Verification of Safety Properties for Concurrent Assembly Code*

Dachuan Yu and Zhong Shao
Department of Computer Science, Yale University
New Haven, CT 06520-8285, U.S.A.
{yu, shao} @ cs.yale.edu

Abstract

Concurrency, as a useful feature of many modern programming languages and systems, is generally hard to reason about. Although existing work has explored the verification of concurrent programs using high-level languages and calculi, the verification of concurrent assembly code remains an open problem, largely due to the lack of abstraction at a low-level. Nevertheless, it is sometimes necessary to reason about assembly code or machine executables so as to achieve higher assurance.

In this paper, we propose a logic-based "type" system for the static verification of concurrent assembly programs, applying the "invariance proof" technique for verifying general safety properties and the "assume-guarantee" paradigm for decomposition. In particular, we introduce a notion of "local guarantee" for the thread-modular verification in a non-preemptive setting.

Our system is fully mechanized. Its soundness has been verified using the Coq proof assistant. A safety proof of a program is semi-automatically constructed with help of Coq, allowing the verification of even undecidable safety properties. We demonstrate the usage of our system using three examples, addressing mutual exclusion, deadlock freedom, and partial correctness respectively.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, mechanical verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

*This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF ITR grant CCR-0081590, 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.

General Terms

Languages, Verification

Keywords

Assembly, concurrency, local guarantee

1 Introduction

The Verifying Compiler, as a Grand Challenge [20] to the programming languages community, solicits a programming framework which guarantees the correctness of a program before running it. The criterion of correctness ranges from type-safety, which has been widely applied in programming practice, to total correctness of critical components, whose application is often limited by an evaluation of the cost and benefits of accurate and complete formalization.

Whereas it is difficult to ensure correctness for sequential systems, it is even more so for concurrent systems due to the interaction and interference between multiple threads. Much existing work has explored the verification of concurrent programs using high-level languages and calculi (*e.g.,* CSP [19], CCS [26], CML [37], TLA [24]). Unfortunately, the verification of concurrent assembly code remains an open problem. On the one hand, the low-level abstraction is arduous to work with; On the other hand, the extreme flexibility offered by assembly languages is tough to manage.

Nevertheless, any high-level program or computation must be translated before it is carried out on actual machines, and the translation process in practice is invariably anything but trivial. It is at least suspectable that the guarantee established at the high-level may be invalidated by tricky compilation and optimization bugs. Therefore, it is not only useful, but sometimes necessary to reason about assembly code or machine executable directly so as to achieve higher assurance. Furthermore, certain critical applications, such as core system libraries and embedded software, are sometimes directly written in assembly code for efficiency and expressiveness; their verification should not be overlooked.

Our previous work [46, 47] investigated a logic-based approach for verifying assembly code. We demonstrated that a simple low-level language based on Hoare logic [17, 18], namely a language for certified assembly programming (CAP), can be used to certify more than type safety; and with help of a proof assistant, semi-automatic proof construction is not unduly difficult, allowing the verification of even undecidable program properties. Moreover, the machine-checkable proofs for the safety of the assembly code directly en-

$$
\begin{array}{rrcl}
(\textit{Program}) & \mathbb{P} & ::= & (\mathbb{C},\mathbb{S},\mathbb{I}) \\
(\textit{CodeHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{H},\mathbb{R}) \\
(\textit{Heap}) & \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{RegFile}) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Register}) & \mathtt{r} & ::= & \{\mathtt{r}_k\}^{k\in\{0\ldots 7\}} \\
(\textit{Labels}) & \mathtt{f},\mathtt{l} & ::= & \mathtt{n}\ (\textit{nat nums}) \\
(\textit{WordVal}) & \mathtt{w} & ::= & \mathtt{n}\ (\textit{nat nums}) \\
(\textit{InstrSeq}) & \mathbb{I} & ::= & \mathtt{c};\mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \\
(\textit{Command}) & \mathtt{c} & ::= & \mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \mid \mathsf{movi}\ \mathtt{r}_d,\mathtt{w} \mid \mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f} \\
 & & & \mid \mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(\mathtt{w}) \mid \mathsf{st}\ \mathtt{r}_d(\mathtt{w}),\mathtt{r}_s \\
 & & & \\
(\textit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\
(\textit{Assert}) & \mathtt{a} & \in & \textit{State} \rightarrow \textit{Prop}
\end{array}
$$

Figure 1. Syntax and verification constructs of Simple CAP.

$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}) \longmapsto \mathbb{P}$ where	
if $\mathbb{I} =$	then $\mathbb{P} =$
jd $\mathtt{f}$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}'$
add $\mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t;\mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\}),\mathbb{I}')$
movi $\mathtt{r}_d,\mathtt{w};\mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{w}\}),\mathbb{I}')$
bgt $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)$; and $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$
ld $\mathtt{r}_d,\mathtt{r}_s(\mathtt{w});\mathbb{I}'$	$(\mathbb{C},(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\}),\mathbb{I}')$ where $(\mathbb{R}(\mathtt{r}_s)+\mathtt{w}) \in \mathrm{dom}(\mathbb{H})$
st $\mathtt{r}_d(\mathtt{w}),\mathtt{r}_s;\mathbb{I}'$	$(\mathbb{C},(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\},\mathbb{R}),\mathbb{I}')$ where $(\mathbb{R}(\mathtt{r}_d)+\mathtt{w}) \in \mathrm{dom}(\mathbb{H})$

Figure 2. Operational semantics of Simple CAP.

able the encapsulation of verified CAP programs as Foundational Proof-Carrying Code (FPCC) [4, 15] packages.

In this paper, we extend the previous work on CAP to the domain of concurrent computation, presenting a concurrent language for certified assembly programming (CCAP). The computation layer of CCAP models a simple yet generic assembly language that supports the non-deterministic interleaved execution of two threads. The "type" layer of CCAP, as is the case of CAP, engages the calculus of inductive constructions (CiC) [35] to support essentially higher-order predicate logic reasonings. To effectively model safety properties and the interaction between threads, we adapt and apply established approaches for reasoning about (high-level) concurrent programs, namely the "invariance proof" technique [24] and the "assume-guarantee" paradigm [27, 23].

The thread control in CCAP is non-preemptive—a thread's execution will not be interrupted until it explicitly yields control. This non-preemptive setting is very useful in clearly separating thread interaction from other orthogonal issues of the verification of assembly programs. Furthermore, besides facilitating simpler presentation and allowing easier understanding, the non-preemptive setting is also more fundamental, because even preemptive thread controls are essentially implemented with help of non-preemption, such as the disabling of interrupts.

Non-preemption also introduces extra difficulties which have not been addressed by previous researches that assume preemption. Since the thread control will not be interrupted arbitrarily, programs can be written in a more flexible manner (for most safety properties, a program that works correctly in a preemptive setting will also work correctly in a non-preemptive setting, but not vice versa). In this paper, we generalize the assume-guarantee paradigm by introducing a notion of "local guarantee" to accommodate such extra flexibility.

It is worth noting that CCAP is strictly more expressive than CAP—it is a generalization to handle more problems in practice, rather than a specialization for a particular problem domain. We also wish to point out that generating FPCC packages for CCAP is as easy as that for CAP, although a detailed account of which is omitted due to space constraints.

We have developed the language CCAP and proved its soundness using the Coq proof assistant [42]. The implementation in Coq is available for download [43]. We illustrate the usage of CCAP by discussing three example programs, whose safety properties of concern are mutual exclusion, deadlock freedom, and partial correctness respectively. For ease of understanding, we present the central idea by modeling a simplified abstract machine, and give a generalized account in the appendix.

2 Background

In this section, we review some key techniques used in the design of CCAP. Section 2.1 presents a simplified CAP in a nutshell. Section 2.2 discusses informally the technique of invariance proof for proving safety properties, along with its connection with the syntactic approach to type soundness [44]. A brief introduction to the assume-guarantee paradigm is given in Section 2.3.

2.1 Simple CAP

As suggested by its name (a language for certified assembly programming), CAP is an assembly language designed for writing "certified programs", *i.e.,* programs together with their formal safety proofs. Here we introduce Simple CAP with a limited instruction set for ease of understanding. In particular, we omit the support for higher-order code pointers (realized by an indirect jump instruction in the original CAP), instead discussing its handling in Section 6.3.

This language is best learned in two steps. Step one puts aside the "certifying" part and focus on a generic assembly language (see the upper part of Figure 1). A complete program consists of a code heap, a dynamic state component made up of the register file and data heap, and an instruction sequence. The instruction set is minimal but extensions are straightforward. The register file is made up of 8 registers and the date heap is potentially infinite. The operational semantics of this language (Figure 2) should pose no surprise. Note that it is illegal to access a heap location (label) which does not exist, in which case the execution gets "stuck."

In step two, we equip this language with a construct Ψ (*Code Heap Specification*) for expressing user-defined safety requirements in Hoare-logic style (see the lower part of Figure 1). A code heap specification associates every code label with an assertion, with the intention that the precondition of a code block is described by the corresponding assertion. CAP programs are written in continuation-passing style because there are no instructions directly in correspondence with the calling and returning in a high-level language. Hence postconditions in Hoare logic do not have an explicit counterpart in CAP; they are interpreted as preconditions of the continuations.

To check the validity of these assertions mechanically, we im-

$$\frac{\Psi \vdash \mathbb{C} \qquad \Psi \vdash \{\mathtt{a}\}\,\mathbb{I} \qquad (\mathtt{a}\ \mathbb{S})}{\Psi \vdash \{\mathtt{a}\}\,(\mathbb{C},\mathbb{S},\mathbb{I})} \quad (\text{PROG})$$

$$\frac{\begin{array}{l}\Psi = \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1 \ldots \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\} \\ \Psi \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \quad \forall i \in \{1 \ldots n\}\end{array}}{\Psi \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1 \ldots \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\}} \quad (\text{CODEHEAP})$$

$$\frac{\forall \mathbb{S}.\, \mathtt{a}\,\mathbb{S} \supset \mathtt{a}_1\,\mathbb{S} \quad \text{where } \Psi(\mathtt{f}) = \mathtt{a}_1}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jd}\ \mathtt{f}} \quad (\text{JD})$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\,\forall \mathbb{R}.\, \mathtt{a}\,(\mathbb{H},\mathbb{R}) \supset \mathtt{a}'\,(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)+\mathbb{R}(\mathtt{r}_t)\}) \\ \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t;\mathbb{I}} \quad (\text{ADD})$$

Figure 3. Sample inference rules of Simple CAP.

plement all CAP language constructs and their semantics using the calculus of inductive constructions (CiC) [42, 35], which is a calculus that corresponds to a variant of higher-order predicate logic via the formulae-as-types principle (Curry-Howard isomorphism [22]). We further define our assertion language (*Assert*) to contain any CiC terms which have type *State*→*Prop*, where *Prop* is a CiC sort corresponding to logical propositions, and the various syntactic categories of the assembly language (such as *State*) have been encoded using inductive definitions. Although CiC is unconventional to most programmers, we implement the CAP language using the Coq proof assistant [42] which implicitly handles the formulae-as-types principle and hence programmers can reason as if they were directly using higher-order predicate logic. (Logic is something a programmer has to learn if formal reasoning is desired. Higher-order logic appears to be fairly close to what people usually use to reason.) For example, a precondition specifying that the registers $\mathtt{r}_1$, $\mathtt{r}_2$ and $\mathtt{r}_3$ store the same value is written in Coq as the assertion (state predicate) `[s:State] let (H,R)=s in R(r1)=R(r2) /\ R(r2)=R(r3)` where `[s:State]` is a binding of variable `s` of type `State`, `let` is a pattern matching construct, and `/\` is the logical connective $\wedge$. Proving the validity of propositions is done by using tactics which correspond to logical inference rules such as "$\wedge$-introduction."

To reason about the validity of an assertion as a precondition of a code block, we define a set of inference rules for proving specification judgments of the following forms:

$\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$	(*well-formed program*)
$\Psi \vdash \mathbb{C}$	(*well-formed code heap*)
$\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$	(*well-formed instruction sequence*)

The intuition of well-formed instruction sequence, for example, is that if the instruction sequence $\mathbb{I}$ starts execution in a machine state which satisfies the assertion $\mathtt{a}$, then executing $\mathbb{I}$ is safe with respect to the specification Ψ.

