Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination

Xinyu Feng and Zhong Shao

Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{feng, shao}@cs.yale.edu

Abstract

Proof-carrying code (PCC) is a general framework that can, in principle, verify safety properties of arbitrary machine-language programs. Existing PCC systems and typed assembly languages, however, can only handle *sequential* programs. This severely limits their applicability since many real-world systems use some form of concurrency in their core software. Recently Yu and Shao proposed a logic-based "type" system for verifying concurrent assembly programs. Their thread model, however, is rather restrictive in that no threads can be created or terminated dynamically and no sharing of code is allowed between threads. In this paper, we present a new formal framework for verifying general multi-threaded assembly code with unbounded dynamic thread creation and termination as well as sharing of code between threads. We adapt and generalize the rely-guarantee methodology to the assembly level and show how to specify the semantics of thread "fork" with argument passing. In particular, we allow threads to have different assumptions and guarantees at different stages of their lifetime so they can coexist with the dynamically changing thread environment. Our work provides a foundation for certifying realistic multi-threaded programs and makes an important advance toward generating proof-carrying concurrent code.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification — correctness proofs, formal methods; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; D.4.5 [*Operating Systems*]: Reliability—verification

General Terms Languages, Verification

Keywords Concurrency Verification, Proof-Carrying Code, Rely-Guarantee, Dynamic Thread Creation

1. Introduction

Proof-carrying code (PCC) [28] is a general framework that can, in principle, verify safety properties of arbitrary machine-language programs. Existing PCC systems [29, 6, 2] and typed assembly languages (TAL) [27, 26], however, can only handle *sequential* programs. This severely limits their applicability since most real-world systems use some form of concurrency in their core software. Certifying low-level concurrent programs is an important task because it helps increase the reliability of software infrastructure and is crucial for scaling the PCC and TAL technologies to realistic systems.

As an important first step, Yu and Shao [42]—at last year's ICFP—proposed a certified formal framework (known as CCAP) for specifying and reasoning about general properties of concurrent programs at the assembly level. They applied the "invariance proof" technique for verifying general safety properties and the rely-guarantee methodology [23] for decomposition. They introduced a notion of "local guarantee" for supporting thread-modular verification even inside the middle of an atomic instruction sequence. Their thread model, however, is rather restrictive in that no threads can be created or terminated dynamically and no sharing of code is allowed between threads; both of these features are widely supported and used in mainstream programming languages such as C, Java, and Concurrent ML [34].

Certifying dynamic thread creation and termination turns out to be a much harder problem than we had originally anticipated [42]. Dynamic thread creation and termination imply a changing thread *environment* (*i.e.,* the collection of all live threads in the system other than the thread under concern). Such dynamic environment cannot be tracked during static verification, yet we still must somehow reason about it. For example, we must ensure that a newly created thread does not interfere with existing live threads, but at the same time we do not want to enforce non-interference for threads that have no overlap in their lifetime. Using one copy of code to create multiple threads also complicates program specification.

Existing work on the verification of concurrent programs almost exclusively uses high-level calculi (*e.g.,* CSP [21], CCS [25], TLA [24]). Also, existing work on the rely-guarantee methodology for shared-memory concurrency only supports properly nested concurrent code in the form of $\mathsf{cobegin}\ P_1 \| \ldots \| P_n\ \mathsf{coend}$ (which is a language construct for parallel composition where code blocks P_1, ..., P_n execute in parallel and all terminate at the coend point). They do not support dynamic thread creation and termination.

Modularity is also needed to make verification scale. Existing work on the rely-guarantee methodology supports thread modularity, *i.e.,* different threads can be verified separately without looking into other threads' code. However, they do not support code reuse very well. In CCAP, if a procedure is called in more than one thread, it must be verified multiple times using different specifications, one for each calling thread. We want a procedure to be specified and verified once so it can be reused for different threads.

In this paper, we propose a new framework for supporting certified multi-threaded assembly programming (CMAP). CMAP is based on a realistic abstract machine which supports dynamic thread creation with argument passing (the "fork" operation) as well as termination (the "exit" operation). Thread "join" can also be

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

implemented in our language using synchronization. Our approach builds on previous work on type systems and program verification but makes the following important new contributions:

- The "fork/join" thread model is more general than "cobegin/coend" in that it supports unbounded dynamic thread creation, which poses new challenges for verification. To our knowledge, our work is the first to successfully apply the rely-guarantee method to verify concurrent programs with dynamic thread creation and termination. Our CMAP framework provides a foundation for certifying realistic multi-threaded programs and makes an important step toward generating concurrent certified code.
- The presence of dynamic threads makes it impossible to track the actual live threads during verification. This poses great challenge in enforcing the rely-guarantee condition. To solve this, we collect all dynamic threads into a single *environment* (*i.e.,* the dynamic thread queue) and reason about the environment's assumption and guarantee requirements as a whole. Although the dynamic thread queue cannot be tracked statically, we can update and approximate the environment's assumption and guarantee at each program point. In fact, we can unify the concepts of the current running thread's assumption/guarantee with its environment's guarantee/assumption. As we will demonstrate in Sections 3 and 4, making this work in a formal framework (*i.e.,* CMAP) is not trivial and it constitutes our main technical contribution.
- To ensure that the dynamic thread environment is well-formed, we enforce the invariant that the active threads in the system never interfere with each other. We maintain this invariant by following the approach used for type checking the dynamic data heap [27]. By combining the type-based proof techniques with the rely-guarantee based reasoning, we get a simple, extensible, and expressive framework for reasoning about the flexible "fork/join" thread model.
- We allow one copy of thread code to be activated multiple times at different places. Different "incarnations" may have different behavior, depending on the value of the thread argument. This allows us to support unbounded thread creation.
- We show how to maintain thread-modular reasoning even in the presence of dynamic thread creation and termination. Unlike CCAP, we allow each code segment to be specified independently of threads. Our work provides great support for code- and verification sharing between threads.
- We have also solved some practical issues such as thread argument passing and the saving and restoring of thread-private data at context switches. These issues are important for realistic multi-threaded programming but as far as we know have never been discussed in existing work.

We have developed CMAP and proved its soundness using the Coq proof assistant [37]. The implementation in Coq is available for download [10]. Our work makes an important advance toward building a complete PCC system for multi-threaded programs. Without formal systems such as CMAP, we cannot formally reason about concurrent assembly code. Still, more work must be done before we can construct a fully practical system. For example, a highly desirable goal is a high-level language with concise human-readable annotations that can be automatically compiled into CMAP programs and proofs. We leave this as future work.

In the rest of this paper, we first give an overview of the rely-guarantee-based reasoning and a detailed explanation of the key challenges in verifying multi-threaded assembly code (Section 2). We then give an informal description of our approach to address these problems in Section 3, and present our work on CMAP with

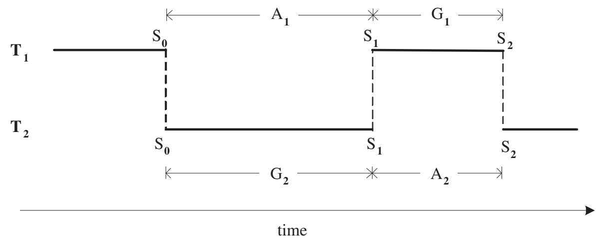

Figure 1. Rely-guarantee-based reasoning

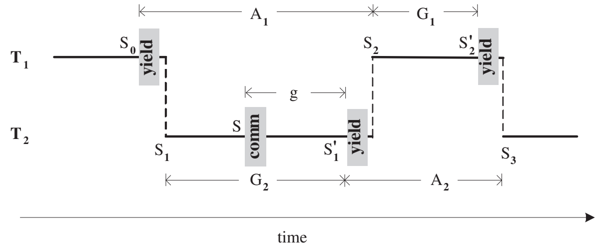

Figure 2. R-G in a non-preemptive setting

formal semantics in Section 4. We use a few examples to illustrate CMAP-based program verification in Section 5. Finally we discuss related work and conclude.

2. Background and Challenges

2.1 Rely-Guarantee-Based Reasoning

The rely-guarantee (R-G) proof method [23] is one of the best-studied approaches to the compositional verification of shared-memory concurrent programs. Under the R-G paradigm, every thread is associated with a pair (A, G), with the meaning that if the environment (*i.e.,* the collection of all of the rest threads) satisfies the assumption A, the thread will meet its guarantee G to the environment. In the shared-memory model, the assumption A of a thread describes what atomic transitions may be performed by other threads, while the guarantee G of a thread must hold on every atomic transition of the thread. They are typically modeled as predicates on a pair of states, which are often called *actions*.

For instance, in Figure 1 we have two interleaving threads T_1 and T_2. T_1's assumption A_1 adds constraints on the transition (S_0, S_1) made by the environment (T_2 in this case), while G_1 describes the transition (S_1, S_2), assuming the environment's transition satisfies A_1. Similarly A_2 describes (S_1, S_2) and G_2 describes (S_0, S_1).

We need two steps to reason about a concurrent program consisting of T_1 and T_2. First, we check that there is no interference between threads, i.e., that each thread's assumption can be satisfied by its environment. In our example, non-interference is satisfied as long as $G_1 \Rightarrow A_2$ (a shorthand for $\forall S, S'.G_1(S, S') \Rightarrow A_2(S, S')$), and $G_2 \Rightarrow A_1$. Second, we check that T_1 and T_2 do not lie, that is, they satisfy their guarantee as long as their assumption is satisfied. As we can see, the first step only uses the specification of each thread, while the second step can be carried out independently without looking at other threads' code. This is how the R-G paradigm achieves thread-modularity.

2.2 R-G in Non-Preemptive Thread Model

CMAP adopts a non-preemptive thread model, in which threads yield control voluntarily with a yield instruction, as shown in Figure 2. The preemptive model can be regarded as a special case of

```
Variables:                Initially:
  nat[100] data;            data[i] = n_i, 0 ≤ i < 100

main1 :                      main2 :
  data[0] := f(0);             nat i := 0;
  fork(chld, 0);               while(i < 100){
    .                            data[i] := f(i);
    .                            fork(chld, i);
  data[99] := f(99);             i := i + 1;
  fork(chld, 99);              }
  . . .                        . . .

void chld(int x){
  data[x] := g(x, data[x]);
}
```

Figure 3. Loop: high-level program

the non-preemptive one, in which an explicit yield is used at every program point. Also, on real machines, programs might run in both preemptive and non-preemptive settings: preemption is usually implemented using interrupts; a program can disable the interrupt to get into non-preemptive setting.

An "atomic" transition in a non-preemptive setting then corresponds to a sequence of instructions between two yields. For instance, in Figure 2 the state pair (S_2, S_2') corresponds to an atomic transition of thread T_1. A difficulty in modeling concurrency in such a setting is that the effect of an "atomic" transition cannot be completely captured until the end. For example, in Figure 2, the transition (S_1, S_1') should satisfy G_2. But when we reach the intermediate state S, we have no idea of what the whole transition (*i.e.,* (S_1, S_1')) will be. At this point, neither (S_1, S) nor (S, S_1') need satisfy G_2. Instead, it may rely on the remaining commands (the commands between comm and yield, including comm) to complete an adequate state transition. In CCAP [42], a "local guarantee" g is introduced for every program point to capture further state changes that must be made by the following commands before it is safe for the current thread to yield control. For instance, the local guarantee g attached to comm in Figure 2 describes the transition (S, S_1').

2.3 Challenges for Dynamic Thread Creation

To prove safety properties of multi-threaded programs, the key problem is to enforce the invariant that *all executing threads must not interfere with each other*. As mentioned in Section 2.1, threads *do not interfere* (or they satisfy the *non-interference* or *interference-free* property) only if each thread's assumption is implied by the guarantee of all other threads[1]. For languages that do not support dynamic thread creation, the code for each thread corresponds to exactly one executing thread. Using the rely-guarantee method, we can assign an assumption and guarantee to each thread code and enforce non-interference by checking all of these assumptions and guarantees, as is done in [42] and [13]. However, the following example shows that this simple approach cannot support dynamic thread creation and and multiple "incarnation" of the thread code.

In Figure 3, the high-level pseudo code (using C-like syntax) shows the code for (two versions of) a main thread and child threads. The main thread initializes 100 pieces of data using some function `f`, and distributes them to 100 child threads that will work on their own data (by applying a function `g`) in parallel. The `fork` function creates a child thread that will execute the function pointed

[1] We will formalize the Non-Interference property in Section 4.4.

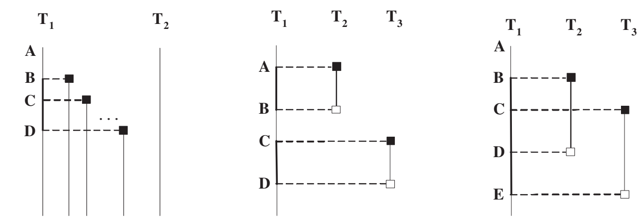

Figure 4. Interleaving of threads

to by the first argument. The second argument of `fork` is passed to the function as argument. The thread `main1` does this in sequential code while `main2` uses code with a loop. We assume that the high level code runs in preemptive mode. In other words, there is an implicit `yield` at any program point.

It is easy to see that both versions are "well-behaved" as long as the function g has no side effects, and all other threads in the rest of the system do not update the array of `data`. However, the simple approach used in [42] and [13] even cannot provide a specification for such trivial code.

1. Figure 4 (a) illustrates the execution of `main1` (time goes downwards). When doing data initialization (at stage A-B, meaning from point A to point B), the main thread needs to assume that no other threads in the environment (say, T_2) can change the array of `data`. However, the composition of the main thread's environment changes after a child thread is created. The assumption used at A-B is no longer appropriate for this new environment since the first child thread will write to `data`[0]. And the environment will keep changing with the execution of the main thread. How can we specify the `main1` thread to support such a dynamic thread environment?

   One possible approach is that the main thread relaxes its assumption to make exceptions for its child threads. However, it is hard to specify the parent-child relationship. Another approach is to use something like the program counter in the assumption and guarantee to indicate the phase of computation. This means the specification of the main thread is sensitive to the implementation. Also the program structure of the main thread has to be exposed to the specification of the child threads, which compromises modularity. The worst thing is that this approach simply won't work for the version `main2`.
2. Since multiple child threads are created, we must make sure that there is no interference between these children. It is easy for the above example since we can let the assumption and guarantee of the `chld` code be parameterized by its argument, and require $G_i \Rightarrow A_j$ given $i \neq j$. However, this approach cannot be generalized for threads that have dummy arguments and their behavior does not depend on their arguments at all. In this case $G_i \equiv G_j$ and $A_i \equiv A_j$ for any i and j. Then requiring $G_i \Rightarrow A_j$ is equivalent to requiring $G_i \Rightarrow A_i$, which cannot be true in general, given the meaning of assumptions and guarantees described in section 2.1. Do we need to distinguish these two kinds of threads and treat them differently? And how do we distinguish them?
3. Another issue introduced by dynamic thread creation, but not shown in this example program, is that the lifetimes of some threads may not overlap. In the case shown in Figure 4 (b), the lifetimes of T_2 and T_3 do not overlap and we should not statically enforce non-interference between them. Again, how can we specify and check the interleaving of threads, which can be as complex as shown in Figure 4 (c)?

