Hoare-Style Logic for Unstructured Programs*

Didrik Lundberg[1,2][0000−0001−9921−3257], Roberto Guanciale[1][0000−0002−8069−6495], Andreas Lindner[1][0000−0001−5311−1781], and Mads Dam[1][0000−0001−5432−6442]

[1] KTH Royal Institute of Technology, Lindstedtsvägen 5, 100 44 Stockholm, Sweden
{didrikl, robertog, andili, mfd}@kth.se
[2] Saab AB, Nettovägen 6, 175 41 Järfälla, Sweden

Abstract. Enabling Hoare-style reasoning for low-level code is attractive since it opens the way to regain structure and modularity in a domain where structure is essentially absent. The field, however, has not yet arrived at a fully satisfactory solution, in the sense of avoiding restrictions on control flow (important for compiler optimization), controlling access to intermediate program points (important for modularity), and supporting total correctness. Proposals in the literature support some of these properties, but a solution that meets them all is yet to be found. We introduce the novel Hoare-style program logic $\mathcal{L}_A$, which interprets postconditions relative to program points when these are first encountered. The logic can support both partial and total correctness, derive contracts for arbitrary control flow, and allows one to freely choose decomposition strategy during verification while avoiding step-indexed approximations and global invariants. The logic can be instantiated for a variety of concrete instruction set architectures and intermediate languages. The rules of $\mathcal{L}_A$ have been verified in the interactive theorem prover HOL4 and integrated with the toolbox HolBA for semi-automated program verification, making it applicable to the ARMv6 and ARMv8 instruction sets.

Keywords: Program Logics · Formal Verification · Theorem Proving · Binary Analysis · Hoare Logic

1 Introduction

Many scenarios require verification of machine code: in microkernels [34,19], the manipulation of processor contexts and hardware configurations have side effects that are not captured by the semantics of high-level languages; in cryptographic routines, resilience to side-channel attacks may require to analyse the exact sequence of memory accesses performed by software [8]; in critical software components, to minimise the trusted computing base the compiler can be

* This work has been supported by the TrustFull project financed by the Swedish Foundation for Strategic Research and the KTH CERCES Center for Resilient Critical Infrastructures financed by the Swedish Civil Contingencies Agency.

designated as untrusted; in software fault isolation techniques [59], it may be required to analyse the binary of closed source software.

One of the main difficulties in verifying machine code is the unstructured and dynamic control flow. This makes it difficult, or in the case of highly optimized code impossible, to adapt logics for high-level languages by mapping binary fragments to high-level statements: the code may be re-ordered by compilation, and one high-level statements may be implemented by multiple overlapping fragments and share fragments with other statements. A number of authors have explored the possibility of regaining Hoare-style reasoning also for the unstructured case [18,55,35,2,21,10,51,53,28,44]. However, we argue that there is still room for progress. In order to provide the formal basis for a binary verification toolkit, it is desirable that a Hoare-style logic for unstructured programs has the following properties:

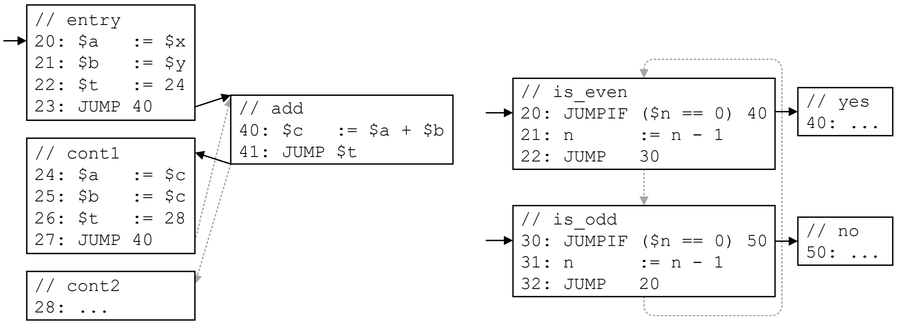
1. ability to express and verify both partial and total correctness;
2. support for verification of programs with arbitrary unstructured control flow, including irreducible loops, i.e., loops with multiple entry points;
3. support for verification of programs with dynamic control flow, e.g., function abstraction and exception handling;
4. freedom to decompose the verification using several strategies, e.g., splitting the program into two fragments which may overlap;

We motivate these requirements in Section 2 via two simple but illustrative examples. Our main contribution is the novel logic $\mathcal{L}_A$ that meets these requirements, which is presented in Section 3. In order to provide a general verification framework that supports programs in arbitrary machine and intermediate languages, the logic abstracts from the axiomatization of primitive state transitions. We show that the logic is sound and complete, and we present a definitional extension that simplifies the verification of binary programs. The logical framework is demonstrated in Section 4, where we verify the two running examples. Finally, $\mathcal{L}_A$ has been formalised in the HOL4 interactive theorem prover, integrated into the HOL4 binary analysis framework HolBA [38][3], and instantiated for two machine models: the ARMv8 ISA and the HolBA intermediate language, which is called BIR. This is described in Section 5. In Section 6, $\mathcal{L}_A$ is compared with related work, in particular, it is argued how existing solutions do not meet all of the requirements listed above. The paper ends with some concluding remarks in Section 7.

2 Two Motivating Examples

We introduce two small programs and their intended properties. Despite their size, the programs present some important challenges for binary verification. They are expressed in an assembly-like pseudolanguage, which represents an abstract unstructured programming language.

[3] The HolBA Github repository is located at `https://github.com/kth-step/HolBA` and our $\mathcal{L}_A$ implementation for this paper is available at the commit tagged `SEFM2020` in the directory `src/theory/abstract_hoare_logic`.



```
// entry
20: $a   := $x
21: $b   := $y
22: $t   := 24
23: JUMP 40

// cont1
24: $a   := $c
25: $b   := $c
26: $t   := 28
27: JUMP 40

// cont2
28: ...

// add
40: $c   := $a + $b
41: JUMP $t
```

(a) function reuse

```
// is_even
20: JUMPIF ($n == 0) 40
21: n      := n - 1
22: JUMP   30

// is_odd
30: JUMPIF ($n == 0) 50
31: n      := n - 1
32: JUMP   20

// yes
40: ...

// no
50: ...
```

(b) optimized mutual recursion

Fig. 1. Example code and control flow graph.

2.1 Example: Function Reuse

The first program consists of a function to add two integers as well as a main program that calls this function twice. Fig. 1a presents the program code with its static control flow graph using program fragments as nodes. Each fragment consists of multiple statements with unique address labels. For example, the statement `40: $c := $a + $b` is located at address `40` and is part of the fragment representing the function `add`. It evaluates `$a + $b`, assigns the result to the variable `$c`, and gives control to the next statement. In this case, the next statement is the indirect jump to the value in the variable `$t`, representing the return from `add`.

The main program consists of three fragments and takes the two parameters `$x` and `$y`. It computes $2 * (\texttt{\$x} + \texttt{\$y})$ by calling `add` twice and assigning the result to `$c`. The fragments for `entry` and `cont1` prepare the parameters of the function, as well as the return address, and call `add` using direct jumps. The ellipses in fragment `cont2` represent the code that follows afterwards.

The program satisfies the following contract:

$$[\texttt{\$x} = v_1 \wedge \texttt{\$y} = v_2]\; \texttt{20} \to \{\texttt{28}\}\; [\texttt{\$c} = 2 * (v_1 + v_2)] \tag{1}$$

It states that whenever an execution reaches the entry point at address `20` and the precondition $\texttt{\$x} = v_1 \wedge \texttt{\$y} = v_2$ holds, then, execution reaches the exit point at address `28` and there the desired postcondition $\texttt{\$c} = 2 * (v_1 + v_2)$ holds.

