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 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

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 susceptible 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 the 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-
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.
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 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 data 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.”

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 data 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 $\Psi$ (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-

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{(Program)} & $P := (C, S, I)$
\hline
\textbf{(CodeHeap)} & $C := \{ f \leadsto \ell \}^*$
\hline
\textbf{(State)} & $S := ([R])$
\hline
\textbf{(Heap)} & $H := \{ (r \leadsto w) \}^*$
\hline
\textbf{(RegFile)} & $R := \{ (x \leadsto w) \}^*$
\hline
\textbf{(Register)} & $r := \{ x \}^{k=0..7}$
\hline
\textbf{(Labels)} & $f, l := n$ (nat nums)
\hline
\textbf{(WordVal)} & $w := n$ (nat nums)
\hline
\textbf{(InstrSeq)} & $I := c; l \mid j d f$
\hline
\textbf{(Command)} & $c := add r_d, r_x, r_y \mid movi r_x, w \mid bgt r_x, r_y, f$
\hline
\textbf{(CdHpSpec)} & $\Psi := \{ f \leadsto a \}^*$
\hline
\textbf{(Assert)} & $a \in \text{State}$
\hline
\end{tabular}
\caption{Syntax and verification constructs of Simple CAP.}
\end{figure}

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
$P$ & $(C, ([R]), I) \rightarrow P$ where
\hline
$C$ & $(C, ([R]), I) \rightarrow I$ where $(C(f)) = I$
\hline
$add r_d, r_x, r_y, I$ & $(C, ([R]), I) \rightarrow (C, ([R]), [r_d \leadsto r_x \leadsto r_y]), I$
\hline
$movi r_x, w, I$ & $(C, ([R]), I) \rightarrow (C, ([R]), [r_x \leadsto w]), I$
\hline
$bgt r_x, r_y, f, I$ & $(C, ([R]), I) \rightarrow (C, ([R]), [r_x \leq r_y], I)$
\hline
$ld r_d, r_y, I$ & $(C, ([R]), I) \rightarrow (C, ([R]), [r_d + w \rightarrow R(r_y)], I)$
\hline
$st r_d, r_y, I$ & $(C, ([R]), I) \rightarrow (C, ([R]), [r_d + w \in \text{dom}(R)], I)$
\hline
\end{tabular}
\caption{Operational semantics of Simple CAP.}
\end{table}
implement 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 formulæ-as-types principle (Curry-Howard isomorphism [22]). We further define our assertion language (a calculus that corresponds to a variant of higher-order predicate logic) to contain all CAP language constructs and their semantics 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 formulæ-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 \( r_1, r_2 \) and \( r_3 \) store the same value is written in Coq as the assertion (state predicate) \( s \vdash H, R \equiv _s a \) in \( R(r_1) = R(r_2) \land R(r_2) = R(r_3) \) where \( s \vdash H, R \) is a binding of variable \( a \) of type \( \text{State} \), \( \vdash \) is a pattern matching construct, and \( \land \) is the logical connective \( \land \). Proving the validity of propositions is done by using tactics which correspond to logical inference rules such as "\( \land \)-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:

\[
\begin{align*}
\Psi \vdash C & \quad \text{(well-formed program)} \\
\Psi \vdash \{a\} I & \quad \text{(well-formed code heap)} \\
\Psi \vdash \{a\} I & \quad \text{(well-formed instruction sequence)}
\end{align*}
\]

The intuition of well-formed instruction sequence, for example, is that if the instruction sequence \( I \) starts execution in a machine state which satisfies the assertion \( a \), then executing \( I \) is safe with respect to the specification \( \Psi \).

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 \( a \) if both the code heap and the instruction sequence are well-formed, and the machine state satisfies the assertion \( a \). 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 "\( \vdash \)" 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 \( a' \) which serves both as the postcondition of the addition (\( a' \) 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 \( a' \), 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 \( \Psi \vdash \{a\} (C, S, I) \), then for all natural number \( n \), there exists a program \( I' \) such that \( (C, S, I) \vdash n \rightarrow I' \), and

- if \( (C, S, I) \vdash I \rightarrow I' \), then \( \Psi(I') S' \); and
- if \( (C, S, I) \vdash I \rightarrow I' \), then \( \Psi(I') (H, R) \).