In the next section we'll show how these issues are resolved in our development of CMAP.

3. Our Approach

In the rest of this paper, to distinguish the executing thread and the thread code, we call the dynamically running thread the "dynamic thread" and the thread code the "static thread". In Figure 3 the function `chld` is the static child thread, from which 100 dynamic child threads are activated.

As explained in Section 2.3, the approach that requiring non-interference of static threads is too rigid to support dynamic thread creation. Our approach, instead, enforces the thread non-interference in a "lazy" way. We maintain a dynamic thread queue which contains all of the active threads in the system. When a new thread is created, it is added to the dynamic thread queue. A thread is removed from the queue when its execution terminates. We also require that, when specifying the program, each static thread be assigned an assumption/guarantee pair. However, we do not check for non-interference between static thread specifications. Instead, each dynamic thread is also assigned an assumption and guarantee at the time of creation, which is an instantiation of the corresponding static thread specification with the thread argument. We require that dynamic threads do not interfere with each other, which can be checked by inspecting their specifications.

Our approach is very flexible in that each dynamic thread does not have to stick to one specification during its lifetime. When its environment changes, its specification can be changed accordingly. As long as the new specification does not introduce interference with other existing dynamic threads, and the subsequent behavior of this thread satisfies the new specification, the whole system is still interference-free. In this way, we can deal with the changing environment resulting from dynamic thread creation and termination. Problem 1 in Section 2.3 can be solved now.

If the lifetimes of two threads do not overlap they will not show up in the system at the same time. Therefore we do not need to check for interference at all. Also, since each dynamic thread has its own specification, we no longer care about the specification of the corresponding static thread. Therefor problems 2 and 3 shown in Section 2.3 are no longer an issue in our approach.

3.1 Typing The Dynamic Thread Queue

We define the dynamic thread queue $\mathbb{Q}$ as a set of thread identifiers $\mathtt{t}_i$, and the assignment Θ of assumption/guarantee pairs to dynamic threads as a partial mapping from $\mathtt{t}_i$ to (A_i, G_i) [2]. The queue $\mathbb{Q}$ is "well-typed" with regard to Θ if:

- $\mathbb{Q} = dom(\Theta)$, where $dom(\Theta)$ is the domain of Θ;
- threads in $\mathbb{Q}$ do not interfere, *i.e.,* $\forall \mathtt{t}_i, \mathtt{t}_j . \mathtt{t}_i \neq \mathtt{t}_j \Rightarrow (G_i \Rightarrow A_j)$; and
- each dynamic thread $\mathtt{t}_i$ is "well-behaved" with regard to (A_i, G_i), *i.e.,* if A_i is satisfied by the environment, $\mathtt{t}_i$'s execution does not get stuck and satisfies G_i.

Therefore, the invariant we need to maintain is that during the execution of the program, for the queue $\mathbb{Q}$ at each step there exists a Θ such that $\mathbb{Q}$ is well-typed with regard to Θ. In fact, we do not require Θ to be part of the program specification. We only need to ensure that there *exists* such a Θ at each step, which may be changing.

The content of the thread queue keeps changing, so how can we track the set of threads in the queue by a static inspection of the program? Here we follow the approach used for type-checking the dynamic data heap [27], which is dynamically updated by the store instruction and extended by the alloc instruction. We can ensure our invariant holds as long as the following conditions are satisfied:

- At the initial state (when the program starts to run) we can find a Θ to type-check the initial $\mathbb{Q}$. Usually the initial $\mathbb{Q}$ only contains the main thread, which will start to execute its first instruction, so we can simply assign the assumption/guarantee in the specification of the static main thread to the dynamic main thread.
- For each instruction in the program, assume that before the execution of the instruction there is a Θ such that $\mathbb{Q}$ is well typed. Then as long as certain constraints are satisfied to execute the instruction, there must exist a Θ' that can type check the resulting $\mathbb{Q}'$. For most instructions which do not change the content of the thread queue, this condition can be trivially satisfied. We are only interested in the "fork" and "exit" operation which will change the content of $\mathbb{Q}$.

For the "exit" instruction, the second condition can also be satisfied by the following lemma which can be trivially proven.

Lemma 3.1 (Thread Deletion)
If $\mathbb{Q}$ is well-typed with regard to Θ, then for all $\mathtt{t} \in \mathbb{Q}$ we know $\mathbb{Q} \setminus \{\mathtt{t}\}$ is well-typed with regard to $\Theta \setminus \{\mathtt{t}\}$.

For the "fork" instruction, things are trickier. We need to ensure that the new child thread does not interfere with threads in the parent thread's environment. We also require that the parent thread does not interfere with the child thread. The following lemma ensures the first requirement.

Lemma 3.2 (Queue Extension I)
Suppose $\mathbb{Q}$ is well-typed with regard to Θ and the current executing thread is $\mathtt{t}$. If

- $\Theta(\mathtt{t}) = (A, G)$;
- *a new thread $\mathtt{t}'$ is created by $\mathtt{t}$;*
- *(A', G') is the instantiation of the corresponding static thread specification by the thread argument;*
- *$A \Rightarrow A'$ and $G' \Rightarrow G$;*

then $\mathbb{Q}' \cup \{\mathtt{t}'\}$ is well-typed with regard to $\Theta'\{\mathtt{t}' \rightsquigarrow (A', G')\}$, where $\mathbb{Q}' = \mathbb{Q} \setminus \{\mathtt{t}\}$ and $\Theta' = \Theta \setminus \{\mathtt{t}\}$.

Here $\mathbb{Q}'$ is the environment of the current thread $\mathtt{t}$. Since $\mathtt{t}$ does not interfere with its environment (because $\mathbb{Q}$ is well-typed), we know that its assumption A is an approximation of what the environment can guarantee (G_e), and similarly that G is an approximation of the environment's assumption (A_e). By this interpretation, we can unify the concepts of the current running thread's assumption/guarantee with its environment's guarantee/assumption. To ensure the new thread $\mathtt{t}'$ does not interfere with $\mathtt{t}$'s environment, we need $G' \Rightarrow A_e$ and $G_e \Rightarrow A'$, which can be derived from $G' \Rightarrow G$ and $A \Rightarrow A'$.

Still, we need to ensure that thread $\mathtt{t}$ does not interfere with $\mathtt{t}'$. As mentioned above, A and G are approximations of G_e and A_e, respectively. Since the environment is extended with the child thread, the guarantee G'_e for the new environment is $G_e \vee G'$ and the assumption for the new environment A'_e is $A_e \wedge A'$. We want to change A and G correspondingly to reflect the environment change. First, the following lemma says that the specification of a dynamic thread can be changed during its lifetime.

Lemma 3.3 (Queue Update)
Suppose $\mathbb{Q}$ is well-typed with regard to Θ and that the current executing thread is $\mathtt{t}$. If

[2] This is a temporary formulation to illustrate our basic idea. We will use different definitions in our formal development of CMAP in Section 4.

- $\Theta(t) = (A, G)$;
- $G'' \Rightarrow G$ *and* $A \Rightarrow A''$;
- *the subsequent behavior of the current thread satisfies* (A'', G'');

then $\mathbb{Q}$ *is well-typed with regard to* $\Theta\{t \rightsquigarrow (A'', G'')\}$.

Now we can change the specification of the parent thread t to let it reflect the change of the environment.

Lemma 3.4 (Queue Extension II)
Suppose $\mathbb{Q}$ *is well-typed with regard to* Θ *and the current executing thread is* t. *If*

- $\Theta(t) = (A, G)$;
- *a new thread* t' *is created by* t;
- (A', G') *is the instantiation of the corresponding static thread specification by the thread argument;*
- $(A \Rightarrow A') \wedge (G' \Rightarrow G)$;
- *the remainder behavior of the thread* t *also satisfies* $(A \vee G', G \wedge A')$;

then $\mathbb{Q} \cup \{t'\}$ *is well-typed with regard to* $\Theta\{t' \rightsquigarrow (A', G'), t \rightsquigarrow (A \vee G', G \wedge A')\}$.

If t later creates another thread t'', because the specification of t already reflects the existence of t', by Lemma 3.2 we know that t'' will not interfere with t' as long as its specification satisfies the constraints. Therefore we do not need to explicitly check that t' and t'' are activated from the same static thread or that multiple activations of a static thread do not interfere with each other.

These lemmas are used to prove the soundness of CMAP. They are somewhat similar to the heap update and heap extension lemmas used in TAL's soundness proof [27]. People familiar with the traditional rely-guarantee method may feel this is nothing but the parallel composition rule used to support nested cobegin/coend. However, by combining the invariant-based proof technique used by type systems and the traditional rely-guarantee method, we can now verify multi-threaded assembly program with a more flexible program structure than the cobegin/coend structure. In particular, programs which are not properly nested, as shown in Figure 4(c) and the `main2` program in Figure 3, can be supported in our system. This is one of the most important contributions of this paper.

3.2 Parameterized Assumption/Guarantee

The assumptions and guarantees are interfaces between threads, which should only talk about shared resources. As we allow multiple activations of static threads, the behavior of a dynamic thread may depend on its arguments, which is the thread's private data. Therefore, to specify a static thread, the assumption and guarantee need to be parameterized over the thread argument.

In our thread model, the flat memory space is shared by all threads, and as in most operating systems, the register file is saved at the moment of context switch. Therefore the register file is thread-private data. The thread argument is stored in a dedicated register.

Rather than letting the assumption and guarantee be parameterized over the thread argument, we let them be parameterized over the whole register file. This makes our specification language very expressive. For instance, we allow the dynamic thread to change its specification during its lifetime to reflect change in the environment. If the thread has private data that tracks the composition of the environment, and its specification is parameterized by such data, then its specification automatically changes with the change of the data, which in turn results from the change of the thread environment. This is the key technique we use to support unbounded dynamic thread creation, as shown in the program `main2` in Figure 3.

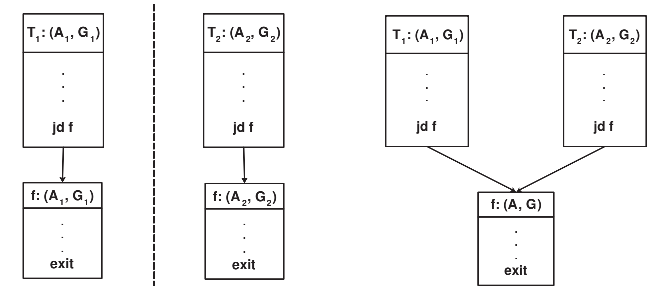

Figure 5. Code sharing between different threads

3.3 Support of Modular Verification

The rely-guarantee method supports thread modularity well, *i.e.,* code of one thread can be verified independently without inspecting other threads' code. However, it does not have good support for code reuse. In CCAP, each thread has its own code heap and there is no sharing of code between threads. As Figure 5(a) shows, if an instruction sequence is used by multiple threads, it has multiple copies in different threads' code heaps, each copy verified with regard to the specifications of these threads.

Based on the rely-guarantee method, the thread modularity is also supported in our system. In addition, using our "lazy" checking of thread non-interference, and by the queue update lemma, we can allow instruction sequences to be specified independently of their calling thread, thus achieving better modularity.

As shown in Figure 5(b), we assign an assumption/guarantee pair to the specification of each instruction sequence, and require the instruction sequence be well-behaved with regard to its own assumption/guarantee. Similar to threads, the instruction sequence is well-behaved if, when it is executed by a dynamic thread, its execution is safe and satisfies its guarantee, as long as other dynamic threads in the environment satisfy the assumption. The instruction sequence only needs to be verified once with respect to its own specification, and can be executed by different threads as long as certain constraints are satisfied.

Intuitively, it is safe for a dynamic thread t with specification (A_i, G_i) to execute the instruction sequence labeled by `f` as long as executing it does not require a stronger assumption than A_i, nor does it violate the guarantee G_i. Therefore, if the specification of `f` is (A, G), t can call `f` as long as $A_i \Rightarrow A$ and $G \Rightarrow G_i$. The intuition is backed up by our queue update lemma.

4. CMAP

The language CMAP is based on an "untyped" low-level abstract machine supporting multi-threaded programs with dynamic thread creation and argument passing. The "type" system of CMAP uses the calculus of inductive constructions (CiC) [32] to essentially support reasoning in higher-order predicate logic.

4.1 The Abstract Machine

Figure 6 shows the definition of our abstract machine.

A CMAP program (corresponding to a complete machine state) is made up of an updatable state $\mathbb{S}$ (which is made up of the shared memory $\mathbb{M}$ and the register file $\mathbb{R}$), a dynamic thread queue $\mathbb{Q}$, two shared code heaps $\mathbb{C}$ (for basic code blocks) and $\mathbb{T}$ (for thread entries), and the current instruction sequence $\mathbb{I}$ of the currently executing thread. Here $\mathbb{C}$ and $\mathbb{T}$ can be merged, but conceptually it is cleaner to have them separated because the specification of $\mathbb{T}$ and is different from that of $\mathbb{C}$ (See Section 4.3: for $\mathbb{T}$ we do not need to specify a local guarantee). Memory is a partial mapping from

$((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}) \longmapsto \mathbb{P}$	
if $\mathbb{I} =$	then $\mathbb{P} =$
$\mathsf{fork}\ \mathtt{h},\mathtt{r};\mathbb{I}''$	$((\mathbb{M},\mathbb{R}),\mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}',\mathbb{I}')\},\mathbb{T},\mathbb{C},\mathbb{I}'')$ where $\mathbb{I}' = \mathbb{T}(\mathtt{h})$, $\mathtt{t} \notin dom(\mathbb{Q})$, $\mathtt{t} \neq \mathbb{R}(\mathsf{rt})$, and $\mathbb{R}' = \{\mathtt{r}_0 \rightsquigarrow \_,\ldots,\mathtt{r}_{15} \rightsquigarrow \_, \mathsf{rt} \rightsquigarrow \mathtt{t}, \mathsf{ra} \rightsquigarrow \mathbb{R}(\mathtt{r})\}$
$\mathsf{yield};\mathbb{I}''$	$((\mathbb{M},\mathbb{R}'),(\mathbb{Q}\{\mathbb{R}(\mathsf{rt}) \rightsquigarrow (\mathbb{R},\mathbb{I}'')\}) \setminus \{\mathtt{t}\},\mathbb{T},\mathbb{C},\mathbb{I}')$ where $\mathtt{t} \in dom(\mathbb{Q})$ and $(\mathbb{R}',\mathbb{I}') = \mathbb{Q}(\mathtt{t})$ or $\mathtt{t} = \mathbb{R}(\mathsf{rt})$ and $(\mathbb{R}',\mathbb{I}') = (\mathbb{R},\mathbb{I}'')$
exit	$((\mathbb{M},\mathbb{R}'),\mathbb{Q} \setminus \{\mathtt{t}\},\mathbb{T},\mathbb{C},\mathbb{I}')$ where $\mathtt{t} \in dom(\mathbb{Q})$ and $(\mathbb{R}',\mathbb{I}') = \mathbb{Q}(\mathtt{t})$
$\mathsf{jd}\ \mathtt{f}$	$((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}')$ where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
$\mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}''$	$((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}'')$ if $\mathbb{R}(\mathtt{r}_s) \leq \mathbb{R}(\mathtt{r}_t)$, $((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}')$ otherwise, where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
$\mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}''$	$((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}'')$ if $\mathbb{R}(\mathtt{r}_s) \neq \mathbb{R}(\mathtt{r}_t)$, $((\mathbb{M},\mathbb{R}),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}')$ otherwise, where $\mathbb{I}' = \mathbb{C}(\mathtt{f})$
$\mathtt{c};\mathbb{I}''$ for remaining cases of $\mathtt{c}$	$(\mathtt{Next}(\mathtt{c},(\mathbb{M},\mathbb{R})),\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}'')$