In Figure 3, we give some sample inference rules which the programmers use to prove the safety of their programs. A program is well-formed (rule PROG) under assertion $\mathtt{a}$ if both the code heap and the instruction sequence are well-formed, and the machine state satisfies the assertion $\mathtt{a}$. A code heap is well-formed (rule CODEHEAP) if every code block is well-formed under the associated assertion. A direct jump is safe (rule JD) if the current assertion is no weaker than the assertion of the target code block as specified by the code heap specification. Here the symbol "$\supset$" is used to denote logical implication. Also note the slight notational abuse for convenience—state meta-variables are reused as state variables.

An instruction sequence preceded by an addition is safe (rule ADD) if we can find an assertion $\mathtt{a}'$ which serves both as the postcondition of the addition ($\mathtt{a}'$ holds on the updated machine state after executing the addition, as captured by the implication) and as the precondition of the tail instruction sequence. A programmer's task, when proving the well-formedness of a program, involves mostly applying the appropriate inference rules, finding intermediate assertions like $\mathtt{a}'$, and proving the logical implications.

The soundness of these inference rules is established with respect to the operational semantics following the syntactic approach of proving type soundness [44]. The soundness theorem below guarantees that given a well-formed program, the current instruction sequence will be able to execute without getting "stuck" (normal type-safety); in addition, whenever we jump to a code block, the specified assertion of that code block (which is an arbitrary state predicate given by the user) will hold.

Theorem 1 (CAP Soundness) If $\Psi \vdash \{\mathtt{a}\}\,(\mathbb{C},\mathbb{S},\mathbb{I})$, then for all natural number n, there exists a program $\mathbb{P}$ such that $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^n \mathbb{P}$, and

- if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},\mathbb{S}',\mathsf{jd}\ \mathtt{f})$, then $\Psi(\mathtt{f})\ \mathbb{S}'$;
- if $(\mathbb{C},\mathbb{S},\mathbb{I}) \longmapsto^* (\mathbb{C},(\mathbb{H},\mathbb{R}),(\mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f}))$ and $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$, then $\Psi(\mathtt{f})\ (\mathbb{H},\mathbb{R})$.

Interested readers are referred to the previous work [46, 47] for a complete modeling of CAP and a certified *malloc/free* library whose correctness criterion involves more than type-safety.

2.2 Invariance Proof

The code heap type of CAP allows the specification of certain program properties in addition to type safety. However, it appears insufficient in modeling general safety properties, which typically disallow undesired things from happening at any program points. Directly accommodating such a property in CAP would require inspection of preconditions at all program points, resulting in an unwieldy specification.

Fortunately, many methods have been proposed for proving safety (invariance) properties for both sequential and concurrent programs [12, 17, 33, 25, 24]. As observed by Lamport [24], "all of these methods are essentially the same—when applied to the same program, they involve the same proof, though perhaps in different orders and with different notation." In particular, the execution of a program can be viewed as transitions of states, and a general invariance proof of a program satisfying a safety property P reduces to finding an invariant I satisfying three conditions:

1. The initial state of the program satisfies I;
2. I implies P;
3. If a state satisfies I, then the next state satisfies I.

Interestingly, this "invariance proof" technique also characterizes certain principles found in the syntactic approach to type soundness [44], as engaged by many modern type systems. In fact, the re-

```
(1)   f1 : movi r1, 1
(2)        ...
(3)        yield
(4)        st r2(0), r1
(5)        jd f2

(6)   f2 : movi r1, 2
(7)        st r2(1), r1
(8)        yield
(9)        ...
(10)       jd f3
```

Figure 4. Example: Non-preemptive thread control.

$$
\begin{array}{rrll}
(\textit{Prog}) & \mathbb{P} & ::= & (\mathbb{S}, \mathbb{C}_1, \mathbb{C}_2, \mathbb{I}_1, \mathbb{I}_2, i) \text{ where } i \in \{1,2\} \\
(\textit{State}) & \mathbb{S} & ::= & (\mathbb{M}, \mathbb{R}) \\
(\textit{Mem}) & \mathbb{M} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{RegFile}) & \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Register}) & \mathtt{r} & ::= & \{\mathtt{r}_k\}^{k \in \{0 \ldots 7\}} \\
(\textit{Labels}) & \mathtt{f}, \mathtt{l} & ::= & \mathtt{n}\ (\textit{nat nums}) \\
(\textit{WordVal}) & \mathtt{w} & ::= & \mathtt{n}\ (\textit{nat nums}) \\
(\textit{CdHeap}) & \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{InstrSeq}) & \mathbb{I} & ::= & \mathtt{c}; \mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \\
(\textit{Comm}) & \mathtt{c} & ::= & \mathsf{yield} \mid \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \mid \mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \\
 & & & \mid \mathsf{movi}\ \mathtt{r}_d, \mathtt{w} \mid \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \mid \mathsf{be}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \\
 & & & \mid \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w}) \mid \mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s
\end{array}
$$

Figure 5. Syntax of CCAP.

semblance is clear once we describe the type-safety proofs of these systems as follows:

1. The program is initially well-typed;
2. If a program is well-typed, then it makes a step (usually referred to as a lemma of "progress");
3. If a program is well-typed, then it is also well-typed after making a step (usually referred to as a lemma of "type-preservation").

It is not difficult to observe that the syntactic approach to type soundness is in essence a special application of the invariance proof where the invariant I is well-typedness of the program and the safety property P is type-safety (*i.e.,* non-stuckness). Since the progress and preservation lemmas of such type systems are proved once and for all, all that remains when writing a program is to verify the (initial) well-typedness by typing rules.

The story of CAP is similar—the invariant I is the well-formedness of the program and the safety property P is type-safety and satisfaction of the code heap specification. The problem we encountered applying CAP directly to general safety properties now becomes apparent: type-safety and satisfaction of the code heap specification is not sufficient in modeling arbitrary safety properties. On the bright side, a solution becomes also apparent: it suffices to enhance the judgment of well-formed programs so that it implies the validity of a user specified predicate *Inv* (referred to as a "global invariant" in the remainder of this paper), which in turn implies a safety property of concern. Such a CAP-like language maintains the flavor of a type system, but the "typing rules" are practically "proof rules" of a logic system.

2.3 Assume-Guarantee

A last piece of the puzzle is a compositional method for reasoning about the interaction between threads. Since Francez and Pnueli [13] developed the first method for reasoning compositionally about concurrency, various methods have been proposed, the most representative ones of which are *assumption-commitment* invented by Misra and Chandy [27] for message-passing systems and *rely-guarantee* by Jones [23] for shared-memory programs. These two methods have thereupon been carefully studied [32, 36, 40, 41, 1, 45, 2, 5, 16, 11] and often referred to as *assume-guarantee*.

Under this assume-guarantee paradigm, every thread (or process) is often associated with a pair (A, G) consisting of a guarantee G that the thread will satisfy provided the environment satisfies the assumption A. Under the shared-memory model, the assumption A of a thread describes what atomic transitions may be performed by other threads, and the guarantee G of a thread must hold on every atomic transition performed by the thread itself. They are typically modeled as predicates on a pair of states, which are often called *actions*. Reasoning about a concurrent program then reduces to reasoning about each thread separately, provided that the guarantee of each thread be no weaker than the assumption of every other thread.

We apply this approach at an assembly level, using assumptions and guarantees to characterize the interaction between threads. In Section 3, our abstract machine adopts a non-preemptive thread model, in which threads yield control voluntarily with a command yield. An atomic transition in a preemptive setting then corresponds to a sequence of commands between two yield in our setting. Figure 4, for example, shows two code blocks that belong to the same thread; the state transition between the two yield at lines (3) and (8) must satisfy the guarantee of the current thread, and the potential state transition at either yield, caused by other threads, is expected to satisfy the assumption.

A difficulty in modeling concurrency in such a setting is that one has to "look inside" an atomic operation. It is different from modeling an atomic operation as a whole, as is the case of some previous work on concurrency verification, where the state transition caused is immediately visible when analyzing the atomic operation. When an atomic operation is made up of a sequence of commands, its effect can not be completely captured until reaching the end. For example, in Figure 4, the state change caused by the st at line (4) need not immediately satisfy the guarantee; instead, it may rely on its following commands (*i.e.,* lines (6) and (7)) to complete an adequate state transition. Hence when verifying the safety of a command using CAP-style inference rules, it is insufficient to simply check the guarantee against the state change cause by that command. In our modeling, we introduce a "local guarantee" g 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.

Such a non-preemptive setting helps to separate thread interaction from other orthogonal issues of assembly verification. For instance, from the example of Figure 4, it is apparent that the thread interaction using yield is orthogonal from the handling of control flow transfer (*e.g.,* direct jump jd or indirect jump jmp). This also provides insights on the verification of preemptive threads, because a preemptive model can be considered as a special case of the non-preemptive model in which explicit yielding is used at all program points.

$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}_1,\mathbb{I}_2,1) \longmapsto \mathbb{P}$ where	
if $\mathbb{I}_1 =$	then $\mathbb{P} =$
jd $\mathtt{f}$	$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}',\mathbb{I}_2,1)$ where $\mathbb{C}_1(\mathtt{f}) = \mathbb{I}'$
yield; $\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}',\mathbb{I}_2,i)$ where $i \in \{1,2\}$
bgt $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}',\mathbb{I}_2,1)$ when $\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)$; and $((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}'',\mathbb{I}_2,1)$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}_1(\mathtt{f}) = \mathbb{I}''$
be $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}',\mathbb{I}_2,1)$ when $\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)$; and $((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}'',\mathbb{I}_2,1)$ when $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}_1(\mathtt{f}) = \mathbb{I}''$
$\mathtt{c};\mathbb{I}'$ for remaining cases of $\mathtt{c}$	$(\mathtt{Next}(\mathtt{c},(\mathbb{M},\mathbb{R})),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}',\mathbb{I}_2,1)$
$((\mathbb{M},\mathbb{R}),\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}_1,\mathbb{I}_2,2) \longmapsto \mathbb{P}$ defined similarly	

Figure 6. Operational semantics of CCAP.

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

Figure 7. Auxiliary state update macro.

$$
\begin{array}{rrcl}
(\mathit{ProgSpec}) & \Phi & ::= & (\mathit{Inv},\Psi_1,\Psi_2,\mathbb{A}_1,\mathbb{A}_2,\mathbb{G}_1,\mathbb{G}_2) \\
(\mathit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* \\
(\mathit{ThrdSpec}) & \Theta & ::= & (\mathit{Inv},\Psi,\mathbb{A},\mathbb{G}) \\
 & & & \\
(\mathit{Invariant}) & \mathit{Inv} & \in & \mathit{State} \to \mathit{Prop} \\
(\mathit{Assertion}) & \mathtt{p} & \in & \mathit{State} \to \mathit{Prop} \\
(\mathit{Assumption}) & \mathbb{A} & \in & \mathit{State} \to \mathit{State} \to \mathit{Prop} \\
(\mathit{Guarantee}) & \mathbb{G},\mathtt{g} & \in & \mathit{State} \to \mathit{State} \to \mathit{Prop}
\end{array}
$$

Figure 8. Verification constructs of CCAP.

3 CCAP

CAP supports mechanical verification on sequential assembly code; invariance proof can be applied to reason about general safety properties; and assume-guarantee allows the decomposition of a concurrent program into smaller pieces which in turn can be reasoned about sequentially. The remaining task is to accommodate these techniques in a single language CCAP. In particular, assume-guarantee must be adapted to work for non-preemptive assembly code, as briefly explained in the previous section.

In this section, we present CCAP based on an abstract machine that supports the concurrent execution of two threads using a shared memory. A generalized version supporting more threads is given in Appendix A.

3.1 Abstract Machine

The abstract machine is a straightforward extension from CAP (see Figure 5 for the syntax). A program $\mathbb{P}$ consists of a shared state component $\mathbb{S}$ made up of a memory $\mathbb{M}$ and a register file $\mathbb{R}$, two code heaps $\mathbb{C}$ (one for each thread), two current instruction sequences $\mathbb{I}$ (one for each thread), and an indicator i indicating the current thread.

Threads may interact using the shared memory, and yield control (yield) voluntarily. Only a small set of commands are modeled for simplicity, but extensions are straightforward. Note that there is no need for instructions such as "test-and-set" because of non-preemption.