The control flow edges for the first and second calls are represented by solid black and dashed grey arrows, respectively. Notice that the static control flow contains two edges from `add` to the two return targets, while the dynamic control flow only uses one return edge per call context, i.e., the first call always returns to `cont1` and the second call returns to `cont2`.

This program illustrates how reuse patterns complicate the treatment of control flow in unstructured languages.

Apparent loops. The static control flow of the example contains a loop: the entry of `add` at the address `40` can reach the exit of `add`; this exit has a control flow edge to 24; the fragment from 24 can jump back to the entry of add at 40. This is caused by the lack of function abstraction in unstructured languages, which requires a form of reconstruction of call contexts for the control flow. Verifying this part of the program as a loop would involve the introduction of an invariant, which is undesirable.

Individual fragments. It is usually desirable to verify the function add independently of the rest of the program. The contract for this function has to capture the initial values of $a, $b and $t when starting execution from address 40:

$$[\mathtt{\$a} = v_1 \land \mathtt{\$b} = v_2 \land \mathtt{\$t} = v_3]\, \mathtt{40} \to \{v_3\}\, [\mathtt{\$c} = v_1 + v_2] \tag{2}$$

The initial value of $t represents the return address of the function. In the state where execution reaches this address, the variable $c must be the sum of the initial parameter values. Notice how the precondition ties the return address to the exit point of the contract with the free variable v_3. This allows expressing properties of reusable fragments in terms of generic contracts that can be instantiated per call.

Overlapping fragments. A natural strategy to verify the whole program is to sequentially compose the following two contracts, which capture the two steps of the main program:

$$[\mathtt{\$x} = v_1 \land \mathtt{\$y} = v_2]\, \mathtt{20} \to \{\mathtt{24}\}\, [\mathtt{\$c} = v_1 + v_2]$$

$$[\mathtt{\$c} = v_1]\, \mathtt{24} \to \{\mathtt{28}\}\, [\mathtt{\$c} = 2 * v_1]$$

The two contracts concern overlapping fragments of the program because the fragment of each contract contains one invocation of the function add: the invocation of the first contract ends in 24, and the other one ends in 28.

2.2 Example: Optimized Mutual Recursion

Fig. 1b presents a program that uses mutual recursion between two functions to determine the parity of a given integer. The function is_even ends in address 40 if the input is even, otherwise in 50. Likewise, is_odd ends in 40 if the input is odd, otherwise in 50. The example does not use a stack, which could be the result of an optimized compilation.

Upon entry to either of the two functions, the control flow has a loop (whose control flow edges are represented by dashed grey arrows) with two exit points yes and no. Considering the entry point is_even, the program meets the following contract:

$$[\mathtt{\$n} = v_1]\, \mathtt{20} \to \{\mathtt{40}, \mathtt{50}\}\, [(\mathtt{\$pc} = \mathtt{40} \land v_1\%2 = 0) \lor (\mathtt{\$pc} = \mathtt{50} \land v_1\%2 = 1)] \tag{3}$$

Here $pc represents the program counter, and it is used in the postcondition to specify that the exit point reached depends on the input parity. For example, execution reaches 40 and not 50 if the initial value of $n modulo 2 is 0. The program satisfies an analogous contract for entry point is_odd.

Multiple exit points. The given example consists of two fragments that, both individually and combined, have two exit points each. To capture this specific case as well as arbitrary branch structures in a natural way, it is necessary to express and compose contracts with multiple exit points.

Irreducible loops. The loop of the example is irreducible because it has multiple entry points, which is not uncommon for optimized code. For this reason, verification of unstructured programs requires the ability to deal with these types of loops.

3 The Program Logic $\mathcal{L}_A$

We assume a machine model consisting of a deterministic transition system. Let Σ be the set of machine states ranged over by s. The execution of a single machine instruction is modeled by the partial function $\mathbf{nxt} : \Sigma \hookrightarrow \Sigma$ with $\mathbf{nxt}^n$ as its nth iteration. The partiality of the transition relation allows one to model failing executions. We assume a function $\mathbf{lbl} : \Sigma \to \Lambda$ from states to a set of control states Λ. This function can be used to retrieve the label or address of the next instruction executed from a state and can be thought of as accessing the program counter. The generality of **lbl** allows it to also include stack pointers and other parts of concrete machine states.

We use the notion of entry/exit points, or labels, to identify program fragments. The weak transition relation $\mathbf{weak}(s, L, s')$ relates an initial state s to the final state s' that is reached when executing the fragment whose entry point is $\mathbf{lbl}(s)$ and exit points are L.

Definition 1 (Weak transition relation).

$$\begin{aligned}\mathbf{weak}(s, L, s') = {} & \exists n.\, n > 0 \,\wedge\, \mathbf{nxt}^n(s) = s' \,\wedge\, \mathbf{lbl}(s') \in L \,\wedge \\ & \forall n' : 0 < n' < n.\ \mathbf{lbl}(\mathbf{nxt}^{n'}(s)) \notin L\end{aligned}$$

The weak transition relation is deterministic, partial (since a program may never reach L from s), and guarantees that no intermediate state has **lbl** in L. That is, when $\mathbf{weak}(s, L, s')$ then s' represents the first encounter of a state with label in L after s.

The Hoare-style judgment of $\mathcal{L}_A$, $[P]\, l \to L\, [Q]$, states total correctness in terms of pre- and postconditions P and Q, entry point l, and set of possible exit points L. In the following, we abstract from the assertion language used for pre- and postconditions.

Definition 2 (Judgment of $\mathcal{L}_A$). *The judgment* $[P]\, l \to L\, [Q]$ *is valid iff*

$$\forall s.\, \mathbf{lbl}(s) = l \,\wedge\, P(s) \implies \exists s'.\, \mathbf{weak}(s, L, s') \,\wedge\, Q(s')\,.$$

Notice that due to Definition 1, we call this a *first-encounter judgment.* A partial-correctness version can be obtained by exchanging the conjunction in the conclusion to an implication. In the following, we use $\mathbf{lbl} = l$ and $\mathbf{lbl} \in L$ for the

$$\frac{[P \wedge C]\, l \to L\, [Q] \qquad [P \wedge \neg C]\, l \to L\, [Q]}{[P]\, l \to L\, [Q]}\ \textsc{Case}$$

$$\begin{pmatrix} \vDash (\mathbf{lbl} = l) \wedge P_2 \implies P_1 \\ \vDash (\mathbf{lbl} \in L) \wedge Q_1 \implies Q_2 \end{pmatrix} \frac{[P_1]\, l \to L\, [Q_1]}{[P_2]\, l \to L\, [Q_2]}\ \textsc{Conseq}$$

$$\frac{[P]\, l \to L_1 \cup L_2\, [Q] \qquad [Q]\, L_1 \to L_2\, [Q]}{[P]\, l \to L_2\, [Q]}\ \textsc{Seq}$$

$$(\vDash l \notin L)\ \frac{\begin{array}{c}[I \wedge C \wedge V = x]\, l \to \{l\} \cup L\, [\mathbf{lbl} = l \wedge I \wedge V < x] \\ [I \wedge \neg C]\, l \to L\, [Q]\end{array}}{[I]\, l \to L\, [Q]}\ \textsc{Loop}$$

Fig. 2. Inference rules for $\mathcal{L}_A$

predicates that constrain the label of a state. Since commonly a program fragment must guarantee different properties for different exit points, we use the notation $l \mapsto Q$ for $(\mathbf{lbl} = l) \wedge Q$. For instance, the following contract describes the effects of the first statement of the program in Fig. 1a:

$$C_1 = [\top]\, \mathtt{20} \to \{\mathtt{21}, \mathtt{22}\}\, [\mathtt{21} \mapsto \mathtt{\$x} = \mathtt{\$a}] \tag{4}$$

Notice that the postcondition of C_1 is equivalent to $\mathbf{lbl} = \mathtt{21} \wedge \mathtt{\$x} = \mathtt{\$a}$, therefore the contract implicitly guarantees that execution starting from 20 reaches the address 21 without encountering the address 22.