Figure 8. Operational semantics of CMAP

$$
\begin{array}{rrl}
(\textit{Program}) & \mathbb{P} & ::= (\mathbb{S},\mathbb{Q},\mathbb{T},\mathbb{C},\mathbb{I}) \\
(\textit{State}) & \mathbb{S} & ::= (\mathbb{M},\mathbb{R}) \\
(\textit{Memory}) & \mathbb{M} & ::= \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Register}) & \mathtt{r} & ::= \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \mid \mathsf{r}_{15} \mid \mathsf{rt} \mid \mathsf{ra} \\
(\textit{CdHeap}) & \mathbb{C} & ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{Labels}) & \mathtt{f},\mathtt{l} & ::= \mathtt{n}\ (\textit{nat nums}) \\
(\textit{WordVal}) & \mathtt{w} & ::= \mathtt{n}\ (\textit{nat nums}) \\
(\textit{TEntries}) & \mathbb{T} & ::= \{\mathtt{h} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{TQueue}) & \mathbb{Q} & ::= \{\mathtt{t} \rightsquigarrow (\mathbb{R},\mathbb{I})\}^* \\
(\textit{THandles}) & \mathtt{h} & ::= \mathtt{n}\ (\textit{nat nums}) \\
(\textit{ThrdID}) & \mathtt{t} & ::= \mathtt{n}\ (\textit{nat nums}) \\
(\textit{InstrSeq}) & \mathbb{I} & ::= \mathtt{c};\mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{exit} \\
(\textit{Commd}) & \mathtt{c} & ::= \mathsf{yield} \mid \mathsf{fork}\ \mathtt{h},\mathtt{r} \mid \mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \mid \mathsf{sub}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \\
& & \quad \mid \mathsf{movi}\ \mathtt{r}_d,\mathtt{w} \mid \mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f} \mid \mathsf{beq}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f} \\
& & \quad \mid \mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(\mathtt{w}) \mid \mathsf{st}\ \mathtt{r}_d(\mathtt{w}),\mathtt{r}_s
\end{array}
$$

Figure 6. The abstract machine

if $\mathtt{c} =$	then $\mathtt{Next}(\mathtt{c},(\mathbb{M},\mathbb{R})) =$
$\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{sub}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)-\mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{movi}\ \mathtt{r}_d,\mathtt{w}$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{w}\})$
$\mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(\mathtt{w})$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\})$ where $(\mathbb{R}(\mathtt{r}_s)+\mathtt{w}) \in dom(\mathbb{M})$
$\mathsf{st}\ \mathtt{r}_d(\mathtt{w}),\mathtt{r}_s$	$(\mathbb{M}\{(\mathbb{R}(\mathtt{r}_d)+\mathtt{w}) \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\},\mathbb{R})$ where $(\mathbb{R}(\mathtt{r}_d)+\mathtt{w}) \in dom(\mathbb{M})$

Figure 7. Auxiliary state update function

memory locations to word-sized values. The register file $\mathbb{R}$ maps registers to word-sized values. In our machine, there are 16 general purpose registers ($\mathtt{r}_0$ - $\mathtt{r}_{15}$) and two special registers (rt and ra) that hold the current thread id and the thread argument. Code heaps map code labels to instruction sequences, which are lists of instructions terminated by a jd or exit instruction. Code labels pointing to thread entries are also called *thread handles*. $\mathbb{T}$ maps thread handles to thread entries (instruction sequences from which a thread starts to execute). Thread entries are also called *static threads*. The current instruction sequence $\mathbb{I}$ plays the role of the program counter of the current executing thread.

For simplicity, we just model the queue of ready threads, which is the dynamic thread queue $\mathbb{Q}$ that maps the dynamic thread id to an execution context of a thread. The dynamic thread id $\mathtt{t}$ is a natural number generated randomly at run time. The thread execution context includes the snapshot of the register file and the program point where the thread will resume its execution. Note that $\mathbb{Q}$ does not contain the current executing thread, which is different from the dynamic thread queue used in Section 3.

The instruction set of CMAP just contains the most basic and common assembly instructions. It also includes primitives fork, exit and yield to support multi-threaded programming which can be viewed as system calls to a thread library. We do not have a join instruction because thread join can be implemented using synchronization. Readers who are eager to see a CMAP program can take a quick look at Figure 10 in Section 5.1 (ignore the specifications and annotations in program for the time being), which is the CMAP implementation of programs `main2` and `chld` shown in Figure 3.

The execution of CMAP programs is modeled as small-step transitions from one program to another, *i.e.,* $\mathbb{P} \longmapsto \mathbb{P}'$. Figure 8 defines the program transition function.

The primitive fork creates a new thread using the static thread $\mathtt{h}$, and passes the value $\mathbb{R}(\mathtt{r})$ to it as the argument. The new thread will be assigned a fresh thread id and placed in the dynamic thread queue waiting for execution. The current thread continues with the subsequent instructions.

At a yield instruction, the current thread will give up the control of the machine. Its execution context is stored in $\mathbb{Q}$. The scheduler will pick one thread non-deterministically from the thread queue (which might be the yielding thread itself), restore its execution context, and execute it.

The exit instruction terminates the execution of the current thread and non-deterministically selects a thread from the thread queue. Here we have the implicit assumption that there is always an idle thread in the thread queue that never changes the state or terminates, ensuring that the thread queue will never be empty.

Semantics for the rest of the instructions are standard. The "next state" function defined in Figure 7 describes the effects of some instructions on the state.

4.2 The Meta-Logic

To encode the specification and proofs, we use the calculus of inductive constructions (CiC) [37, 32], which is an extension of the calculus of constructions (CC) [7] with inductive definitions. CC corresponds to Church's higher-order predicate logic via the Curry-Howard isomorphism.

CiC has been shown strongly normalizing [38], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [37], which we have used to implement CMAP.

In the remainder of this paper, we will mainly use the more familiar mathematical and logical notations, instead of strict CiC or Coq representation. We use *Prop* to denote the type of all propositions. No knowledge of CiC and Coq is required to understand them.

4.3 Program Specifications

The verification constructs of CMAP are defined in Figure 9. The program specification Φ is a global invariant (*Inv*), a static thread

$$
\begin{array}{rrcl}
(\mathit{ProgSpec}) & \Phi & ::= & (\mathit{Inv}, \Delta, \Psi) \\
(\mathit{ThrdSpec}) & \Delta & ::= & \{\mathtt{h} \rightsquigarrow \theta\}^* \\
(\mathit{ThrdType}) & \theta & ::= & (\mathtt{p}, \mathbb{A}, \mathbb{G}) \\
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G})\}^* \\
(\mathit{ActTSpec}) & \Theta & ::= & \{\mathtt{t} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\}^* \\
\\
(Invariant) & \mathit{Inv} & \in & \mathit{Mem} \rightarrow \mathit{Prop} \\
(Assertion) & \mathtt{p} & \in & \mathit{State} \rightarrow \mathit{Prop} \\
(Assumption) & \mathbb{A} & \in & \mathit{RegFile} \rightarrow \mathit{Mem} \rightarrow \mathit{Mem} \rightarrow \mathit{Prop} \\
(Guarantee) & \mathbb{G}, \mathtt{g} & \in & \mathit{RegFile} \rightarrow \mathit{Mem} \rightarrow \mathit{Mem} \rightarrow \mathit{Prop}
\end{array}
$$

Figure 9. Verification constructs of CMAP

specification Δ, and a code heap specification Ψ. The invariant *Inv* is a programmer specified predicate, which implies a safety property of concern. It must hold throughout the execution of the program. The static thread specification Δ contains a specification θ for each static thread in $\mathbb{T}$. Each θ contains a precondition $\mathtt{p}$ to invoke this thread, and an assumption $\mathbb{A}$ and guarantee $\mathbb{G}$ for this thread with regard to the environment at its creation time.

A code heap specification Ψ assigns a quadruple $(\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G})$ to each instruction sequence. The assertion $\mathtt{p}$ is the precondition to execute the code sequence. The local guarantee $\mathtt{g}$, as introduced in Section 2.2, describes a valid state transition – it is safe for the current thread to yield control only after making a state transition described by $\mathtt{g}$. As explained in Section 3.3,, we assign a pair of $\mathbb{A}$ and $\mathbb{G}$ to each instruction sequence as part of its specification. The instruction sequence can be verified with regard to its own specification without knowing which thread executes it. Here the $\mathbb{A}$ and $\mathbb{G}$ reflect knowledge of the dynamic thread environment at the time the instruction sequence is executed.

The global invariant *Inv* is a CiC term of type *Mem* $\rightarrow$ *Prop*, *i.e.,* a predicate over memory. *Inv* does not specify the register file, which contains thread-private data and keeps changing. In contrast to *Inv*, assertions ($\mathtt{p}$) are predicates over the whole state. Assumptions and guarantees (*e.g.,* $\mathbb{A}$, $\mathbb{G}$ and $\mathtt{g}$) are CiC terms with type *RegFile* $\rightarrow$ *Mem* $\rightarrow$ *Mem* $\rightarrow$ *Prop*, which means predicates over a register file and two instances of memory. Assumptions and guarantees specify the behavior of threads by describing the change of shared memory. As mentioned in Section 3.2, they are parameterized over the register file, which contains the private data of threads.

We also define the specification Θ of active threads in the thread queue $\mathbb{Q}$. Within Θ, each triple $(\mathtt{p}, \mathbb{A}, \mathbb{G})$ describes a dynamic thread at its yield point (or at the beginning if it has just forked). The assertion $\mathtt{p}$ gives the constraint of the state when the thread gets control back to execute its remaining instructions. The assumption and guarantee used by the thread at the yield point are given by $\mathbb{A}$ and $\mathbb{G}$. As we said in Section 3.1, the $\mathbb{A}$ and $\mathbb{G}$ of each dynamic thread may change during the lifetime of the thread. Notice that Θ is not part of the program specification. It is used only in the soundness proof.

4.4 Inference Rules

We use the following judgement forms to define the inference rules:

$\Phi; \Theta; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}$	(well-formed program)
$\Phi; \Theta; (\mathtt{g}, \mathbb{S}) \vdash \mathbb{Q}$	(well-formed dynamic threads)
$\Phi \vdash \mathbb{T}$	(well-formed static threads)
$\Phi \vdash \mathbb{C}$	(well-formed code heap)
$\Phi; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{I}$	(well-formed instr. sequence)

Before introducing the inference rules, we first define some shorthands in Table 1 to simplify our presentation.

Well-formed programs. The PROG rule shows the invariants that need to be maintained during program transitions.

Representation	Definition
$\mathbb{S}(\mathtt{r})$	$\mathbb{R}(\mathtt{r})$ where $\mathbb{S} = (\mathbb{M}, \mathbb{R})$
$\mathit{Inv} \wedge \mathtt{p}$	$\lambda(\mathbb{M}, \mathbb{R}).\mathit{Inv}\ \mathbb{M} \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})$
$\mathtt{g}\ \mathbb{S}\ \mathbb{M}'$	$\mathtt{g}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}'$, where $\mathbb{S} = (\mathbb{M}, \mathbb{R})$
$\mathbb{A}\ \mathbb{S}\ \mathbb{M}'$, $\mathbb{G}\ \mathbb{S}\ \mathbb{M}'$	similar to $\mathtt{g}\ \mathbb{S}\ \mathbb{M}'$
$\mathtt{p} \circ \mathtt{c}$	$\lambda\mathbb{S}.\mathtt{p}\ (\mathtt{Next}(\mathtt{c}, \mathbb{S}))$
$\mathtt{g} \circ \mathtt{c}$	$\lambda\mathbb{S}.\lambda\mathbb{M}'.\ \mathtt{g}\ (\mathtt{Next}(\mathtt{c}, \mathbb{S}))\ \mathbb{M}'$
$\mathbb{A} \circ \mathtt{c}$, $\mathbb{G} \circ \mathtt{c}$	similar to $\mathtt{g} \circ \mathtt{c}$
$\mathtt{p} \Rightarrow \mathtt{p}'$	$\forall\mathbb{S}.\mathtt{p}\ \mathbb{S} \Rightarrow \mathtt{p}'\ \mathbb{S}$
$\mathtt{p} \Rightarrow \mathtt{g}$	$\forall\mathbb{R}, \mathbb{M}.\mathtt{p}\ (\mathbb{M}, \mathbb{R}) \Rightarrow \mathtt{g}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}$
$\mathtt{p} \Rightarrow \mathtt{g} \Rightarrow \mathtt{g}'$	$\forall\mathbb{S}, \mathbb{M}'.\mathtt{p}\ \mathbb{S} \Rightarrow \mathtt{g}\ \mathbb{S}\ \mathbb{M}' \Rightarrow \mathtt{g}'\ \mathbb{S}\ \mathbb{M}'$
$\mathbb{A} \Rightarrow \mathbb{A}'$	$\forall\mathbb{R}, \mathbb{M}, \mathbb{M}'.\mathbb{A}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathbb{A}'\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}'$
$\mathbb{G} \Rightarrow \mathbb{G}'$	$\forall\mathbb{R}, \mathbb{M}, \mathbb{M}'.\mathbb{G}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathbb{G}'\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}'$
$\mathtt{p} \xRightarrow{\mathbb{R}} \mathtt{g} \xRightarrow{\mathbb{R}'} \mathtt{p}'$	$\forall\mathbb{M}, \mathbb{M}'.\mathtt{p}\ (\mathbb{M}, \mathbb{R}) \Rightarrow \mathtt{g}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathtt{p}\ (\mathbb{M}', \mathbb{R}')$
$(\mathit{Inv}, \mathtt{p} \circlearrowleft \mathbb{A}, \mathbb{R})$	$(\mathit{Inv} \wedge \mathtt{p}) \xRightarrow{\mathbb{R}} \mathbb{A} \xRightarrow{\mathbb{R}} \mathtt{p}$

Table 1. Assertion definitions and syntactic sugar

$$
\frac{\begin{array}{c}(\mathit{Inv}, \Delta, \Psi) = \Phi \quad (\mathbb{M}, \mathbb{R}) = \mathbb{S} \quad \mathtt{t} = \mathbb{R}(\mathsf{rt}) \\ \Phi \vdash \mathbb{T} \quad \Phi \vdash \mathbb{C} \quad (\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \quad \Phi; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{I} \\ \Phi; \Theta; (\mathtt{g}, \mathbb{S}) \vdash \mathbb{Q} \quad \mathtt{NI}(\Theta\{\mathtt{t} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\}, \mathbb{Q}\{\mathtt{t} \rightsquigarrow (\mathbb{R}, \mathbb{I})\})\end{array}}{\Phi; \Theta; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash (\mathbb{S}, \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})} \ (\text{PROG})
$$

The well-formedness of a program is judged with respect to the program specification Φ, the dynamic thread specification Θ, and the specification of the current executing thread $(\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G})$. Compared with the triples in Θ, we need the local guarantee $\mathtt{g}$ here to specify the transition which the current thread must make before it yields control.

