules of Fig. 5. Classes are typed via judgment $\vdash_{\text{cls}}$, and method bodies via $\vdash_{\text{m}}$. All typing rules are implicitly parameterized by the fixed codebase C. Types are always of the form $\mathsf{c}@\alpha$, with c the class name analogous to Java's object type, and α the type variable associated with a specific object instance, needed for the polymorphic type system. Typing environment Γ maps variables x, field names f, and **this** to their types. We delay the discussion of class-indexed constraint store $\mathcal{C}$ and method-indexed constraint store $\mathcal{M}$ until judgments $\vdash_{\text{cls}}$ and $\vdash_{\text{m}}$ are explained. Per-method constraint store $\mathcal{K}$ is a set of constraints collected for each method body. Each constraint is represented by metavariable κ. The meanings of specific constraints will be clarified later, but in general a $\leq$ constraint is intuitively a constraint recording flow, a $\overset{\theta}{\dashrightarrow}$ constraint is intuitively a constraint recording access. Special variables $\hat{\alpha}$ are used only in $\leq$ constraints.

Access Constraints A main goal of our type system is to generate the static access graph that was informally described in Section 2. The nodes of such a graph are type variables representing individual objects. The edges, as collected in the modular type checking phase, are called *access constraints*. They are of the form $\texttt{thost} \overset{\theta}{\dashrightarrow} \alpha$, meaning an object represented by type variable α is accessed. Constraint label θ, also called *access mode*, ranges over three values: when $\theta = \mathsf{R}$, the constraint asserts that object α is non-exclusively "read"; when $\theta = \mathsf{W}$, α is exclusively "written"; when $\theta = \mathsf{T}$, α is the entry/facade object for a "protected zone" of sharing. In our type system, $\overset{\mathsf{T}}{\dashrightarrow}$ constraints are generated when α is a shared task object or a shared ordinary object. Observe that both kinds of objects are dynamically protected upon entry, and the objects completely hiding behind them need not be dynamically protected. The $\overset{\mathsf{T}}{\dashrightarrow}$ constraints make the static access graph aware of the dynamic protection points and reason accordingly.

Placeholder type variable `thost` denotes the "accessor." Intuitively, the accessor should have been a type variable

$$\text{(T-Read)}\ \frac{\Gamma(\mathsf{f}) = \mathsf{c}@\alpha}{\Gamma \vdash \mathsf{f}^{\zeta} : \mathsf{c}@\alpha \backslash aC(\mathsf{R}, \Gamma(\textbf{this}), \zeta)} \qquad \text{(T-Write)}\ \frac{\Gamma(\mathsf{f}) = \mathsf{c}@\alpha_1 \qquad \Gamma \vdash e : \mathsf{c}@\alpha_2 \backslash \mathcal{K}}{\Gamma \vdash \mathsf{f}^{\zeta} := e : \mathsf{c}@\alpha_1 \backslash \mathcal{K} \cup \{\alpha_2 \leq \alpha_1\} \cup aC(\mathsf{W}, \Gamma(\textbf{this}), \zeta)}$$

$$\text{(T-Msg)}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau \backslash \mathcal{K} \qquad \tau = \mathsf{c}@\alpha_0 \qquad \mathit{mtype}(\langle \mathsf{c}; \mathsf{m} \rangle) = \forall \mathcal{A}.(\mathsf{c}_1 \to \mathsf{c}_2) \\ \Gamma \vdash e' : \mathsf{c}_1@\alpha' \backslash \mathcal{K}' \qquad * \text{ matches } \mathit{modifier}(\mathsf{c}) \text{ returns } \mathsf{c}_2 \qquad \kappa = [\alpha]^{\alpha_0, \mathsf{m}, \alpha'} \qquad \text{any } \zeta\end{array}}{\Gamma \vdash e *_{[\alpha]} \mathsf{m}(e') : \mathsf{c}_2@\alpha \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\kappa\} \cup aC(\mathsf{T}, \tau, \zeta)}$$

$$\text{(T-New)}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{c}_0@\alpha_0 \backslash \mathcal{K}_0 \\ \mathcal{A} = [\alpha, \ldots] \qquad \mathit{mtype}(\langle \mathsf{c}; \mathsf{c} \rangle) = \forall \mathcal{A}'.(\mathsf{c}_0 \to \mathsf{c}) \qquad \kappa = [\alpha]^{\alpha, \mathsf{c}, \alpha_0} \qquad \tau = \mathsf{c}@\alpha \qquad \text{any } \zeta\end{array}}{\Gamma \vdash \textbf{new}_{\mathcal{A}}\ \mathsf{c}(e) : \tau \backslash \mathcal{K}_0 \cup \{\mathcal{A}^{\mathsf{c}} \leq \alpha, \kappa\} \cup aC(\mathsf{T}, \tau, \zeta)}$$

$$\text{(T-NewTask)}\ \frac{\Gamma \vdash \textbf{new}_{\mathcal{A}_1}\ \mathsf{c}(e) : \mathsf{c}@\alpha_1 \backslash \mathcal{K}_1 \qquad \Gamma \vdash \textbf{new}_{\mathcal{A}_2}\ \mathsf{c}(e) : \mathsf{c}@\alpha_2 \backslash \mathcal{K}_2 \qquad \textbf{task} \in \mathit{modifier}(\mathsf{c})}{\Gamma \vdash \textbf{new}_{\mathcal{A}_1, \mathcal{A}_2}\ \mathsf{c}(e) : \mathsf{c}@\alpha_1 \backslash \mathcal{K}_1 \cup \mathcal{K}_2 \cup \{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_1\}}$$

$$\text{(T-Sub)}\ \frac{\Gamma \vdash e : \mathsf{c}_0@\alpha \backslash \mathcal{K} \qquad \mathsf{c} \in \mathit{supers}(\mathsf{c}_0)}{\Gamma \vdash e : \mathsf{c}@\alpha \backslash \mathcal{K}} \qquad \text{(T-Cast)}\ \frac{\Gamma \vdash e : \mathsf{c}_0@\alpha \backslash \mathcal{K}}{\Gamma \vdash (\mathsf{c})\ e : \mathsf{c}@\alpha \backslash \mathcal{K}} \qquad \text{(T-Var)}\ \Gamma \vdash x : \Gamma(x) \backslash \emptyset$$

$$\text{(T-This)}\ \Gamma \vdash \textbf{this} : \Gamma(\textbf{this}) \backslash \emptyset \qquad \text{(T-Null)}\ \Gamma \vdash \textbf{null} : \tau \backslash \emptyset$$

$$\text{(T-Cls)}\ \frac{\begin{array}{c}\vdash_{\text{cls}} \mu\ \textbf{class}\ \mathsf{c}_0 \ldots \backslash \mathcal{M} \\ \mathit{fields}(\mathsf{c}) = \forall \mathcal{A}.\Gamma \qquad \Gamma \vdash_{\text{m}} \mathit{mbody}(\pi_j) : \mathit{mtype}(\pi_j) \backslash \forall \mathcal{A}_j.\forall \mathcal{S}_j.\mathcal{K}_j \text{ for all } \mathsf{m}_j \in \text{dom}(M), \pi_j = \langle \mathsf{c}; \mathsf{m}_j \rangle\end{array}}{\vdash_{\text{cls}} \mu\ \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}_0\ \{F\ M\} \backslash \forall \mathcal{A}.(\mathcal{M} \rhd \overline{\mathsf{m}_j \mapsto \forall \mathcal{A}_j.\forall \mathcal{S}_j.\mathcal{K}_j})}$$

$$\text{(T-ClsTop)}\ \frac{\mathit{fields}(\texttt{Object}_{\mu}) = \forall \mathcal{A}.\Gamma}{\vdash_{\text{cls}} \mu\ \textbf{class}\ \texttt{Object}_{\mu} \backslash \forall \mathcal{A}.[\,]}$$

$$\text{(T-Md)}\ \frac{\Gamma \uplus [x \mapsto \mathsf{c}_1@\alpha_1] \vdash\ e : \mathsf{c}_2@\alpha_3 \backslash \mathcal{K} \qquad \mathcal{A} = [\alpha_1, \alpha_2] \qquad \mathcal{S} = \mathit{labels}(e)}{\Gamma \vdash_{\text{m}} x.e : \forall \mathcal{A}.(\mathsf{c}_1 \to \mathsf{c}_2) \backslash \forall \mathcal{A}.\forall \mathcal{S}.(\mathcal{K} \cup \{\alpha_3 \leq \alpha_2\})}$$

.	matches	ϵ	returns	c
!.	matches	**shared**	returns	c
!->	matches	**shared task**	returns	c
->	matches	**task**	returns	`Unit`