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-
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 jdf 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.

### 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 \( A \)
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 P consists of a shared state component S made up of a memory M and a register file R, two code heaps C (one for each thread), two current instruction sequences 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 is 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 (A) and guarantees (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 G and an assumption 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 p describing valid states, but also a guarantee g describing valid state transitions—it is safe for the current thread to yield control only after making a state transition allowed by g. We call g a “local guarantee” to distinguish it from 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 A and guarantee 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:
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.

\[
\Phi: \{p_1, p_2, g\} \vdash \mathcal{P} \quad \Theta: \{p, g\} \vdash \mathcal{C} \quad \Theta: (p, g) \vdash \mathcal{I}
\]

(Well-formed program) (Well-formed code heap) (Well-formed instr. seq.)

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 \( \Phi \), but also an assertion \( p_1 \) describing the current state, an assertion \( p_2 \) describing the state in which thread 2 may take control, and a relation \( g \) 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 \( \Phi \). Lines 2 requires that both the global invariant \( \mathcal{I} \) and the current assertion \( p_1 \) must hold on the current state; it also restricts the assertion \( p_2 \) of thread 2 and the current guarantee \( g \) of thread 1: if some operations satisfying \( g \) is performed on the current state, then the result state satisfies \( p_2 \). This indicates that it is sufficient to refer to \( g \) 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 \( p_2 \) will not be disturbed by allowed (assumed) transitions of thread 1. Hence line 3 restricts the assertion \( p_2 \) to preserve itself under state transitions that satisfy the assumption \( a_2 \). 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 \( g_2 \) 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., \( G_1 \supseteq A_2 \) and \( G_2 \supseteq A_1 \). 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**

\[
\Phi = (\text{Inv}, \Psi, A, G) \\
\Psi = \{ f_j \supset (p_j, g_j) \}_{j \in \{1, ... n\}} \\
\Theta: (p_j, g_j) \vdash I_j \quad \forall j \in \{1, ... n\} \\
\Theta: (p, g) \vdash I \\
\Theta: (p, g) \vdash \text{yield}; I
\]

(WELL-FORMED-HEAP)

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**

\[
\Phi = (\text{Inv}, \Psi, A, G) \\
\forall S. \mathcal{I} \supset (g \supset S) \\
\forall S. \forall S'. (\mathcal{I} \supset \mathcal{P}(g \supset S')) \supset (g \supset S') \\
\Theta: (p, g) \vdash I \\
\Theta: (p, g) \vdash I \\
\Theta: (p, g) \vdash \text{yield}; I
\]

(WELL-FORMED-EXECUTION)

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

\[
\Theta = (\text{Inv}, \Psi, A, G) \\
\forall S. (\mathcal{I} \supset (p \supset \text{next}(c, S))) \\
\forall S. (\mathcal{I} \supset (p \supset (g \text{next}(c, S)')) \supset (g \supset S')) \\
\Theta: (p, g') \vdash I \\
\Theta: (p, g) \vdash c : I
\]

(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 \( (p', g') \) under which the tail instruction sequence \( I \) is well-formed. Moreover, the global invariant \( \mathcal{I} \) and the intermediate assertion \( p' \) must hold on the updated machine state, and the intermediate guarantee \( g' \) applied to the updated machine state must be no weaker than the current guarantee \( g \) applied to the current state.