In the first line, we give the composition of the program specification, the current state and the current thread id. Here we use a pattern match representation, which will be used in the rest of this paper.

We require the code, including the thread entry points $\mathbb{T}$ and the code heap $\mathbb{C}$, always be well-formed with respect to the program specification. Since Φ, $\mathbb{T}$ and $\mathbb{C}$ do not change during program transitions, the check for the first two premises in line 2 can be done once and for all.

The next premise shows the constraints on the current state $\mathbb{S}$: it must satisfy both the global invariant *Inv* and the assertion $\mathtt{p}$ of the current thread. The last premise in line 2 essentially requires it be safe for the current thread to execute the remainder instruction sequence $\mathbb{I}$.

Premises in line 3 require the well-formedness of dynamic threads in $\mathbb{Q}$, which is checked by the rule DTHRDS, and the non-interference between all the live threads.

Non-interference. The non-interference macro $\mathtt{NI}(\Theta, \mathbb{Q})$ requires that each dynamic thread be compatible with all the other. It is formally defined as:

$$
\begin{array}{l}\forall \mathtt{t}_i, \mathtt{t}_j \in \mathit{dom}(\Theta).\forall\mathbb{M}, \mathbb{M}'. \\ \quad \mathtt{t}_i \neq \mathtt{t}_j \Rightarrow \mathbb{G}_i\ \mathbb{R}_i\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathbb{A}_j\ \mathbb{R}_j\ \mathbb{M}\ \mathbb{M}',\end{array}
$$

where $(\_, \mathbb{A}_i, \mathbb{G}_i) = \Theta(\mathtt{t}_i)$, $(\_, \mathbb{A}_j, \mathbb{G}_j) = \Theta(\mathtt{t}_j)$, $\mathbb{R}_i = \mathbb{Q}(\mathtt{t}_i)$ and $\mathbb{R}_j = \mathbb{Q}(\mathtt{t}_j)$.

As explained in Section 3, here we enforce the non-interference by a lazy check of specifications Θ of dynamic threads in $\mathbb{Q}$, instead of checking the specification Δ of all static threads in $\mathbb{T}$.

Well-formed dynamic threads. The rule DTHRDS ensures each dynamic thread in $\mathbb{Q}$ is in good shape with respect to the specification Θ, the current program state $\mathbb{S}$, and the transition $\mathtt{g}$ that the current thread need do before other threads can take control.

$$\frac{\begin{array}{l}(\mathbb{R}_k, \mathbb{I}_k) = \mathbb{Q}(\mathtt{t}_k),\ \ (\mathtt{p}_k, \mathbb{A}_k, \mathbb{G}_k) = \Theta(\mathtt{t}_k),\ \ \forall \mathtt{t}_k \in dom(\mathbb{Q}) \\ (Inv, \mathtt{p}_k \circlearrowleft \mathbb{A}_k, \mathbb{R}_k) \quad \forall \mathbb{M}'.\mathtt{g}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathtt{p}_k\ (\mathbb{M}', \mathbb{R}_k) \\ (Inv, \Delta, \Psi); (\mathtt{p}_k, \mathbb{G}_k, \mathbb{A}_k, \mathbb{G}_k) \vdash \mathbb{I}_k\end{array}}{(Inv, \Delta, \Psi); \Theta; (\mathtt{g}, (\mathbb{M}, \mathbb{R})) \vdash \mathbb{Q}} \quad \text{(DTHRDS)}$$

The first line gives the specification of each thread when it reaches a yield point, and its execution context.

The first premise in line 2 requires that if it is safe for a thread to take control at certain state, it should be also safe to do so after any state transition satisfying its assumption. Note that the state transition will not affect the thread-private data in $\mathbb{R}_k$.

The second premise describes the relationship between the local guarantee of the current thread and the preconditions of other threads. For any transitions starting from the current state $(\mathbb{M}, \mathbb{R})$, as long as it satisfies $\mathtt{g}$, it should be safe for other threads to take control at the result state.

Line 3 requires that for each thread its remainder instruction sequence must be well formed. Note we use $\mathbb{G}_k$ as the local guarantee because after yield, a thread starts a new "atomic transition" described by its global guarantee.

Well-formed static threads. This rule checks the static thread entry is well formed with respect to its specification in Δ.

$$\frac{\begin{array}{l}(\mathtt{p}_k, \mathbb{A}_k, \mathbb{G}_k) = \Delta(\mathtt{h}_k),\ \ \mathbb{I}_k = \mathbb{T}(\mathtt{h}_k),\ \ \forall \mathtt{h}_k \in dom(\mathbb{T}) \\ \forall \mathbb{R}.(Inv, \mathtt{p}_k \circlearrowleft \mathbb{A}_k, \mathbb{R}) \quad (Inv, \Delta, \Psi); (\mathtt{p}_k, \mathbb{G}_k, \mathbb{A}_k, \mathbb{G}_k) \vdash \mathbb{I}_k\end{array}}{(Inv, \Delta, \Psi) \vdash \mathbb{T}} \quad \text{(THRDS)}$$

The first line gives the specification for each static thread in $\mathbb{T}$. We implicitly require $dom(\Delta) = dom(\mathbb{T})$.

The first premise in line 2 says that if it is safe to invoke the new thread at certain state, it should also be safe to delay the invocation after any transition satisfying the assumption of this thread.

The initial instruction sequence of each thread must be well-formed with respect to the thread specification. This is required by the last premise.

Well-formed code heap. A code heap is well formed if every instruction sequence is well-formed with respect to its corresponding specification in Ψ.

$$\frac{dom(\Psi) = dom(\mathbb{C}) \quad (Inv, \Delta, \Psi); \Psi(\mathtt{f}) \vdash \mathbb{C}(\mathtt{f}), \forall \mathtt{f} \in dom(\Psi)}{(Inv, \Delta, \Psi) \vdash \mathbb{C}} \quad \text{(CDHP)}$$

Thread creation. The FORK rule describes constraints on new thread creation, which enforces the non-interference between the new thread and existing threads.

$$\frac{\begin{array}{l}(\mathtt{p}', \mathbb{A}', \mathbb{G}') = \Delta(\mathtt{h}) \\ \mathbb{A} \Rightarrow \mathbb{A}'' \quad \mathbb{G}'' \Rightarrow \mathbb{G} \quad (\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}' \quad \mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'') \\ \forall \mathbb{R}, \mathbb{R}'.\ (\mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \wedge \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathsf{ra})) \Rightarrow ((Inv \wedge \mathtt{p}) \overset{\mathbb{R}}{\Longrightarrow} \mathtt{g} \overset{\mathbb{R}'}{\Longrightarrow} \mathtt{p}') \\ (Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}'', \mathbb{G}'') \vdash \mathbb{I}\end{array}}{(Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{fork}\ \mathtt{h}, \mathtt{r}; \mathbb{I}} \quad \text{(FORK)}$$

As explained in Section 3.1, the parent thread can change its specification to reflect the change of the environment. To maintain the non-interference invariant, constraints between specifications of the parent and child threads have to be satisfied, as described in Lemma 3.4. Here we enforce these constraints by premises in line 2, where $(\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}'$ is the shorthand for:

$$\begin{array}{l}\forall (\mathbb{M}, \mathbb{R}).\forall \mathbb{M}', \mathbb{M}''.\forall \mathbb{R}'.(Inv \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R}) \Rightarrow \mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \\ \quad \Rightarrow \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathsf{ra}) \Rightarrow (\mathbb{A} \vee \mathbb{G}'')\ \mathbb{R}\ \mathbb{M}'\ \mathbb{M}'' \Rightarrow \mathbb{A}'\ \mathbb{R}'\ \mathbb{M}'\ \mathbb{M}'',\end{array}$$

and $\mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'')$ for:

$$\begin{array}{l}\forall (\mathbb{M}, \mathbb{R}).\forall \mathbb{M}', \mathbb{M}''.\forall \mathbb{R}'.(Inv \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R}) \Rightarrow \mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \\ \quad \Rightarrow \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathsf{ra}) \Rightarrow \mathbb{G}'\ \mathbb{R}'\ \mathbb{M}'\ \mathbb{M}'' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'')\ \mathbb{R}\ \mathbb{M}'\ \mathbb{M}''.\end{array}$$

Above non-interference checks use the extra knowledge that:

- the new thread id is different with its parent's, *i.e.,* $\mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt})$;
- the argument of the new thread comes from the parent's register $\mathtt{r}$, *i.e.,* $\mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathsf{ra})$;
- the parent's register file satisfies the precondition, *i.e.,* $(Inv \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R})$.

In most cases, the programmer can just pick $(\mathbb{A} \vee \widehat{\mathbb{G}'})$ and $(\mathbb{G} \wedge \widehat{\mathbb{A}'})$ as $\mathbb{A}''$ and $\mathbb{G}''$ respectively, where $\widehat{\mathbb{G}'}$ and $\widehat{\mathbb{A}'}$ are instantiations of $\mathbb{G}'$ and $\mathbb{A}'$ using the value of the child's argument.

The premise in line 3 says that after the current thread completes the transition described by $\mathtt{g}$, it should be safe for the new thread to take control with its new register file $(\mathbb{R}')$, whose relationship between the parents register file $\mathbb{R}$ is satisfied.

The last premise checks the well-formedness of the remainder instruction sequence. Since the fork instruction does not change states, we need not change the precondition $\mathtt{p}$ and $\mathtt{g}$.

Yielding and termination.

$$\frac{\forall \mathbb{R}.\ (Inv, \mathtt{p} \circlearrowleft \mathbb{A}, \mathbb{R}) \quad (Inv \wedge \mathtt{p}) \Rightarrow \mathtt{g} \quad (Inv, \Delta, \Psi); (\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{I}}{(Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{yield}; \mathbb{I}} \quad \text{(YIELD)}$$

The YIELD rule requires that it is safe for the yielding thread to take back control after any state transition satisfying the assumption $\mathbb{A}$. Also the current thread cannot yield until it completes the required state transition, *i.e.,* an identity transition satisfies the local guarantee $\mathtt{g}$. Lastly, one must verify the remainder instruction sequence with the local guarantee reset to $\mathbb{G}$.

$$\frac{(Inv \wedge \mathtt{p}) \Rightarrow \mathtt{g}}{(Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{exit}} \quad \text{(EXIT)}$$

The rule EXIT is simple: it is safe for the current thread to terminate its execution only after it finishes the required transition described by $\mathtt{g}$, which is an identity transition.

Type-checking other instructions.

$$\frac{\begin{array}{l}\mathtt{c} \in \{\mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t,\ \mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t,\ \mathsf{movi}\ \mathtt{r}_d, \mathtt{w}\}, \mathtt{r}_d \notin \{\mathsf{rt}, \mathsf{ra}\} \\ (Inv \wedge \mathtt{p}) \Rightarrow (Inv \wedge \mathtt{p}') \circ \mathtt{c} \quad (Inv \wedge \mathtt{p}) \Rightarrow (\mathtt{g}' \circ \mathtt{c}) \Rightarrow \mathtt{g} \\ \mathbb{A} \Rightarrow \mathbb{A}' \circ \mathtt{c} \quad \mathbb{G}' \circ \mathtt{c} \Rightarrow \mathbb{G} \quad (Inv, \Delta, \Psi); (\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}') \vdash \mathbb{I}\end{array}}{(Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathtt{c}; \mathbb{I}} \quad \text{(SIMP)}$$

The rule SIMP covers the verification of instruction sequences starting with a simple command such as add, sub or movi. We require that the program not update registers rt and ra. In these cases, one must find an intermediate precondition $(\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}')$ under which the remainder instruction sequence $\mathbb{I}$ is well-formed.

The global invariant *Inv* and the intermediate assertion $\mathtt{p}'$ must hold on the updated machine state, and the intermediate guarantee $\mathtt{g}'$ applied to the updated machine state must be no weaker than the current guarantee $\mathtt{g}$ applied to the current state.

Since $\mathbb{A}$ and $\mathbb{G}$ are parameterized over $\mathbb{R}$, which will be changed by the instruction $\mathtt{c}$, one may change $\mathbb{A}$ and $\mathbb{G}$ to ensure the assumption does not become stronger and the guarantee does not become weaker.

$$\frac{\begin{array}{l}(\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}') = \Psi(\mathtt{f}) \\ (Inv \wedge \mathtt{p}) \Rightarrow \mathtt{p}' \quad (Inv \wedge \mathtt{p}) \Rightarrow \mathtt{g}' \Rightarrow \mathtt{g} \quad \mathbb{A} \Rightarrow \mathbb{A}' \quad \mathbb{G}' \Rightarrow \mathbb{G}\end{array}}{(Inv, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{jd}\ \mathtt{f}} \quad \text{(JD)}$$

The JD rule checks the specification of the target instruction sequence. As mentioned before, each instruction sequence can have its own specification, independent of the thread that will jump to it. It is safe for a thread to execute an instruction sequence as long as executing the instruction sequence does not require a stronger assumption than the thread's assumption $\mathbb{A}$, nor does it break the guarantee $\mathbb{G}$ of the thread.