τ	$::=$	$\mathsf{c}@\alpha$	*types*
Γ	$::=$	$\overline{\mathsf{t} \mapsto \tau}$	*typing environment*
t	$::=$	$x \mid \textbf{this} \mid \mathsf{f}$	*environment variable*
$\mathcal{S}$	$::=$	$\overline{\mathcal{A}}$	*$\mathcal{A}$ sequence*
$\mathcal{K}$	$::=$	$\overline{\kappa}$	*per-method constraint store*
κ	$::=$	$\hat{\alpha} \leq \alpha \mid \mathcal{A}^{\alpha, \mathsf{m}, \alpha'} \mid \texttt{thost} \overset{\theta}{\dashrightarrow} \alpha$	*constraint*
$\hat{\alpha}$	$::=$	$\alpha \mid \mathcal{A}^{\mathsf{c}}$	*flow element*
θ	$::=$	$\mathsf{T} \mid \mathsf{R} \mid \mathsf{W}$	*access mode*
$\mathcal{C}$	$::=$	$\overline{\mathsf{c} \mapsto \forall \mathcal{A}.\mathcal{M}}$	*class-indexed constraint store*
$\mathcal{M}$	$::=$	$\overline{\mathsf{m} \mapsto \forall \mathcal{A}.\forall \mathcal{S}.\mathcal{K}}$	*method-indexed constraint store*

$$aC(\theta, \mathsf{c}@\alpha, \zeta) \overset{\text{def}}{=} \begin{cases} \{\texttt{thost} \overset{\theta}{\dashrightarrow} \alpha\} & \text{if } \mathit{modifier}(\mathsf{c}) = \epsilon, \theta = \mathsf{R} \text{ or } \mathsf{W}, \zeta = \mathsf{reg} \\ & \text{or } \mathit{modifier}(\mathsf{c}) \in \textbf{shared}, \theta = \mathsf{T} \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 5. Typing Rules

representing the entry facade object for a protected zone. A placeholder is used because no concrete type variable naming this object is known during the modular type checking phase. Consider for example the case when the `toLoad Counter` object is written by object `Loader` because `toLoad`'s field `v` is written in method `dec`. When class `Counter` is typed, we cannot directly determine in what "protected zone" method `dec` is invoked from – one needs to backtrack on all possible call paths to find the first non-ordinary object. We therefore put `thost` here and rely on the closure phase (Sec. 3.4) to instantiate it; such a scheme is common in inter-procedural analyses with a modular phase.

Access constraints are collected by typing rules dependent on $aC(\theta, \tau, \zeta)$. This convenience function collects θ constraints when the type of the accessed entity is τ and the scope of access is ζ. The function is defined in Fig. 5. (T-Msg) puts $aC(\mathsf{T}, \tau, \zeta)$ into the constraint set, *i.e.* it collects $\overset{\mathsf{T}}{\dashrightarrow}$ constraints when the message receiver is a shared task object or a shared ordinary object (regardless of the scope). This is consistent with our previous discussion of "protected zones." The same constraint is collected in (T-New) for constructor calls. For accessing non-shared ordinary objects, an access occurs when the field of that object is read or written. Related constraints are collected in (T-Read) and (T-Write) respectively. Observe that when the field read/write happens in a constructor, the access is not recorded as a constraint. This is sound because, when an object is constructed, its reference is not yet created, let alone leaked to another task and accessed by it – such read/write access fundamentally does not lead to atomicity violations.

Expressions with Polymorphic Typing In principle, polymorphic typing behaves rather like let-polymorphism: the type constraints of the polymorphic code – the invokee's method body here – should be "refreshed" and added to the invoker's constraint set. What complicates matters here is the fundamentally recursive nature of OO programs: naively merging the invokee's constraints may lead to non-termination due to an infinite regress of refreshing along a recursive call. It is for this reason our type system in the modular phase only places a *delayed contour marker* in the constraint set, and delays the task of refreshing and merging to the phase of closure. Marker $[\alpha]^{\alpha_0, \mathsf{m}, \alpha'}$ added in (T-Msg) indicates the need to merge (at closure time) the constraints of method m of object α_0, with argument being α' and return value being α. For instance, typing the annotated expression $\texttt{cf!.}_{[\alpha_{100}]}\texttt{newCtr}(\textbf{null})$ in class `Loader` generates the following marker constraint:

$$\mathcal{K}_{\texttt{loader1}} = \{[\alpha_{100}]^{\alpha_{200}, \texttt{newCtr}, \alpha_{100}}\}$$

where α_{200} is the type variable associated with variable `cf` – the latter is the argument of the constructor; its associated type variable is computed by $mtype(\langle \texttt{Loader}; \texttt{Loader} \rangle)$ earlier. The choice of type variables to type **null** is irrelevant; we use α_{100} here. (T-Msg) also contains a predicate "$*$ matches μ returns c", which matches different method invocation symbols ($*$) with class modifiers μ. In addition, it requires that asynchronous top-level task creation has no interesting return values.

In (T-New), the type variable representing the object instantiated by $\textbf{new}_{\mathcal{A}}\mathsf{c}(e)$ is the first element of $\mathcal{A}$, consistent with how $\mathcal{A}$ is constructed in the formal syntax. Since all such annotations include disjoint type variables, **new** expressions at different program points are given different type variables. Rule (T-New) also contains a flow constraint of the form $\mathcal{A}^{\mathsf{c}} \leq \alpha$. This constraint says that instantiation site $\mathcal{A}^{\mathsf{c}}$ flows into type variable α. This constraint, together with the transitivity of $\leq$ as defined in the closure phase, is used to trace back any type variable to its concrete instantiation point(s) – a concrete type analysis scheme essential for languages with aliases. Lastly, a similar delayed contour marker is added to a constructor call. For instance, typing the annotated expression $\textbf{new}_{[\alpha_{102}, \alpha_{103}]}\ \texttt{WorkUnit}(\textbf{this})$ in class `Loader` leads to the following constraints:

$$\mathcal{K}_{\texttt{loader2}} = \{[\alpha_{102}, \alpha_{103}]^{\texttt{WorkUnit}} \leq \alpha_{102}, [\alpha_{102}]^{\alpha_{102}, \texttt{WorkUnit}, \alpha_{300}}\}$$

assuming **this** was given type α_{300}.

(T-NewTask) types expression $\textbf{new}_{\mathcal{A}_1, \mathcal{A}_2}\ \mathsf{c}(e)$, which is used when $modifier(\mathsf{c}) \in \textbf{task}$. This is how task twinning is reflected in the type system: for each statically known task object, the type system instantiates them twice, with $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively.

Other Rules Standard nominal subtyping is supported by (T-Sub). Casting is typed by (T-Cast); we do not single out stupid cast as warnings [IPW99] as this does not affect soundness. (T-Read), (T-Write), (T-This), (T-Var) rely on the typing environment. Given a class μ **class** c **extends** c' $\{F\ M\}$ in codebase C where $F = [\mathsf{c}_1\ \mathsf{f}_1, \ldots, \mathsf{c}_n\ \mathsf{f}_n]$, a typing environment with field and **this** type information is prepared via the following deterministic function:

$$\begin{array}{rcl} fields(\mathsf{c}) & \stackrel{\text{def}}{=} & \forall\mathcal{A}.((\Gamma \uplus [\mathsf{f}_1 \mapsto \mathsf{c}_1@\alpha_1, \ldots \mathsf{f}_n \mapsto \mathsf{c}_n@\alpha_n]) \\ & & \quad \triangleright [\textbf{this} \mapsto \mathsf{c}@\alpha_0]) \\ & \text{if} & fields(\mathsf{c}') = \forall\mathcal{A}'.\Gamma \\ & & \mathcal{A} = \mathcal{A}' \uplus [\alpha_1, \ldots, \alpha_n] \\ & & \Gamma(\textbf{this}) = \mathsf{c}'@\alpha_0, \alpha_1, \ldots, \alpha_n \text{ distinct} \end{array}$$

and the base case is $fields(\texttt{Object}_{\mu}) = \forall[\alpha].[\textbf{this} \mapsto \texttt{Object}_{\mu}@\alpha]$. This function deterministically assigns each field a distinct type variable, as well as assigning one for **this**. The function is implicitly parameterized by the fixed codebase C. It is able to support field inheritance but disallows field shadowing for simplicity. As an example, the function has the following behavior on class `Loader` :

$$fields(\texttt{Loader}) = \forall[\alpha_{300}, \alpha_{301}]. \left[\begin{array}{l} \texttt{toLoad} \mapsto \texttt{Counter}@\alpha_{301} \\ \textbf{this} \mapsto \texttt{Loader}@\alpha_{300} \end{array}\right]$$