Since unstructured code can have multiple entry points, many program logics feature multiple-entry judgments [10,51,53,44,26]. These are actually equivalent to conjuncts of multiple judgments. In a slight abuse of notation, we use $[P]\, L_1 \to L_2\, [Q]$ to refer to the set of all $[P]\, l \to L_2\, [Q]$ such that $l \in L_1$, interpreted conjunctively. Multiple-exit judgments cannot be so reduced. Consider the example of a conditional jump with targets l_1 and l_2: if it is required that the contract states that l_1 is not visited before l_2, this cannot be phrased using single-exit judgments only.

3.1 Inference Rules

Fig. 2 shows the inference rules of $\mathcal{L}_A$. Note that there are no rules for primitive transitions (execution of only one transition) - these are added when the inference system is instantiated for a specific ISA (see Section 5), for example, contract C_1 of Eq. 4 is obtained in this fashion.

The Case rule allows for combining judgments on different paths of execution (split by a condition C on the initial state) that both end up in states with program counters in L. This is useful when verifying branching structures.

The Conseq rule can strengthen the precondition and weaken the postcondition. Note that in contrast to the standard Consequence rule [31], Conseq only requires that the implications hold for states in the entry and exit points (l and in L, respectively). For instance, contract C_1 of Eq. 4 can be weakened to $[\top]\, \mathtt{20} \to \{\mathtt{21}, \mathtt{22}\}\, [(\mathtt{21} \mapsto \mathtt{\$x} = \mathtt{\$a}) \vee (\mathtt{23} \mapsto \top)]$ (since $\mathtt{23}$ is not in the exit label set).

The Seq rule allows for sequential composition of two fragments, with the label set L_1 designating the midpoints. As an example, consider the two first statements of the `entry` function in Fig. 1a. The contract C_1 of Eq. 4 holds for the first statement. Similarly for the second statement, the contract

$$C_2 = [\mathtt{21} \mapsto \mathtt{\$x} = \mathtt{\$a}]\, \mathtt{21} \to \{\mathtt{22}\}\, [\mathtt{22} \mapsto \mathtt{\$x} = \mathtt{\$a} \wedge \mathtt{\$y} = \mathtt{\$b}]$$

holds. In order to sequentially compose these two contracts, the precondition of C_2 and the postcondition of both C_1 and C_2 has to be identical. Just like for the composition rule in Hoare logic, it is required the same predicate on the point(s) of composition holds. Let Q be $(\mathtt{21} \mapsto \mathtt{\$x} = \mathtt{\$a}) \vee (\mathtt{22} \mapsto \mathtt{\$x} = \mathtt{\$a} \wedge \mathtt{\$y} = \mathtt{\$b})$. Since $\mathtt{21} \mapsto \mathtt{\$x} = \mathtt{\$a}$ implies Q, the contract C_1 can be weakened to $[\top]\, \mathtt{20} \to \{\mathtt{21}, \mathtt{22}\}\, [Q]$. In fact, since C_1 guarantees that $\mathtt{22}$ is not encountered between $\mathtt{20}$ and $\mathtt{21}$, we can enforce any property in the (impossible) case of ending the fragment in $\mathtt{22}$. Also, since $\mathtt{22}$ is not the entry point and $\mathtt{21}$ is not an exit point of C_2, the Conseq rule can be used to weaken C_2 to $[Q]\, \mathtt{21} \to \{\mathtt{22}\}\, [Q]$. This enables to use Seq rule to infer $[\top]\, \mathtt{20} \to \{\mathtt{22}\}\, [Q]$ and Conseq rule to obtain $[\top]\, \mathtt{20} \to \{\mathtt{22}\}\, [\mathtt{\$x} = \mathtt{\$a} \wedge \mathtt{\$y} = \mathtt{\$b}]$. Note the shape of the premises of Seq, which are due to unstructured control flows: the fragment starting from l may reach the endpoints L_2 without encountering the midpoints L_1, for instance if the first fragment contains the compilation of a `break` statement. For this reason the rule requires that the first fragment directly establishes Q if it reaches the endpoints L_2 before L_1.

The premises of Loop are contracts for the loop body and loop exit. For the loop body, the invariant I and the condition C entail that execution starting from l does not reach any of the exit points in L before returning to l, preserves the invariant, and strictly decreases the variant. For the loop exit, the invariant I and the negation of the condition C guarantee that execution starting from l reaches the exit points L and establishes Q. The side condition $l \notin L$ ensures that execution with entry point l and exit points L establishes Q on the first encounter of L also in the case when C holds. Also, notice that if $l \in L$ then the fragment associated with $[I]\, l \to L\, [Q]$ corresponds to the loop body, since the weak transition does not loop through l.

The version for partial correctness simply disregards the variant. Note that unlike similar rules for structured loops [39], the Loop rule must take into account possible side effects of exiting the loop as well as multiple exit points. For this reason, the postcondition in the conclusion is Q and not $I \wedge \neg C$. Multiple loop entry points can also be handled by formulating one invariant and variant per entry point, allowing for analysis of irreducible loops. Moreover, the condition C is not syntactically extracted from the program.

The following rule is not necessary for completeness, but is used to unify contracts stating different properties on the final states:

$$\frac{[P]\, l \to L\, [Q_1] \qquad [P]\, l \to L\, [Q_2]}{[P]\, l \to L\, [Q_1 \wedge Q_2]}\ \textsc{Conj}$$

Theorem 1 (Soundness). *If $[P]\, l \to L\, [Q]$ can be derived from valid assumptions using the inference rules of $\mathcal{L}_A$, then $[P]\, l \to L\, [Q]$ is valid.*

Proof. By structural induction over the individual rules.

3.2 Completeness of $\mathcal{L}_A$

Note that since $\mathcal{L}_A$ is agnostic with regard to the concepts of programs, statements and instructions, the completeness theorem is formulated relative to a sound and complete oracle for primitive transitions.

Theorem 2 (Completeness of $\mathcal{L}_A$). *Given that the logic used for stating pre- and postconditions is sufficiently expressive to state invariants, variants, weakest preconditions and strongest postconditions, and that there exists a sound and complete oracle for contracts of primitive transitions, the first-order theory of the underlying program and the theory of well-founded orders, then $\vdash [P]\, l \to L\, [Q]$ if $[P]\, l \to L\, [Q]$ is valid.*

Proof. To prove completeness of $\mathcal{L}_A$, the following definition is introduced for the control-flow graph of $[P]\, l \to L\, [Q]$:

$$\mathit{CFG}(P, l, L) = \left\{ (\mathbf{lbl}(\mathbf{nxt}^{n}(s)), \mathbf{lbl}(\mathbf{nxt}^{n+1}(s))) \;\middle|\; \begin{array}{l} P(s) \wedge \mathbf{lbl}(s) = l \wedge n \geq 0 \wedge \\ \forall 0 < n' \leq n.\, \mathbf{lbl}(\mathbf{nxt}^{n'}(s)) \notin L \end{array} \right\}$$

The edges of the control-flow graph $\mathit{CFG}(P, l, L)$ are the possible transitions starting from the state with label l for which the precondition P holds, up to and including edges into the exit label set L.

The proof is then by induction over set inclusion on $\mathit{CFG}(P, l, L)$.