Inference rules for memory operations are quite similar to the SIMP rule. Here we also need ensure the memory address is in the domain of the data heap.

$$\frac{\begin{array}{l} \mathtt{c} = \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w}) \quad \mathtt{r}_d \notin \{\mathsf{rt}, \mathsf{ra}\} \\ \forall \mathbb{M}.\forall \mathbb{R}.(\mathit{Inv} \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R}) \Rightarrow ((\mathbb{R}(\mathtt{r}_s) + \mathtt{w}) \in \mathit{dom}(\mathbb{M})) \\ (\mathit{Inv} \wedge \mathtt{p}) \Rightarrow (\mathit{Inv} \wedge \mathtt{p}') \circ \mathtt{c} \quad (\mathit{Inv} \wedge \mathtt{p}) \Rightarrow (\mathtt{g}' \circ \mathtt{c}) \Rightarrow \mathtt{g} \\ \forall \mathbb{S}, \mathbb{M}, \mathbb{M}'.\mathbb{A}\ (\mathbb{S}.\mathbb{R})\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathbb{A}'\ (\mathtt{Next}(\mathtt{c}, \mathbb{S}).\mathbb{R})\ \mathbb{M}\ \mathbb{M}' \\ \forall \mathbb{S}, \mathbb{M}, \mathbb{M}'.\mathbb{G}'\ (\mathtt{Next}(\mathtt{c}, \mathbb{S}).\mathbb{R})\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathbb{G}\ (\mathbb{S}.\mathbb{R})\ \mathbb{M}\ \mathbb{M}' \\ (\mathit{Inv}, \Delta, \Psi); (\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}') \vdash \mathbb{I} \end{array}}{(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathtt{c}; \mathbb{I}} \quad (\text{LD})$$

$$\frac{\begin{array}{l} \mathtt{c} = \mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s \\ \forall \mathbb{M}.\forall \mathbb{R}.(\mathit{Inv} \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R}) \Rightarrow ((\mathbb{R}(\mathtt{r}_d) + \mathtt{w}) \in \mathit{dom}(\mathbb{M})) \\ (\mathit{Inv} \wedge \mathtt{p}) \Rightarrow (\mathit{Inv} \wedge \mathtt{p}') \circ \mathtt{c} \quad (\mathit{Inv} \wedge \mathtt{p}) \Rightarrow (\mathtt{g}' \circ \mathtt{c}) \Rightarrow \mathtt{g} \\ (\mathit{Inv}, \Delta, \Psi); (\mathtt{p}', \mathtt{g}', \mathbb{A}, \mathbb{G}) \vdash \mathbb{I} \end{array}}{(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathtt{c}; \mathbb{I}} \quad (\text{ST})$$

Rules for conditional branching instructions are similar to the JD rule, which are straightforward to understand.

$$\frac{\begin{array}{l} (\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}') = \Psi(\mathtt{f}) \quad \mathbb{A} \Rightarrow \mathbb{A}' \quad \mathbb{G}' \Rightarrow \mathbb{G} \\ \forall \mathbb{S}.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) > \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{p}'\ \mathbb{S}) \\ \forall \mathbb{S}.\forall \mathbb{M}'.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) > \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{g}'\ \mathbb{S}\ \mathbb{M}') \Rightarrow (\mathtt{g}\ \mathbb{S}\ \mathbb{M}') \\ \forall \mathbb{S}.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) \leq \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{p}''\ \mathbb{S}) \\ \forall \mathbb{S}.\forall \mathbb{M}'.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) \leq \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{g}''\ \mathbb{S}\ \mathbb{M}') \Rightarrow (\mathtt{g}\ \mathbb{S}\ \mathbb{M}') \\ (\mathit{Inv}, \Delta, \Psi); (\mathtt{p}'', \mathtt{g}'', \mathbb{A}, \mathbb{G}) \vdash \mathbb{I} \end{array}}{(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \quad (\text{BGT})$$

$$\frac{\begin{array}{l} (\mathtt{p}', \mathtt{g}', \mathbb{A}', \mathbb{G}') = \Psi(\mathtt{f}) \quad \mathbb{A} \Rightarrow \mathbb{A}' \quad \mathbb{G}' \Rightarrow \mathbb{G} \\ \forall \mathbb{S}.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) = \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{p}'\ \mathbb{S}) \\ \forall \mathbb{S}.\forall \mathbb{M}'.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) = \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{g}'\ \mathbb{S}\ \mathbb{M}') \Rightarrow (\mathtt{g}\ \mathbb{S}\ \mathbb{M}') \\ \forall \mathbb{S}.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) \neq \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{p}''\ \mathbb{S}) \\ \forall \mathbb{S}.\forall \mathbb{M}'.(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S} \Rightarrow (\mathbb{S}(\mathtt{r}_s) \neq \mathbb{S}(\mathtt{r}_t)) \Rightarrow (\mathtt{g}''\ \mathbb{S}\ \mathbb{M}') \Rightarrow (\mathtt{g}\ \mathbb{S}\ \mathbb{M}') \\ (\mathit{Inv}, \Delta, \Psi); (\mathtt{p}'', \mathtt{g}'', \mathbb{A}, \mathbb{G}) \vdash \mathbb{I} \end{array}}{(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathsf{beq}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \quad (\text{BEQ})$$

4.5 Soundness of CMAP

The soundness of CMAP inference rules with respect to the operational semantics of the machine is established following the syntactic approach of proving type soundness [39]. From the "progress" and "preservation" lemmas, we can guarantee that given a well-formed program under compatible assumptions and guarantees, the current instruction sequence will be able to execute without getting "stuck". Furthermore, any safety property derivable from the global invariant will hold throughout the execution. We define $\mathbb{P} \overset{n}{\longmapsto} \mathbb{P}'$ as the relation of n-step ($n \geq 0$) program transitions. The soundness of CMAP is formally stated as Theorem 4.3.

Lemma 4.1 (Progress)
$\Phi = (\mathit{Inv}, \Delta, \Psi)$. *If there exist* Θ, p, g, $\mathbb{A}$ *and* $\mathbb{G}$ *such that* $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash ((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$*, then* $(\mathit{Inv}\ \mathbb{M})$*, and there exists a program* $\widetilde{\mathbb{P}}$ *such that* $((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I}) \longmapsto \widetilde{\mathbb{P}}$.

Lemma 4.2 (Preservation)
If $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}$ *and* $\mathbb{P} \longmapsto \widetilde{\mathbb{P}}$*, where* $\mathbb{P} = (\mathbb{S}, \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$ *and* $\widetilde{\mathbb{P}} = (\widetilde{\mathbb{S}}, \widetilde{\mathbb{Q}}, \mathbb{T}, \mathbb{C}, \widetilde{\mathbb{I}})$*, then there exist* $\widetilde{\Theta}$, $\widetilde{p}$, $\widetilde{g}$, $\widetilde{\mathbb{A}}$ *and* $\widetilde{\mathbb{G}}$ *such that* $\Phi; \widetilde{\Theta}; (\widetilde{p}, \widetilde{g}, \widetilde{\mathbb{A}}, \widetilde{\mathbb{G}}) \vdash \widetilde{\mathbb{P}}$.

Theorem 4.3 (Soundness)
$\Phi = (\mathit{Inv}, \Delta, \Psi)$. *If there exist* Θ, p, g, $\mathbb{A}$ *and* $\mathbb{G}$ *such that* $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}_0$*, then for any* $n \geq 0$*, there exist* $\mathbb{M}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{T}$, $\mathbb{C}$ *and* $\mathbb{I}$ *such that* $\mathbb{P}_0 \overset{n}{\longmapsto} ((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$ *and* $(\mathit{Inv}\ \mathbb{M})$.

The proofs for these two lemmas and the soundness theorem are given in Appendix A. We have also implemented the complete CMAP system [10] in the Coq proof assistant so we are confident that CMAP is indeed sound and can be used to certify general multi-threaded programs.

5. Examples

5.1 Unbounded Dynamic Thread Creation

In Figure 3 we showed a small program `main2` which spawns child threads within a while loop. This kind of unbounded dynamic thread creation cannot be supported using the cobegin/coend structure. We show how such a program is specified and verified using our logic. To simplify the specification, we trivialize the function `f` and `g` and let $\mathtt{f}(i) = i$ and $\mathtt{g}(x, \_) = x + 1$.

We assume the high-level program works in a preemptive mode. Figure 10 shows the CMAP implementation, where yield instructions are inserted to simulate the preemption. This also illustrates that our logic is general enough to simulate preemptive thread model.

To verify the safety property of the program, the programmer need find a global invariant and specifications for each static thread and the code heap. In Figure 10 we show definitions of assertions that are used to specify the program. Proof sketches are also inserted in the program. For ease of reading, we use named variables as short-hands for their values in memory. The primed variables represent the value of the variable after state transition. We also introduce the shorthand $[\mathtt{r}]$ for $\mathbb{R}(\mathtt{r})$.

The following formulae show the specifications of static threads, and the initial memory and instruction sequence.

$$\begin{array}{l} \mathit{Inv} \equiv \mathtt{True} \\ \Delta\ \equiv \{\mathtt{main} \rightsquigarrow (\mathtt{True}, \mathbb{A}_0, \mathbb{G}_0), \mathtt{chld} \rightsquigarrow (\mathtt{p}, \mathbb{A}, \mathbb{G})\} \\ \Psi\ \equiv \{\mathtt{loop} \rightsquigarrow (\mathtt{p}', \mathbb{G}_1, \mathbb{A}_1, \mathbb{G}_1), \mathtt{cont} \rightsquigarrow (\mathtt{p}_3, \mathbb{G}_3, \mathbb{A}_3, \mathbb{G}_3)\} \\ \text{Initial } \mathbb{M} \equiv \{\mathtt{data} \rightsquigarrow \_, \ldots, \mathtt{data} + 99 \rightsquigarrow \_\} \\ \text{Initial } \mathbb{I}\ \equiv \mathsf{movi}\ \mathtt{r}_0, 0;\ \mathsf{movi}\ \mathtt{r}_1, 1;\ \mathsf{movi}\ \mathtt{r}_2, 100;\ \mathsf{jd}\ \mathtt{loop} \end{array}$$

For each child thread, it is natural to assume that no other threads will touch its share of the data entry and guarantee that other threads' data entry will not be changed, as specified by $\mathbb{A}$ and $\mathbb{G}$. For the main thread, it assumes at the beginning that no other threads in the environment will change any of the data entries ($\mathbb{A}_0$).

Specifying the loop body is not easy. We need find a loop invariant $(\mathtt{p}', \mathbb{G}_1, \mathbb{A}_1, \mathbb{G}_1)$ to attach to the code label `loop`. At first glance, $\mathbb{A}_1$ and $\mathbb{G}_1$ can be defined the same as $\mathbb{A}_3$ and $\mathbb{G}_3$, respectively. However, this does not work because our FORK rule requires $\mathbb{G} \Rightarrow \mathbb{G}_1$, which cannot be satisfied. Instead, our $\mathbb{A}_1$ and $\mathbb{G}_1$ are polymorphic over the loop index $\mathtt{r}_0$, which reflects the composition of the changing thread environments. At the point that a new child thread is forked but the value of $\mathtt{r}_0$ has not been changed to reflect the environment change, we explicitly change the assumption and guarantee to $\mathbb{A}_2$ and $\mathbb{G}_2$. When the value of $\mathtt{r}_0$ is increased, we cast $\mathbb{A}_2$ and $\mathbb{G}_2$ back to $\mathbb{A}_1$ and $\mathbb{G}_1$.

5.2 The Readers-Writers Problem

Our logic is general enough to specify and verify general properties of concurrent programs. In this section, we give a simple solution of the readers-writers problem and show that there is no race conditions. This example also shows how P/V operations and lock primitives can be implemented and specified in CMAP.

Figure 11 shows the C-like pseudo code for readers and writers. Note that this simple code just ensures that there is no race con-

$\mathrm{p} \equiv 0 \le [\mathrm{ra}] < 100 \qquad \mathbb{A} \equiv \mathtt{data}[\mathrm{ra}] = \mathtt{data}'[\mathrm{ra}]$
$\mathbb{G} \equiv \forall i.(0 \le i < 100 \wedge i \ne [\mathrm{ra}]) \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$

$\mathbb{A}_0 \equiv \forall i. 0 \le i < 100 \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i]) \qquad \mathbb{G}_0 \equiv \mathtt{True}$

$\mathbb{A}_1 \equiv \forall i.(0 \le i < 100 \wedge i \ge [\mathrm{r}_0]) \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$
$\mathbb{G}_1 \equiv \forall i.(0 \le i < 100 \wedge i < [\mathrm{r}_0]) \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$
$\mathbb{A}_2 \equiv \forall i.(0 \le i < 100 \wedge i > [\mathrm{r}_0]) \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$
$\mathbb{G}_2 \equiv \forall i.(0 \le i < 100 \wedge i \le [\mathrm{r}_0]) \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$

$\mathbb{A}_3 \equiv \mathtt{True} \qquad \mathbb{G}_3 \equiv \forall i. 0 \le i < 100 \Rightarrow (\mathtt{data}[i] = \mathtt{data}'[i])$
$\mathrm{p}' \equiv 0 \le [\mathrm{r}_0] < 100 \wedge [\mathrm{r}_1] = 1 \wedge [\mathrm{r}_2] = 100 \qquad \mathrm{p}_3 \equiv [\mathrm{r}_0] = 100$

```
main : −{(True, A0, G0)}
       movi r0, 0
       movi r1, 1
       movi r2, 100
       jd loop
loop : −{(p', G1, A1, G1)}
       beq r0, r2, cont
       yield
       st r0(data), r0
       yield
       fork chld, r0
       −{(p', G1, A2, G2)}
       yield
       add r0, r0, r1
       −{(p', G1, A1, G1)}
       yield
       jd loop
cont : −{(p3, G3, A3, G3)}
       ...

chld : −{(p, A, G)}
       movi r0, data
       yield
       add r0, r0, ra
       yield
       ld r1, r0(0)
       yield
       movi r2, 1
       yield
       add r1, r1, r2
       yield
       st r0(0), r1
       exit
```

Figure 10. Loop: the CMAP program

```
Variables :
  int[100] rf, wf;
  int cnt, writ, l, v;
Initially :
  rf[i] = wf[i] = 0, 0 ≤ i < 100;
  cnt = 0 ∧ writ = 1 ∧ l = 0;

writer(int x){
  while(true){
    P(writ);

    wf[x] := 1;

    write v ...

    wf[x] := 0;

    V(writ);
  }
}

reader(int x){
  while(true){
    lock_acq(l);
      if (cnt = 0){
        P(writ);
      }
      cnt := cnt + 1;
      rf[x] := 1;
    lock_rel(l);

    read v ...

    lock_acq(l);
      rf[x] := 0;
      cnt := cnt − 1;
      if(cnt = 0){
        V(writ);
      }
    lock_rel(l);
}}
```

Figure 11. Readers & writers : the high-level program

ditions. It does not ensure fairness. Verification of liveness properties is part of our future work. We assume that 100 readers and writers will be created by a main thread. The main thread and its specification will be very similar to the main program shown in the previous section, so we omit it here and just focus on the readers and writers code. The array of `rf` and `wf` are not necessary for the implementation. They are introduced as auxiliary variables just for specification and verification purpose.