To make the parametric nature of constraints more explicit, a class-indexed constraint store $\mathcal{C}$ computed in (T-Cls) and (T-ClsTop) is always of the form $\forall\mathcal{A}.\mathcal{M}$, where $\mathcal{A}$ is computed by the previous *fields* function. A method-indexed constraint store $\mathcal{M}$ computed in (T-Md) is always of the form $\forall\mathcal{A}.\forall\mathcal{S}.\mathcal{K}$, with $\mathcal{A}$ containing the two type variables representing the argument and the return value of the method, and $\mathcal{S}$ being the set of $\mathcal{A}$ labels appearing in the body of the method. $\mathcal{S}$ is computed by the following function:

$$\begin{array}{rcl} \mathit{labels}(x) & \stackrel{\text{def}}{=} & \emptyset \\ \mathit{labels}(\mathbf{new}_{\mathcal{A}}\ \mathsf{c}(e)) & \stackrel{\text{def}}{=} & \{\mathcal{A}\} \cup \mathit{labels}(e) \\ \mathit{labels}(\mathbf{new}_{\mathcal{A}_1,\mathcal{A}_2}\ \mathsf{c}(e)) & \stackrel{\text{def}}{=} & \{\mathcal{A}_1,\mathcal{A}_2\} \cup \mathit{labels}(e) \\ \mathit{labels}(e *_{\mathcal{A}} \mathsf{m}(e')) & \stackrel{\text{def}}{=} & \{\mathcal{A}\} \cup \mathit{labels}(e) \cup \mathit{labels}(e') \\ \mathit{labels}((\mathsf{c})e) & \stackrel{\text{def}}{=} & \mathit{labels}(e) \\ & \cdots & \end{array}$$

We require the type variables computed by *fields*, by *mtype*, and by *labels* to be pairwise disjoint. We now provide a complete picture of all the constraints produced for class `Loader`:

$$\begin{array}{rcl} \mathcal{C}(\texttt{Loader}) & = & \forall[\alpha_{300},\alpha_{301}].\mathcal{M} \\ \mathcal{M}(\texttt{Loader}) & = & \forall[\alpha_{200},\alpha_{201}].\forall[[\alpha_{100}]]. \\ & & \mathcal{K}_{\texttt{loader1}} \cup \{\alpha_{300} \le \alpha_{201}\} \\ & & \quad \cup\{\texttt{thost} \stackrel{\mathsf{T}}{\dashrightarrow} \alpha_{200}\} \\ \mathcal{M}(\texttt{loadWorkUnit}) & = & \forall[\alpha_{202},\alpha_{203}]. \\ & & \forall[[\alpha_{101}],[\alpha_{102},\alpha_{103}]]. \\ & & \mathcal{K}_{\texttt{loader2}} \cup \{[\alpha_{101}]^{\alpha_{301},\mathrm{dec},\alpha_{101}}\} \\ & & \quad \cup\{\alpha_{102} \le \alpha_{203}\} \end{array}$$

3.4 Type Closure

We first define reflexive and transitive binary relation $\Omega \stackrel{\Delta}{\hookrightarrow} \Omega'$ by the proof system in Fig. 6. This relation denotes Ω closes to Ω' under calling context $\stackrel{\Delta}{\hookrightarrow}$. A calling context Δ is represented as a sequence of tuples δ, each of the form $\langle\beta;\mathsf{c};\mathsf{m}\rangle$ denoting a call to method m of object β of class c on the call chain. We use β, $\mathcal{B}$, ω, Ω to represent the closure-time counterparts of modular-typing-time α, $\mathcal{A}$, κ, $\mathcal{K}$, respectively. We differentiate these syntactic entities to highlight the fact that β's and α's are chosen to be disjoint. The set of all β's are denoted $\mathbb{CTV}$. It includes one special type variable `tmain` to represent the bootstrapping task.

Type closure $\circlearrowleft \mathcal{C}$, as used in (T-Program), is defined as the largest set Ω where relation $\texttt{boot} \stackrel{[\,]}{\hookrightarrow} \Omega$ holds under implicit $\mathcal{C}$, and `boot` is sugar for $\{[\texttt{tmain}]^{\texttt{tmain},\texttt{main},\texttt{tmain}}, [\texttt{tmain}]^{\texttt{Main}} \le \texttt{tmain}\}$. Intuitively the delayed contour marker in this set indicates that the `main` method of the `Main` class is invoked, with `tmain` representing the "main" task and irrelevant arguments and return values. The overall goal of type closure is to merge local per-class and per-method constraints into one global set, so that all access constraints $\beta \stackrel{\theta}{\dashrightarrow} \beta'$ can form one static access graph for the key well-formedness check. Rule (C-Canon) "canonizes" access constraints, *i.e.* it traces back the chain of the flow constraints so that the type variable generated at instantiation point is used as a canonical name for the object. This is *de facto* applying a concrete type analysis to the object aliases, expressed as type variables here, appearing in access constraints. We use constraint form $\beta \stackrel{\theta}{\rightarrow} \beta'$ to represent the canonized version. To facilitate the process of instantiation-point back-tracing, rules (C-Flow=) and (C-Flow+) assert that flow constraints $\le$ are reflexive and transitive. The transitivity rule (C-Flow+) enables any type variable to ultimately find its instantiation point(s) via the previously explained flow constraint placed in (T-New). This is why in the `map` function of the `Mapper` class, our type system ultimately will find out what objects flow into `ul`, even though `ul` is not instantiated in its scope. (C-Task=) and (C-Task+) define reflexivity and transitivity for $\stackrel{\mathsf{T}}{\rightarrow}$ – if a protected zone β_1 encloses protected zone β_2, and β_2 encloses β_3, then β_1 can be viewed as enclosing β_3.

The main complexity of the closure algorithm arises from context sensitivity, captured by (C-Contour). Recall that for a context-sensitive algorithm, the type constraints associated with the method body need to be "refreshed" according to the specific calling context. We define a function for picking type variables:

$$\mathit{gen}(\Delta,\mathcal{A}) \stackrel{\text{def}}{=} \mathit{generate}(\mathit{collapse}(\Delta),\mathcal{A})$$

where *generate* is a deterministic function defined as follows:

$$\mathit{generate}(\Delta,\mathcal{A}) \stackrel{\text{def}}{=} \mathcal{B} \text{ where } |\mathcal{A}| = |\mathcal{B}|, \text{ all elements in } \mathcal{B} \text{ distinct}$$

with the additional requirement that for any $\mathcal{A}_1$, $\mathcal{A}_2$, the type variables in sequences $\mathit{generate}(\Delta,\mathcal{A}_1)$ and $\mathit{generate}(\Delta',\mathcal{A}_2)$ are disjoint if $\mathit{collapse}(\Delta) \neq \mathit{collapse}(\Delta')$ or $\mathcal{A}_1 \neq \mathcal{A}_2$ Note this requirement can be concretely satisfied by indexing the variables on the call string.

Function *collapse* determines whether a recursive invocation has been made, and if so, reuses the results of the initial invocation:

$$\begin{array}{rcl} \mathit{collapse}([\,]) & \stackrel{\text{def}}{=} & [\,] \\ \mathit{collapse}(\delta:\Delta) & \stackrel{\text{def}}{=} & [\delta,\delta_1,\ldots,\delta_n] \\ & & \text{if } \ \mathit{collapse}(\Delta) = [\ldots,\delta,\delta_1,\ldots,\delta_n] \\ \mathit{collapse}(\delta:\Delta) & \stackrel{\text{def}}{=} & \delta : \mathit{collapse}(\Delta) \\ & & \text{if } \ \delta \notin \mathit{collapse}(\Delta) \end{array}$$

Given calling context $\Delta = [\langle\beta_1;\mathsf{c}_1;\mathsf{m}_1\rangle,\ldots,\langle\beta_n;\mathsf{c}_n;\mathsf{m}_n\rangle]$, partial function $\mathit{pzone}(\Delta)$ is used to compute the "current protected zone." It is defined as β_i for some $i \in [1..n]$ where $\mathit{modifier}(\mathsf{c}_i) \neq \epsilon$, and for any j such that $1 \le j < i$, $\mathit{modifier}(\mathsf{c}_i) = \epsilon$. Substitution notation $\bullet[\sigma]$ replaces all type variables $\alpha \in \mathrm{dom}(\sigma)$ in $\bullet$ with $\sigma(\alpha)$. Let us now illustrate one inductive step of closure – when the constraints associated with `Mapper` class's `map` method have been

$$\text{(C-Canon)}\ \{\beta_1' \xrightarrow{\theta}\!\!\!\!\!\rightarrow\ \beta_2', (\beta_1 : \mathcal{B}_1)^{\mathsf{c}_1} \leq \beta_1', (\beta_2 : \mathcal{B}_2)^{\mathsf{c}_2} \leq \beta_2'\} \overset{\Delta}{\hookrightarrow} \{\beta_1 \xrightarrow{\theta} \beta_2\} \qquad \text{(C-Flow=)}\ \emptyset \overset{\Delta}{\hookrightarrow} \{\beta \leq \beta\}$$