For the base case, only one edge exists. This means that every state that satisfies P and whose **lbl** is l immediately reaches L in one transition. The contract describing this is derivable directly from contracts for the primitive transitions.

For the inductive case, the induction hypothesis is that if $\mathit{CFG}(P', l', L')$ is a strict subgraph of $\mathit{CFG}(P, l, L)$, then $[P']\, l' \to L'\, [Q'] \implies \vdash [P']\, l' \to L'\, [Q']$.

First, consider the case where all edges in $\mathit{CFG}(P, l, L)$ go directly from l to L. Then, $\vdash [P]\, l \to L\, [Q]$ follows directly from the rules for primitive transitions and the Case rule.In the case where not all edges go directly to L from l, consider the case for which the labels of $CFG(P, l, L) \setminus (\Lambda \times L)$ can be partitioned into *sequential sets* L_a, L_b such that

$$(CFG(P, l, L) \setminus (\Lambda \times L)) \cap (L_b \times L_a) = \emptyset\,.$$

Let C_b be the predicate on states in l identifying all states from which execution reaches L_b before L (this is provided by the oracle in the assumptions of the theorem). Furthermore, let L_{mid} be the subset of L_b which has incoming edges from L_a. Then, $\vdash [P \wedge C_b]\, l \to L_{mid} \cup L\, [R_1]$ follows from the inductive hypothesis for some strongest postcondition R_1 provided by the oracle in the assumptions (note that this contract will not visit L by definition of C_b). $\vdash [R_2]\, L_{mid} \to L\, [Q]$ follows for similar reasons, where R_2 is some weakest precondition provided by the oracle. After unifying the midcondition using the Conseq rule (by precondition strengthening), then the two contracts can be composed using the Seq rule to obtain $\vdash [P \wedge C_a]\, l \to L\, [Q]$. If $C_b = \top$, then this derives the complete contract $[P]\, l \to L\, [Q]$. If not, then $[P \wedge \neg C_b]\, l \to L\, [Q]$ must be valid, and derivable since it is a strict subgraph of $CFG(P, l, L)$. $[P]\, l \to L\, [Q]$ is then derivable by the Case rule.

In case the partition into sequential sets is not possible, then l is in the transitive closure for every label except those in L , and since no edges go directly from l to L, then $l \notin L$. Let C_l be the predicate on states in l identifying all states such that execution from l goes back to l before reaching L:

$$C_l = \left\{ \begin{array}{l} s.\, I(s) \wedge \exists n.\, n > 0 \wedge \mathbf{nxt}^n(s) = s' \wedge \mathbf{lbl}(s) = l \wedge \\ \forall n'.n' < n \wedge \mathbf{nxt}^{n'}(s) = s' \implies s' \notin L \end{array} \right\}$$

I is the predicate identifying all possible states in l:

$$I = \left\{ \begin{array}{l} s'.\, \exists s, n.\, P(s) \wedge \mathbf{lbl}(s) = l \wedge n \geq 0 \wedge \\ \quad \mathbf{nxt}^n(s) = s' \wedge \mathbf{lbl}(s') = l \wedge \forall n'.\, n' < n \\ \quad \mathbf{nxt}^{n'}(s) = s'' \implies s'' \notin L \end{array} \right\}$$

and V is a loop variant (which must exist, due to the total-correctness judgment in the antecedent of the proof obligation). Then the contract for the loop body $[I \wedge C_l \wedge V = x]\, l \to \{l\} \cup L\, [\mathbf{lbl} = l \wedge I \wedge V < x]$ must be valid, and derivable since it lacks any edge to L. Also, $[I \wedge \neg C_l]\, l \to L\, [Q]$ will be valid by definition of C_l, and lack all edges going back to l, and accordingly be derivable. Since $P \implies I$ (the invariant also holds by definition in the very initial state) then the two contracts can be used with the Loop rule to obtain $[P]\, l \to L\, [Q]$, which was to be proved, completing the proof.

3.3 $\mathcal{L}_{\mathbf{AS}}$ - Definitional extension of $\mathcal{L}_A$

There are some common strategies that simplify verification via $\mathcal{L}_A$. First, for sequential composition it is useful to clearly identify which labels in the exit label set are not encountered by execution in the first contract, since it is required to prove the exit label set of the first contract must include the exit points of the second contract. Secondly, while analysing concrete binary code it is usually necessary to demonstrate preservation of invariants, such as the code being in memory, the stack pointer being outside the code memory, and so on.

For these reasons we introduce a definitional extension of $\mathcal{L}_A$. The judgment for $\mathcal{L}_{\text{AS}}$ $[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]$ adds an invariant I, which must hold in the initial and final states of execution, and two disjoint exit label sets L_W and L_B. The sets L_W and L_B are referred to as *whitelist* and *blacklist* respectively: the labels in L_B must not be encountered before reaching exit points in L_W.

Definition 3 (Judgment of $\mathcal{L}_{\text{AS}}$). *Given that the* whitelist *and* blacklist *satisfy $L_W \cap L_B = \emptyset$ and $L_W \neq \emptyset$, the judgment $[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]$ is defined as $[P \wedge I]\, l \rightarrow L_W \cup L_B\, [(\mathbf{lbl} \notin L_B) \wedge Q \wedge I]$.*

The set of rules for $\mathcal{L}_{\text{AS}}$ are derived from $\mathcal{L}_A$ and for brevity we just introduce them with short comments. It is possible to weaken a contract by freely dropping labels from the blacklist or moving them to the whitelist:

$$\frac{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]}{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \setminus L \rangle\, [Q]}\ \text{BL-S}\textsc{ubset}$$

$$(\vDash L \subseteq L_B)\ \frac{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]}{[P]\, l \xrightarrow{I} \langle L_W \cup L \mid L_B \setminus L \rangle\, [Q]}\ \text{BL-}\textsc{to}\text{-WL}$$

On the contrary, to move a label from the whitelist to the blacklist, the postcondition must entail that the label is not encountered:

$$\begin{pmatrix} \vDash L \subset L_W \\ \vDash Q \implies \mathbf{lbl} \notin L \end{pmatrix} \frac{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]}{[P]\, l \xrightarrow{I} \langle L_W \setminus L \mid L_B \cup L \rangle\, [Q]}\ \text{WL-}\textsc{to}\text{-BL}$$

Finally, a simplified and more conventional sequential composition is possible when (i) the whitelist of the second fragment is included in this blacklist of the first one and (ii) the midpoints L_W do not overlap with the final endpoints L'_W:

$$\begin{pmatrix} \vDash L'_W \subseteq L_B \\ \vDash L_W \cap L'_W = \emptyset \end{pmatrix} \frac{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [R] \qquad [R]\, L_W \xrightarrow{I} \langle L'_W \mid L'_B \rangle\, [Q]}{[P]\, l \xrightarrow{I} \langle L'_W \mid L_B \cap L'_B \rangle\, [Q]}\ \text{S-S}\textsc{eq}$$

In contrast to $\mathcal{L}_A$, the consequence rule has been split up into the two separate rules to remove the need for unnecessary computation in the implementation of proof procedures. The rule Pre-Str is for precondition strengthening and Post-Weak for postcondition weakening.

$$(\vDash (\mathbf{lbl} = l) \wedge P_2 \implies P_1)\ \frac{[P_1]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]}{[P_2]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q]}\ \text{P}\textsc{re}\text{-S}\textsc{tr}$$