Figure 13 shows the CMAP implementation of the high-level pseudo code. Yielding is inserted at all the intervals of the atomic operations of the high-level program. The lock primitives and P/V operations, as shown in Figure 12, are implemented as program macros parameterized by the return label. They will be instantiated

```
acq(f) : −{}
         yield
         movi r0, 0
         movi r1, l
         ld r2, r1(0)
         bgt r2, r0, acq
         st r1(0), rt
         jd f

rel(f) : −{}
         yield
         movi r0, 0
         st r0(l), r0
         jd f

p_writ(f) : −{}
            yield
            movi r0, 0
            movi r1, writ
            ld r2, r1(0)
            beq r2, r0, p_writ
            st r1(0), r0
            jd f

v_writ(f) : −{}
            movi r1, 1
            movi r2, writ
            st r2(0), r1
            jd f
```

Figure 12. Lock & semaphore primitives

```
reader :  −{(p1, g, Ar, Gr)}
          JD acq(cont_1)

cont_1 :  −{(p2, g, Ar, Gr)}
          yield
          movi r0, 0
          movi r1, cnt
          ld r2, r1(0)
          yield
          beq r2, r0, getw
          jd inc_cnt

getw :    −{(p3, g, Ar, Gr)}
          JD p_writ(inc_cnt)

inc_cnt : −{(p4, g, Ar, Gr)}
          yield
          movi r1, cnt
          ld r2, r1(0)
          yield
          movi r3, 1
          add r2, r2, r3
          st r1(0), r2
          yield
          movi r1, 1
          st ra(rf), r1
          −{(p5, g, Ar, Gr)}
          JD rel(cont_2)

cont_2 :  −{(p6, g, Ar, Gr)}
          yield
          ...
          yield
          −{(p6, g, Ar, Gr)}
          JD acq(cont_3)

cont_3 :  −{(p7, g, Ar, Gr)}
          yield
          movi r0, 0
          st ra(rf), r0
          yield
          movi r1, cnt
          ld r2, r1(0)
          yield
          movi r3, 1
          sub r2, r2, r3
          st r1(0), r2
          yield
          beq r2, r0, relw
          −{(p8, g, Ar, Gr)}
          JD rel(reader)

relw :    −{(p9, g, Ar, Gr)}
          yield
          JD v_writ(cont_4)
cont_4 :  −{(p8, g, Ar, Gr)}
          JD rel(reader)

writer :  −{(p10, g, Aw, Gw)}
          JD p_writ(cont_5)

cont_5 :  −{(p11, g1, Aw, Gw)}
          movi r1, 1
          st ra(wf), r1
          yield
          ...
          yield
          movi r0, 0
          st ra(wf), r0
          −{(p11, g2, Aw, Gw)}
          JD v_writ(writer)
```

Figure 13. Readers & writers : the CMAP program

and inlined in the proper position of the code. We introduce a pseudo instruction JD to represent the inlining of the macro.

We define the global invariant and reader/writer's assumptions and guarantees in Figure 14. The code heap specifications are embedded in the code as annotations, as shown in Figure 13. Specifications of lock primitives and P/V operations are given at places they are inlined. Definitions of assertions and local guarantees used in code heap specifications are shown in Figure 14.

The assertion inv_1 says that out of the critical section protected by the lock, the value of the counter `cnt` is always consistent and reflects the number of the readers that can read the value of `v`; inv_2 says at one time there is at most one writer that can change the value of `v`; while inv_3 and inv_4 states the relationship between the counter `cnt`, the semaphore variable `writ` and the actual number

$$inv_1 \equiv \mathtt{l} = 0 \Rightarrow \sum_i \mathtt{rf}[i] = \mathtt{cnt} \qquad inv_2 \equiv \sum_i \mathtt{wf}[i] \leq 1$$
$$inv_3 \equiv \sum_i \mathtt{wf}[i] = 1 \Rightarrow (\mathtt{cnt} = 0 \land \mathtt{writ} = 0)$$
$$inv_4 \equiv \mathtt{writ} = 1 \Rightarrow \mathtt{cnt} = 0 \qquad \mathit{Inv} \equiv inv_1 \land inv_2 \land inv_3 \land inv_4$$

$$idr \equiv \forall i.\mathtt{rf}[i] = \mathtt{rf}'[i]$$
$$idr_1(i) \equiv \forall j.i \neq j \Rightarrow \mathtt{rf}[j] = \mathtt{rf}'[j] \qquad idr_2(i) \equiv \mathtt{rf}[i] = \mathtt{rf}'[i]$$
$$idw \equiv \forall i.\mathtt{wf}[i] = \mathtt{wf}'[i]$$
$$idw_1(i) \equiv \forall j.i \neq j \Rightarrow \mathtt{wf}[j] = \mathtt{wf}'[j] \qquad idw_2(i) \equiv \mathtt{wf}[i] = \mathtt{wf}'[i]$$

$$\mathbb{A}_r \equiv idr_2([\mathsf{ra}]) \land (\mathtt{rf}[\mathsf{ra}] = 1 \Rightarrow \mathtt{v} = \mathtt{v}') \land (\mathtt{l} = [\mathsf{rt}] \Rightarrow \mathtt{cnt} = \mathtt{cnt}')$$
$$\mathbb{G}_r \equiv idw \land idr_1([\mathsf{ra}]) \land \mathtt{v} = \mathtt{v}' \land (\mathtt{l} \notin \{[\mathsf{rt}], 0\} \Rightarrow \mathtt{cnt} = \mathtt{cnt}')$$

$$\mathbb{A}_w \equiv idw_2([\mathsf{ra}]) \land (\mathtt{wf}[\mathsf{ra}] = 1 \Rightarrow \mathtt{v} = \mathtt{v}')$$
$$\mathbb{G}_w \equiv idr \land idw_1([\mathsf{ra}]) \land ((\mathtt{writ} = 0 \land \mathtt{wf}[\mathsf{ra}] = 0) \Rightarrow \mathtt{v} = \mathtt{v}')$$

$$\mathrm{g} \equiv \lambda\mathbb{R}, \mathbb{M}, \mathbb{M}'.\mathbb{M} = \mathbb{M}' \qquad \mathrm{g}_1 \equiv \lambda\mathbb{R}, \mathbb{M}, \mathbb{M}'.\mathbb{M}\{\mathtt{wf}[\mathsf{ra}] \rightsquigarrow 1\} = \mathbb{M}'$$
$$\mathrm{g}_2 \equiv \lambda\mathbb{R}, \mathbb{M}, \mathbb{M}'.\mathbb{M}\{\mathtt{writ} \rightsquigarrow 1\} = \mathbb{M}' \qquad \mathrm{p}_1 \equiv \mathtt{rf}[\mathsf{ra}] = 0 \land \mathtt{l} \neq [\mathsf{rt}]$$
$$\mathrm{p}_2 \equiv \mathtt{rf}[\mathsf{ra}] = 0 \land \mathtt{l} = [\mathsf{rt}] \qquad \mathrm{p}_3 \equiv \mathrm{p}_2 \land \mathtt{cnt} = 0$$
$$\mathrm{p}_4 \equiv \mathrm{p}_2 \land \mathtt{writ} = 0 \qquad \mathrm{p}_5 \equiv \mathtt{rf}[\mathsf{ra}] = 1 \land \mathtt{l} = [\mathsf{rt}]$$
$$\mathrm{p}_6 \equiv \mathtt{rf}[\mathsf{ra}] = 1 \land \mathtt{l} \neq [\mathsf{rt}] \land \mathtt{cnt} > 0$$
$$\mathrm{p}_7 \equiv \mathtt{rf}[\mathsf{ra}] = 1 \land \mathtt{l} = [\mathsf{rt}] \land \mathtt{cnt} > 0 \qquad \mathrm{p}_8 \equiv \mathtt{l} = [\mathsf{rt}] \land \mathtt{rf}[\mathsf{ra}] = 0$$
$$\mathrm{p}_9 \equiv \mathtt{l} = [\mathsf{rt}] \land \mathtt{writ} = 0 \land \mathtt{cnt} = 0 \land \mathtt{rf}[\mathsf{ra}] = 0$$
$$\mathrm{p}_{10} \equiv \mathtt{wf}[\mathsf{ra}] = 0 \qquad \mathrm{p}_{11} \equiv \mathtt{wf}[\mathsf{ra}] = 0 \land \mathtt{writ} = 0 \land \mathtt{cnt} = 0$$

Figure 14. Program specifications

of writers that can write the data, which must be maintained to ensure the mutual exclusive access between readers and writers. The program invariant *Inv*, which is the conjunction of the four, must be satisfied at any step of the execution.

The assumptions and guarantees of readers and writers are parameterized by their thread id (in rt) and thread argument (in ra). The reader assumes ($\mathbb{A}_r$) that if it has the right to read the data $\mathtt{v}$ (*i.e.,* $\mathtt{rf}[\mathsf{ra}] = 1$), nobody can change the data. Also, if it owns the lock, nobody else can change the value of the counter. Finally, it assumes its auxiliary variable $\mathtt{rf}[\mathsf{ra}]$ will never be changed by others. Readers guarantee ($\mathbb{G}_r$) that they will never change the shared data $\mathtt{v}$ and auxiliary variables of other threads, and it will not revise the counter unless it owns the lock. Writers' assumption ($\mathbb{A}_w$) and guarantee ($\mathbb{G}_w$) are defined in a similar way to ensure the non-interference.

5.3 More Examples

Yu and Shao [42] have shown by examples that CCAP is expressive enough to verify general properties of concurrent programs, like mutual exclusion, deadlock freedom, and partial correctness. CMAP, as an extension of CCAP with dynamic thread creation, is more expressive than CCAP (it is straightforward to translate the CCAP code and verification to PCC packages in CMAP). Therefore, CMAP also applies for those CCAP examples. In Appendix B, we give a variation of the GCD program shown in [42], where collaborating threads are forked by a main thread. We also show how thread join can be implemented and reasoned in CMAP.

6. Related Work and Conclusion

We have presented a certified programming framework for verifying multi-threaded assembly code with unbounded dynamic thread creation. Our work is related to two directions of research: concurrency verification and PCC. The rely-guarantee method [23, 35, 36, 1, 13, 42] is one of the best studied technique for compositional concurrent program verification. However, most of the work on it are based on high-level languages or calculi, and none of them support unbounded dynamic thread creation. On the other hand, many PCC frameworks [28, 27, 2, 18, 41, 8, 42] have been proposed for machine/assembly code verification, but most of them only support sequential code. The only intersection point of these two directions is the work on CCAP [42], which applies the R-G method at the assembly level to verify general concurrency properties, like mutual exclusion and deadlock-freedom. Unfortunately, CCAP does not support dynamic thread creation either.

Recently, O'Hearn and others [31, 4, 3] applied separation logic to reason about concurrent programs. Bornat et al. [3] showed how to verify the race-free property of a program solving the readers and writers problem, which is similar to the program presented in this paper. However, their work heavily depends on the higher-level language features, such as resources and conditional critical regions. As other traditional work on concurrency verification, they only support nested cobegin/coend structure. It is not clear how their technique can be applied to support assembly code verification with unbounded dynamic thread creation.

A number of model checkers have been developed for concurrent software verification, but most of them only check programs with a fixed finite number of threads [22, 9, 19, 5, 16, 14]. The CIRC algorithm [20] supports unbounded number of threads, but it doesn't model the dynamic thread creation and the changing thread environment. Qadeer and Rehof [33] pointed out that verification of concurrent boolean program with unbounded parallelism is decidable if the number of context switches is bounded, but they do not directly verify dynamic thread creation either. Given a context bound k, they reduce a dynamic concurrent program P to a program Q with k+1 threads and verify Q instead. 3VMC [40] supports both unbounded number of threads and dynamic thread creation, but it is not based on the rely-guarantee method and does not support compositional verification.

Many type systems are also proposed to reason about concurrent programs [11, 12, 15, 17]. Unlike CMAP, which uses high-order predicate logic to verify general program properties, they are designed to automatically reason about specific properties of programs, like races, deadlocks and atomicity. Also, they do not directly generate proofs about program properties. Instead, proofs are implicitly embedded in their soundness proofs.

CMAP extends previous work on R-G method and CCAP with dynamic thread creation. We unify the concepts of a thread's assumption/guarantee and its environment's guarantee/assumption, and allow a thread to change its assumption and guarantee to track the changing environment caused by dynamic thread creation. Code segments in CMAP can be specified and verified once and used in multiple threads with different assumptions and guarantees, therefore CMAP achieves better modularity than CCAP. Some practical issues, such as argument passing at thread creation, thread local data, and multiple invocation of one copy of thread code, are also discussed to support practical multi-threaded programming. CMAP has been developed using the Coq proof assistant, along with a formal soundness proof and the verified example programs.

The goal of our work is to provide an explicit and general framework (with smaller TCB) such that code and specifications at the higher level can be compiled down to the assembly level. Although directly specifying and proving CMAP programs may be daunting, it is simpler at the higher level. Also, there are common concurrent idioms that would permit the assumption/guarantee to be generated automatically during compilation. For example, for critical sections protected by lock, the assumption is always like "nobody else will update the protected data if I am holding the lock..." (see Figure 14). Another scenario is illustrated by the example shown in Figure 3 and 10, where each thread is exclusively responsible for a single piece of global data. In these scenarios, the assumption is always like "nobody else will touch my share of data" and the guarantee is like "I will not update other threads' data". Automatically generating assumptions/guarantees and proofs for common idioms, and compiling higher level concurrent programs and specifications down to CMAP, will be our future work.

The thread primitives in CMAP are higher-level pseudo instructions. They can be replaced by system calls to certified thread libraries, which is part of our ongoing work. Also we separate issues in multi-threaded programming from the embedded code pointer problem which is addressed in a companion paper [30]. Applying that framework to our thread model will be part of the future work.

7. Acknowledgment

We thank Andrew McCreight and anonymous referees for suggestions and comments on an earlier version of this paper. Rodrigo Ferreira helped implement the CMAP language and its soundness proof in the Coq proof system. This research is based on work supported in part by grants from Intel and Microsoft, and NSF grant CCR-0208618. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems*, 17(3):507–535, 1995.

[2] A. W. Appel. Foundational proof-carrying code. In *Proc. 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258. IEEE Computer Society, June 2001.

[3] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proc. 32th ACM Symp. on Principles of Prog. Lang.*, pages 259–270, 2005.

[4] S. D. Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 16–34, 2004.

[5] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03: International Conference on Software Engineering*, pages 385–395, 2003.

[6] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, 2000. ACM Press.

[7] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[8] K. Crary. Toward a foundational typed assembly language. In *Proc. 30th ACM Symp. on Principles of Prog. Lang.*, pages 198–212, 2003.

[9] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 261–276, London, UK, 1999. Springer-Verlag.

[10] R. Ferreira and X. Feng. Coq (v8.0) implementation for CMAP language and the soundness proof. `http://flint.cs.yale.edu/publications/cmap.html`, Mar. 2005.

[11] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99 – Concurrency Theory, volume 1664 of LNCS*, pages 288–303. Springer-Verlag, 1999.

[12] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108. Springer-Verlag, 1999.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. 2002 European Symposium on Programming*, pages 262–277. Springer-Verlag, 2002.

[14] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. 10th SPIN workshop*, pages 213–224, May 2003.