Consider a sequence of states \( S_1 \ldots S_n \) between two “safe points” (i.e., between two yield commands). Suppose we have a sequence of guarantees \( g_1 \ldots g_n \) such that:

\[
(g_2 S_2 S_n) \supset (g_1 S_1 S_n); \\
(g_3 S_3 S_n) \supset (g_2 S_2 S_n); \\
... \\
(g_{n-1} S_{n-1} S_n) \supset (g_{n-2} S_{n-2} S_n); \\
(g_n S_n S_n) \supset (g_{n-1} S_{n-1} S_n).
\]

(LOCAL-GUARD)

From these we know that the state transition from \( S_1 \) to \( S_n \) satisfies the original guarantee \( g_1 \) if only \( g_n S_n S_n \), which will be established at the yield command.

\[
\Theta = (\text{Inv}, \Psi, A, G) \\
\forall M. \forall \mathcal{R}. (\mathcal{I} \supset (M \supset p (M, R))) \supset ((\mathcal{R}(r_x) + w) \in \text{dom}(M)) \\
\forall S. (\mathcal{I} \supset (p \supset \text{next}(c, S))) \\
\forall S. (\mathcal{I} \supset (p \supset (g \text{next}(c, S)') \supset (g \supset S'))) \\
\Theta: (p, g') \vdash I \\
\Theta: (p, g) \vdash c = \text{id}_{x, \text{fin}}(x)(w) \\
\Theta: (p, g) \vdash c; I
\]

(LD)
\[ \Theta = (\text{Im}, \Psi, A, G) \]
\[ \forall M. \forall \Psi. (\text{Im} \land p(M, R)) \rightarrow (\Psi(\tau(\tau)), \tau) \in \text{dom}(M) \]
\[ \forall S. (\text{Im} \land p(S)) \rightarrow (\Psi(\tau(c)), \tau) \in S \]
\[ \Theta(p, g) \vdash c = \text{dt}(a(p), r) \]
\[ \Theta(p, g) = c, \Pi \]

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.

\[ \Theta = (\text{Im}, \Psi, A, G) \]
\[ \forall M. \forall \Psi. (\text{Im} \land p(M, R)) \rightarrow (\Psi(\tau(\tau)), \tau) \in \text{dom}(M) \]
\[ \forall S. (\text{Im} \land p(S)) \rightarrow (\Psi(\tau(c)), \tau) \in S \]
\[ \Theta(p, g) \vdash c = \text{dt}(a(p), r), c' \]

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 machine invariant \( \text{Im} \) explicitly, because the machine state remains the same.

\[ \Theta = (\text{Im}, \Psi, A, G) \]
\[ \forall M. \forall \Psi. (\text{Im} \land p(M, R)) \rightarrow (\Psi(\tau(\tau)), \tau) \in \text{dom}(M) \]
\[ \forall S. (\text{Im} \land p(S)) \rightarrow (\Psi(\tau(c)), \tau) \in S \]
\[ \Theta(p', g') \vdash \pi \]

Proof sketch: By induction on the structure of \( \text{Im} \). (Im \( S \)) holds by definition of well-formed program. In the cases where \( \tau \) 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 \( \tau \) starts with Id 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 \( \tau \) starts with bgt or be, where \( \tau \) 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 \[ \Theta = (\text{Im}, \Psi, A, G) \]
\[ \Psi(\tau) = (p, g) \] and \( \Psi(\tau) = (p, g) \), then \( \Theta(p, g) \vdash \Pi \).

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

Lemma 4 (Preservation) \[ \Phi = (\text{Im}, \Psi, A, G_1, G_2) \]
\[ \forall S. S' \rightarrow (\text{Im} \land p(M, R)) \rightarrow (\Psi(\tau(c)), \tau) \in S \]
\[ \forall S. S' \rightarrow (\text{Im} \land p(M, R)) \rightarrow (\Psi(\tau(c)), \tau) \in S \]
\[ \Phi(p_1, p_2, g) \vdash (S, C_1, C_2, i_1, i_2, i) \rightarrow P \]

Proof sketch: By induction on the structure of \( \tau \). We must establish the premises (lines 2–5) of Rule PROG1 or PROG2.