The operational semantics is defined in Figures 6 and 7. Figure 7 defines a "next state" macro relation detailing the effect of some commands on the machine state. For memory access or update commands (ld, st), the macro is defined only if the side conditions are met. In Figure 6, we only show the cases where thread 1 is the current thread; the other cases are similar. At a yield command, the machine picks a thread non-deterministically without affecting the state. Control flow commands (jd, bgt and be) do not affect the state either. Implicit from these two figures is that the machine gets stuck when executing a memory access or update but the side condition as specified by the macro is not met.

3.2 Inference Rules

Verification constructs of CCAP are introduced in Figure 8. A program specification Φ consists of a global invariant (*Inv*), assumptions ($\mathbb{A}$) and guarantees ($\mathbb{G}$), and code heap specifications (Ψ). A programmer must find a global invariant *Inv* for the program; *Inv* should be no weaker than the safety property of interest and hold throughout the execution of the program. A programmer must also find for each thread a guarantee $\mathbb{G}$ and an assumption $\mathbb{A}$ describing allowed atomic state transitions of the thread and its environment respectively.

Every thread also has its own code heap specification Ψ. As is the case of CAP, Ψ is a mapping from code labels to preconditions. What's different now is that a precondition in CCAP contains not only an assertion $\mathtt{p}$ describing valid states, but also a guarantee $\mathtt{g}$ describing valid state transitions—it is safe for the current thread to yield control only after making a state transition allowed by $\mathtt{g}$. We call $\mathtt{g}$ a "local guarantee" to distinguish it from $\mathbb{G}$. To prove the safety of a program, the programmer needs to find appropriate preconditions for all program points, and apply the inference rules to be introduced later.

CCAP's reasoning is thread-modular: the verification of a concurrent program can be decomposed into the verification of its component threads, and each thread is reasoned about separately with respect to its own thread specification. We use Θ to denote such a thread specification, which consists of the global invariant *Inv*, and the code heap specification Ψ, assumption $\mathbb{A}$ and guarantee $\mathbb{G}$ of the thread. The program specification can then be considered as a union of the thread specification of every component thread.

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

$$\Phi;(\mathtt{p}_1,\mathtt{p}_2,\mathtt{g})\vdash\mathbb{P} \qquad (\textit{well-formed program})$$
$$\Theta\vdash\mathbb{C} \qquad (\textit{well-formed code heap})$$
$$\Theta;(\mathtt{p},\mathtt{g})\vdash\mathbb{I} \qquad (\textit{well-formed instr. seq.})$$

These judgment forms bear much resemblance to those of CAP, noting that preconditions have evolved to include local guarantees. When verifying for a well-formed program, an extra assertion is used to describe a state expected by the idling thread; however, no extra local guarantee is needed.

Well-formed program There are two rules for checking the well-formedness of a program. Rule PROG1 is used when thread 1 is the current thread, and Rule PROG2 is used when thread 2 is the current thread.

$$\frac{\begin{array}{l}\Phi=(\mathit{Inv},\Psi_1,\Psi_2,\mathbb{A}_1,\mathbb{A}_2,\mathbb{G}_1,\mathbb{G}_2)\\ (\mathit{Inv}\wedge\mathtt{p}_1\ \mathbb{S}) \qquad \forall\mathbb{S}''.\,(\mathtt{g}\ \mathbb{S}\ \mathbb{S}'')\supset(\mathtt{p}_2\ \mathbb{S}'')\\ \forall\mathbb{S}'.\forall\mathbb{S}''.\,(\mathit{Inv}\wedge\mathtt{p}_2\ \mathbb{S}')\supset(\mathbb{A}_2\ \mathbb{S}'\ \mathbb{S}'')\supset(\mathtt{p}_2\ \mathbb{S}'')\\ (\mathit{Inv},\Psi_1,\mathbb{A}_1,\mathbb{G}_1)\vdash\mathbb{C}_1 \qquad (\mathit{Inv},\Psi_1,\mathbb{A}_1,\mathbb{G}_1);(\mathtt{p}_1,\mathtt{g})\vdash\mathbb{I}_1\\ (\mathit{Inv},\Psi_2,\mathbb{A}_2,\mathbb{G}_2)\vdash\mathbb{C}_2 \qquad (\mathit{Inv},\Psi_2,\mathbb{A}_2,\mathbb{G}_2);(\mathtt{p}_2,\mathbb{G}_2)\vdash\mathbb{I}_2\end{array}}{\Phi;(\mathtt{p}_1,\mathtt{p}_2,\mathtt{g})\vdash(\mathbb{S},\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}_1,\mathbb{I}_2,1)} \qquad (\text{PROG1})$$

$$\frac{\begin{array}{l}\Phi=(\mathit{Inv},\Psi_1,\Psi_2,\mathbb{A}_1,\mathbb{A}_2,\mathbb{G}_1,\mathbb{G}_2)\\ (\mathit{Inv}\wedge\mathtt{p}_2\ \mathbb{S}) \qquad \forall\mathbb{S}''.\,(\mathtt{g}\ \mathbb{S}\ \mathbb{S}'')\supset(\mathtt{p}_1\ \mathbb{S}'')\\ \forall\mathbb{S}'.\forall\mathbb{S}''.\,(\mathit{Inv}\wedge\mathtt{p}_1\ \mathbb{S}')\supset(\mathbb{A}_1\ \mathbb{S}'\ \mathbb{S}'')\supset(\mathtt{p}_1\ \mathbb{S}'')\\ (\mathit{Inv},\Psi_2,\mathbb{A}_2,\mathbb{G}_2)\vdash\mathbb{C}_2 \qquad (\mathit{Inv},\Psi_2,\mathbb{A}_2,\mathbb{G}_2);(\mathtt{p}_2,\mathtt{g})\vdash\mathbb{I}_2\\ (\mathit{Inv},\Psi_1,\mathbb{A}_1,\mathbb{G}_1)\vdash\mathbb{C}_1 \qquad (\mathit{Inv},\Psi_1,\mathbb{A}_1,\mathbb{G}_1);(\mathtt{p}_1,\mathbb{G}_1)\vdash\mathbb{I}_1\end{array}}{\Phi;(\mathtt{p}_1,\mathtt{p}_2,\mathtt{g})\vdash(\mathbb{S},\mathbb{C}_1,\mathbb{C}_2,\mathbb{I}_1,\mathbb{I}_2,2)} \qquad (\text{PROG2})$$

Since these two rules are symmetrical, we only explain Rule PROG1, where thread 1 is the current thread. In this rule, the well-formedness of a program is determined with respect to not only the program specification Φ, but also an assertion $\mathtt{p}_1$ describing the current state, an assertion $\mathtt{p}_2$ describing the state in which thread 2 may take control, and a relation $\mathtt{g}$ describing the discrepancy between these two states.

The premises of Rule PROG1 may appear complex, but are rather intuitive. Line 1 gives us the components of Φ. Lines 2 requires that both the global invariant *Inv* and the current assertion $\mathtt{p}_1$ must hold on the current state; it also restricts the assertion $\mathtt{p}_2$ of thread 2 and the current guarantee $\mathtt{g}$ of thread 1: if some operations satisfying $\mathtt{g}$ is performed on the current state, then the result state satisfies $\mathtt{p}_2$. This indicates that it is sufficient to refer to $\mathtt{g}$ for the expectation of thread 2 when checking thread 1.

When thread 1 is executing, thread 2 should be in a "safe state" whose properties described by $\mathtt{p}_2$ will not be disturbed by allowed (assumed) transitions of thread 1. Hence line 3 restricts the assertion $\mathtt{p}_2$ to preserve itself under state transitions that satisfy the assumption $\mathbb{A}_2$. Note that "$\supset$" is right associative, *i.e.,* $a\supset b\supset c$ is interpreted as $a\supset(b\supset c)$.

The last two lines decompose the verification task into two parts, each performed independently for a single thread based on the corresponding thread specification. For thread 1 (line 4), the code heap must be well-formed, and the current instruction sequence must be well-formed under the current precondition. The requirements for thread 2 is similar (line 5), except that the guarantee $\mathbb{G}_2$ is used in the precondition of thread 2, indicating necessary state transitions before thread 2 may yield.

Another requirement is that the assumptions and the guarantees be compatible, *i.e.,* $\mathbb{G}_1\supset\mathbb{A}_2$ and $\mathbb{G}_2\supset\mathbb{A}_1$. Noting that these relations are constants and concern only the specification, we do not include this requirement in the inference rules, but will use it as a premise of a soundness lemma.

Well-formed code heap

$$\frac{\begin{array}{l}\Theta=(\mathit{Inv},\Psi,\mathbb{A},\mathbb{G})\\ \Psi=\{\mathtt{f}_j\leadsto(\mathtt{p}_j,\mathtt{g}_j)\}^{j\in\{1\ldots n\}}\\ \Theta;(\mathtt{p}_j,\mathtt{g}_j)\vdash\mathbb{I}_j \quad \forall j\in\{1\ldots n\}\end{array}}{\Theta\vdash\{\mathtt{f}_j\leadsto\mathbb{I}_j\}^{j\in\{1\ldots n\}}} \qquad (\text{CODEHEAP})$$

A code heap is well-formed if every code block is well-formed under the associated precondition. Note that after the decomposition performed by Rules PROG1 and PROG2, every thread is verified separately under its own thread specification.

Well-formed instruction sequence

$$\frac{\begin{array}{l}\Theta=(\mathit{Inv},\Psi,\mathbb{A},\mathbb{G})\\ \forall\mathbb{S}.\,(\mathit{Inv}\wedge\mathtt{p}\ \mathbb{S})\supset(\mathtt{g}\ \mathbb{S}\ \mathbb{S})\\ \forall\mathbb{S}.\forall\mathbb{S}'.\,(\mathit{Inv}\wedge\mathtt{p}\ \mathbb{S})\supset(\mathbb{A}\ \mathbb{S}\ \mathbb{S}')\supset(\mathtt{p}\ \mathbb{S}')\\ \Theta;(\mathtt{p},\mathbb{G})\vdash\mathbb{I}\end{array}}{\Theta;(\mathtt{p},\mathtt{g})\vdash\mathsf{yield};\mathbb{I}} \qquad (\text{YIELD})$$

Rule YIELD is used to verify an instruction sequence starting with yield. It is safe to yield control under a state $\mathbb{S}$ only if required state transitions are complete, *i.e.,* an identity transition from $\mathbb{S}$ satisfies the local guarantee $\mathtt{g}$. Furthermore, the assertion $\mathtt{p}$ must preserve itself under transitions allowed by the assumption $\mathbb{A}$, indicating that it is safe to execute any number of atomic operations by the other thread. Lastly, one must verify the remainder instruction sequence $\mathbb{I}$ with the local guarantee reset to $\mathbb{G}$.

$$\frac{\begin{array}{l}\Theta=(\mathit{Inv},\Psi,\mathbb{A},\mathbb{G})\\ \forall\mathbb{S}.\,(\mathit{Inv}\wedge\mathtt{p}\ \mathbb{S})\supset(\mathit{Inv}\wedge\mathtt{p}'\ \mathtt{Next}(\mathsf{c},\mathbb{S}))\\ \forall\mathbb{S}.\forall\mathbb{S}'.\,(\mathit{Inv}\wedge\mathtt{p}\ \mathbb{S})\supset(\mathtt{g}'\ \mathtt{Next}(\mathsf{c},\mathbb{S})\ \mathbb{S}')\supset(\mathtt{g}\ \mathbb{S}\ \mathbb{S}')\\ \Theta;(\mathtt{p}',\mathtt{g}')\vdash\mathbb{I} \qquad \mathsf{c}\in\{\mathsf{add},\mathsf{sub},\mathsf{movi}\}\end{array}}{\Theta;(\mathtt{p},\mathtt{g})\vdash\mathsf{c};\mathbb{I}} \qquad (\text{SIMPL})$$

Rule SIMPL covers the verification of instruction sequences starting with a simple command such as add, sub or movi. In these cases, one must find an intermediate precondition $(\mathtt{p}',\mathtt{g}')$ under which the tail instruction sequence $\mathbb{I}$ is well-formed. Moreover, 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.