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[16] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02: Automated Software Engineering*, pages 3–12, 2002.

[17] D. Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN International workshop on Types in Languages Design and Implementation*, pages 13–25, 2003.

[18] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, pages 89–100. IEEE Computer Society, July 2002.

[19] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[20] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI'04: Programming Language Design and Implementation*, pages 1–13, 2004.

[21] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[22] G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proc. 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 453–455. Springer, 1995.

[23] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Programming Languages and Systems*, 5(4):596–619, 1983.

[24] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[25] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.

[26] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: a realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pages 85–97. ACM Press, Jan. 1998.

[28] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Prog. Lang.*, pages 106–119. ACM Press, Jan. 1997.

[29] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998.

[30] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. Technical report, Dec. 2004. `http://flint.cs.yale.edu/publications/ecap.html`.

[31] P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.

[32] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[33] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS'05: Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[34] J. H. Reppy. CML: A higher concurrent language. In *Proc. 1991 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, Ontario, Canada, 1991. ACM Press.

[35] E. W. Stark. A proof technique for rely/guarantee properties. In *Proc. 5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *LNCS*, pages 369–391, 1985.

[36] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *Proc. 2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 510–525, 1991.

[37] The Coq Development Team. The Coq proof assistant reference manual. The Coq release v7.1, Oct. 2001.

[38] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

[39] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[40] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. 28th ACM Symp. on Principles of Prog. Lang.*, pages 27–40, New York, NY, USA, 2001. ACM Press.

[41] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming, LNCS Vol. 2618*, pages 363–379, 2003.

[42] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Proc. 2004 ACM SIGPLAN Int'l Conf. on Functional Prog.*, September 2004.

A. The Soundness of CMAP

Lemma A.1 (Progress)
$\Phi = (\mathit{Inv}, \Delta, \Psi)$. *If there exists* Θ, p, g, $\mathbb{A}$ *and* $\mathbb{G}$ *such that* $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash ((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$*, then* $(\mathit{Inv}\ \mathbb{M})$*, and there exists a program* $\widetilde{\mathbb{P}}$ *such that* $((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I}) \longmapsto \widetilde{\mathbb{P}}$.

Proof sketch: By induction over the structure of $\mathbb{I}$. $(\mathit{Inv}\ \mathbb{M})$ holds by the assumption and the inversion of the rule PROG. In the cases where $\mathbb{I}$ starts with add, sub, movi, fork, exit, or yield, the program can always make a step by the definition of the operational semantics (recall in our abstract machine we always assume there is an "idle" thread in $\mathbb{Q}$, therefore the instructions fork and exit can go through). In the cases where $\mathbb{I}$ starts with ld and st, the side conditions for make a step, as defined by the operational semantics, are established by the rules LD and ST. In the cases where $\mathbb{I}$ starts with bgt or beq, or where $\mathbb{I}$ is jd, the operatinal semantics may fetch a code block from the code heap; such a code block exists by the inversion of the rule CDHP. $\square$

Lemma A.2 (Preservation)
If $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}$ *and* $\mathbb{P} \longmapsto \widetilde{\mathbb{P}}$*, where* $\mathbb{P} = (\mathbb{S}, \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$ *and* $\widetilde{\mathbb{P}} = (\widetilde{\mathbb{S}}, \widetilde{\mathbb{Q}}, \mathbb{T}, \mathbb{C}, \widetilde{\mathbb{I}})$*, then there exist* $\widetilde{\Theta}$, $\widetilde{p}$, $\widetilde{g}$, $\widetilde{\mathbb{A}}$ *and* $\widetilde{\mathbb{G}}$ *such that* $\Phi; \widetilde{\Theta}; (\widetilde{p}, \widetilde{g}, \widetilde{\mathbb{A}}, \widetilde{\mathbb{G}}) \vdash \widetilde{\mathbb{P}}$.

Proof sketch. By the assumption $\Phi; \Theta; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}$ and the inversion of the rule PROG, we know that

1. $(\mathit{Inv}, \Delta, \Psi) = \Phi$, $(\mathbb{M}, \mathbb{R}) = \mathbb{S}$, $\mathtt{t} = \mathbb{S}(\mathsf{rt})$;
2. $\Phi \vdash \mathbb{T}$;
3. $\Phi \vdash \mathbb{C}$;
4. $(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S}$;
5. $\Phi; (\mathtt{p}, \mathtt{g}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{I}$;
6. $\Phi; \Theta; (\mathtt{g}, \mathbb{S}) \vdash \mathbb{Q}$;
7. $\mathtt{NI}(\Theta\{\mathtt{t} \leadsto (\mathtt{p}, \mathbb{A}, \mathbb{G})\}, \mathbb{Q}\{\mathtt{t} \leadsto (\mathbb{R}, \mathbb{I})\})$ (see Section 4.4 for the definition of $\mathtt{NI}$).

We prove the lemma by induction over the structure of $\mathbb{I}$. Here we only give the detailed proof of cases where $\mathbb{I}$ starts with fork and yield. Proof for the rest cases are trivial.

Case $\mathbb{I} = \mathsf{fork}\ \mathtt{h}, \mathtt{r}; \mathbb{I}''$.
By the operational semantics, we know that $\widetilde{\mathbb{S}} = \mathbb{S}$, $\widetilde{\mathbb{Q}} = \mathbb{Q}\{\mathtt{t}' \leadsto (\mathbb{R}', \mathbb{I}')\}$, and $\widetilde{\mathbb{I}} = \mathbb{I}''$, where $\mathbb{R}' = \{\mathtt{r}_0 \leadsto \_, \ldots, \mathtt{r}_{15} \leadsto \_, \mathsf{rt} \leadsto \mathtt{t}', \mathsf{ra} \leadsto \mathbb{S}(\mathtt{r})\}$, $\mathbb{I}' = \mathbb{T}(\mathtt{h})$, $\mathtt{t}' \notin \mathit{dom}(\mathbb{Q})$ and $\mathtt{t}' \neq \mathtt{t}$. According to 5 and the inversion of the FORK rule, we know that there exist $\mathtt{p}'$, $\mathbb{A}'$, $\mathbb{G}'$, $\mathbb{A}''$ and $\mathbb{G}''$ such that

f.1 $(\mathtt{p}', \mathbb{A}', \mathbb{G}') = \Delta(\mathtt{h})$;
f.2 $\mathbb{A} \Rightarrow \mathbb{A}''$, $\mathbb{G}'' \Rightarrow \mathbb{G}$, $(\mathbb{A} \vee \mathbb{G}'') \Rightarrow \mathbb{A}'$, and $\mathbb{G}' \Rightarrow (\mathbb{G} \wedge \mathbb{A}'')$;
f.3 $\forall (\mathbb{M}, \mathbb{R}). \forall \mathbb{R}'. \forall \mathbb{M}'. (\mathit{Inv} \wedge \mathtt{p})\ (\mathbb{M}, \mathbb{R}) \Rightarrow \mathbb{R}(\mathsf{rt}) \neq \mathbb{R}'(\mathsf{rt}) \Rightarrow \mathbb{R}(\mathtt{r}) = \mathbb{R}'(\mathsf{ra}) \Rightarrow \mathtt{g}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}' \Rightarrow \mathtt{p}'\ (\mathbb{M}', \mathbb{R}')$;
f.4 $(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathtt{g}, \mathbb{A}'', \mathbb{G}'') \vdash \mathbb{I}''$

Then we let $\widetilde{\Theta} = \Theta\{\mathtt{t}' \leadsto (\mathtt{p}', \mathbb{A}', \mathbb{G}')\}$, $\widetilde{\mathtt{p}} = \mathtt{p}$, $\widetilde{\mathtt{g}} = \mathtt{g}$, $\widetilde{\mathbb{A}} = \mathbb{A}''$, and $\widetilde{\mathbb{G}} = \mathbb{G}''$.

According to the rule PROG, to prove $\Phi; \widetilde{\Theta}; (\widetilde{\mathtt{p}}, \widetilde{\mathtt{g}}, \widetilde{\mathbb{A}}, \widetilde{\mathbb{G}}) \vdash \widetilde{\mathbb{P}}$, we need prove the following:

- $\Phi \vdash \mathbb{T}$ and $\Phi \vdash \mathbb{C}$. They trivially follow 2 and 3. Since these two conditions never change, we will omit the proof of them in the following cases.
- $(\mathit{Inv} \wedge \mathtt{p})\ \mathbb{S}$. By 4.
- $\Phi; (\mathtt{p}, \mathtt{g}, \mathbb{A}'', \mathbb{G}'') \vdash \mathbb{I}''$. By f.4.
- $\Phi; \widetilde{\Theta}; (\mathtt{g}, \mathbb{S}) \vdash \widetilde{\mathbb{Q}}$. By 6 and the inversion of the rule DTHRDS, we need check the following for the new thread $\mathtt{t}'$:
  - $\forall \mathbb{M}', \mathbb{M}''. (\mathit{Inv} \wedge \mathtt{p}')(\mathbb{M}', \mathbb{R}') \Rightarrow \mathbb{A}'\ \mathbb{R}'\ \mathbb{M}'\ \mathbb{M}'' \Rightarrow \mathtt{p}'(\mathbb{M}'', \mathbb{R}')$. By 2 and the inversion of the rule THRDS.
  - $(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}') \vdash \mathbb{I}'$. By 2 and the inversion of the rule THRDS.
  - $\forall \mathbb{M}''. \mathtt{g}\ \mathbb{S}\ \mathbb{M}'' \Rightarrow \mathtt{p}'\ (\mathbb{M}'', \mathbb{R}')$. By 4, and f.3.
- $\mathtt{NI}(\widetilde{\Theta}\{\mathtt{t} \leadsto (\mathtt{p}, \mathbb{A}'', \mathbb{G}'')\}, \widetilde{\mathbb{Q}}\{\mathtt{t} \leadsto (\mathbb{R}, \mathbb{I}'')\})$. By 4, 7, f.2, and Lemma 3.4.

Case $\mathbb{I} = \mathsf{yield}; \mathbb{I}''$.
By the operational semantics, we know that $\widetilde{\mathbb{S}} = (\mathbb{M}, \mathbb{R}')$, $\widetilde{\mathbb{Q}} = (\mathbb{Q}\{\mathbb{R}(\mathsf{rt}) \leadsto (\mathbb{R}, \mathbb{I}'')\}) \setminus \{\mathtt{t}\}$, and $\widetilde{\mathbb{I}} = \mathbb{I}'$, where $\mathtt{t} \in \mathit{dom}(\mathbb{Q})$ and $(\mathbb{R}', \mathbb{I}') = \mathbb{Q}(\mathtt{t})$ or $\mathtt{t} = \mathbb{R}(\mathsf{rt})$ and $(\mathbb{R}', \mathbb{I}') = (\mathbb{R}, \mathbb{I}'')$.

If $\mathtt{t} = \mathbb{R}(\mathsf{rt})$, the yield instruction is like a no-op instruction. The proof is trivial. So we just consider the case that $\mathtt{t} \in \mathit{dom}(\mathbb{Q})$ and $(\mathbb{R}', \mathbb{I}') = \mathbb{Q}(\mathtt{t})$.

We let $\widetilde{\Theta} = (\Theta\{\mathbb{R}(\mathsf{rt}) \leadsto (\mathtt{p}, \mathbb{A}, \mathbb{G})\}) \setminus \{\mathtt{t}\}$, $\widetilde{\mathtt{p}} = \mathtt{p}'$, $\widetilde{\mathtt{g}} = \mathbb{G}'$, $\widetilde{\mathbb{A}} = \mathbb{A}'$, and $\widetilde{\mathbb{G}} = \mathbb{G}'$, where $(\mathtt{p}', \mathbb{A}', \mathbb{G}') = \Theta(\mathtt{t})$. According to the rule PROG, to prove $\Phi; \widetilde{\Theta}; (\widetilde{\mathtt{p}}, \widetilde{\mathtt{g}}, \widetilde{\mathbb{A}}, \widetilde{\mathbb{G}}) \vdash \widetilde{\mathbb{P}}$, we need prove:

- $(\mathit{Inv} \wedge \mathtt{p}')\ (\mathbb{M}, \mathbb{R}')$. By 4 we know that $\mathit{Inv}\ (\mathbb{M}, \mathbb{R}')$ (note that Inv is independent of $\mathbb{R}$).

  By 6 and the inversion of the DTHRDS rule, we know that $\forall \mathbb{M}''. \mathtt{g}\ \mathbb{S}\ \mathbb{M}'' \Rightarrow \mathtt{p}'\ (\mathbb{M}'', \mathbb{R}')$, also by 4, 5 and the inversion of the YIELD rule we get $\mathtt{g}\ \mathbb{S}\ \mathbb{M}$. Therefore we can prove $\mathtt{p}'\ (\mathbb{M}, \mathbb{R}')$.
- $\Phi; (\mathtt{p}', \mathbb{G}', \mathbb{A}', \mathbb{G}') \vdash \mathbb{I}'$. By 6 and the inversion of DTHRDS.
- $\Phi; \widetilde{\Theta}; (\mathbb{G}', (\mathbb{M}, \mathbb{R}')) \vdash \widetilde{\mathbb{Q}}$. To prove this, we only check the following for the yielding thread:
  - $\forall \mathbb{M}', \mathbb{M}''. (\mathit{Inv} \wedge \mathtt{p})(\mathbb{M}', \mathbb{R}) \Rightarrow \mathbb{A}\ \mathbb{R}\ \mathbb{M}'\ \mathbb{M}'' \Rightarrow \mathtt{p}(\mathbb{M}'', \mathbb{R})$. This follows 5 and the inversion of the rule YIELD.
  - $(\mathit{Inv}, \Delta, \Psi); (\mathtt{p}, \mathbb{G}, \mathbb{A}, \mathbb{G}) \vdash \mathbb{I}''$. This follows 5 and the inversion of the rule YIELD.
  - $\forall \mathbb{M}''. \mathbb{G}'\ \mathbb{R}'\ \mathbb{M}\ \mathbb{M}'' \Rightarrow \mathtt{p}\ (\mathbb{R}, \mathbb{M}'')$. By 7 we know $\forall \mathbb{M}', \mathbb{M}''. \mathbb{G}'\ \mathbb{R}'\ \mathbb{M}'\ \mathbb{M}'' \Rightarrow \mathbb{A}\ \mathbb{R}\ \mathbb{M}'\ \mathbb{M}''$. Therefore, we only need prove: $\forall \mathbb{M}''. \mathbb{A}\ \mathbb{R}\ \mathbb{M}\ \mathbb{M}'' \Rightarrow \mathtt{p}\ (\mathbb{M}'', \mathbb{R})$. By 5, the inversion of the rule YIELD, and 4 we get it.
- $\mathtt{NI}(\widetilde{\Theta}\{\mathtt{t} \leadsto (\mathtt{p}', \mathbb{A}', \mathbb{G}')\}, \widetilde{\mathbb{Q}}\{\mathtt{t} \leadsto (\mathbb{R}', \mathbb{I}')\})$, which follows 7.