Abbr.	Formula
$P_{\texttt{add}}$	$\texttt{40} \mapsto \texttt{\$a} = v_1 \wedge \texttt{\$b} = v_2 \wedge \texttt{\$t} = v_3$
$R_{\texttt{add}}$	$\texttt{41} \mapsto \texttt{\$c} = v_1 + v_2 \wedge \texttt{\$t} = v_3$
$Q_{\texttt{add}}$	$v_3 \mapsto \texttt{\$c} = v_1 + v_2$
$P_{\texttt{main\_1}}$	$\texttt{20} \mapsto \texttt{\$x} = x \wedge \texttt{\$y} = y$
$R_{\texttt{main\_1}}$	$\texttt{40} \mapsto \texttt{\$a} = x \wedge \texttt{\$b} = y \wedge \texttt{\$t} = \texttt{24}$
$S_{\texttt{main\_1}}$	$\texttt{24} \mapsto \texttt{\$c} = x + y$
$P_{\texttt{main\_2}}$	$S_{\texttt{main\_1}}$
$R_{\texttt{main\_2}}$	$\texttt{40} \mapsto \texttt{\$a} = \texttt{\$b} = x + y \wedge \texttt{\$t} = \texttt{28}$
$S_{\texttt{main\_2}}$	$\texttt{28} \mapsto \texttt{\$c} = (x + y) + (x + y)$
$Q_{\texttt{main}}$	$\texttt{28} \mapsto \texttt{\$c} = 2 * (x + y)$

Table 1. Function reuse formulas.

Abbr.	Formula
`cond`	$1 < \texttt{\$n}$
`var`	$\texttt{\$n}$
`inv`	$v_1\%2 = \texttt{\$n}\%2$
$P_{\texttt{loop\_b}}$	$\texttt{inv} \wedge \texttt{cond} \wedge \texttt{var} = v_2$
$Q_{\texttt{loop\_b}}$	$\texttt{inv} \wedge \texttt{var} < v_2$
$P_{\texttt{loop\_e}}$	$\texttt{inv} \wedge \neg\texttt{cond}$
$Q_{\texttt{loop\_e}}$	$(\texttt{40} \mapsto v_1\%2{=}0) \vee (\texttt{50} \mapsto v_1\%2{=}1)$
$R_{\texttt{loop}}$	$\texttt{inv}$
$Q_{\texttt{loop}}$	$Q_{\texttt{loop\_e}}$
$P_{\texttt{loop}}$	$\texttt{\$n} = v_1$

Table 2. Mutual recursion formulas.

$$(\vDash (\mathbf{lbl} \in L_W) \wedge Q_1 \implies Q_2)\ \frac{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q_1]}{[P]\, l \xrightarrow{I} \langle L_W \mid L_B \rangle\, [Q_2]}\ \textsc{Post-Weak}$$

The loop rule S-Loop is very similar to Loop of $\mathcal{L}_A$. Here the loop invariant uses the place of the $\mathcal{L}_{\mathrm{AS}}$ invariant, and the side condition accounts for the split of the end labels into whitelist and blacklist.

$$\begin{pmatrix} \vDash l \notin L_W \\ \vDash l \notin L_B \end{pmatrix} \ \frac{\begin{array}{c} [C \wedge V = x]\, l \xrightarrow{I} \langle \{l\} \cup L_W \mid L_B \rangle\, [\mathbf{lbl} = l \wedge V < x] \\ [\neg C \wedge I]\, l \xrightarrow{\top} \langle L_W \mid L_B \rangle\, [Q] \end{array}}{[I]\, l \xrightarrow{\top} \langle L_W \mid L_B \rangle\, [Q]}\ \textsc{S-Loop}$$

4 Verification of the examples

To exemplify the usage of our logic, we verify the programs from Section 2.

4.1 Function Reuse

We first establish a generic contract for the function `add` that is then instantiated for the two function invocations. Fig. 3 visualizes the verification flow and Table 1 collects the predicates for this example. In the following, we simply omit the blacklists if they are empty. The contract **add generic** corresponds to Eq. 2 and it is generic in terms of the function arguments v_1 and v_2 as well as the return label v_3. For simplicity, we set the blacklist as the set $\{\texttt{20} \ldots \texttt{28}\} \setminus \{v_3\}$, which are all the addresses of the example outside of `add`, except the return address. In a more general use case, the blacklist is a variable set, where the return label v_3 and all function labels except the entry cannot be included.

The contract **add return** refers to a single primitive transition; hence it is not derived from the inference rules of $\mathcal{L}_A$. Instead, it is established using the semantics of the single indirect jump at program address `41`, which guarantees $[R_{\texttt{add}}]\, \texttt{41} \rightarrow \langle \{\texttt{20} \ldots \texttt{28}, v_3\} \mid \rangle\, [Q_{\texttt{add}}]$. This contract is weakened to **add**

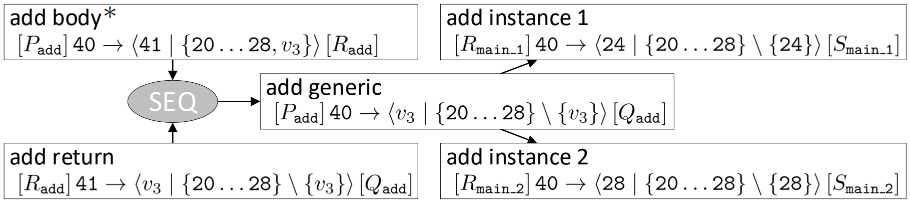

add body*
$[P_{\mathtt{add}}]\,40 \to \langle 41 \mid \{20 \ldots 28, v_3\}\rangle\,[R_{\mathtt{add}}]$
SEQ
add return
$[R_{\mathtt{add}}]\,41 \to \langle v_3 \mid \{20 \ldots 28\} \setminus \{v_3\}\rangle\,[Q_{\mathtt{add}}]$
add generic
$[P_{\mathtt{add}}]\,40 \to \langle v_3 \mid \{20 \ldots 28\} \setminus \{v_3\}\rangle\,[Q_{\mathtt{add}}]$
add instance 1
$[R_{\mathtt{main\_1}}]\,40 \to \langle 24 \mid \{20 \ldots 28\} \setminus \{24\}\rangle\,[S_{\mathtt{main\_1}}]$
add instance 2
$[R_{\mathtt{main\_2}}]\,40 \to \langle 28 \mid \{20 \ldots 28\} \setminus \{28\}\rangle\,[S_{\mathtt{main\_2}}]$

Fig. 3. Verification workflow for function add with the state predicates from Table 1.

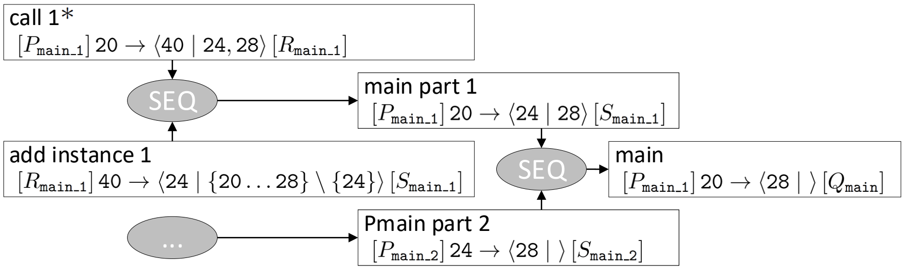

call 1*
$[P_{\mathtt{main\_1}}]\,20 \to \langle 40 \mid 24, 28\rangle\,[R_{\mathtt{main\_1}}]$
SEQ
add instance 1
$[R_{\mathtt{main\_1}}]\,40 \to \langle 24 \mid \{20 \ldots 28\} \setminus \{24\}\rangle\,[S_{\mathtt{main\_1}}]$
main part 1
$[P_{\mathtt{main\_1}}]\,20 \to \langle 24 \mid 28\rangle\,[S_{\mathtt{main\_1}}]$
SEQ
main
$[P_{\mathtt{main\_1}}]\,20 \to \langle 28 \mid \rangle\,[Q_{\mathtt{main}}]$
...
Pmain part 2
$[P_{\mathtt{main\_2}}]\,24 \to \langle 28 \mid \rangle\,[S_{\mathtt{main\_2}}]$

Fig. 4. Verification workflow for main program with the state predicates from Table 1.

return with the rule WL-TO-BL. Similarly, **add body** is derived from the semantics of the single assignment at program address 40, which guarantees $[P_{\mathtt{add}}]\,40 \to \langle\{20, \ldots, 28, 41, v_3\} \mid \rangle\,[R_{\mathtt{add}}]$. Here, we need to impose the side condition $v_3 \neq 41$ in order to use the rule WL-TO-BL to obtain **add body**. The label v_3 is included into the blacklist to allow subsequent sequential composition with **add return** to **add body**.