Consider a sequence of states $\mathbb{S}_1\ldots\mathbb{S}_n$ between two "safe points" (*i.e.,* between two yield commands). Suppose we have a sequence of guarantees $\mathtt{g}_1\ldots\mathtt{g}_n$ such that:

$$\begin{array}{l}(\mathtt{g}_2\ \mathbb{S}_2\ \mathbb{S}_n)\supset(\mathtt{g}_1\ \mathbb{S}_1\ \mathbb{S}_n);\\ (\mathtt{g}_3\ \mathbb{S}_3\ \mathbb{S}_n)\supset(\mathtt{g}_2\ \mathbb{S}_2\ \mathbb{S}_n);\\ \ldots;\\ (\mathtt{g}_{n-1}\ \mathbb{S}_{n-1}\ \mathbb{S}_n)\supset(\mathtt{g}_{n-2}\ \mathbb{S}_{n-2}\ \mathbb{S}_n);\\ (\mathtt{g}_n\ \mathbb{S}_n\ \mathbb{S}_n)\supset(\mathtt{g}_{n-1}\ \mathbb{S}_{n-1}\ \mathbb{S}_n).\end{array} \qquad (\text{LOCAL-GUAR})$$

From these we know that the state transition from $\mathbb{S}_1$ to $\mathbb{S}_n$ satisfies the original guarantee $\mathtt{g}_1$ if only $(\mathtt{g}_n\ \mathbb{S}_n\ \mathbb{S}_n)$, which will be established at the yield command.

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

$$\frac{\begin{array}{l}\Theta = (Inv, \Psi, \mathbb{A}, \mathbb{G}) \\ \forall \mathbb{M}. \forall \mathbb{R}. (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \supset ((\mathbb{R}(\mathtt{r}_d) + \mathtt{w}) \in \mathrm{dom}(\mathbb{M})) \\ \forall \mathbb{S}. (Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (Inv \wedge \mathtt{p}'\ \mathtt{Next}(\mathtt{c}, \mathbb{S})) \\ \forall \mathbb{S}. \forall \mathbb{S}'. (Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{g}'\ \mathtt{Next}(\mathtt{c}, \mathbb{S})\ \mathbb{S}') \supset (\mathtt{g}\ \mathbb{S}\ \mathbb{S}') \\ \Theta; (\mathtt{p}', \mathtt{g}') \vdash \mathbb{I} \qquad \mathtt{c} = \mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s\end{array}}{\Theta; (\mathtt{p}, \mathtt{g}) \vdash \mathtt{c}; \mathbb{I}} \quad (\mathrm{ST})$$

Verifying memory access or update with Rules LD or ST is in spirit similar to verifying simple commands, except one must also establish the side condition as specified by the macro `Next`.

$$\frac{\begin{array}{l}\Theta = (Inv, \Psi, \mathbb{A}, \mathbb{G}) \qquad \Psi(\mathtt{f}) = (\mathtt{p}', \mathtt{g}') \\ \forall \mathbb{S}. (Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{p}'\ \mathbb{S}) \\ \forall \mathbb{S}. \forall \mathbb{S}'. (Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{g}'\ \mathbb{S}\ \mathbb{S}') \supset (\mathtt{g}\ \mathbb{S}\ \mathbb{S}')\end{array}}{\Theta; (\mathtt{p}, \mathtt{g}) \vdash \mathsf{jd}\ \mathtt{f}} \quad (\mathrm{JD})$$

Direct jump is easily verified by fetching the precondition from the code heap specification, as shown in Rule JD. There is no need to re-establish the global invariant *Inv* explicitly, because the machine state remains the same.

$$\frac{\begin{array}{l}\Theta = (Inv, \Psi, \mathbb{A}, \mathbb{G}) \qquad \Psi(\mathtt{f}) = (\mathtt{p}', \mathtt{g}') \\ \forall \mathbb{M}. \forall \mathbb{R}. (\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \supset (\mathtt{p}'\ (\mathbb{M}, \mathbb{R})) \\ \forall \mathbb{M}. \forall \mathbb{R}. \forall \mathbb{S}'. (\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \\ \qquad \supset (\mathtt{g}'\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \\ \forall \mathbb{M}. \forall \mathbb{R}. (\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \supset (\mathtt{p}''\ (\mathbb{M}, \mathbb{R})) \\ \forall \mathbb{M}. \forall \mathbb{R}. \forall \mathbb{S}'. (\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \\ \qquad \supset (\mathtt{g}''\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \\ \Theta; (\mathtt{p}'', \mathtt{g}'') \vdash \mathbb{I}\end{array}}{\Theta; (\mathtt{p}, \mathtt{g}) \vdash \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \quad (\mathrm{BGT})$$

$$\frac{\begin{array}{l}\Theta = (Inv, \Psi, \mathbb{A}, \mathbb{G}) \qquad \Psi(\mathtt{f}) = (\mathtt{p}', \mathtt{g}') \\ \forall \mathbb{M}. \forall \mathbb{R}. (\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \supset (\mathtt{p}'\ (\mathbb{M}, \mathbb{R})) \\ \forall \mathbb{M}. \forall \mathbb{R}. \forall \mathbb{S}'. (\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \\ \qquad \supset (\mathtt{g}'\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \\ \forall \mathbb{M}. \forall \mathbb{R}. (\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \supset (\mathtt{p}''\ (\mathbb{M}, \mathbb{R})) \\ \forall \mathbb{M}. \forall \mathbb{R}. \forall \mathbb{S}'. (\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M}, \mathbb{R})) \\ \qquad \supset (\mathtt{g}''\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M}, \mathbb{R})\ \mathbb{S}') \\ \Theta; (\mathtt{p}'', \mathtt{g}'') \vdash \mathbb{I}\end{array}}{\Theta; (\mathtt{p}, \mathtt{g}) \vdash \mathsf{be}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \quad (\mathrm{BE})$$

Lastly, a conditional command is safe if both branches can be verified, hence Rule BGT or BE can be understood as a combination of Rules JD and SIMPL, noting that the comparison result can be used in verifying the branches.

3.3 Soundness

The soundness of these inference rules with respect to the operational semantics of the machine is established following the syntactic approach of proving type soundness. From the "progress" and "preservation" lemmas (proved by induction on $\mathbb{I}$; see the implementation for detailed Coq proofs [43]), 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.

Lemma 1 (Code Lookup) If $(\_, \Psi, \_, \_) \vdash \mathbb{C}$ and $\Psi(\mathtt{f}) = (\mathtt{p}, \mathtt{g})$, then there exists $\mathbb{I}$ such that $\mathbb{C}(\mathtt{f}) = \mathbb{I}$.

Proof sketch: By definition of well-formed code heap. □

Lemma 2 (Progress) $\Phi = (Inv, \Psi_1, \Psi_2, \mathbb{A}_1, \mathbb{A}_2, \mathbb{G}_1, \mathbb{G}_2)$. If $\Phi; (\mathtt{p}_1, \mathtt{p}_2, \mathtt{g}) \vdash (\mathbb{S}, \mathbb{C}_1, \mathbb{C}_2, \mathbb{I}_1, \mathbb{I}_2, i)$ where $i \in \{1, 2\}$, then $(Inv\ \mathbb{S})$ and there exists a program $\mathbb{P}$ such that $(\mathbb{S}, \mathbb{C}_1, \mathbb{C}_2, \mathbb{I}_1, \mathbb{I}_2, i) \longmapsto \mathbb{P}$.

Proof sketch: By induction on the structure of $\mathbb{I}_i$. $(Inv\ \mathbb{S})$ holds by definition of well-formed program. In the cases where $\mathbb{I}_i$ starts with add, sub, movi or yield, the program can always make a step by the definition of the operational semantics. In the cases where $\mathbb{I}_i$ starts with ld or st, the side conditions for making a step, as defined by the operational semantics, are established by the corresponding inference rules. In the cases where $\mathbb{I}_i$ starts with bgt or be, or where $\mathbb{I}_i$ is jd, the operational semantics may fetch a code block from the code heap; such a code block exists by Lemma 1. □

Lemma 3 (Code Well-formedness) If $\Theta = (\_, \Psi, \_, \_)$, $\Theta \vdash \mathbb{C}$, $\Psi(\mathtt{f}) = (\mathtt{p}, \mathtt{g})$ and $\mathbb{C}(\mathtt{f}) = \mathbb{I}$, then $\Theta; (\mathtt{p}, \mathtt{g}) \vdash \mathbb{I}$.

Proof sketch: By definition of well-formed code heap. □

Lemma 4 (Preservation) $\Phi = (Inv, \Psi_1, \Psi_2, \mathbb{A}_1, \mathbb{A}_2, \mathbb{G}_1, \mathbb{G}_2)$.
Suppose $\forall \mathbb{S}'. \forall \mathbb{S}''. (\mathbb{G}_1\ \mathbb{S}'\ \mathbb{S}'') \supset (\mathbb{A}_2\ \mathbb{S}'\ \mathbb{S}'')$ and
$\forall \mathbb{S}'. \forall \mathbb{S}''. (\mathbb{G}_2\ \mathbb{S}'\ \mathbb{S}'') \supset (\mathbb{A}_1\ \mathbb{S}'\ \mathbb{S}'')$.
If $\Phi; (\mathtt{p}_1, \mathtt{p}_2, \mathtt{g}) \vdash (\mathbb{S}, \mathbb{C}_1, \mathbb{C}_2, \mathbb{I}_1, \mathbb{I}_2, i)$ where $i \in \{1, 2\}$,
and $(\mathbb{S}, \mathbb{C}_1, \mathbb{C}_2, \mathbb{I}_1, \mathbb{I}_2, i) \longmapsto \mathbb{P}$,
then there exists $\mathtt{p}_1'$, $\mathtt{p}_2'$ and $\mathtt{g}'$ such that $\Phi; (\mathtt{p}_1', \mathtt{p}_2', \mathtt{g}') \vdash \mathbb{P}$.

Proof sketch: By induction on the structure of $\mathbb{I}_i$. We must establish the premises (lines 2–5) of Rule PROG1 or PROG2.

The first half of line 2 is easily established: for any command that updates the machine state, its corresponding inference rule dictates that the global invariant and the postcondition hold on the updated machine state. The second half of line 2 also follows naturally, because the inference rules yield a sequence of "strengthening" local guarantees, as illustrated by the equations LOCAL-GUAR.

Unless $\mathbb{I}_i$ starts with yield, line 3 is trivially preserved because the precondition of the idling thread remains unchanged; in the case where $\mathbb{I}_i$ starts with yield, the current thread may become the idling thread, and line 3 follows from a premise of Rule YIELD.

The first half of lines 4 and 5 (well-formed code heaps) is not affected by a program step, hence it is trivially preserved. For the second half of lines 4 and 5 (well-formed instruction sequences), in the cases where $\mathbb{I}_i$ starts with add, sub, movi, yield, ld or st, the well-formedness of the instruction sequence in the next state can be directly derived from the corresponding inference rules; in the cases where $\mathbb{I}_i$ starts with bgt or be, or is jd, we apply Lemma 3. □

4 Examples

CCAP is a realization of established verification techniques at the assembly level. The expressiveness of these techniques and their application to high-level programs are well-known. In this section, we give 3 simple examples to demonstrate the mechanized verification of interesting safety properties for assembly code and, in particular, illustrate the usage of local guarantees (g) as part of the preconditions.

4.1 Mutual Exclusion

Dekker's algorithm [10] is the first solution to mutual exclusion problem for the two-process case. Flanagan *et al.* [11] have given a modeling under the assume-guarantee paradigm for a high-level parallel language of atomic operations. In this section, we show that the subtle synchronization used in Dekker's algorithm can be reasoned about as easily in CCAP, where atomic operations are composed of sequences of instructions.