$$\text{(C-Flow+)}\ \{\hat{\beta}_1 \leq \beta, \beta \leq \beta_2\} \overset{\Delta}{\hookrightarrow} \{\hat{\beta}_1 \leq \beta_2\} \qquad \text{(C-Task=)}\ \emptyset \overset{\Delta}{\hookrightarrow} \{\beta \xrightarrow{\mathsf{T}} \beta\}$$

$$\text{(C-Task+)}\ \{\beta_1 \xrightarrow{\mathsf{T}} \beta_2, \beta_2 \xrightarrow{\mathsf{T}} \beta_3\} \overset{\Delta}{\hookrightarrow} \{\beta_1 \xrightarrow{\mathsf{T}} \beta_3\}$$

$$\text{(C-Contour)}\ \frac{\begin{array}{c}\delta = \langle \beta; \mathsf{c}; \mathsf{m}\rangle \qquad \Delta' = \delta : \Delta \qquad \mathcal{C}(\mathsf{c}) = \forall \mathcal{A}_1.\mathcal{M} \\ \mathcal{M}(\mathsf{m}) = \forall \mathcal{A}_2.\forall \mathcal{S}.\mathcal{K} \qquad \sigma = (\mathcal{A}_1 \longmapsto \mathcal{B}) \uplus (\mathcal{A}_2 \longmapsto [\beta_{\text{arg}}, \beta_{\text{ret}}]) \uplus \biguplus_{\mathcal{A} \in \mathcal{S}} \mathcal{A} \longmapsto gen(\Delta', \mathcal{A})\end{array}}{\{[\beta_{\text{ret}}]^{\beta, \mathsf{m}, \beta_{\text{arg}}}, \mathcal{B}^{\mathsf{c}} \leq \beta\} \overset{\Delta}{\hookrightarrow} \{\langle \mathcal{K}[\sigma]\rangle^{\delta}\}}$$

$$\text{(C-GlobalIntro)}\ \{\hat{\beta} \leq \beta, \langle \Omega\rangle^{\delta}\} \overset{\Delta}{\hookrightarrow} \langle \Omega \cup \{\hat{\beta} \leq \beta\}\rangle^{\delta}$$

$$\text{(C-GlobalElim)}\ \frac{\mathcal{B}^{\beta_{\text{ret}}, \mathsf{m}, \beta_{\text{arg}}} \notin \Omega'}{\{\langle \Omega \cup \Omega'\rangle^{\delta}\} \overset{\Delta}{\hookrightarrow} \Omega'[\texttt{thost} \mapsto pzone(\delta : \Delta)] \cup \{\langle \Omega\rangle^{\delta}\}} \qquad \text{(C-Union)}\ \frac{\Omega_1 \overset{\Delta}{\hookrightarrow} \Omega_2}{\Omega \cup \Omega_1 \overset{\Delta}{\hookrightarrow} \Omega \cup \Omega_2}$$

$$\text{(C-Subset)}\ \frac{\Omega \overset{\Delta}{\hookrightarrow} \Omega_1 \cup \Omega_2}{\Omega \overset{\Delta}{\hookrightarrow} \Omega_1} \qquad \text{(C-Context)}\ \frac{\Omega_1 \overset{\delta:\Delta}{\hookrightarrow} \Omega_2}{\langle \Omega_1\rangle^{\delta} \overset{\Delta}{\hookrightarrow} \langle \Omega_2\rangle^{\delta}}$$

$$\begin{array}{lll}
\beta & \in \mathbb{C}\mathbb{T}\mathbb{V} & \textit{type variables in closure} \\
\mathcal{B} & ::= \overline{\beta} & \beta \textit{ sequence} \\
\Delta & ::= \overline{\delta} & \textit{calling context} \\
\delta & ::= \langle \beta; \mathsf{c}; \mathsf{m}\rangle & \textit{call site} \\
\Omega & ::= \overline{\omega} & \textit{closure} \\
\omega & ::= \hat{\beta} \leq \beta \mid \mathcal{B}^{\beta, \mathsf{m}, \beta'} \mid \beta \xrightarrow{\theta}\!\!\!\!\!\rightarrow \beta' \mid \beta \xrightarrow{\theta} \beta' \mid \langle \Omega\rangle^{\delta} & \textit{constraints in closure} \\
\hat{\beta} & ::= \beta \mid \mathcal{B}^{\mathsf{c}} & \textit{flow elements in closure} \\
\sigma & ::= \overline{\alpha \mapsto \beta} & \textit{substitution}
\end{array}$$

$$WF(\Omega) \overset{\text{def}}{=} \forall \beta.\beta' \xrightarrow{\mathsf{W}} \beta \in \Omega \implies cutExists(\Omega, \{\beta' | \beta' \xrightarrow{\theta} \beta\})$$

$$cutExists(\Omega, \mathcal{B}) \overset{\text{def}}{=} \exists \beta_{\mathsf{c}}. \bigwedge_{\beta \in \mathcal{B}} (\beta_{\mathsf{c}} \xrightarrow{\mathsf{T}} \beta) \wedge (\forall \beta_{\mathsf{c}}' \neq \beta_{\mathsf{c}}. \bigwedge_{\beta \in \mathcal{B}} (\beta_{\mathsf{c}}' \xrightarrow{\mathsf{T}} \beta) \implies (\beta_{\mathsf{c}}' \xrightarrow{\mathsf{T}} \beta_{\mathsf{c}}))$$

Figure 6. Type Constraint Closure and Isolation Preservation

merged, we show how (C-Contour) helps merge in the constraints for the method body of `loadWorkUnit`. Let us assume the closure at that step includes:

$$[\beta_{401}]^{\beta_{402}, \texttt{loadWorkUnit}, \beta_{403}}$$

which results from typing `ul!->`$_{[\alpha_{401}]}$`loadWorkUnit()` in `Mapper` and

$$[\beta_{404}, \beta_{405}]^{\texttt{Loader}} \leq \beta_{402}$$

from typing **new**$_{[\alpha_{404}, \alpha_{405}]}$ `Loader(cf)` in `Main` and by flow transitivity. Using the definition for $\mathcal{C}$(`Loader`) given previously, the substitution built up in (C-Contour) thus is

$$\mathcal{A}_1 \longmapsto \mathcal{B} \quad \text{is} \quad \begin{bmatrix} \alpha_{300} \mapsto \beta_{404} \\ \alpha_{301} \mapsto \beta_{405} \end{bmatrix}$$

$$\mathcal{A}_2 \longmapsto [\beta_{\text{arg}}, \beta_{\text{ret}}] \quad \text{is} \quad \begin{bmatrix} \alpha_{202} \mapsto \beta_{403} \\ \alpha_{203} \mapsto \beta_{401} \end{bmatrix}$$

$$\biguplus_{\mathcal{A} \in \mathcal{S}} \mathcal{A} \longmapsto gen(\Delta', \mathcal{A}) \quad \text{is} \quad \begin{bmatrix} \alpha_{101} \mapsto \beta_{601} \\ \alpha_{102} \mapsto \beta_{602} \\ \alpha_{103} \mapsto \beta_{603} \end{bmatrix}$$

and given β_{406} represents the `Mapper` object, and $\Delta' = [\langle \beta_{402}; \texttt{Loader}; \texttt{loadWorkUnit}\rangle, \langle \beta_{406}; \texttt{Mapper}; \texttt{map}\rangle, \langle \texttt{tmain}; \texttt{Main}; \texttt{main}\rangle]$, the *gen* function is:

$$\begin{array}{rcl} gen(\Delta', [\alpha_{101}]) & = & [\beta_{601}] \\ gen(\Delta', [\alpha_{102}, \alpha_{103}]) & = & [\beta_{602}, \beta_{603}] \end{array}$$

There are two interesting points here. First, if function `loadWorkUnit` were invoked via different call chains, the gen function would map Δ' to different type variable lists, so that different instances of `WorkUnit` instantiated from `loadWorkUnit` can be differentiated. This is also why different `Counter` instances in Fig. 2 can be approximated statically even though they are all instantiated from one program point. Second, rather than immediately merging the substituted constraints into the type closure, a special *contextual constraint* of the form $\langle\Omega\rangle^\delta$ is used, a marker denoting constraints Ω in the calling context of δ is to be merged to the type closure. The real merging happens at (C-GlobalElim). Here any constraint other than delayed contour markers can be "yanked" out of the contextual constraint – in other words, these constraints are not "context-sensitive". Note that when this happens, the placeholder for the "current protected zone," `thost`, needs to be properly replaced. Continuing with the example above, any constraint inside the $\langle\rangle$ of the contextual constraint obtained via (C-Contour) can be "yanked" out with `thost` replaced with β_{402}. It represents the `Loader` object itself. This is the value of `thost` here because shared tasks create protected zones of their own. Other rules related to contextual constraints, (C-GlobalIntro) and (C-Context), are self-explanatory.