This contract is instantiated twice for the two function invocations, where concrete values are substituted for the return address v_3 and the function arguments v_1 and v_2. For the first function call (i.e. **add instance 1**) $v_1 = x$, $v_2 = y$, $v_3 = 24$ and for the second function call (i.e. **add instance 2**) $v_1 = v_2 = x + y$, $v_3 = 28$. Fig. 4 shows the flow to verify the main program. For brevity, it omits the source contracts of the second function invocation. The verification consists of several sequential compositions, where the whitelists of subsequent contracts have to be included in blacklists of preceding contracts. For example, **main part 1** has 28 in the blacklist so that we can compose it with **main part 2**, where 28 is in the whitelist. Finally, we use weakening to obtain the overall contract **main**, which corresponds to Eq. 1.

4.2 Mutual Recursion

The verification steps of the two recursion entry points is_even and is_odd are virtually equivalent. Hence, we only present the steps for is_even as shown in Fig. 5. Because the control flow is a loop, we start by identifying the condition to stay in the loop cond as well as loop variant var and loop invariant inv, which are

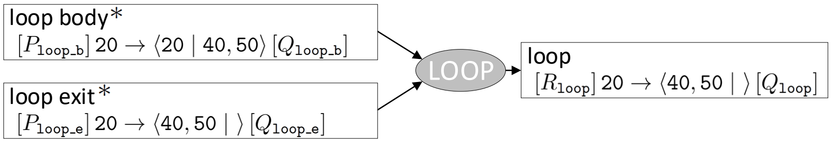

loop body*
$[P_{\texttt{loop\_b}}]\ 20 \to \langle 20 \mid 40, 50 \rangle\ [Q_{\texttt{loop\_b}}]$
loop exit*
$[P_{\texttt{loop\_e}}]\ 20 \to \langle 40, 50 \mid \ \rangle\ [Q_{\texttt{loop\_e}}]$
LOOP
loop
$[R_{\texttt{loop}}]\ 20 \to \langle 40, 50 \mid \ \rangle\ [Q_{\texttt{loop}}]$

Fig. 5. Verification workflow for the recursive function `is_even` with the state predicates from Table 2.

defined together with the state predicates in Table 2. These predicates allow us to establish the contracts **loop body** and **loop exit** using sequential reasoning, as for the previous example. These contracts are then combined using the loop composition rule. Finally, precondition strengthening allows us to obtain the desired contract corresponding to Eq. 3 from **loop** because $P_{\texttt{loop}}$ implies $R_{\texttt{loop}}$.

5 Implementation

We implemented our logic and verified its soundness in HOL4. We also instantiated the logic for two transition systems: the formal model of ARMv8 and the machine-independent intermediate language, BIR, of the binary analysis framework HolBA [38].

HolBA supports the following verification workflow: first, the HolBA transpiler translates ARMv8 (among other ISAs) binary code to BIR together with a bisimulation proof. The bisimulation relation guarantees correspondence of ARMv8 state components with BIR variable assignments and the BIR program counter. Second, the BIR program can be verified to meet contracts - for these examples, this has been done using a weakest precondition generator and prover. Third, the contracts are transferred to the ARMv8 model using the bisimulation. Transpilation to and from BIR as well as proving contracts for acyclic program fragments is fully automated, meaning the verification code consists of specification as well as fitting together contracts for acyclic program fragments, where applying one rule typically takes one LOC. The integration of our logic in HolBA allows composing ARMv8 contracts either directly, or indirectly by composing BIR contracts and transferring the result to ARMv8.

The ARMv8 instantiation uses the L3-based machine model [29]. An ARMv8 machine state consists of registers, processor flags and memory. The program counter is one of the registers and indicates which memory location contains the next instruction to execute. Consequently, for this model, **lbl** retrieves the program counter from the state and Λ are all memory addresses. The state transition function `NextStateARM8` represents the execution of a single instruction and corresponds to **nxt**. Because the program is stored in memory, it may change by memory operations. The invariant of $\mathcal{L}_{\mathrm{AS}}$ can be used to fix the program under analysis and exclude self-modifying code, by requiring that the program binary is loaded in system memory.

Component	Total	Subcomponents
Logics	863	$\mathcal{L}_A$: 372, $\mathcal{L}_{\text{AS}}$: 491
Instantiations	1040	BIR: 974, ARMv8: 66
BIR composition tools	742	
Examples	1180	Function reuse: 428, Mutual recursion: 382, ARMv8: 370

Table 3. HOL4 code sizes in lines of code (LOC).

The BIR instantiation uses the model defined in HolBA. BIR is designed to have as few language primitives as possible, to simplify the construction of analysis tools. In fact, it is very similar to the pseudolanguage used in Section 2. A BIR state consists of a map of variable names to values, a program counter, and execution status flags to indicate whether the program is running or is in an exceptional state. The variable map is used to represent both register and memory assignments. In BIR, the program is not part of the state like in ARMv8 a program is in system memory. It can therefore change only due to an explicit jump statement. For this model, **lbl** extracts the program counter of the BIR state and **nxt** is the execution of the sequence of BIR statements that simulate a single machine instruction. The transition function **nxt** is only defined for transitions to non-exceptional states.

We chose to use boolean BIR expressions as state predicates in order to reuse the existing automation of HolBA. Because BIR expressions can only refer to variable assignments, but not to the program counter, this choice restricts the expressiveness of pre- and postconditions. For this reason, BIR judgments are defined as a specialisation of $\mathcal{L}_{\text{AS}}$ where the postconditions are maps from labels to BIR expressions following the syntax of $\mapsto$ from Section 3. We also proved in HOL4 a program composition rule for BIR, i.e., contracts of subprograms can be applied to larger programs.

We extended HolBA to support our specialized BIR judgments in the existing verification infrastructure. Specifically, we modified the weakest precondition tool of HolBA to enable automatic proofs of contracts for non-looping BIR statement sequences without indirect jumps by using the existing SMT solver integration and contract entailment. This allowed us, for example, to establish the contracts marked with $*$ in Figs. 3,4 and 5. We created a library to automate the application of composition rules and used it to verify the two example programs according to the workflows presented in Section 4. This was also used together with the HolBA transpiler to verify an ARMv8 program that has been compiled from C code.

Table 3 shows the code sizes (calculated using `cloc` [20]) for our verification of the logics, their instantiations, the supporting HOL4 tools to semi-automate contract composition, and the verification of the examples. Each example of this paper amounts to about 10 BIR statements and can be verified in less than 10 seconds on an Intel i7-4800MQ with our proof-of-concept implementations. The

Logic	Total correctness	Partial correctness	First encounter	Overlap. fragments	Completeness
Myreen, Gordon [44]	■	□	□	■	□
ISCAP [56]	□	■	Only function ret.	□	□
Saabas, Uustalu [51,7]	■	■	■	□	■
Tan, Appel [53]	□	■	■	■	■
Benton [10]	□	■	■	■	□

Table 4. Summary of features of existing logics for unstructured programs.