The algorithm is revised using 4 boolean variables [11] (Figure 9). It is easy to observe that variable `cs_i` is `true` if thread i is in its crit-

```
Variables:                          Initially:
  boolean a1, a2, cs1, cs2;           ¬cs1 ∧ ¬cs2

Thread1:                            Thread2 :
  while (true){                       while (true){
    a1 := true;                         a2 := true;
    cs1 := ¬a2;                         cs2 := ¬a1;
    if (cs1) {                          if (cs2) {
      // critical sec.                    // critical sec.
      cs1 := false;                       cs2 := false;
    }                                   }
    a1 := false;                        a2 := false;
  }                                   }
```

Figure 9. Dekker's mutual exclusion algorithm.

$$\begin{aligned}
\mathit{Inv} &\equiv \mathbb{M}(\mathtt{cs}_1),\mathbb{M}(\mathtt{cs}_2),\mathbb{M}(\mathtt{a}_1),\mathbb{M}(\mathtt{a}_2) \in \{0,1\}\\
&\quad \wedge\neg(\mathbb{M}(\mathtt{cs}_1)\wedge\mathbb{M}(\mathtt{cs}_2))\\
&\quad \wedge(\mathbb{M}(\mathtt{cs}_1)\supset\mathbb{M}(\mathtt{a}_1))\wedge(\mathbb{M}(\mathtt{cs}_2)\supset\mathbb{M}(\mathtt{a}_2))\\
\mathbb{A}_1,\mathbb{G}_2 &\equiv (\mathbb{M}(\mathtt{a}_1)=\mathbb{M}'(\mathtt{a}_1))\wedge(\mathbb{M}(\mathtt{cs}_1)=\mathbb{M}'(\mathtt{cs}_1))\\
\mathbb{A}_2,\mathbb{G}_1 &\equiv (\mathbb{M}(\mathtt{a}_2)=\mathbb{M}'(\mathtt{a}_2))\wedge(\mathbb{M}(\mathtt{cs}_2)=\mathbb{M}'(\mathtt{cs}_2))
\end{aligned}$$

Initial $\mathbb{M}\equiv\{\mathtt{cs}_1\rightsquigarrow 0,\mathtt{cs}_2\rightsquigarrow 0,\mathtt{a}_1\rightsquigarrow \_,\mathtt{a}_2\rightsquigarrow \_\}$
Initial thread: i where $i\in\{1,2\}$
Initial instruction sequences: $\mathbb{I}_i \equiv \mathsf{jd}\ loop_i$
Initial precondition triple: $(\mathtt{p}_1,\mathtt{p}_2,\mathtt{g})\equiv(\mathsf{True},\mathsf{True},\mathbb{G}_i)$

```
loop1 : −{(True, G1)}                 csec1 : −{(True, G1)}
        yield                                 yield
        movi r1, a1                           −{(True, G1)}
        movi r2, 1                            movi r1, cs1
        st r1(0), r2                          −{(R(r1) = cs1, G1)}
        −{(H(a1) = 1, G1)}                    movi r2, 0
        yield                                 −{(R(r1) = cs1 ∧
        movi r1, cs1                              R(r2) = 0, G1)}
        movi r2, 1                            st r1(0), r2
        movi r3, a2                           −{(H(cs1) = 0, G1)}
        ld r4, r3(0)                          yield
        sub r5, r2, r4                        −{(H(cs1) = 0, G1)}
        st r1(0), r5                          jd reset1
        −{(True, G1)}
        yield                         reset1 : −{(H(cs1) = 0, G1)}
        movi r1, cs1                          yield
        ld r2, r1(0)                          movi r1, a1
        movi r3, 0                            movi r2, 0
        bgt r2, r3, csec1                     st r1(0), r2
        −{(H(cs1) = 0, G1)}                   −{(True, G1)}
        yield                                 yield
        jd reset1                             jd loop1
```

Figure 10. CCAP implementation for Dekker's algorithm.

ical section; hence the mutual exclusion property of this algorithm can be expressed as $\neg(\mathtt{cs}_1\wedge\mathtt{cs}_2)$.

To decompose the verification problem into two sequential ones, thread i makes the assumption that the other thread not modify variables $\mathtt{a_i}$ and $\mathtt{cs_i}$. Furthermore, the mutual exclusion property is strengthened to the invariant $\neg(\mathtt{cs}_1\wedge\mathtt{cs}_2)\wedge(\mathtt{cs}_1\supset\mathtt{a}_1)\wedge(\mathtt{cs}_2\supset\mathtt{a}_2)$.

Some key components of a corresponding CCAP implementation of the algorithm are shown in Figure 10. Only the code of thread 1 is given, since that of thread 2 is similar. Yielding is inserted at all the intervals of the atomic operations of the original program.

```
Variables:                      Initially:
  nat fork1, fork2;               fork1 = 0 ∧ fork2 = 0

Thread1:                        Thread2 :
  while (true){                   while (true){
    acquire(fork1, 1);              acquire(fork1, 2);
    acquire(fork2, 1);              acquire(fork2, 2);
    // eat                          // eat
    release(fork2);                 release(fork2);
    release(fork1);                 release(fork1);
    // think                        // think
  }                               }
```

Figure 11. Dining philosophers algorithm.

The code heap specification is separated and given as preconditions (surrounded by $-\{\}$) at the beginning of each code block for ease of reading. Extra preconditions are spread throughout the code as an outline of the proof—the inference rules can be easily applied once preconditions are given at all program points.

For expository convenience, we use variable names as short-hands for memory locations. The boolean values `false` and `true` are encoded using 0 and 1 respectively. We also omit the variable bindings from the predicates. To be exact, a "$\lambda\mathbb{S}.$" is omitted from the global invariant and every assertion; a "$\lambda\mathbb{S}.\lambda\mathbb{S}'.$" is omitted from every assumption and guarantee. In these predicates, $\mathbb{M}$ and $\mathbb{R}$ refer to the memory component and the register file component of the bounded variable $\mathbb{S}$ respectively, and $\mathbb{M}'$ and $\mathbb{R}'$ refer to those of the bounded variable $\mathbb{S}'$.

For this example, state predicate True is used as the assertion $\mathtt{p}$ at some of the yield instructions, because *Inv* alone is sufficient in characterizing the states at these points. Moreover, since thread 1 never modifies the value of $\mathtt{a}_2$ or $\mathtt{cs}_2$, the guarantee $\mathbb{G}_1$ can be used as the local guarantee $\mathtt{g}$ throughout the code of thread 1. Take the code block labeled $csec_1$ as an example. There are two move instructions used to put intermediate values into registers; their effect is apparent from the associated assertions. A following store instruction writes into the memory location $\mathtt{cs}_1$, after which the global invariant is still maintained. It is a simple task to check the well-formedness of this code block with respect to the preconditions, because every assembly instruction only incur little change to the machine state. The use of a proof assistant helps automate most simple steps and keep the would-be arduous task manageable.

4.2 Deadlock Freedom

One way to prevent deadlock is to assign a total ordering to all resources and stick to it when acquiring them. We give a simplified example of dining philosophers in Figure 11, where two philosophers (Thread_1 and Thread_2) share two forks (represented by memory locations $\mathtt{fork}_1$ and $\mathtt{fork}_2$). A philosopher picks up (`acquire`) a fork by writing the thread id (1 or 2) into the memory location representing the fork, and puts down (`release`) a fork by writing 0. Both philosophers acquire $\mathtt{fork}_1$ before acquiring $\mathtt{fork}_2$.

The deadlock freedom property for this example can be expressed as $(\mathtt{fork}_2=0)\vee(\mathtt{fork}_2=\mathtt{fork}_1)$, indicating that either a fork is free or a philosopher is holding both forks. In the more general case of three or more philosophers, the deadlock freedom property can be expressed as $(\mathtt{fork_n}=0)\vee(\exists i.\ \mathtt{fork_n}=\mathtt{fork_i})$, where n is the number of philosophers and $\mathtt{fork_n}$ is the "greatest fork." Note that

$Inv \equiv (\mathbb{M}(\texttt{fork}_2) = 0) \vee (\mathbb{M}(\texttt{fork}_2) = \mathbb{M}(\texttt{fork}_1))$
$\mathbb{A}_1, \mathbb{G}_2 \equiv (\mathbb{M}'(\texttt{fork}_2) = 2 \supset \mathbb{M}'(\texttt{fork}_1) = 2)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_1) = 1 \leftrightarrow \mathbb{M}'(\texttt{fork}_1) = 1)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_2) = 1 \leftrightarrow \mathbb{M}'(\texttt{fork}_2) = 1)$
$\mathbb{A}_2, \mathbb{G}_1 \equiv (\mathbb{M}'(\texttt{fork}_2) = 1 \supset \mathbb{M}'(\texttt{fork}_1) = 1)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_1) = 2 \leftrightarrow \mathbb{M}'(\texttt{fork}_1) = 2)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_2) = 2 \leftrightarrow \mathbb{M}'(\texttt{fork}_2) = 2)$

Initial $\mathbb{M} \equiv \{\texttt{fork}_1 \rightsquigarrow 0, \texttt{fork}_2 \rightsquigarrow 0\}$
Initial thread: i where $i \in \{1,2\}$
Initial instruction sequences: $\mathbb{I}_i \equiv$ jd fa_i
Initial precondition triple: $(\mathtt{p}_1, \mathtt{p}_2, \mathtt{g}) \equiv ($
$\quad \mathbb{M}(\texttt{fork}_1) \neq 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1,$
$\quad \mathbb{M}(\texttt{fork}_1) \neq 2 \wedge \mathbb{M}(\texttt{fork}_2) \neq 2, \mathbb{G}_i)$

$fa_1 : -\{(\mathbb{M}(\texttt{fork}_1) \neq 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1, \mathbb{G}_1)\}$
yield
movi $\mathtt{r}_1, \texttt{fork}_1$
ld $\mathtt{r}_2, \mathtt{r}_1(0)$
movi $\mathtt{r}_3, 0$
bgt $\mathtt{r}_2, \mathtt{r}_3, fa_1$
$-\{(\mathbb{M}(\texttt{fork}_1) = 0 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1$
$\quad \wedge \mathbb{R}(\mathtt{r}_1) = \texttt{fork}_1, \mathbb{G}_1)\}$
movi $\mathtt{r}_2, 1$
$-\{(\mathbb{M}(\texttt{fork}_1) = 0 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1$
$\quad \wedge \mathbb{R}(\mathtt{r}_1) = \texttt{fork}_1 \wedge \mathbb{R}(\mathtt{r}_2) = 1,$
$\quad (\mathbb{M}'(\texttt{fork}_2) = 1 \supset \mathbb{M}'(\texttt{fork}_1) = 1)$
$\quad \wedge (\mathbb{M}'(\texttt{fork}_1) \neq 2)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_2) = 2 \leftrightarrow \mathbb{M}'(\texttt{fork}_2) = 2))\}$
st $\mathtt{r}_1(0), \mathtt{r}_2$
$-\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1,$
$\quad (\mathbb{M}'(\texttt{fork}_2) = 1 \supset \mathbb{M}'(\texttt{fork}_1) = 1)$
$\quad \wedge (\mathbb{M}'(\texttt{fork}_1) \neq 2)$
$\quad \wedge (\mathbb{M}(\texttt{fork}_2) = 2 \leftrightarrow \mathbb{M}'(\texttt{fork}_2) = 2))\}$
yield
$-\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1, \mathbb{G}_1)\}$
jd fb_1

$fb_1 : -\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1, \mathbb{G}_1)\}$
yield
movi $\mathtt{r}_1, \texttt{fork}_2$
ld $\mathtt{r}_2, \mathtt{r}_1(0)$
movi $\mathtt{r}_3, 0$
bgt $\mathtt{r}_2, \mathtt{r}_3, fb_1$
movi $\mathtt{r}_2, 1$
st $\mathtt{r}_1(0), \mathtt{r}_2$
$-\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) = 1,$
$\quad (\mathbb{M}'(\texttt{fork}_2) = 1 \supset \mathbb{M}'(\texttt{fork}_1) = 1)$
$\quad \wedge (\mathbb{M}'(\texttt{fork}_1) \neq 2)$
$\quad \wedge (\mathbb{M}'(\texttt{fork}_2) \neq 2))\}$
yield
$-\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) = 1, \mathbb{G}_1)\}$
movi $\mathtt{r}_1, \texttt{fork}_2$
movi $\mathtt{r}_2, 0$
st $\mathtt{r}_1(0), \mathtt{r}_2$
yield
$-\{(\mathbb{M}(\texttt{fork}_1) = 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1, \mathbb{G}_1)\}$
movi $\mathtt{r}_1, \texttt{fork}_1$
movi $\mathtt{r}_2, 0$
st $\mathtt{r}_1(0), \mathtt{r}_2$
yield
$-\{(\mathbb{M}(\texttt{fork}_1) \neq 1 \wedge \mathbb{M}(\texttt{fork}_2) \neq 1, \mathbb{G}_1)\}$
jd fa_1

Figure 12. CCAP implementation for dining philosophers algorithm.