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 cs1 is true if thread i is in its crit-
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 "\{\}" is omitted from the global invariant and every assertion; a "\{\}" is omitted from every assumption and guarantee. In these predicates, \( M \) and \( R \) refer to the memory component and the register file component of the bounded variable \( S \) respectively, and \( M' \) and \( R' \) refer to those of the bounded variable \( S' \).

For this example, state predicate True is used as the assertion \( p \) 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 \( a_2 \) or \( c_2 \), the guarantee \( G_1 \) can be used as the local guarantee \( g \) throughout the code of thread 1. Take the code block labeled \( cse1 \) 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 \( c_1 \); 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 (\( T_1 \) and \( T_2 \)) share two forks (represented by memory locations \( fork_1 \) and \( fork_2 \)). 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 \( fork_1 \) before acquiring \( fork_2 \).

The deadlock freedom property for this example can be expressed as \( \forall (fork_2 = 0) \lor (fork_2 = fork_1) \), 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 \( \forall (fork_2 = 0) \lor (\exists n. fork_{2n} = fork_{2n+1}) \), where \( n \) is the number of philosophers and \( fork_n \) is the "greatest fork." Note that

<table>
<thead>
<tr>
<th>Variables:</th>
<th>Initially:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean ( a_1, a_2, c_1, c_2 )</td>
<td>( \neg c_1 \land \neg c_2 )</td>
</tr>
</tbody>
</table>

**Thread 1:**

\[
\text{while (true)} \{
  a_1 := \text{true};
  c_1 := \neg a_2;
  \text{if (c_1)} \{
    //critical sec.
    c_1 := \text{false};
  \}
  a_1 := \text{false};
\}
\]

**Thread 2:**

\[
\text{while (true)} \{
  a_2 := \text{true};
  c_2 := \neg a_1;
  \text{if (c_2)} \{
    //critical sec.
    c_2 := \text{false};
  \}
  a_2 := \text{false};
\}
\]

**Figure 9.** Dekker’s mutual exclusion algorithm.

<table>
<thead>
<tr>
<th>Variables:</th>
<th>Initially:</th>
</tr>
</thead>
<tbody>
<tr>
<td>nat ( fork_1, fork_2 )</td>
<td>( fork_1 = 0 \land fork_2 = 0 )</td>
</tr>
</tbody>
</table>

**Thread 1:**

\[
\text{while (true)} \{
  \text{acquire(fork_1, 1)};
  \text{acquire(fork_2, 1)};
  // eat
  \text{release(fork_2)};
  \text{release(fork_1)};
  // think
\}
\]

**Thread 2:**

\[
\text{while (true)} \{
  \text{acquire(fork_1, 2)};
  \text{acquire(fork_2, 2)};
  // eat
  \text{release(fork_2)};
  \text{release(fork_1)};
  // think
\}
\]

**Figure 11.** Dining philosophers algorithm.

To decompose the verification problem into two sequential ones, thread \( i \) makes the assumption that the other thread not modify variables \( a_i \) and \( c_i \). Furthermore, the mutual exclusion property is strengthened to the invariant \( \neg (c_1 \land c_2) \land (c_1 \land \neg a_1) \land (c_2 \land \neg a_2) \).

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.

**Figure 10.** CCAP implementation for Dekker’s algorithm.
Inv \equiv (M(fork_2) = 0) \lor (M(fork_2) = M(fork_1))
A_1, G_2 \equiv (M(fork_2) = 2 \lor M'(fork_1) = 2)
\land (M(fork_1) = 1 \leftrightarrow M'(fork_1) = 1)
\land (M(fork_2) = 1 \leftrightarrow M'(fork_2) = 1)
A_2, G_1 \equiv (M'(fork_2) = 1 \lor M'(fork_1) = 1)
\land (M(fork_1) = 2 \leftrightarrow M'(fork_1) = 2)
\land (M(fork_2) = 2 \leftrightarrow M'(fork_2) = 2)