$\square$

Theorem A.3 (Soundness)
$\Phi = (\mathit{Inv}, \Delta, \Psi)$. *If there exists* Θ, p, g, $\mathbb{A}$ *and* $\mathbb{G}$ *such that* $\Phi; \Theta; (p, g, \mathbb{A}, \mathbb{G}) \vdash \mathbb{P}_0$*, then for any* $n \geq 0$*, there exist* $\mathbb{M}$, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{T}$, $\mathbb{C}$ *and* $\mathbb{I}$ *such that* $\mathbb{P}_0 \overset{n}{\longmapsto} ((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$ *and* $(\mathit{Inv}\ \mathbb{M})$.

Proof sketch. Given the progress and the preservation lemmas, this theorem can be easily proved by induction over n. However, instead of proving $(\mathit{Inv}\ \mathbb{M})$, we need strengthen the induction hypothesis and prove " $\Phi; \widetilde{\Theta}; (\widetilde{\mathtt{p}}, \widetilde{\mathtt{g}}, \widetilde{\mathbb{A}}, \widetilde{\mathbb{G}}) \vdash ((\mathbb{M}, \mathbb{R}), \mathbb{Q}, \mathbb{T}, \mathbb{C}, \mathbb{I})$ for some $\widetilde{\Theta}$, $\widetilde{\mathtt{p}}$, $\widetilde{\mathtt{g}}$, $\widetilde{\mathbb{A}}$ and $\widetilde{\mathbb{G}}$". $\square$

B. Partial Correctness of A Lock-Free Program

Figure 16 and 17 give the CMAP implementation of the GCD algorithm shown in Figure 15, where two threads collaborate to compute the greatest common divisor of two numbers. This is a lock-free algorithm, *i.e.,* no synchronization is required to ensure the non-interference even if atomic operations are machine instructions. To show the lock-free property, we insert yield instructions at every program point of the chld thread (some yield instructions are omitted in the main thread for clarity).

In Figure 18 we show definitions of assertions used to specify the program. The following formulae show the specifications of static threads, and the initial memory and instruction sequence.

$\mathit{Inv} \equiv \mathtt{True}$
$\Delta \equiv \{\mathtt{main} \leadsto (\mathtt{True}, \mathbb{A}_0, \mathbb{G}_0), \mathtt{chld} \leadsto (\mathtt{p}_2 \wedge \mathtt{p}_8, \mathbb{A}_g, \mathbb{G}_g)\}$
Initial $\mathbb{M} \equiv \{\mathtt{a} \leadsto \_, \mathtt{b} \leadsto \_, \mathtt{flag} \leadsto \_, (\mathtt{flag} + 1) \leadsto \_\}$
Initial $\mathbb{I} \equiv \mathsf{jd}\ \mathtt{begn}$

```
Variables:
  nat a, b;
  nat[2] flag;

Main :                        void gcd(int x){
  a := α;   b := β;             while (a ≠ b){
  flag[0] := 0;                   if ((a > b)&&(x = 0))
  fork(gcd, 0);                     a := a − b;
  flag[1] := 0;                   else if ((a < b)&&(x = 1))
  fork(gcd, 1);                     b := b − a;
  while (!flag[0])              }
    yield;                      flag[x] := 1;
  while (!flag[1])            }
    yield;
  post proc ...
```

Figure 15. GCD: high-level program

```
main : −{(True, A0, G0)}          next :  −{p2, g, A1, G1}
       jd begn                            yield
begn : −{(True, G0, A0, G0)}              st r1(1), r0
       movi r0, α                         yield
       movi r1, a                         movi r2, 1
       st r1(0), r0                       fork chld, r2
       yield                      join1 : −{p2 ∧ p3, g, A2, G2}
       movi r0, β                         yield
       movi r1, b                         ld r2, r1(0)
       st r1(0), r0                       beq r0, r2, join1
       yield                      join2 : −{p2 ∧ p4, g, A2, G2}
       movi r0, 0                         yield
       movi r1, flag                      ld r2, r1(1)
       st r1(0), r0                       beq r0, r2, join2
       yield                      post :  −{(p2 ∧ p5, g, A2, G2)}
       movi r2, 0                         yield
       fork chld, r2                      ...
```

Figure 16. GCD: the `main` thread

Inv is simply set to $\mathtt{True}$. The partial correctness of the gcd algorithm is inferred by the fact that p_2 is established at the entry point and $\mathbb{G}_g$ is satisfied.

Two child threads are created using the same static thread `chld`. They use thread arguments to distinguish their task. Correspondingly, the assumption $\mathbb{A}_g$ and $\mathbb{G}_g$ of the static thread `chld` also use the thread argument to distinguish the specification of different dynamic copies. The child thread does not change its assumption and guarantee throughout its lifetime. Therefore we omit $\mathbb{A}_g$ and $\mathbb{G}_g$ in the code heap specifications. Since we insert yield instructions at every program point, every local guarantee is simply $\mathbb{G}_g$, which is also omitted in the specifications.

At the data initiation stage, the main thread assumes no threads in the environment changes a, b and the flags, and guarantees nothing, as shown in $\mathbb{A}_0$ and $\mathbb{G}_0$ (see Figure 18). After creating child threads, the main thread changes its assumption and guarantee to reflect the changing environment. The new assumption is just the disjunction of the previous assumption and the guarantee of the new thread (instantiated by the thread id and thread argument), similarly the new guarantee is the conjunction of the previous guarantee and the new thread's assumption.

This example also shows how thread join can be implemented in CMAP by synchronization. We use one flag for each thread to indicate if the thread is alive or not. The assumptions and guarantees of the main thread also take advantages of these flags so that it can weaken its guarantee and strengthen its assumption after the child threads die.

```
chld : −{(p2 ∧ p8, Ag, Gg)}        calc1 : −{p2 ∧ p9 ∧ p12}
       jd gcd                               yield
gcd :  −{p2 ∧ p8}                           beq r0, ra, cld1
       yield                                yield
       movi r0, 0                           jd loop
loop : −{p2 ∧ p8 ∧ p9}             cld1 :  −{p2 ∧ p9 ∧ p13}
       yield                                yield
       ld r1, r0(a)                         sub r3, r1, r2
       yield                                yield
       −{p2 ∧ p9 ∧ p10}                     st r0(a), r3
       ld r2, r0(b)                         yield
       yield                                jd loop
       −{p2 ∧ p9 ∧ p11}            calc2 : −{p2 ∧ p9 ∧ p14}
       beq r1, r2, done                     yield
       yield                                movi r3, 1
       bgt r1, r2, calc1                    yield
       yield                                beq r3, ra, cld2
       jd calc2                             yield
done : −{p1 ∧ p2 ∧ p8}                      jd loop
       yield                       cld2 :  −{p2 ∧ p9 ∧ p15}
       movi r0, 1                           yield
       yield                                sub r3, r2, r1
       st ra(flag), r0                      yield
       exit                                 st r0(b), r3
                                            yield
                                            jd loop
```

Figure 17. GCD: the `chld` thread

$$\mathrm{id}_1 \equiv \mathtt{a} = \mathtt{a}' \qquad \mathrm{id}_2 \equiv \mathtt{b} = \mathtt{b}'$$
$$\mathrm{id}_3 \equiv (\mathtt{flag}[0] = \mathtt{flag}'[0]) \wedge (\mathtt{flag}[1] = \mathtt{flag}'[1])$$

$$\mathbb{G}_0 \equiv \mathtt{True} \qquad \mathbb{A}_0 \equiv \mathrm{id}_1 \wedge \mathrm{id}_2 \wedge \mathrm{id}_3$$

$$\mathrm{p} \equiv \mathrm{id}_2 \wedge (\mathtt{a} > \mathtt{b} \Rightarrow (\mathtt{GCD}(\mathtt{a},\mathtt{b}) = \mathtt{GCD}(\mathtt{a}',\mathtt{b}'))) \wedge (\mathtt{a} \leq \mathtt{b} \Rightarrow \mathrm{id}_1)$$
$$\mathrm{p}' \equiv \mathrm{id}_1 \wedge (\mathtt{a} < \mathtt{b} \Rightarrow (\mathtt{GCD}(\mathtt{a},\mathtt{b}) = \mathtt{GCD}(\mathtt{a}',\mathtt{b}'))) \wedge (\mathtt{a} \geq \mathtt{b} \Rightarrow \mathrm{id}_2)$$

$$\begin{aligned}\mathbb{G}_g \equiv\ & (\mathtt{flag}'[\mathsf{ra}] = 0 \Rightarrow (([\mathsf{ra}] = 0 \wedge \mathrm{p}) \vee ([\mathsf{ra}] = 1 \wedge \mathrm{p}')))\\ & \wedge(\mathtt{flag}'[\mathsf{ra}] = 1 \Rightarrow (\mathrm{id}_1 \wedge \mathrm{id}_2 \wedge \mathtt{a}' = \mathtt{b}'))\\ & \wedge(\mathtt{flag}[\mathsf{ra}] = 1 \Rightarrow \mathbb{S} = \mathbb{S}')\\ & \wedge(\mathtt{flag}[1-\mathsf{ra}] = \mathtt{flag}'[1-\mathsf{ra}])\end{aligned}$$
$$\begin{aligned}\mathbb{A}_g \equiv\ & (\mathtt{flag}'[\mathsf{ra}] = 0) \Rightarrow ((\mathtt{flag}[\mathsf{ra}] = \mathtt{flag}'[\mathsf{ra}])\\ & \qquad \wedge(([\mathsf{ra}] = 0 \wedge \mathrm{p}') \vee ([\mathsf{ra}] = 1 \wedge \mathrm{p})))\end{aligned}$$

$$\mathbb{G}_1 \equiv \mathbb{G}_0 \wedge (\mathtt{flag}'[0] = 0 \Rightarrow ((\mathtt{flag}[0] = \mathtt{flag}'[0]) \wedge \mathrm{p}'))$$
$$\begin{aligned}\mathbb{A}_1 \equiv\ & \mathbb{A}_0 \vee ((\mathtt{flag}[1] = \mathtt{flag}'[1]) \wedge (\mathtt{flag}'[0] = 0 \Rightarrow \mathrm{p})\\ & \wedge(\mathtt{flag}'[0] = 1 \Rightarrow (\mathrm{id}_1 \wedge \mathrm{id}_2 \wedge \mathtt{a}' = \mathtt{b}')))\\ & \wedge(\mathtt{flag}[0] = 1 \Rightarrow \mathbb{S} = \mathbb{S}'))\end{aligned}$$

$$\mathbb{G}_2 \equiv \mathbb{G}_1 \wedge (\mathtt{flag}'[1] = 0 \Rightarrow ((\mathtt{flag}[1] = \mathtt{flag}'[1]) \wedge \mathrm{p}))$$
$$\begin{aligned}\mathbb{A}_2 \equiv\ & \mathbb{A}_1 \vee ((\mathtt{flag}[0] = \mathtt{flag}'[0]) \wedge (\mathtt{flag}'[1] = 0 \Rightarrow \mathrm{p}')\\ & \wedge(\mathtt{flag}'[1] = 1 \Rightarrow (\mathrm{id}_1 \wedge \mathrm{id}_2 \wedge \mathtt{a}' = \mathtt{b}')))\\ & \wedge(\mathtt{flag}[1] = 1 \Rightarrow \mathbb{S} = \mathbb{S}'))\end{aligned}$$

$$\mathrm{g} \equiv \mathbb{S} = \mathbb{S}' \qquad \mathrm{p}_1 \equiv \mathtt{a} = \mathtt{b} \qquad \mathrm{p}_2 \equiv \mathtt{GCD}(\mathtt{a},\mathtt{b}) = \mathtt{GCD}(\alpha,\beta)$$
$$\mathrm{p}_3 \equiv \mathtt{flag}[0] = 0 \vee (\mathtt{flag}[0] = 1 \wedge \mathrm{p}_1)$$
$$\mathrm{p}_4 \equiv (\mathtt{flag}[1] = 0 \vee \mathtt{flag}[1] = 1) \wedge \mathtt{flag}[0] = 1 \wedge \mathrm{p}_1$$
$$\mathrm{p}_5 \equiv \mathtt{flag}[0] = 1 \wedge \mathtt{flag}[1] = 1 \wedge \mathrm{p}_1$$

$$\mathrm{p}_6 \equiv [\mathsf{ra}] = 0 \wedge \mathtt{flag}[0] = 0 \qquad \mathrm{p}_7 \equiv [\mathsf{ra}] = 1 \wedge \mathtt{flag}[1] = 0$$
$$\mathrm{p}_8 \equiv \mathrm{p}_6 \vee \mathrm{p}_7 \qquad \mathrm{p}_9 \equiv [\mathrm{r}_0] = 0$$
$$\mathrm{p}_{10} \equiv (\mathrm{p}_6 \wedge [\mathrm{r}_1] = \mathtt{a}) \vee (\mathrm{p}_7 \wedge ([\mathrm{r}_1] \leq \mathtt{b} => [\mathrm{r}_1] = \mathtt{a}))$$
$$\begin{aligned}\mathrm{p}_{11} \equiv\ & (\mathrm{p}_6 \wedge [\mathrm{r}_1] = \mathtt{a} \wedge ([\mathrm{r}_1] \geq [\mathrm{r}_2] => [\mathrm{r}_2] = \mathtt{b}))\\ & \vee(\mathrm{p}_7 \wedge [\mathrm{r}_2] = \mathtt{b} \wedge ([\mathrm{r}_1] \leq [\mathrm{r}_2] => [\mathrm{r}_1] = \mathtt{a}))\end{aligned}$$

$$\mathrm{p}_{12} \equiv [\mathrm{r}_1] > [\mathrm{r}_2] \wedge ([\mathsf{ra}] = 0 \Rightarrow (\mathtt{flag}[0] = 0 \wedge [\mathrm{r}_1] = \mathtt{a} \wedge [\mathrm{r}_2] = \mathtt{b}))$$
$$\mathrm{p}_{13} \equiv \mathrm{p}_6 \wedge [\mathrm{r}_1] > [\mathrm{r}_2] \wedge [\mathrm{r}_1] = \mathtt{a} \wedge [\mathrm{r}_2] = \mathtt{b}$$
$$\mathrm{p}_{14} \equiv [\mathrm{r}_1] < [\mathrm{r}_2] \wedge ([\mathsf{ra}] = 1 \Rightarrow (\mathtt{flag}[1] = 0 \wedge [\mathrm{r}_1] = \mathtt{a} \wedge [\mathrm{r}_2] = \mathtt{b}))$$
$$\mathrm{p}_{15} \equiv \mathrm{p}_7 \wedge [\mathrm{r}_1] < [\mathrm{r}_2] \wedge [\mathrm{r}_1] = \mathtt{a} \wedge [\mathrm{r}_2] = \mathtt{b}$$

Figure 18. GCD: Assertion Definitions