Variables:	Initially:
nat a, b;	$\mathtt{a} = \alpha \wedge \mathtt{b} = \beta$
Thread_1:	Thread_2 :
while $(\mathtt{a} \neq \mathtt{b})$	while $(\mathtt{b} \neq \mathtt{a})$
if $(\mathtt{a} > \mathtt{b})$	if $(\mathtt{b} > \mathtt{a})$
$\mathtt{a} := \mathtt{a} - \mathtt{b}$;	$\mathtt{b} := \mathtt{b} - \mathtt{a}$;

Figure 13. GCD algorithm.

we cannot use CCAP as is to reason about livelock or starvation freedom, which are liveness properties and require different proof methods [34].

The corresponding CCAP specification and program are shown in Figure 12, where the same notational convention is used as established in the previous section, and only the code for Thread_1 is given. In this example, *Inv* is exactly the desired property. Thread 1 guarantees (see $\mathbb{G}_1$) to acquire $\texttt{fork}_2$ only after acquiring $\texttt{fork}_1$ ($\mathbb{M}'(\texttt{fork}_2) = 1 \supset \mathbb{M}'(\texttt{fork}_1) = 1$). Due to the nature of our simulation using the values of $\texttt{fork}_1$ and $\texttt{fork}_2$ to indicate the occupiers of the resources, Thread 1 also guarantees that it does not release a resource held by Thread 2 or acquire a resource for Thread 2 ($\mathbb{M}(fork_i) = 2 \leftrightarrow \mathbb{M}'(fork_i) = 2$, where $\leftrightarrow$ means "if and only if"). The case for Thread 2 is similar.

We explain the code block labeled fa_1, which corresponds to the first acquiring operation of Thread_1. The precondition of this block indicates that Thread_1 owns neither resource at the beginning of the loop. A busy waiting on $\texttt{fork}_1$ is coded using the branch instruction (bgt). During the waiting, the guarantee is trivially maintained because no state change is incurred. The thread executes past the branch instruction only if $\texttt{fork}_1$ is free, *i.e.,* $\mathbb{M}(\texttt{fork}_1) = 0$, as captured by the assertion immediately after the branch. At this point, based on the current value of $\texttt{fork}_1$, which does not equal to 2, we can simplify the local guarantee at the next step following Rule SIMPL. The difference between the new guarantee and the original $\mathbb{G}_1$ lies in the handling of $\texttt{fork}_1$—the new guarantee requires that $\mathbb{M}'(\texttt{fork}_1) \neq 2$, because it is known that the original value of $\texttt{fork}_1$ was not 2. The verification of the remainder of this code block is straightforward. At the yielding instruction before jumping to fb_1, the local guarantee is instantiated twice on the current state; its validity is trivial. After the yielding instruction, the local guarantee is reset to $\mathbb{G}_1$ as required by Rule YIELD.

4.3 Partial Correctness

We consider a program for computing a Greatest Common Divisor (GCD) as shown in Figure 13. This program satisfies the partial correctness property that a and b become equal to the GCD of their initial values when the program exits the loop. α and β are rigid variables ranging over natural numbers; they are introduced for reasoning purposes.

Figure 14 gives the corresponding CCAP specification and program. Only the code of Thread_1 is shown. The global invariant of this program is that the GCD of a and b remains the same. Thread_1, for example, makes the assumption that Thread_2 may change the values of a or b only if $\mathbb{M}(\mathtt{a}) < \mathbb{M}(\mathtt{b})$ and only in such a way that the GCD can be preserved. The partial correctness property is expressed as the precondition of the code block labeled $done_1$—if the execution ever reaches here, the value stored in a and b must be the expected GCD.

Let δ be a shorthand for $\gcd(\alpha,\beta)$.

$$
\begin{array}{ll}
Inv & \equiv \gcd(\mathbb{M}(\mathtt{a}),\mathbb{M}(\mathtt{b})) = \delta \\
\mathbb{A}_1,\mathbb{G}_2 & \equiv (\mathbb{M}(\mathtt{a}) \geq \mathbb{M}(\mathtt{b}) \supset (\mathbb{M}(\mathtt{a}) = \mathbb{M}'(\mathtt{a}) \wedge \mathbb{M}(\mathtt{b}) = \mathbb{M}'(\mathtt{b}))) \\
& \quad \wedge (\mathbb{M}(\mathtt{a}) < \mathbb{M}(\mathtt{b}) \supset (\gcd(\mathbb{M}'(\mathtt{a}),\mathbb{M}'(\mathtt{b})) = \delta)) \\
\mathbb{A}_2,\mathbb{G}_1 & \equiv (\mathbb{M}(\mathtt{b}) \geq \mathbb{M}(\mathtt{a}) \supset (\mathbb{M}(\mathtt{a}) = \mathbb{M}(\mathtt{a}) \wedge \mathbb{M}(\mathtt{b}) = \mathbb{M}'(\mathtt{b}))) \\
& \quad \wedge (\mathbb{M}(\mathtt{b}) < \mathbb{M}(\mathtt{a}) \supset (\gcd(\mathbb{M}'(\mathtt{a}),\mathbb{M}'(\mathtt{b})) = \delta))
\end{array}
$$

Initial $\mathbb{M} \equiv \{\mathtt{a} \rightsquigarrow \alpha, \mathtt{b} \rightsquigarrow \beta\}$
Initial thread: i where $i \in \{1,2\}$
Initial instruction sequences: $\mathbb{I}_i \equiv$ jd $loop_i$
Initial precondition triple: $(\mathtt{p}_1,\mathtt{p}_2,\mathtt{g}) \equiv$
$(\gcd(\mathbb{M}(\mathtt{a}),\mathbb{M}(\mathtt{b})) = \delta, \gcd(\mathbb{M}(\mathtt{a}),\mathbb{M}(\mathtt{b})) = \delta, \mathbb{G}_i)$

$loop_1$: $-\{(\mathsf{True},\mathbb{G}_1)\}$
yield
movi $\mathtt{r}_1,\mathtt{a}$
ld $\mathtt{r}_2,\mathtt{r}_1(0)$
movi $\mathtt{r}_3,\mathtt{b}$
ld $\mathtt{r}_4,\mathtt{r}_3(0)$
be $\mathtt{r}_2,\mathtt{r}_4,done_1$
$-\{(\mathsf{True},\mathbb{G}_1)\}$
yield
movi $\mathtt{r}_1,\mathtt{a}$
ld $\mathtt{r}_2,\mathtt{r}_1(0)$
movi $\mathtt{r}_3,\mathtt{b}$
ld $\mathtt{r}_4,\mathtt{r}_3(0)$
bgt $\mathtt{r}_2,\mathtt{r}_4,calc_1$
$-\{(\mathsf{True},\mathbb{G}_1)\}$
yield
jd $loop_1$

$calc_1$: $-\{(\mathbb{M}(\mathtt{a}) > \mathbb{M}(\mathtt{b}),\mathbb{G}_1)\}$
yield
$-\{(\mathbb{M}(\mathtt{a}) > \mathbb{M}(\mathtt{b}),\mathbb{G}_1)\}$
movi $\mathtt{r}_1,\mathtt{a}$
$-\{(\mathbb{M}(\mathtt{a}) > \mathbb{M}(\mathtt{b}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{a}, \gcd(\mathbb{M}'(\mathtt{a}),\mathbb{M}'(\mathtt{b})) = \delta)\}$
ld $\mathtt{r}_2,\mathtt{r}_1(0)$
movi $\mathtt{r}_3,\mathtt{b}$
ld $\mathtt{r}_4,\mathtt{r}_3(0)$
sub $\mathtt{r}_5,\mathtt{r}_2,\mathtt{r}_4$
st $\mathtt{r}_1(0),\mathtt{r}_5$
$-\{(\mathsf{True}, \gcd(\mathbb{M}'(\mathtt{a}),\mathbb{M}'(\mathtt{b})) = \delta)\}$
yield
jd $loop_1$

$done_1$: $-\{(\mathbb{M}(\mathtt{a}) = \mathbb{M}(\mathtt{b}) = \delta,\mathbb{G}_1)\}$
yield
jd $done_1$

Figure 14. CCAP implementation for GCD algorithm.

The most interesting code block is that labeled $calc_1$. After the first yielding instruction, the comparison result of $\mathbb{M}(\mathtt{a}) > \mathbb{M}(\mathtt{b})$ can be used to simplify the next local guarantee to $\gcd(\mathbb{M}'(\mathtt{a}),\mathbb{M}'(\mathtt{b})) = \delta$. The actual proof of this certified program involves also applying mathematical properties such as $\gcd(a,a) = a$ and $a > b \supset \gcd(a,b) = \gcd(a-b,b)$.

5 Implementation

In Section 2.1, we mentioned the use of Coq and the underlying CiC for the mechanical verification of CAP programs. The same approach is applied to implementing CCAP. We encode the syntax of CCAP using inductive definitions, and define the operational semantics and inference rules as a collection of relations. Embedding the entire CCAP in Coq allows us to make use of Coq's full expressiveness, including its facility for inductive definitions, to code the verification constructs in Figure 8. This also allows us to write down the soundness lemmas as Coq terms and formally prove their validity as theorems using the Coq proof assistant. In particular, our implementation contains about 350 lines of Coq code for the language definition of CCAP, and about 550 lines of Coq tactics for the soundness proof.

We have also verified the examples of Section 4 using this Coq encoding (see the actual code [43] for details). The proof construction of each of these examples, with help of the Coq proof assistant, took no more than a few hours. On average, 11.5 lines of proof in Coq tactics is written for every assembly instruction. As we have expected, many premises of the CCAP inference rules can be automatically verified given the intermediate preconditions. This is especially true in the cases of simple instructions which do not concern the memory. Human "smartness" is required occasionally, however, to find proper intermediate preconditions and apply mathematical properties such as those of the GCD. In principle, the programer should possess informal ideas on why their programs "work" when programming. Our system simply requires the programer to present those in logic, something necessary if formal reasoning is desired.

Keen readers may have observed that these examples do not involve any data structures, hence the simplicity in verifying them is not surprising. Previous work [46, 47] has studied similar verification on list-like data structures, and shown that the verification task is also straightforward after developing proper lemmas detailing the interaction between various instructions (in particular st) and data structures.

6 Related and Future Work

6.1 Program Verification

The CAP languages (*i.e.,* CAP and CCAP) approach the "verifying compiler" grand challenge from the aspect of program verification. They are developed as a complementary approach to Proof-Carrying Code (PCC) [29, 28] where the safety proof of a program is developed semi-automatically by the programmer with help of a proof assistant. The human factor involved achieves safety in a more general sense, allowing the verification of even undecidable program properties. Some useful applications include core system libraries and critical software components. Much further work is required, however, on improving the modularity of the CAP languages before they can be applied to large scale applications. This work includes modular support for higher-order code pointers (see Section 6.3) and safe linking of software components [14]. In the long run, it is conceivable to develop verified software components one by one and eventually have even a complete operation system verified. The modularity support for the CAP languages will be crucial in realizing this ambition.

The CAP languages address program safety directly at the level of machine code. The code being verified is so close to the actual executable that the translation between them is very trustworthy. This contrasts with model checking, which operates on models of programs. The non-trivial abstraction step to construct a model from a program indicates that a verified model may not necessarily imply a verified program. On the other hand, focusing on high-level models enables model checking to be effective in practice.

Type systems are another popular approach of reasoning about programs. Types document programmers' intent and enable compilers to detect certain programming errors automatically. In practice, type systems have only been applied to reasoning about a limited scope of properties, including non-stuckness of programs, information flow, finite-state rules, races and deadlocks, and atomicity of multithreaded programs. In essence, types are simply restricted predicates, and the complexity of a type system is determined by the complexity of the underlying property that it enforces. This makes it hard to apply type systems to low-level code such as those found in storage allocation libraries, whose properties often demand very specialized type systems. The CAP languages, in contrast, directly employ higher-order predicate logic, intending to formally present the reasoning of a programmer. Proof construction is not unduly difficult with help of a proof assistant.