Initial M \equiv \{fork_1 \sim 0, fork_2 \sim 0\}
Initial thread: i where i \in \{1, 2\}
Initial instruction sequence: i \equiv jd fa_i
Initial precondition triple: (p_1, p_2, G) \equiv (M(fork_1) \neq 1 \land M(fork_2) \neq 1), (M(fork_1) \neq 2 \land M(fork_2) \neq 2, G_i)

fa_1: \neg (M(fork_1) \neq 1 \land M(fork_2) \neq 1, G_1)
yield
mov r_1, fork_1
ld r_2, r_1(0)
mov r_3, 0
bgt r_2, r_3, fa_1
\neg (M(fork_1) = 0 \land M(fork_2) \neq 1
\land R(r_1) = fork_1, G_1)
mov r_2, 1
\neg (M(fork_1) = 0 \land M(fork_2) \neq 1
\land R(r_1) = fork_1 \land R(r_2) = 1,
(M'(fork_2) = 1 \lor M'(fork_1) = 1)
\land (M'(fork_1) \neq 2)
\land (M(fork_2) = 2 \leftrightarrow M'(fork_2) = 2))
sl r_1(0), r_2
\neg (M(fork_1) = 1 \land M(fork_2) \neq 1,
(M'(fork_2) = 1 \lor M'(fork_1) = 1)
\land (M'(fork_1) \neq 2)
\land (M(fork_2) = 2 \leftrightarrow M'(fork_2) = 2))
yield
\neg (M(fork_1) = 1 \land M(fork_2) \neq 1, G_1)
jd fb_1

fb_1: \neg (M(fork_1) = 1 \land M(fork_2) \neq 1, G_1)
yield
mov r_1, fork_2
ld r_2, r_1(0)
mov r_3, 0
bgt r_2, r_3, fb_1
mov r_2, 1
sl r_1(0), r_2
\neg (M(fork_1) = 1 \land M(fork_2) = 1,
(M'(fork_2) = 1 \lor M'(fork_1) = 1)
\land (M'(fork_1) \neq 2)
\land (M'(fork_2) \neq 2))
yield
\neg (M(fork_1) = 1 \land M(fork_2) \neq 1, G_1)
mov r_1, fork_2
mov r_2, 0
sl r_1(0), r_2
yield
\neg (M(fork_1) = 1 \land M(fork_2) \neq 1, G_1)
mov r_1, fork_2
mov r_2, 0
sl r_1(0), r_2
yield
\neg (M(fork_1) \neq 1 \land M(fork_2) \neq 1, G_1)
jd fa_1

Figure 13. GCD algorithm.

Variables:
- Initially:
  - nat a, b;
  - a = α \land b = β
- Thread:
  - while (a \neq b)
  - if (a > b)
    - a := a - b;
  - if (b > a)
    - b := b - a;

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 G_1) to acquire fork_2 only after acquiring fork_1, (M'(fork_2) = 1 \lor M'(fork_1) = 1). Due to the nature of our simulation using the values of fork_1 and 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 (M(fork_1) = 2 \land \neg M'(fork_1) = 2), where \neg means “if and only if”.

Figure 12. CCAP implementation for dining philosophers algorithm.

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 M(a) < M(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 \( \delta \) be a shorthand for \( \gcd(a, \beta) \).

\[
\begin{align*}
\text{Im} & \equiv \gcd(M(a), M(b)) = \delta \\
A_1, G_2 & \equiv (M(a) \geq M(b) \land (M(a) = M'(a) \land M(b) = M'(b))) \\
A_2, G_1 & \equiv (M(b) \geq M(a) \land (M(a) = M'(a) \land M(b) = M'(b))) \\
& \land (M(b) < M(a) \land (\gcd(M'(a), M'(b)) = \delta))
\end{align*}
\]

Initial \( M \equiv \{a \sim a, b \sim b\} \)

Initial thread: \( i \) where \( i \in \{1, 2\} \)

Initial instruction sequences: \( i \equiv \text{jd } \text{loop}_i \)