3.5 Isolation Preservation

Isolation is enforced by the WF function in Fig. 6. It checks that for any non-shared ordinary object, either it is only accessed once, all accesses are reads, or the accessing tasks as in $\mathcal{B}$ must satisfy $cutExists(\Omega, \mathcal{B})$.

The $cutExists(\Omega, \mathcal{B})$ function in Fig. 6 formally defines the notion of cut vertex alluded to in Sec. 2.2: the subgraph including all static access paths ending with a type variable in $\mathcal{B}$ must have a cut vertex (or articulation point), and if there is more than one cut vertex for that subgraph, there must be a "least upper bound" of them. The definition here is phrased so as to allow a shared task (or the ordinary objects it owns) to access the objects belonging to its "ancestor" tasks, as long as the cut vertex invariant is not violated.

4. Operational Semantics and Formal Properties

In this section we briefly describe the operational semantics of our language, with a focus on features that are related to stating the proven formal properties. Small-step reductions $S \Rightarrow S'$ are defined over configurations $S = \langle H; \Sigma; e\rangle$ for H the object heap, Σ the dynamic constraint set, and e the expression. The reductions are implicitly parameterized by class list C. We use $S \Rightarrow_* S'$ to represent multi-step reduction, which is defined as the transitive closure of $\Rightarrow$. We use $\Rightarrow_C S$ to represent a computation of program C starting in the initial state and computing in multiple steps to S. We use $S \Uparrow$ to mean there is some computation of S that computes forever.

$$
\begin{array}{rcll}
H & ::= & \overline{o \mapsto \langle \mathcal{B}^{\mathsf{c}}; Fd\rangle} & \textit{heap}\\
\Sigma & ::= & \overline{p \xrightarrow{\theta} o \mid p \overset{\theta}{\rightsquigarrow} o} & \textit{dynamic constraint set}\\
\beta & ::= & \cdots \mid o & \textit{extended type variable}\\
\omega & ::= & \cdots \mid \beta \overset{\theta}{\rightsquigarrow} \beta' & \textit{extended constraint}\\
Fd & ::= & \overline{\mathsf{f} \mapsto v} & \textit{field store}\\
v & ::= & o \mid \mathbf{null} & \textit{value}\\
o, p, q & \in & \mathbb{OID} & \textit{object ID}\\
e & ::= & \cdots \mid \langle e\rangle^\delta \mid \frac{o}{\theta} \mid e; e & \textit{extended expressions}\\
 & \mid & v \mid \mathbf{post}\ e \mid e \parallel e & \\
\mathbf{E} & ::= & \bullet \mid \mathbf{E} \parallel e \mid e \parallel \mathbf{E} & \textit{evaluation context}\\
 & \mid & \mathsf{f}^{\zeta} := \mathbf{E} \mid (\mathsf{c})\mathbf{E} & \\
 & \mid & \mathbf{E} *_{[\beta]} \mathsf{m}(e) \mid v *_{[\beta]} \mathsf{m}(\mathbf{E}) & \\
 & \mid & \mathbf{new}_{\mathcal{B}}\ \mathsf{c}(\mathbf{E}) & \\
 & \mid & \mathbf{new}_{\mathcal{B}_1,\mathcal{B}_2}\ \mathsf{c}(\mathbf{E}) & \\
 & \mid & \mathbf{E}; e \mid \langle \mathbf{E}\rangle^\delta &
\end{array}
$$

Figure 7. Dynamic Semantics Definitions

Fig. 7 gives defines related data structures. To reuse data structures that have been defined for static semantics, we extend type variables β to include o's as well. As a result, the previous definition of calling context – a sequence of call sites in the form of $\langle\beta; \mathsf{c}; \mathsf{m}\rangle$'s – can also be viewed as that for the dynamic calling context, in the form of a list of $\langle o; \mathsf{c}; \mathsf{m}\rangle$'s. H is a mapping from objects o to field stores (Fd), and its program point information $\mathcal{B}^{\mathsf{c}}$ of their instantiation. Expressions are extended with values v, which are either object ID's or **null**. Auxiliary expression $\langle e\rangle^\delta$ is used to represent an expression e evaluated inside a dynamic call site δ. Expression $\frac{o}{\theta}$ is a helper expression to indicate o is accessed in mode θ, and **post** e is a helper expression to "post" a "root" task for later execution. Parallel operator $e \parallel e'$ is commutative. Dynamic constraint set Σ includes two kinds of constraints: $\xrightarrow{\theta}$ and $\overset{\theta}{\rightsquigarrow}$. We reuse the $\xrightarrow{\theta}$ constraint from the static type system, which is computed only for stating theorems and is not collected by the compiler. We use $\mathbf{E}$ to represent evaluation contexts. The initial configuration is $\langle H_{\text{init}}; \emptyset; e_{\text{init}}\rangle$ where $H_{\text{init}} = [o \mapsto \langle[\texttt{tmain}]^{\texttt{Main}}; [\,]\rangle]$ for some o and $e_{\text{init}} = o{-}{>}_{[\texttt{tmain}]}\texttt{main}(\mathbf{null})$.

Operational Semantics A complete definition of operational semantics can be found online [TT]. We now only focus on what is relevant for stating meta-theories, especially access invariants. First, reductions under a particular dynamic calling context Δ is represented by the $S \overset{\Delta}{\Longrightarrow} S'$, and this relation is connected with $\Rightarrow$ by a self-evident context rule.

$$H, \Sigma, \mathbf{E}[\,e\,] \Rightarrow H', \Sigma', \mathbf{E}[\,e'\,] \text{ if } H, \Sigma, e \overset{cxt(\mathbf{E})}{\Longrightarrow} H', \Sigma', e'$$

where

$$
\begin{aligned}
cxt(\bullet) &\stackrel{\text{def}}{=} [\,] \\
cxt([\![\,\mathbf{E}\,]\!]^o) &\stackrel{\text{def}}{=} cxt(\mathbf{E}) \\
cxt(\langle\mathbf{E}\rangle^\delta) &\stackrel{\text{def}}{=} cxt(\mathbf{E}) : [\delta] \\
cxt((\mathsf{c})\mathbf{E}) &\stackrel{\text{def}}{=} cxt(\mathbf{E}) \\
&\dots
\end{aligned}
$$

The reductions related to access take the following shape, where H' and e' are irrelevant information we elide here:

$$
\begin{aligned}
H, \Sigma, \mathsf{f}^{\mathsf{reg}} := v &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H', \Sigma, \tfrac{o}{\mathsf{W}}; e' \\
&\text{if } \Delta = \langle o; \mathsf{c}; \mathsf{m}\rangle : \Delta' \\
H, \Sigma, \mathsf{f}^{\mathsf{reg}} &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H, \Sigma, \tfrac{o}{\mathsf{R}}; e' \\
&\text{if } \Delta = \langle o; \mathsf{c}; \mathsf{m}\rangle : \Delta' \\
H, \Sigma, o *_{[\beta]} \mathsf{m}(v) &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H, \Sigma, \tfrac{o}{\mathsf{T}}; e' \\
H, \Sigma, \tfrac{o}{\theta} &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H, \Sigma \cup dset(\mu, \theta, p, o), e' \\
&\text{if } progressable(H, \Sigma, \mu, \theta, p, o), p = pzone(\Delta) \\
H, \Sigma, \langle v\rangle^{\langle o;\mathsf{c};\mathsf{m}\rangle} &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H, \Sigma - \{o_1 \overset{\theta}{\rightsquigarrow} o_2 \mid o_1 \text{ or } o_2 \text{ is } o\}, v \\
&\text{if } H(o) = \langle \mathcal{B}^{\mathsf{c}}; Fd\rangle, modifier(\mathsf{c}) \in \textbf{task} \\
H, \Sigma, \langle v\rangle^{\langle o;\mathsf{c};\mathsf{m}\rangle} &\xrightarrow{\Delta}\!\!\!\!\Longrightarrow H, \Sigma, v \\
&\text{if } H(o) = \langle \mathcal{B}^{\mathsf{c}}; Fd\rangle, modifier(\mathsf{c}) \notin \textbf{task}
\end{aligned}
$$

In essence, all object access controls (field read/write or messaging) have been delegated to the reduction of $\frac{o}{\theta}$, where function $progressable(H, \Sigma, \mu, \theta, p, o)$ is defined as:

μ \ θ	T	R or W
ϵ	true	true
shared	$roots(H, \Sigma, o) \subseteq roots(H, \Sigma, p)$	true
shared task	$roots(H, \Sigma, o) \subseteq roots(H, \Sigma, p)$	true
task	$(o \overset{T}{\rightsquigarrow} o) \notin \Sigma$	true

and function $dset(\mu, \theta, p, o)$ is defined as

μ \ θ	T	R or W
ϵ	$\emptyset$	$\{p \overset{\theta}{\rightsquigarrow} o, p \xrightarrow{\theta} o\}$
shared	$\{p \overset{T}{\rightsquigarrow} o, p \xrightarrow{\mathsf{T}} o\}$	$\emptyset$
shared task	$\{p \overset{T}{\rightsquigarrow} o, p \xrightarrow{\mathsf{T}} o\}$,	$\emptyset$
task	$\{o \overset{T}{\rightsquigarrow} o, o \xrightarrow{\mathsf{T}} o\}$	$\emptyset$

By now the difference between $\xrightarrow{\theta}$ and $\overset{\theta}{\rightsquigarrow}$ should be clear: $\xrightarrow{\theta}$ constraints monotonically grow in the dynamic constraint set, recording the entire "history" of the dynamic access since the program is bootstrapped. $\overset{\theta}{\rightsquigarrow}$ constraints on the other hand are removed whenever the invocation to a (shared or non-shared) task object ends – intuitively, a task "frees" all its accessing objects at the end of its execution. For that reason, $\overset{\theta}{\rightsquigarrow}$ only records the access relation at a specific runtime snapshot. We next define a function to compute the set of "root" task objects (*i.e.* non-shared task objects) that are currently accessing o.