Shao *et al.* [39] developed a type system for certified binaries (TSCB). An entire proof system (CiC) is integrated into a compiler intermediate language so as to perform reasoning about certified programs in a calculus but essentially using higher-order predicate logic via the formulae-as-types principle [22]. Being a type system, the soundness of TSCB only guarantees type-safety (non-stuckness of well-typed programs). In comparison, the CAP languages allow programmers to write arbitrary predicates in the specification to account for more general safety properties. Another convenient trait of the CAP languages is that, using a proof assistant like Coq, the well-formedness reasoning can be performed without the knowledge about the CiC calculus, which may be desirable to some programmers.

6.2 Concurrency Verification

There has been much work on concurrency verification (see [9, 21] for a systematic and comprehensive introduction). This paper adapts and applies established techniques on this issue for assembly code, a domain not covered by existing work. The proof methods pertinent to this paper have been discussed in Sections 2.2 and 2.3. In particular, our modeling has benefited from previous work by Lamport [24] and Flanagan *et al.* [11], as elaborated below.

Lamport [24] proposed the Temporal Logic of Actions (TLA) as a logic for specifying and reasoning about concurrent systems. He pointed out a unified view of existing methods for proving invariance (safety) properties, which can be described formally in TLA as applications of a temporal inference rule named `INV1`. Some simple temporal reasoning following `INV1` shows that a general invariance proof can be reduced to finding an invariant I satisfying three conditions. All these conditions are assertions about predicates and actions, rather than temporal formulas; hence the proof for invariance properties can be completed in ordinary mathematics following the three conditions.

CCAP engaged a "syntactic approach" usually found in type systems. Well-formedness of the program is used as the invariant I, and the soundness lemmas, namely progress and preservation, cover two of the conditions. The last condition, *i.e.,* initial well-formedness of the program, is left for the programmer to establish using "typing" rules (inference rules). It is not difficult to see that the lack of temporal reasoning in CCAP does not limit its expressiveness in proving for safety properties, following the observation from TLA.

Proving liveness properties, on the other hand, requires further work. Existing work handles liveness properties using counting-down arguments or proof lattices [34]. As observed by Lamport [24], although liveness properties are expressed by a variety of temporal formulas, their proofs can always be reduced to the proof of leads-to properties—formulas of the form $P \leadsto Q$. It is an interesting future work to investigate how to apply these existing approaches to the mechanical verification of assembly code.

Flanagan *et al.* [11] investigated the mechanical checking of proof obligations for Java programs. Their automatic checker takes as input a program together with annotations describing appropriate assumptions, invariants and correctness properties. The key techniques used by the checker, including assume-guarantee decomposition, are derived from a parallel language of atomic operations.

6.3 Higher-Order Code Pointers

Focusing on concurrency verification, we have left out from this paper the orthogonal issue of higher-order code pointers. Higher-order code pointers are the reflection of higher-order procedures at the assembly level, and a common example is the return pointers which are frequently used in most programs. Unfortunately, to the authors' knowledge, the modularity support for higher-order code pointers (or procedures) in Hoare-logic systems has been a long-standing open problem.

A preliminary support for higher-order code pointers in CAP can be found in previous work [46, 47]. However, the solution there is not sufficiently modular. Take a library function as an example, both the caller's resource and the callee's resource need to be explicitly written in the code heap specification for a CAP routine. Since a library function is to be called at various places, its code heap specification entry in CAP is essentially a template to be instantiated upon used.

Yu *et al.* [48] encountered the problem of "functional parameters" when checking the correctness of MC68020 object code programs. They handle it by asserting the correctness of the functional parameters using "constraints." The correctness of the program is proved assuming these constraints, and the correctness theorem of the program is used repeatedly by substituting the functional parameters with specific functions as long as these functions meet the imposed constraints. Although plausible, the actual mechanization of this idea, as explained by Yu *et al.* [48], is "extremely difficult." Furthermore, this approach is not truly satisfactory because the specification of a program and its correctness theorem are essentially templates which need to be instantiated upon used. CAP's approach, as describe earlier, is very close to this: the actual code heap specification is used when forming the "constraints." In practice, we find it awkward when certifying programs with extensive use of code pointers, especially when these uses are unconventional compared with return pointers.

Similar problems surfaced in some PCC systems as well. In particular, Configurable PCC (CPCC) systems, as proposed by Necula and Schneck [30], statically check program safety using symbolic predicates which are called "continuations." For checking the safety of an indirect jump instruction which transfers the program control given a code pointer, a trusted "decoder" generates an "indirect continuation" whose safety needs to be verified; this continuation is indirect because the target address cannot be determined by the decoder statically. For verification purpose, an untrusted "VCGen extension" is responsible for proposing some "direct continuations" (direct meaning that the target addresses are known statically) whose safety implies the safety of the "indirect continuation" given by the decoder. In practice, the extension works by listing all the possible values of the code pointer (essentially replacing the code pointer in the continuations with all concrete functions that it could stand for), which requires whole-program analysis and hence is contradictory with the goal of modularity.

Chang *et al.* [6] presented a refined CPCC system in which "local invariants" refine "continuations." A local invariant essentially consists of two related components—an "assumption" of the current state and a list of "progress continuations" which are used for handling code pointers. To allow the VCGen extension to manipulate predicates using first-order logic, only a syntactically restricted form of invariants are used. Although this is necessary for automatic proof construction for type-safety, it is insufficient in handling higher-order code pointers in general. As a result, these local invariants are only used to handle more gracefully certain fixed patterns of code pointers, such as return pointers. Other situations, such as virtual dispatch, would still require whole-program analysis for the VCGen extension to discharge the decoder's indirect continuations. In particular, it is unclear how this approach extends to support arbitrary safety policies and nested continuations.

Reynolds [38] identified a similar problem in separation logic and referred to it as "embedded" code pointers, which are "difficult to describe in the first-order world of Hoare logic." He speculated that a potential solution lies in marrying separation logic with continuation-passing style. The idea was only described briefly and informally, and a convincing development remains yet to come forth. Recently, O'Hearn *et al.* [31] investigated proof rules for modularity and information hiding for first-order procedures using separation logic. However, it is unclear how their approach extends to support higher-order features. It is also worth mentioning that related problems on higher-order procedures in Hoare logic can be traced back to the late seventies [7, 8].

We are currently working on a system which addresses this issue more properly. It involves a unified framework for type systems and Hoare logic, and allows reasoning using both types and predicates. A similar idea on deploying a logic in a type system for an assembly language is due to Ahmed and Walker [3]. We intend to present this work, which is sufficiently interesting and self-contained, in a separate paper.

6.4 Other Future Work

CCAP is based on an abstract concurrent machine which shields us from the details of thread management such as creation and scheduling. However, many concurrent programs are executed on sequential machines, relying on thread libraries to take charge in thread management. One interesting possibility is to implement a certified thread library using a sequential CAP language by verifying the implementation of thread primitives, which are really sequential programs with advanced handling of code pointers. This work helps bring CCAP to a more solid ground regarding trustworthy computing.

We are also working on the application of the CAP languages to a realistic machine model, such as the Intel Architecture (x86). The intention is to verify the very same assembly code as those running on actual processors. Our experience indicates that the tasks involved here are mostly engineering issues, including the handling of fixed-size integers, byte-aligned (as opposed to word-aligned) addressing mode, finite memory model (restricted by word-size), and encoding and decoding of variable-length instructions. We believe, however, that these aspects are orthogonal to the reasoning of most program properties of interest and hence their handling can be facilitated with proper proof libraries.

7 Conclusion

We have presented a language CCAP for verifying safety properties of concurrent assembly code. The language is modeled based on a concurrent abstract machine, adapting established techniques for concurrency verification at an assembly level. CCAP has been developed using the Coq proof assistant, alone with a formal soundness proof and the verified example CCAP programs. Only small examples are given in this paper for illustrative purposes, but practical applications are within reach, especially for small-scale software such as core libraries, critical components or embedded systems. Its potential application to large-scale software calls for a further development on modularity.

8 Acknowledgment

We thank the anonymous referees for suggestions on improving the presentation.

9 References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. on Programming Languages and Systems*, 15(1):73–132, 1993.

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

[3] A. Ahmed and D. Walker. The logical approach to stack typing. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 74–85. ACM Press, 2003.

[4] 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.

[5] A. Cau and P. Collette. Parallel composition of assumption-commitment specifications. *Acta Informatica*, 33(2):153–176, 1996.

[6] B.-Y. E. Chang, G. C. Necular, and R. R. Schneck. Extensible code verification. Unpublished manuscript, 2003.

[7] E. M. Clarke. Programming language constructs for which it is impossible to obtain good Hoare axiom systems. In *Journal of the Association for Computing Machinery*, pages 129–147, Jan. 1979.

[8] E. M. Clarke. The characterization problem for Hoare logics. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 89–106. Prentice Hall, 1985.

[9] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, Cambridge, UK, 2001.

[10] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[11] 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.

[12] R. W. Floyd. Assigning meanings to programs. In A. M. Society, editor, *Proceedings of the Symposium on Applied Math. Vol. 19*, pages 19–31, Providence, R.I., 1967.

[13] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.

[14] N. Glew and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.*, pages 250–261. ACM Press, Jan. 1999.

[15] 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.

[16] T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In *Formal Methods in Computer-Aided Design*, pages 421–432, 1998.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[18] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, Jan. 1971.

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

[20] T. Hoare. The verifying compiler: A grand challenge for computing research. In *Proc. 2003 International Conference on Compiler Construction (CC'03), LNCS Vol. 2622*, pages 262–272, Warsaw, Poland, Apr. 2003. Springer-Verlag Heidelberg.

[21] J. Hooman, W.-P. de Roever, P. Pandya, Q. Xu, P. Zhou, and H. Schepers. A compositional approach to concurrency and its applications. Incomplete manuscript. `http://www.informatik.uni-kiel.de/inf/deRoever/books/`, Apr. 2003.

[22] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[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] L. Lamport and F. B. Schneider. The "Hoare logic" of CSP, and all that. *ACM Trans. on Programming Languages and Systems*, 6(2):281–296, Apr. 1984.

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

[27] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

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

[29] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. 2nd USENIX Symp. on Operating System Design and Impl.*, pages 229–243, 1996.

[30] G. C. Necula and R. R. Schneck. A sound framework for untrustred verification-condition generators. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 248–260. IEEE Computer Society, July 2003.

[31] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. 29th ACM Symp. on Principles of Prog. Lang.*, pages 268–280, Venice, Italy, Jan. 2004. ACM Press.

[32] M. Ossefort. Correctness proofs of communicating processes: Three illustrative examples from the literature. *ACM Trans. on Programming Languages and Systems*, 5(4):620–640, Oct. 1983.

[33] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[34] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Languages and Systems*, 4(3):455–495, July 1982.

[35] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*, volume 664 of *LNCS*. Springer-Verlag, 1993.

[36] A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, pages 123–144, 1985.

[37] 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.

[38] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Los Alamitos, California, July 2002. IEEE Computer Society.

[39] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symp. on Principles of Prog. Lang.*, pages 217–232, Portland, OR, Jan. 2002. ACM Press.

[40] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Proc. 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *LNCS*, pages 369–391, New Delhi, 1985. Springer-Verlag.

[41] C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1-3):347–359, 1988.

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

[43] The FLINT Project. Coq (v7.3.1) implementation for CCAP language, soundness and examples. `http://flint.cs.yale.edu/flint/publications/vsca.html` (17k), Mar. 2004.

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

[45] Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In *International Conference on Concurrency Theory*, pages 267–282, 1994.

[46] 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, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[47] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3):101–127, 2004.