ARMv8 example consists of 48 BIR statements and takes less than 40 seconds for transpilation and verification.

6 Related Work

When Tony Hoare introduced the formal system that later became known as *Hoare logic* in 1969, he did not initially treat arbitrary jumps and noted that these likely present a problem with complex solutions [31]. The following years, several extensions of Hoare logic were proposed to deal with unstructured code [18,17,35,2] which were shown to be unsound [3,2,50].

In 1976, Wang [55] proposed a program logic for total correctness of unstructured Algol-like programs, introducing multi-exit postconditions with different postconditions depending on exit label. However, in this logic, the judgments do not guarantee that the postcondition is met at the initial encounter of an associated label, unless this happens to entail exiting the program segment. In 1981, de Bruin [21] introduced a logic for partial correctness of unstructured programs. This logic is dependent on a list of global label invariants, which must hold every time the corresponding label is reached. This could be cumbersome to handle during practical verification.

Years later, in the early 2000s, computational resources had made program verification possible on a larger scale, facilitating the extension of the scope of analysis from idealized high-level languages to machine code. In particular, typed assembly languages [41] and proof-carrying code [48,1] originated a renewal of interest in the logical foundations for reasoning about programs. Table 4 contains a summary of recently proposed unstructured program logics.

Starting in 2002, the FLINT project began working on the CAP family of unstructured languages and their program logics [57,28,49,56,15,16]. Various dialects of these logics have been formalized in Coq [23,22], instantiated for x86 and SPARCv8 ISAs [58], and used to verify simplified OS kernels [27]. The initial CAP logics are written in continuation-passing style (in order to support first-class code pointers easily) and incorporate separation logic. In 2011, they presented ISCAP, a direct-style logic that supports a partial-correctness judgment only for entire functions [56]. The CAP family does not include any logic for total correctness like $\mathcal{L}_A$.

In 2005, Benton [10] proposed a program logic for partial correctness with multiple entry points and multiple exit points. Benton formalised a version of his logic in Coq [11] and used it for proofs of type safety in a certified compiler between two languages [13,12]. Benton uses continuation-passing style reasoning with step-indexed approximations to provide the partial-correctness judgment. This has some limitations; for example, a label continuation can never be both in the pre- and postcondition. Also, the logic has global label invariants, similar to those of de Bruin.

Tan and Appel introduced a program logic for partial correctness similar to Benton's, basing it on continuation-passing style reasoning with step-indexed approximations [53]. This logic was used in the Foundational Proof-Carrying Code project for type-safety proofs of SPARC machine code. No total-correctness version of this logic is known to exist by the authors.

In 2006, Saabas and Uustalu [51] constructed a program logic for partial correctness. Bartels et al. [6,7,5,32,33] used a derived logic with totally correct judgment formalised in Isabelle/HOL to reason about communicating unstructured code. Marti et al. [40] used another formalisation in Coq to reason about MIPS assembly in a minimal OS. In contrast to $\mathcal{L}_A$, these logics are compositional only over non-overlapping code fragments.

Another program logic for unstructured code with totally correct judgment was suggested in 2007 by Myreen and Gordon [44], most famously used in the CakeML verified compiler [36]. In the implementation of the logic, the axiomatization of single instructions is done via *decompilation into logic* [45,46,42,47] and the logic has been used to verify a bignum implementation [43], validate the compilation of seL4 [52], verify device drivers [25,24], and provide machine-checkable proofs of security properties of realistic executables [54]. Unlike $\mathcal{L}_A$, the judgments of this logic do not guarantee that the postcondition is met at the first encounter of the exit labels, which is equivalent to relaxing the condition $\forall n' : 0 < n' < n.\ \mathbf{lbl}(\mathbf{nxt}^{n'}(s)) \notin L$ in the weak transition relation. For this reason, the judgment of the logic cannot express contracts such as C_1 in Eq. 4, which disallow intermediate visits to certain labels. Also, a counterpart to the Conj rule of $\mathcal{L}_A$ is not possible in this type of logic, since it is not possible to guarantee that if two contracts with the same entry and exit labels hold, then the program establishes both postconditions at the same time. Having a stronger judgment comes with other benefits of clarity as well. For example, with first-encounter judgments stating loop variants and invariants (meaning $\{l\} = L$), it is always known that the invariant holds on every iteration of the loop, whereas this is impossible in the other case, where invariants could hold only every nth loop.

Several authors [4,9,14,37] have proposed mechanisms for semi-automatic verification of contracts for unstructured programs. These approaches use different variations of the weakest precondition calculus to generate verification conditions in the same style that we followed in Section 5 to verify loop-free BIR fragments. However, these works do not introduce a general program logic to enable the composition of contracts that have been established using different verification methods.

7 Conclusion

We have presented a Hoare-style program logic $\mathcal{L}_A$ which combines total correctness and postconditions stated on the first encounter of the endpoints. This program logic been defined inside the ITP HOL4, and integrated with the HolBA toolbox to perform semi-automated verification of binary programs. We also prove relative completeness of $\mathcal{L}_A$. In practical verification, the drawback of a first-encounter judgment is that the exit points of the final contract to be derived need to form a subset of the exit labels in every sub-contract. Usually, this is handled by generously populating the blacklist, for example with the complement of the labels touched by execution in the contract.

The verification procedures we integrated into the toolbox HolBA are unoptimized prototypes to exemplify the usage of the logic and do not present a complete verification tool yet. Because of this, a significant amount of code in the verification examples consists of ad-hoc procedures for special cases and should be properly factored out into the toolbox. We are currently working on improving our tool to enable the verification of small and critical low-level components like, for example, microkernels and cryptographic routines.

In this work, we assume a deterministic transition relation. However, we believe that a partial correctness version of $\mathcal{L}_A$ can be straightforwardly extended to non-deterministic systems. However, the completeness proof might require changes to the assumption on primitive transitions, since these can be non-deterministic. Extending the total correctness version of the logic to deal with non-determinism would likely be more complicated since the contract will have to reason about execution traces.

Our work abstracts from the assertion language of $\mathcal{L}_A$, which is typically restricted when the logic is instantiated. For instance, in our implementation for BIR, we restricted the assertion language to use BIR boolean expressions. Other possibilities for the assertion language can be the assertion language of separation logic for enabling a frame rule, and rely-guarantee for reasoning about concurrency. However, these restrictions of the assertion language do not, in general, allow to directly transfer the completeness proof.

We are currently extending $\mathcal{L}_A$ with *strong invariants* in the sense of Hähnle and Mostowski [30]. This would allow stating properties at every execution step, as opposed to the existing invariants of $\mathcal{L}_{\text{AS}}$ which only hold at the initial and final states. Adding such invariants would allow expressing continuous integrity of shared memory sections.