Definition 1 (Roots)**.** *Function $roots(H, \Sigma, o)$ is defined as the largest set of o_0 such that $H(o_0) = \langle \mathcal{B}^{\mathsf{c}}; Fd\rangle$, $modifier(\mathsf{c}) =$ **task**, and there exists some $n \geq 0$, $\{o_0 \overset{T}{\rightsquigarrow} o_1, \dots o_{n-1} \overset{T}{\rightsquigarrow} o_n, o_n \overset{\theta}{\rightsquigarrow} o\} \subseteq \Sigma$.*

Let us now study liveness. The reduction for $\frac{o}{\theta}$ shows field read/write is never blocked, and messaging to non-shared ordinary objects is never blocked. This is precisely why declaring objects as non-shared ordinary objects improves performance. When the messaging target is a shared ordinary/task object, the execution is not blocked iff the target is not accessed by any task ($roots(H, \Sigma, o) = \emptyset$) or it is a reentrant access ($roots(H, \Sigma, o) = roots(H, \Sigma, p)$). These two cases, combined with the fact that $|roots(H, \Sigma, p)| = 1$ (by the nature of how call stacks grow and shrink), can be summarized with predicate $roots(H, \Sigma, o) \subseteq roots(H, \Sigma, p)$. If the target is a non-shared task object, messages are processed one at a time, as we discussed in Sec. 2. This is why pre-condition $(o \overset{T}{\rightsquigarrow} o) \notin \Sigma$ is used in that case.

Properties We next discuss the properties of our language, starting with some definitions. With a language runtime with blockings, deadlocks are possible:

Definition 2 (Deadlock)**.** *$S = \langle H; \Sigma; \langle \mathbf{E}[\frac{o_0}{\mathsf{T}}]\rangle^{\langle p_0;\mathsf{c}_0;\mathsf{m}_0\rangle} \parallel \dots \parallel \langle \mathbf{E}[\frac{o_n}{\mathsf{T}}]\rangle^{\langle p_n;\mathsf{c}_n;\mathsf{m}_n\rangle} \parallel e\rangle$ is a deadlock configuration iff for $i = 0..n$, $H(o_i) = \langle \mathcal{B}_i^{\mathsf{c}_i}; Fd_i\rangle$, and*

$$p_i \in roots(H, \Sigma, o_{(i+1) \bmod (n+1)})$$

The definition here shows how Task Types programming can help programmers reduce the likelihood of deadlocks by encouraging the default non-shared memory model. Deadlock cannot happen on non-shared ordinary objects: if there are no locks, there are no deadlocks. Next, some run-time exceptions standard in Java-like languages are possible in our language:

Definition 3 (Null Pointer Exception)**.** *S leads to a null pointer exception iff $S = \langle H; \Sigma; \mathbf{E}[\textbf{null} *_{[\beta]} \mathsf{m}(v)]\rangle$.*

Definition 4 (Bad Cast)**.** *Configuration S leads to a bad cast exception iff $S = \langle H; \Sigma; \mathbf{E}[(\mathsf{c}')o]\rangle$ where $H(o) = \langle \mathcal{B}^{\mathsf{c}}; Fd\rangle$ and $\mathsf{c}' \notin supers(\mathsf{c})$.*

The type soundness property is as follows.

Theorem 1 (Type Soundness)**.** *If $\vdash_{\mathrm{p}} C : \mathcal{C}$, then there exists S such that $\Rightarrow_C S$, where either $S \Uparrow$, or $S = \langle H; \Sigma; v\rangle$, or S is a deadlock configuration, or it leads to a null pointer or bad cast exception for some H, Σ.*

This Theorem states that the execution of a statically typed program either diverges, leads to a deadlock configuration, leads to a standard exception, or computes to a value. The Theorem is established by showing Lemmas of Subject Reduction, Progress, and the trivial fact that the bootstrapping process leads to a well-typed initial state.

Theorem 2 (Type Decidability)**.** *For any program C it is decidable whether there exists a $\mathcal{C}$ such that $\vdash_{\mathrm{p}} C : \mathcal{C}$.*

We now move on to state theorems related to the non-shared memory model. Let us first introduce the definition of a well-partitioned heap:

Definition 5 (Well Partitioned Heap)**.** $partitioned(H, \Sigma)$ *holds iff for all* o *such that* $o_0 \overset{W}{\rightsquigarrow} o \in \Sigma$, $|roots(H, \Sigma, o)| = 1$.

The predicate says that at runtime, if a non-shared ordinary object is write accessed, then all accesses must be initiated by only one non-shared task. We now describe some properties related to task isolation.

Theorem 3 (Static Task Isolation)**.** *If* $\Rightarrow_C \langle H; \Sigma; e\rangle$*, then* $partitioned(\Sigma)$.

This theorem says that a written non-shared ordinary object cannot be accessed by more than one "root" task at the same time. This theorem may appear appealing, but it in fact is weaker than what Task Types enforce for non-shared ordinary objects, because it cannot prevent a non-shared ordinary object from first being accessed by one task, released, and then accessed by another. The theorem that fully reflects the spirit of non-shared ordinary objects is the following:

Theorem 4 (Thread Locality)**.** *If* $\vdash_{\mathrm{p}} C : \mathcal{C}$ *and* $\Rightarrow_C \langle H; \Sigma; e\rangle$*, then* $WF(\Sigma)$.

This crucial theorem says the run-time constraint set we are computing in fact "conforms" to the statically checked one, in that cut vertices still exist for all non-shared ordinary objects. Note that WF works on $\overset{\theta}{\rightarrow}$ constraints, which in this case include the entire "history" of access. This theorem implies each non-shared ordinary object is accessed by at most one task *over its entire lifetime*.

Let us now state some concurrency properties:

Theorem 5 (Race Freedom)**.** $\vdash_{\mathrm{p}} C : \mathcal{C}$*, then there are no race conditions for field access in any execution of* C.

This theorem can be easily derived from Thm. 3, which states the most important subcase that fields with non-shared ordinary objects can never be accessed by more than one "root" tasks. For fields of other kinds of objects, the precondition $progressable()$ suffices to guarantee race freedom. Last, we can prove Task Type programs preserve atomicity:

Theorem 6 (Atomicity)**.** $\vdash_{\mathrm{p}} C : \mathcal{C}$*, then pervasive quantized atomicity defined in [LLS08] is preserved for all executions of* C.

Since this property is identical to the one defined in [LLS08], we defer it to an accompanying technical report online [TT]. Informally it is the pervasive, partitioned atomicity property described in Sec. 2.2. The definition of that property is based on the Theory of Reductions [Lip75], a standard method for proving atomicity properties.

5. Implementation and Evaluation

A prototype implementation of Task Types has been built on top of the CoqaJava compiler [LLS08], a compiler built using the Polyglot Java framework [NCM03]. We support the core syntax presented in Fig. 4, plus standard Java features including primitive values, local variable declarations and assignment, local method invocations, public field access, multiple-argument methods, return statement, field initializers, arrays, static classes and methods, method and constructor overloading, super invocations in constructors, super invocations for regular methods, and control flow expressions (loops, branches, and exception handling). We rely on Polyglot 2.4's built-in inner class remover to process inner classes, and conservatively wrap static fields as shared ordinary objects (with lock protection) among all instances since they are global variables that any access to may constitute an atomicity break point. The current implementation does not support reflection or native methods.

We now report some preliminary benchmarks. We picked two programs on the opposite ends of the contention spectrum: the low-contention RayTracer and the high-contention PuzzleSolver. All benchmarks were performed on a 4 x 6-Core Intel E7450 2.4GHz CPU machine (24 cores total) using 64GB RAM, running Debian GNU/Linux 2.6.26.