[48] Y. Yu. *Automated proofs of object code for a widely used microprocessor*. PhD thesis, University of Texas at Austin, Austin, TX, 1992.

$$
\begin{array}{rcl}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{S}, [\mathbb{T}_1 \ldots \mathbb{T}_n], i) \text{ where } i \in \{1 \ldots n\} \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{M}, \mathbb{R}) \\
(\mathit{Memory}) & \mathbb{M} & ::= \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= \{\mathtt{r}_k\}^{k \in \{0 \ldots 7\}} \\
(\mathit{Thread}) & \mathbb{T} & ::= (\mathbb{C}, \mathbb{I}) \\
(\mathit{CdHeap}) & \mathbb{C} & ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{l} & ::= \mathtt{n}\ (\mathit{nat\ nums}) \\
(\mathit{WordVal}) & \mathtt{w} & ::= \mathtt{n}\ (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathsf{c}; \mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \\
(\mathit{Comm}) & \mathsf{c} & ::= \mathsf{yield} \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{be}\ \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 \\
(\mathit{ProgSpec}) & \Phi & ::= (\mathit{Inv}, [\Psi_1 \ldots \Psi_n], [\mathbb{A}_1 \ldots \mathbb{A}_n], [\mathbb{G}_1 \ldots \mathbb{G}_n]) \\
(\mathit{CHSpec}) & \Psi & ::= \{\mathtt{f} \rightsquigarrow (\mathtt{p}, \mathtt{g})\}^* \\
(\mathit{ThrdSpec}) & \Theta & ::= (\mathit{Inv}, \Psi, \mathbb{A}, \mathbb{G}) \\
(\mathit{Invariant}) & \mathit{Inv} & \in \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Assert}) & \mathtt{p} & \in \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Assume}) & \mathbb{A} & \in \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop} \\
(\mathit{Guarantee}) & \mathbb{G}, \mathtt{g} & \in \mathit{State} \rightarrow \mathit{State} \rightarrow \mathit{Prop}
\end{array}
$$

Figure 15. Syntax of a generalized CCAP.

A CCAP with Multiple Threads

The CCAP presented in Section 3 supports only two threads, which is easy to understand yet sufficient in demonstrating all the key ideas. This appendix gives a generalized account. Figures 15 and 16 give the syntax and operational semantics of a generalized CCAP supporting more than two threads. The auxiliary state update macro is the same as shown in Figure 7. Inference rules and soundness lemmas follow. In particular, a generalized inference rule for well-formed program is given to support multiple threads. The inference rules for well-formed code heap and well-formed instruction sequence, in contrast, remain exactly the same as in the two-threaded version (these rules are gathered here for ease of reference). This demonstrates that CCAP is indeed thread-modular.

$((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}_i)\ldots],i) \longmapsto \mathbb{P}$ where	
if $\mathbb{I}_i =$	then $\mathbb{P} =$
$\mathsf{jd}\ \mathtt{f}$	$((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}')\ldots],i)$ where $\mathbb{C}_i(\mathtt{f}) = \mathbb{I}'$
$\mathsf{yield};\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}')\ldots],i')$ where $i' \in \{1\ldots n\}$
$\mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}')\ldots],i)$ when $\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)$; and $((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}'')\ldots],i)$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}_i(\mathtt{f}) = \mathbb{I}''$
$\mathsf{be}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}'$	$((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}')\ldots],i)$ when $\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)$; and $((\mathbb{M},\mathbb{R}),[\ldots(\mathbb{C}_i,\mathbb{I}'')\ldots],i)$ when $\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}_i(\mathtt{f}) = \mathbb{I}''$
$\mathtt{c};\mathbb{I}'$ for remaining cases of $\mathtt{c}$	$(\mathtt{Next}(\mathtt{c},(\mathbb{M},\mathbb{R})),[\ldots(\mathbb{C}_i,\mathbb{I}')\ldots],i)$

Figure 16. Operational semantics of a generalized CCAP.

Note that we do not support the dynamic creation and termination of threads in this language, but extensions on them are natural and have little to do with the concurrency verification techniques presented in this paper. To be specific, even for a dynamically created thread, the code has to be written beforehand, and the behavior has to obey the global invariant, the assumptions and the guarantees. Hence the thread's counterparts exist statically in the code heap and the specification. The safety reasoning used for a CCAP program remains unchanged, no matter how many individual threads are created for the same code. Therefore, the addition of thread creation and termination mostly affects the operational semantics, and the inference rules can be adapted with little effort.

Well-formed program $\boxed{\Phi;([\mathtt{p}_1\ldots\mathtt{p}_n],\mathtt{g}) \vdash \mathbb{P}}$

$$\frac{\begin{array}{l}\Phi = (Inv,[\Psi_1\ldots\Psi_n],[\mathbb{A}_1\ldots\mathbb{A}_n],[\mathbb{G}_1\ldots\mathbb{G}_n])\\ \Theta_k = (Inv,\Psi_k,\mathbb{A}_k,\mathbb{G}_k) \quad \Theta_k \vdash \mathbb{C}_k \quad \forall k \in \{1\ldots n\}\\ (Inv \wedge \mathtt{p}_i\ \mathbb{S}) \qquad \forall \mathbb{S}''.(\mathtt{g}\ \mathbb{S}\ \mathbb{S}'') \supset (\mathtt{p}_k\ \mathbb{S}'') \quad \forall k \ne i\\ \forall \mathbb{S}'.\forall \mathbb{S}''.(Inv \wedge \mathtt{p}_k\ \mathbb{S}') \supset (\mathbb{A}_k\ \mathbb{S}'\ \mathbb{S}'') \supset (\mathtt{p}_k\ \mathbb{S}'') \quad \forall k \ne i\\ \Theta_i;(\mathtt{p}_i,\mathtt{g}) \vdash \mathbb{I}_i \qquad \Theta_k;(\mathtt{p}_k,\mathbb{G}_k) \vdash \mathbb{I}_k \quad \forall k \ne i\end{array}}{\Phi;([\mathtt{p}_1\ldots\mathtt{p}_n],\mathtt{g}) \vdash (\mathbb{S},[(\mathbb{C}_1,\mathbb{I}_1)\ldots(\mathbb{C}_n,\mathbb{I}_n)],i)} \quad (\text{PROG})$$

Well-formed code heap $\boxed{\Theta \vdash \mathbb{C}}$

$$\frac{\begin{array}{l}\Theta = (Inv,\Psi,\mathbb{A},\mathbb{G})\\ \Psi = \{\mathtt{f}_j \rightsquigarrow (\mathtt{p}_j,\mathtt{g}_j)\}^{j\in\{1\ldots m\}}\\ \Theta;(\mathtt{p}_j,\mathtt{g}_j) \vdash \mathbb{I}_j \quad \forall j \in \{1\ldots m\}\end{array}}{\Theta \vdash \{\mathtt{f}_j \rightsquigarrow \mathbb{I}_j\}^{j\in\{1\ldots m\}}} \quad (\text{CODEHEAP})$$

Well-formed instruction sequence $\boxed{\Theta;(\mathtt{p},\mathtt{g}) \vdash \mathbb{I}}$

$$\frac{\begin{array}{l}\Theta = (Inv,\Psi,\mathbb{A},\mathbb{G})\\ \forall \mathbb{S}.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{g}\ \mathbb{S}\ \mathbb{S})\\ \forall \mathbb{S}.\forall \mathbb{S}'.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathbb{A}\ \mathbb{S}\ \mathbb{S}') \supset (\mathtt{p}\ \mathbb{S}')\\ \Theta;(\mathtt{p},\mathbb{G}) \vdash \mathbb{I}\end{array}}{\Theta;(\mathtt{p},\mathtt{g}) \vdash \mathsf{yield};\mathbb{I}} \quad (\text{YIELD})$$

$$\frac{\begin{array}{l}\Theta = (Inv,\Psi,\mathbb{A},\mathbb{G})\\ \forall \mathbb{S}.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (Inv \wedge \mathtt{p}'\ \mathtt{Next}(\mathtt{c},\mathbb{S}))\\ \forall \mathbb{S}.\forall \mathbb{S}'.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{g}'\ \mathtt{Next}(\mathtt{c},\mathbb{S})\ \mathbb{S}') \supset (\mathtt{g}\ \mathbb{S}\ \mathbb{S}')\\ \Theta;(\mathtt{p}',\mathtt{g}') \vdash \mathbb{I} \qquad \mathtt{c} \in \{\mathsf{add},\mathsf{sub},\mathsf{movi}\}\end{array}}{\Theta;(\mathtt{p},\mathtt{g}) \vdash \mathtt{c};\mathbb{I}} \quad (\text{SIMPL})$$

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

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

$$\frac{\begin{array}{l}\Theta = (Inv,\Psi,\mathbb{A},\mathbb{G}) \qquad \Psi(\mathtt{f}) = (\mathtt{p}',\mathtt{g}')\\ \forall \mathbb{S}.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{p}'\ \mathbb{S})\\ \forall \mathbb{S}.\forall \mathbb{S}'.(Inv \wedge \mathtt{p}\ \mathbb{S}) \supset (\mathtt{g}'\ \mathbb{S}\ \mathbb{S}') \supset (\mathtt{g}\ \mathbb{S}\ \mathbb{S}')\end{array}}{\Theta;(\mathtt{p},\mathtt{g}) \vdash \mathsf{jd}\ \mathtt{f}} \quad (\text{JD})$$

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

$$\frac{\begin{array}{l}\Theta = (Inv,\Psi,\mathbb{A},\mathbb{G}) \qquad \Psi(\mathtt{f}) = (\mathtt{p}',\mathtt{g}')\\ \forall \mathbb{M}.\forall \mathbb{R}.(\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M},\mathbb{R})) \supset (\mathtt{p}'\ (\mathbb{M},\mathbb{R}))\\ \forall \mathbb{M}.\forall \mathbb{R}.\forall \mathbb{S}'.(\mathbb{R}(\mathtt{r}_s) = \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M},\mathbb{R}))\\ \qquad \supset (\mathtt{g}'\ (\mathbb{M},\mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M},\mathbb{R})\ \mathbb{S}')\\ \forall \mathbb{M}.\forall \mathbb{R}.(\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M},\mathbb{R})) \supset (\mathtt{p}''\ (\mathbb{M},\mathbb{R}))\\ \forall \mathbb{M}.\forall \mathbb{R}.\forall \mathbb{S}'.(\mathbb{R}(\mathtt{r}_s) \ne \mathbb{R}(\mathtt{r}_t)) \supset (Inv \wedge \mathtt{p}\ (\mathbb{M},\mathbb{R}))\\ \qquad \supset (\mathtt{g}''\ (\mathbb{M},\mathbb{R})\ \mathbb{S}') \supset (\mathtt{g}\ (\mathbb{M},\mathbb{R})\ \mathbb{S}')\\ \Theta;(\mathtt{p}'',\mathtt{g}'') \vdash \mathbb{I}\end{array}}{\Theta;(\mathtt{p},\mathtt{g}) \vdash \mathsf{be}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f};\mathbb{I}} \quad (\text{BE})$$

Lemma 5 (Progress) Let $\Phi = (Inv,[\Psi_1\ldots\Psi_n],[\mathbb{A}_1\ldots\mathbb{A}_n],[\mathbb{G}_1\ldots\mathbb{G}_n])$. If $\Phi;([\mathtt{p}_1\ldots\mathtt{p}_n],\mathtt{g}) \vdash (\mathbb{S},[(\mathbb{C}_1,\mathbb{I}_1)\ldots(\mathbb{C}_n,\mathbb{I}_n)],i)$ where $i \in \{1\ldots n\}$, then $(Inv\ \mathbb{S})$ and there exists a program $\mathbb{P}$ such that $(\mathbb{S},[(\mathbb{C}_1,\mathbb{I}_1)\ldots(\mathbb{C}_n,\mathbb{I}_n)],i) \longmapsto \mathbb{P}$.

Lemma 6 (Preservation) Let $\Phi = (Inv,[\Psi_1\ldots\Psi_n],[\mathbb{A}_1\ldots\mathbb{A}_n],[\mathbb{G}_1\ldots\mathbb{G}_n])$. Suppose $\forall \mathbb{S}'.\forall \mathbb{S}''.(\mathbb{G}_j\ \mathbb{S}'\ \mathbb{S}'') \supset (\mathbb{A}_k\ \mathbb{S}'\ \mathbb{S}'')$ for all $j \ne k$. If $\Phi;([\mathtt{p}_1\ldots\mathtt{p}_n],\mathtt{g}) \vdash (\mathbb{S},[(\mathbb{C}_1,\mathbb{I}_1)\ldots(\mathbb{C}_n,\mathbb{I}_n)],i)$ where $i \in \{1\ldots n\}$, and $(\mathbb{S},[(\mathbb{C}_1,\mathbb{I}_1)\ldots(\mathbb{C}_n,\mathbb{I}_n)],i) \longmapsto \mathbb{P}$, then there exists $\mathtt{p}'_1,\ldots,\mathtt{p}'_n$ and $\mathtt{g}'$ such that $\Phi;([\mathtt{p}'_1\ldots\mathtt{p}'_n],\mathtt{g}') \vdash \mathbb{P}$.