References

1. Appel, A.W.: Foundational proof-carrying code. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 247–256. IEEE (2001)
2. Arbib, M.A., Alagić, S.: Proof rules for gotos. Acta Informatica **11**(2), 139–148 (1979)
3. Ashcroft, E., Hoare, C.A.R., et al.: Remarks on "program proving: Jumps and functions" by m. clint and c.a.r. hoare. Acta Informatica **6** (1976)

4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 82–87 (2005)
5. Bartels, B.: A mechanized verification environment for real-time process algebras and low-level programming languages. Ph.D. thesis, Technical University of Berlin (2014)
6. Bartels, B., Glesner, S.: Verification of distributed embedded real-time systems and their low-level implementations using timed csp. In: 2011 18th Asia-Pacific Software Engineering Conference. pp. 195–202. IEEE (2011)
7. Bartels, B., Jähnig, N.: Mechanized, compositional verification of low-level code. In: NASA Formal Methods Symposium. pp. 98–112. Springer (2014)
8. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 328–343. IEEE (2018)
9. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: International Workshop on Formal Aspects in Security and Trust. pp. 112–126. Springer (2005)
10. Benton, N.: A typed, compositional logic for a stack-based abstract machine. In: Asian Symposium on Programming Languages and Systems. pp. 364–380. Springer (2005)
11. Benton, N.: Abstracting allocation. In: International Workshop on Computer Science Logic. pp. 182–196. Springer (2006)
12. Benton, N., Zarfaty, U.: Formalizing and verifying semantic type soundness for a simple compiler (preliminary report). Tech. rep., Technical Report MSR-TR-2007-31, Microsoft Research (2007)
13. Benton, N., Zarfaty, U.: Formalizing and verifying semantic type soundness of a simple compiler. In: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 1–12 (2007)
14. Burdy, L., Pavlova, M.: Java bytecode specification and verification. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1835–1839 (2006)
15. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. pp. 234–245 (2011)
16. Chlipala, A.: The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In: Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. pp. 391–402 (2013)
17. Clint, M.: Program proving: coroutines. Acta informatica **2**(1), 50–63 (1973)
18. Clint, M., Hoare, C.A.R.: Program proving: Jumps and functions. Acta informatica **1**(3), 214–224 (1972)
19. Dam, M., Guanciale, R., Nemati, H.: Machine code verification of a tiny arm hypervisor. In: Proceedings of the 3rd international workshop on Trustworthy embedded devices. pp. 3–12 (2013)
20. Danial, A.: Count lines of code (cloc). `https://github.com/AlDanial/cloc` (2020), version 1.86
21. De Bruin, A.: Goto statements: Semantics and deduction systems. Acta Informatica **15**(4), 385–424 (1981)
22. Dong, Y., Ren, K., Wang, S., Zhang, S.: Certify once, trust anywhere: Modular certification of bytecode programs for certified virtual machine. In: Asian Symposium on Programming Languages and Systems. pp. 275–293. Springer (2009)

23. Dong, Y., Wang, S., Zhang, L., Yang, P.: Modular certification of low-level intermediate representation programs. In: 2009 33rd Annual IEEE International Computer Software and Applications Conference. vol. 1, pp. 563–570. IEEE (2009)
24. Duan, J.: Formal verification of device drivers in embedded systems. Ph.D. thesis, The University of Utah (2013)
25. Duan, J., Regehr, J.: Correctness proofs for device drivers in embedded systems. In: SSV (2010)
26. Feng, X., Ni, Z., Shao, Z., Guo, Y.: An open framework for foundational proof-carrying code. In: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation. pp. 67–78 (2007)
27. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Combining domain-specific and foundational logics to verify complete software systems. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 54–69. Springer (2008)
28. Feng, X., Shao, Z., Vaynberg, A., Xiang, S., Ni, Z.: Modular verification of assembly code with stack-based control abstractions. ACM SIGPLAN Notices **41**(6), 401–414 (2006)
29. Fox, A.: Directions in isa specification. In: International Conference on Interactive Theorem Proving. pp. 338–344. Springer (2012)
30. Hähnle, R., Mostowski, W.: Verification of safety properties in the presence of transactions. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. pp. 151–171. Springer (2004)
31. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)
32. Jähnig, N., Göthel, T., Glesner, S.: A denotational semantics for communicating unstructured code. arXiv preprint arXiv:1503.04913 (2015)
33. Jähnig, N., Göthel, T., Glesner, S.: Refinement-based verification of communicating unstructured code. In: International Conference on Software Engineering and Formal Methods. pp. 61–75. Springer (2016)
34. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220 (2009)
35. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta informatica **7**(4), 357–360 (1977)
36. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: a verified implementation of ml. ACM SIGPLAN Notices **49**(1), 179–191 (2014)
37. Lehner, H., Müller, P.: Formal translation of bytecode into BoogiePL. Electronic Notes in Theoretical Computer Science **190**(1), 35–50 (2007)
38. Lindner, A., Guanciale, R., Metere, R.: Trabin: Trustworthy analyses of binaries. Science of Computer Programming **174**, 72–89 (2019)
39. Manna, Z., Pnueli, A.: Axiomatic approach to total correctness of programs. Acta Informatica **3**(3), 243–263 (1974)
40. Marti, N.: Formal Verification of Low-Level Software. Ph.D. thesis, University of Tokyo (2008)
41. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system f to typed assembly language. ACM Transactions on Programming Languages and Systems (TOPLAS) **21**(3), 527–568 (1999)
42. Myreen, M.O.: Formal verification of machine-code programs. Tech. rep., University of Cambridge, Computer Laboratory (2009)

43. Myreen, M.O., Curello, G.: Proof pearl: A verified bignum implementation in x86-64 machine code. In: International Conference on Certified Programs and Proofs. pp. 66–81. Springer (2013)
44. Myreen, M.O., Gordon, M.J.: Hoare logic for realistically modelled machine code. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 568–582. Springer (2007)
45. Myreen, M.O., Gordon, M.J.: Verification of machine code implementations of arithmetic functions for cryptography. Theorem Proving in Higher Order Logics: Emerging Trends Proceedings. Dept. of Computer Science, University of Kaiserslautern (2007)
46. Myreen, M.O., Gordon, M.J., Slind, K.: Machine-code verification for multiple architectures-an application of decompilation into logic. In: 2008 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2008)
47. Myreen, M.O., Gordon, M.J., Slind, K.: Decompilation into logic—improved. In: 2012 Formal Methods in Computer-Aided Design (FMCAD). pp. 78–81. IEEE (2012)
48. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 106–119 (1997)
49. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 320–333 (2006)
50. O'Donnell, M.J.: A critique of the foundations of hoare style programming logics. Communications of the ACM **25**(12), 927–935 (1982)
51. Saabas, A., Uustalu, T.: A compositional natural semantics and hoare logic for low-level languages. Electronic Notes in Theoretical Computer Science **156**(1), 151–168 (2006), http://www.sciencedirect.com/science/article/pii/S1571066106002222
52. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified os kernel. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. pp. 471–482 (2013)
53. Tan, G., Appel, A.W.: A compositional logic for control flow. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 80–94. VMCAI'06, Springer-Verlag, Berlin, Heidelberg (2006)
54. Tan, J., Tay, H.J., Gandhi, R., Narasimhan, P.: Auspice: Automatic safety property verification for unmodified executables. In: VSSTE. pp. 202–222. Springer (2015)
55. Wang, A.: An axiomatic basis for proving total correctness of goto-programs. BIT Numerical Mathematics **16**(1), 88–102 (1976)
56. Wang, W., Shao, Z., Jiang, X., Guo, Y.: A simple model for certifying assembly programs with first-class function pointers. In: 2011 Fifth International Conference on Theoretical Aspects of Software Engineering. pp. 125–132. IEEE (2011)
57. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for pcc: Dynamic storage allocation. In: European Symposium on Programming. pp. 363–379. Springer (2003)
58. Zha, J., Feng, X., Qiao, L.: Modular verification of sparcv8 code. In: Asian Symposium on Programming Languages and Systems. pp. 245–263. Springer (2018)
59. Zhao, L., Li, G., De Sutter, B., Regehr, J.: Armor: fully verified software fault isolation. In: Proceedings of the ninth ACM international conference on Embedded software. pp. 289–298 (2011)