RayTracer RayTracer is a computationally intensive algorithm for rendering 3D images and is taken from the Java Grande Suite [SBO01]. We tweaked the program to fit the new Task Types syntax. In the test runs we created 12 non-shared tasks to process individual `Scene` objects concurrently; each scene contains 150 * 500 pixels. The programs were executed 50 times and the first 2 runs were discarded. Table. 1 below reports the elapsed time in milliseconds, which is the median of the 48 hot runs.

	# Cores				
	1	2	4	6	12
Coqa	741	593	450	364	340
Task Types	622	540	279	264	289

Table 1. RayTracer Benchmark: Coqa vs. Task Types

In the table, the "Task Types" results are an implementation of RayTracer with the minimal sharing declarations needed for the program to pass the Task Types typechecker. The program contains 1955 lines of code in 16 classes. We are able to declare 12 classes out of them to be non-shared ordinary classes ($\mu = \epsilon$); two are non-shared task classes, *i.e.* thread launchers, and two needed to be declared **shared**. The fact that the vast majority of classes are typable as non-shared ordinary classes confirms our earlier assertion that most objects can be coded normally – no special sharing declaration is needed. "Coqa" is a re-implementation of the same program with all 16 ordinary objects above explicitly declared as **shared**, meaning the program will follow the Coqa model and all non-task objects will be guarded with runtime locks so at most one task can use them at a time.

As can be seen, the use of non-shared objects led to a nontrivial performance improvement, averaging about 20% faster across the cases here. This preliminary result confirms

that a nontrivial speedup will be obtained compared to the purely dynamic approach of Coqa.

PuzzleSolver The PuzzleSolver benchmark was discussed earlier in Sec. 2.3.1. We benchmarked both the Task Types re-implementation of PuzzleSolver as well as the original Java implementation. To give meaningful multicore data, we created 1 task in the 1-core run, 2 tasks in the 2-core run, and so on. The programs were executed 20 times and the first 2 runs were discarded. Table. 2 below reports the elapsed time in milliseconds, which is the median of the rest of 18 hot runs.

	# Cores				
	1	2	4	6	12
Task Types	123	617	94	99	100
Java	80	564	91	74	82

Table 2. PuzzleSolver Benchmark Results

Observe that for a single-core execution, the Task Type implementation is about 35% slower. As the number of cores increases, the slowdown decreases to around 20%. Observe also that due to high contention this benchmark is not appreciably faster with multiple cores; it also exhibits an odd slowdown at two cores which is hard to explain but is in the underyling Java concurrency implementation since both implementations exhibit it. Our implementation of runtime shared task locking is still inefficient and can be improved, and we are also unnecessarily locking `PuzzlePosition` accesses in this benchmark as was pointed out in Sec. 2.3.1. So, we expect these numbers to improve considerably in a production implementation. That is not to say we expect there will be no runtime penalty at all; it is unlikely to go all the way to zero as some of the locking will not be necessary for correctness.

Compilation Time Compilation of both benchmarks took only a few seconds; the constraint closure step took around 1/3 sec. in each case, which shows it is not a significant factor here. More efficient implementations of context-sensitive/polymorphic type closure have been investigated [MRR05; EGH94; Age96; WS01; WL04]. As we move on to larger code samples, we will consider implementing various optimizations such as BDDs [BLQ$^+$03; WL04] and constraint garbage collection [WS01] if they are needed.

6. Related Work

Type Systems, Static Analyses, and Logics Flanagan and Qadeer [FQ03] first explored type system support for atomicity; their work takes a shared-memory model as basis, and supports reasoning about the composibility of atomic blocks over the shared memory, whereas Task Types can be viewed as carving out a non-shared-memory model in which atomicity violations cannot arise.

STM systems primarily use dynamic means to enforce atomicity [HF03; WJH04; CMC$^+$06]. The problem of weak atomicity has attracted significant interest recently, and a number of static or hybrid solutions exist to ensure transactional code and non-transactional code do not interfere with each other. Such a property is enforced by Shpeisman et. al. [SMAT$^+$07] 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 handoff": 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 [ABHI08] describes a conceptually static procedure to guarantee violation-freedom of transactional code and non-transactional code. Harris et. al. [HMPJH05] 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 shared task objects, race conditions do not occur in our system.

Static inference techniques have been designed to automatically insert locks to enforce atomicity for atomic blocks [MZGB06; HFP06; EFJM07; CCG08]. These techniques assume that atomic blocks are a fundamentally "local" construct, and make assumptions that are unrealistic for supporting the larger atomic regions we aim for: Autolocker [MZGB06] requires all invocations in an atomic block to be inlined, a static bound on lock resources is needed in others [EFJM07; CCG08; HFP06], and some require that all objects accessed in the atomic block not be accessed elsewhere [HFP06].

Our locality enforcement algorithm is related to ownership/region types, and particularly their inference [CCQR04; Wre03; DM07; LS08]. Cut vertices are evocative of dominators in the owners-as-dominators ownership type systems; in Task Types, DAGs can be freely formed inside the boundary of the dominator. A major challenge of our design – the problem that motivated task twinning – is not an issue for ownership/region types. When recursion happens, there is nothing wrong for ownership/region type declaration/inference systems to assign the same type variable to all recursive occurrences, but this would lead to atomicity violation in our situation; "two copies" of the task static instances are needed. .

There are many static analysis algorithms for tracking the flow of objects. Closest to our work are several thread escape analyses [CGS$^+$99; WR99; Bla99] 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 shared task-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.

A type system is a logical system and our work has a distant relation to program logics such as separation logic. In separation logic, local properties of the heap can be guaranteed by partitioning the heap, reasoning about the components, and then soundly re-composing to get the property over the full heap. Task Types share this philosophy of heap partitioning in how the access relations partition objects. Although there are currently no separation logics for concurrent object-oriented languages as we know of, there has been work on separation logics for Java [PB05; PB08] and for concurrency [OHe07; Bro07]. We believe the locality property of relation $cutExists$ is difficult to express with heap partitioning since they are a subtle form of heap sharing; these may be useful areas on which to focus extensions to separation logic.

Language Designs for Atomicity The shortcomings of explicitly declared atomic blocks are summarized in [VTD06]. 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 shared tasks for different atomic sets in Task Types. Their data-centric approach is a 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.

The Actor model and related message-passing languages [Agh90; Arm96; SM08] achieve atomicity by imposing stronger limitations on object sharing: threads communicate only at thread launch. A primary appeal of this model is each actor message handler thread executes atomically. If a synchronous communication is needed however, the sender needs to have an explicit message handler for processing the return value. The synchronous sender must thus be coded as two handlers, the code for actions up to and including the send, and the post-send return value handler (the continuation); this breaks the sender into two different atomicity regions. Additionally, this breaks the obvious control flow that would be apparent in synchronous messaging syntax, making programming more difficult. Some implemented Actor-based languages do include implicit CPS transformation syntax to ease coding, but that convolutes the code meaning (variable scoping for example) and does not repair the fact that the span of an atomic region was broken. Kilim [SM08] 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 design spaces: Kilim on message passing, and Task Types on atomicity.

A recent work that is closer to our spirit of pervasive atomicity is AME [IB07; ABHI08]. 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. This is fundamentally different from pervasive atomicity. In addition, AME does not support synchronous messaging, and does not overlap with the static type system aspect of our work.

A more conservative approach than atomicity is to design *determinism* into a concurrent language, such as building a type and effect system to guarantee threads are deterministic [BAD+09]. While this system is very useful for some algorithms, it is surprisingly subtle the properties that make algorithms deterministic and their system is often too weak to detect the determinism. Also, many algorithms are in fact not deterministic because some choices can be arbitrary. Note that both of our benchmarks are fundamentally arbitrary – the PuzzleSolver for example has tasks competing for next moves and the processor timing will dictate which task wins). The pervasive atomicity of this paper is a less rigid notion for the programmer in that threads must be divided into deterministic segments but need not be wholly deterministic. The two approaches are compatible; perhaps an ideal language would use the DPJ approach when determinism was feasible to guarantee, and fallback to Task Types when not.

7. Discussion

In our initial experience we have not found it difficult to port basic concurrency patterns to Task Types, but further work is needed to increase accuracy of the system to cover the full spectrum of programming patterns. Since increased accuracy also brings increased complexity, there needs to be a strong justification for an extension before it is made. We left out several potentially useful extensions in this paper, which we discuss now.

Simple Extensions Several features are left out from the formal system for simplicity, but can be added with minimal change. The first feature is per-object sharing declarations (rather than per-class sharing declarations as found in the core calculus). For instance, the `PuzzlePosition` class

of Sec. 2.3.1 is a case where a per-object declaration of the sharing policy, in place of the current per-class declaration, would have allowed the program to typecheck without the need for runtime locks. This extension is technically trivial. Because our polymorphic type system is able to differentiate objects of the same class anyways, the only change to implement it would be (1) turning types from c@α to a more verbose form such as c@α **of** μ, and (2) whenever a flow constraint is generated, a constraint asserting both sides have compatible modifiers is collected.