Initial precondition triple: \( (P_1, P_2, G) \equiv (\text{gcd}(M(a), M(b)) = \delta, \text{gcd}(M'(a), M'(b)) = \delta, G_i) \)

\[
\begin{align*}
\text{loop}_1 : \{\{\text{True}, G_1\}\} & \text{ yield} \\
\text{movi } r_1, a & \text{ movi } r_1, a \\
\text{ld } r_2, r_1(0) & \{\{M(a) > M(b), G_1\}\} \\
\text{movi } r_3, b & \text{gcd}(M'(a), M'(b)) = \delta \}
\end{align*}
\]

\[
\begin{align*}
\text{ld } r_1, r_3(0) & \text{ ld } r_1, r_3(0) \\
\text{be } r_2, r_4, \text{done}_1 & \text{ movi } r_2, r_3 \\
\{\{\text{True}, G_1\}\} & \text{ld } r_4, r_3(0) \\
\text{movi } r_1, a & \text{movi } r_1, a \\
\text{ld } r_2, r_1(0) & \{\{\text{True}, G_1\}\} \\
\text{movi } r_3, b & \text{gcd}(M'(a), M'(b)) = \delta \}
\end{align*}
\]

\[
\begin{align*}
\text{ld } r_1, r_4(0) & \text{ yield} \\
\text{bgt } r_2, r_4, \text{calc}_1 & \text{bgt } r_2, r_4, \text{calc}_1 \\
\{\{\text{True}, G_1\}\} & \text{jd } \text{loop}_1 \\
\text{yield} & \{\{\text{True}, G_1\}\} \\
\text{jd } \text{loop}_1 & \text{jd } \text{loop}_1 \\
\text{done}_1 & \{\{M(a) = M(b), G_1\}\} \\
\text{yield} & \{\{M(a) = M(b), G_1\}\} \\
\text{jd } \text{done}_1 & \{\{M(a) = M(b), G_1\}\}
\end{align*}
\]

Figure 14. CCAP implementation for GCD algorithm.

The most interesting code block is that labeled \( \text{calc}_1 \). After the first yielding instruction, the comparison result of \( M(a) > M(b) \) can be used to simplify the next local guarantee to gcd. 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.

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-stickness 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 I8V1. Some simple temporal reasoning following I8V1 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 longstanding 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 described 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 “VC-Gen 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 VC-Gen 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

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 programs 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.

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.
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

\[ \Phi; (p_1 \ldots p_n, g) \vdash F \]

Well-formed code heap

\[ \Theta = (Inv, \Psi, A, G) \]

\[ \Psi = \{ f_j \leftarrow (p_j, g_j) \} \in \{1 \ldots m\} \]

\[ \Theta; (p_j, g_j) \vdash l_j \quad \forall j \in \{1 \ldots m\} \]

\[ \Theta; f_j \leftarrow l_j \in \{1 \ldots m\} \]

Well-formed instruction sequence

\[ \Theta; (p, g) \vdash \Pi \]

Lemma 5 (Progress) Let

\[ \Phi = (Inv, \Psi, A, G) \]

\[ \forall m. \forall r. (Inv \land p (M, R)) \supset ((R(r_d) + w) \in \text{dom}(M)) \]

\[ (Inv \land p S) \supset (Inv \land p' \text{Next}(c, S)) \]

\[ (Inv \land p S) \supset (g' \text{Next}(c, S)') \supset (g S' S') \]

\[ \Theta; (p, g') \vdash I \]

\[ \Theta; (p, g) \vdash \text{yield}; I \]

Lemma 6 (Preservation) Let

\[ \Phi = (Inv, \Psi, A, G) \]

\[ \forall n. \forall r. (Inv \land p (M, R)) \supset ((R(r_d) + w) \in \text{dom}(M)) \]

\[ (Inv \land p S) \supset (Inv \land p' \text{Next}(c, S)) \]

\[ (Inv \land p S) \supset (g' \text{Next}(c, S)') \supset (g S' S') \]

\[ \Theta; (p, g) \vdash c = \text{id} \]

\[ \Theta; (p, g) \vdash c; I \]