A second extension which would have allowed the same program above to typecheck is a call-by-copy parameter passing mechanism. This is a variation on immutability which allows object data to be freely passed from one task to another since the callee gets a fresh copy and thus no back-channel will be created.

A third extension in which the granularity of sharing could be made finer is to take individual fields of objects as the atomic units of sharing, meaning some fields could be owned by one thread and other fields by another thread. This approach may indeed lead to more programs to typecheck and it is not difficult to implement, but it is not clear to us that this is a good idea for object-oriented programmers (as opposed to C programmers). It violates a principle of object-based design in how an object has dual allegiance; in fact this may very well be a sign that such an object needs to be refactored into two different objects.

On the design spectrum with full declaration on one end and full inference on the other, Task Types lie somewhere in the middle: it is a type inference system but shared and task classes and messaging must be explicitly declared. In general, the design principle here is that sharing policies and execution policies for classes are essential and should be declared, but enforcement can be the job of a program analysis, in this case a polymorphic type inference algorithm, for programmability reasons. The ideas here are to some degree independent of this spectrum: it would be possible to infer sharing when needed, and it would also be possible to have an explicit declarative type system to capture the necessary well-formedness properties.

Non-Trivial Extensions Task Types disallow a non-shared ordinary object to be used in "phases", each of which belongs to a different task: "from bootstrapping to a particular point of execution, object `o` belongs to task `X`; *from that point on however*, `o` will never be used by `X` again, and can be grabbed by task `Y`." To see why this could in some cases be a useful feature, consider the Swing AWT `invokeAndWait` method. This method allows one thread to pass some GUI-manipulating code to the AWT thread where it will be queued up as an event and handled. Since Swing is not thread-safe this is the only safe way another thread can interact with Swing widgets outside of listeners. Consider the following example, taken from the Swing threading documentation:

```
void printTextField() throws Exception {
   final String[] myStrings = new String[2];
   Runnable getTextFieldText = new Runnable() {
         public void run() {
               myStrings[0] = textField0.getText();
               myStrings[1] = textField1.getText();
         }
     };
   SwingUtilities.invokeAndWait(getTextFieldText);
   System.out.println(myStrings[0] + " " + myStrings[1]);
}
```

Here `invokeAndWait` will pass the code snippet in `run()` for execution on the AWT thread; the current thread will block until `run()` completes. If this Java code was ported directly to Task Types, the `myStrings` array would want to be statically owned by both the main task and the AWT task and so typing would fail; not even making the array a shared ordinary object would help since even after `run()` completes the AWT thread will not release the array since it has not completed. A call-by-copy semantics, as discussed above, would restore typability since the original task will have a fresh copy of the data; however, this adds unnecessary overhead. This example shows it may be useful to add an *object transfer* feature. As can be seen in the above example, tasks hold on to shared ordinary objects even if they are in fact finished with them, but early release at an existing atomicity break point would not increase the number of atomic regions and would allow programs such as the above to statically typecheck: at the point where `run()` completes, the `myStrings` array can safely be transferred back to the original task. For the above example a flow-insensitive type analysis should be able to determine that `myStrings` does not escape `run()` and if a singular capability to `run()` were passed to the AWT task, object transfer upon return is sound. In more complex cases it may be necessary to use a flow-sensitive analysis to detect when an object is no longer used.

We discussed task twinning in Sec. 2.2. This method is relatively easy to prove correct, but in practice this conservative definition can be refined in several situations. such as a program point is obviously not instantiating more than one task, or if it does, the instantiated tasks are obviously not accessing the same objects. One refinement was mentioned in Sec. 2.2: twinning is not needed when a simple analysis determines a program point is not in a recursive context. Another refinement would be to track information flow in and out of the task object: no sharing can happen between two tasks instantiated from the program point – twinning is not needed as a result – if there is no flow leading them to share.

Limitations A common complaint of constraint-based type systems such as this one is they are non-modular, since the whole program is needed to compute the constraint closure. This mainly leads to two problems: (1) the burden of typechecking overhead at dynamic loading time *a la* Java, and (2) the difficulty of printing precise error messages. As was discussed in Sec. 3, problem (1) motivated us to formulate our system in two phases, so that many type errors

can in fact be discovered modularly. Furthermore, if dynamically loaded code was certified by signature to be identical to the compile-time code, no additional dynamic link checking would be needed. We agree that problem (2) is a real challenge. Indeed, any system with type inference across modularity boundaries – such as the type inference of ML – is faced with this challenge. As our compiler matures, we are interested in studying various error localization approaches.

8. Conclusion

We have presented an approach to achieve atomicity in multithreaded object-oriented programs, by making the following contributions:

- We develop a top-down, mostly static approach to enforce atomicity as opposed to the usual bottom-up, dynamic approach. Atomicity with Task Types is top-down in the sense that it is pervasive instead of building islands of atomic blocks. Principled language constructs for sharing between tasks are provided, and strong atomicity is achieved for all code regardless of sharing.
- We provide a programming model whose language syntax requires only minor changes to standard object syntax, but the philosophy of a non-shared memory model as a default, and explicit support for data sharing between threads, is significantly different than the norm. As a consequence, the object sharing between tasks is brought front and center for the programmer, where it should be.
- Task Types incorporate a precise and provably sound polymorphic/context-sensitive analysis which statically verifies that non-shared ordinary objects are appropriately partitioned between tasks. Since the partitioning is verified statically, there is no need for dynamic partitioning of non-shared objects between tasks, meaning there is no additional runtime overhead.
- The viability of Task Types has been confirmed by a prototype compiler, and initial benchmark results are reasonable.

Task Types are a complex type system; we believe this stems from complexity inherent in writing shared-memory concurrent programs correctly. Although it could be argued that means shared-memory concurrency needs to be abandoned, we are perhaps too far down that road to turn around, and Task Types represent a compromise between the "Wild West" of Java programming today and the rigid straightjacket of non-shared memory concurrency. Programmers also need not understand every detail of the type system, only that some object is shared more than the type system is allowing, and either its use must be restricted or its class must be lifted to be a shared class.

As in many type system approaches, the programmer burden of typing under Task Types will be nontrivial. However, that is not necessarily a bad thing: the ML language by analogy has a type system that is challenging for new programmers, but greatly increases productivity in the long term, because the type system catches what would have been difficult run-time bugs and so the extra time spent typechecking is more than made up in the time saved in later debugging. We believe Task Types will offer a similar advantage for concurrent programming.

The webpage of Task Types [TT] includes all reduction rules, a complete set of proofs for all the theorems presented in this paper, as well as the source code of the compiler.

References

[ABHI08] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08*, pages 63–74, 2008.

[Age96] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA, 1996.

[Agh90] Gul Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.

[Arm96] 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.

[BAD+09] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA '09*, pages 97–116, 2009.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. *SIGPLAN Not.*, 34(10):20–34, 1999.

[BLQ+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03*, pages 103–114, 2003.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02*, pages 211–230, 2002.

[Bro07] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227 – 270, 2007.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI'08*, pages 304–315, 2008.

[CCQR04] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *PLDI '04*, pages 243–254, 2004.

[CGS+99] Jong-deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA'99*, pages 1–19, 1999.

[Cla01] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July

2001.

[CMC+06] 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.

[CPN98] 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.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, 2004.

[DM07] W. Dietl and P. Müller. Runtime universe type inference. In *IWACO'07*, 2007.

[EFJM07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07*, pages 291–296, 2007.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI'94*, pages 242–256, 1994.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI'03*, pages 338–349, 2003.

[GLS94] William D. Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI — Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1994.

[Gro03] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03*, pages 13–25, 2003.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA'03*, pages 388–402, 2003.

[HFP06] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *TRANSACT'06*, June 2006.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60, 2005.

[IB07] 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.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java - a minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *In OOPSLA'98*, pages 36–44. ACM Press, 1998.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LLS08] Yu David Liu, Xiaoqi Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity. In *CC'08*, March 2008.

[LS08] Yu David Liu and Scott Smith. Pedigree types. In *4th International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, pages 63–71, July 2008.

[MRR05] 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.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pages 346–358, 2006.

[NCM03] 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.

[OHe07] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL'05*, pages 247–258, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86, New York, NY, USA, 2008. ACM.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP'08*, 2008.

[SMAT+07] 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.

[TT] `http://www.cs.binghamton.edu/~davidL/tasktypes`.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06*, pages 334–345, 2006.

[WJH04] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP'04*, pages 519–542, 2004.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*, pages 131–144, 2004.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[Wre03] Alisdair Wren. Ownership type inference. Master's thesis, Imperial College, 2003.

[WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP'01*, pages 99–117, 2001.