A Compositional Logic for Control Flow

Gang Tan[1] and Andrew W. Appel[2]

[1] Computer Science Department, Boston College
gtan@cs.bc.edu
[2] Computer Science Department, Princeton University
appel@cs.princeton.edu

Abstract. We present a program logic, $\mathcal{L}_c$, which modularly reasons about unstructured control flow in machine-language programs. Unlike previous program logics, the basic reasoning units in $\mathcal{L}_c$ are multiple-entry and multiple-exit program fragments. $\mathcal{L}_c$ provides fine-grained composition rules to compose program fragments. It is not only useful for reasoning about unstructured control flow in machine languages, but also useful for deriving rules for common control-flow structures such as while-loops, repeat-until-loops, and many others. We also present a semantics for $\mathcal{L}_c$ and prove that the logic is both sound and complete with respect to the semantics. As an application, $\mathcal{L}_c$ and its semantics have been implemented on top of the SPARC machine language, and are embedded in the Foundational Proof-Carrying Code project to produce memory-safety proofs for machine-language programs.

1 Introduction

Hoare Logic [1] has long been used to verify properties of programs written in high-level programming languages. In Hoare Logic, a triple $\{p\}s\{q\}$ describes the relationship between exactly two states—the normal entry and exit states—associated with a program execution. That is, if the state before execution of s satisfies the assertion p, then the state after execution satisfies q. For a high-level programming language with structured control flow, a program logic based on Hoare triples works fine.

However, programs in high-level languages are compiled into machine code to execute. Since it is hard to prove that a compiler with complex optimizations produces correct machine code from verified high-level-language programs, substantial research effort [2, 3, 4] during recent years has been devoted to verifying properties directly at the machine-language level.

Machine-language programs contain goto statements with unrestricted destinations. Therefore, a program fragment or a collection of statements possibly contains multiple exits and multiple entries to which goto statements might jump. In Hoare Logic, since a triple $\{p\}s\{q\}$ is tailored to describe the relationship between the normal entry and the normal exit states, it is not surprising that trouble arises in considering program fragments with more than one entry/exit.

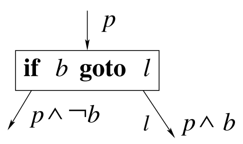
To address the problem of reasoning about control flow in machine-language programs, this paper makes two main contributions:

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 80–94, 2006.

- We propose a program logic, $\mathcal{L}_c$, which modularly reasons about machine-language program fragments. Its basic reasoning units are multiple-entry and multiple-exit program fragments. The logic composes program fragments in a set of fine-grained composition rules. As a result, $\mathcal{L}_c$ is more modular than previous program logics for control flow.
- We also develop for $\mathcal{L}_c$ a semantics. We will show that a naive semantics does not work. We need to use a semantics based on approximations of counting computation steps. Based on this semantics, soundness and (relative) completeness of $\mathcal{L}_c$ are proved.

Before a full technical development, we present an overview of $\mathcal{L}_c$ and its related work.

Overview of $\mathcal{L}_c$. Two design features give $\mathcal{L}_c$ its modularity: its judgment (form of specification) and its composition rules. The judgment in $\mathcal{L}_c$ is directly on multiple-entry and multiple-exit program fragments. For example, $\mathcal{L}_c$ treats a conditional-branch statement "**if** b **goto** l" as a one-entry and two-exit fragment. $\mathcal{L}_c$ then provides for "**if** b **goto** l" a rule, which associates the entries and exits with appropriate invariants, depicted as follows:



The above graph associates the invariant p with the entry, and associates $p \wedge \neg b$ and $p \wedge b$ with two exits, respectively. As a note to our convention, we put invariants on the right of edges; we put labels, when they exist, on the left.

$\mathcal{L}_c$ also provides a set of inference rules to compose judgments on program fragments. These inference rules reason about control flow in smaller steps than Hoare Logic. For example, to reason about while loops, Hoare Logic provides a while rule:

$$\frac{\{p \wedge b\}s\{p\}}{\{p\}\textbf{while } b \textbf{ do } s\{p \wedge \neg b\}}\ \mathsf{while} \qquad\qquad \begin{array}{l} l:\ \textbf{if } \neg b \textbf{ goto } l'; \\ l_1 : s; \\ l_2 : \textbf{goto } l \\ l' : \end{array} \tag{1}$$

A while loop, however, is a high-level language construct. When mapped to machine code, it is implemented by a sequence of more primitive statements. One implementation is shown on the right of the previous figure. Since the implementation contains unstructured control flow, Hoare logic cannot reason about it. In contrast, our logic can treat each statement in the implementation as a multiple-entry and multiple-exit fragment. Using its composition rules, the logic can combine fragments and eliminate intermediate entries and exits. In the end, from its composition rules, the logic can derive the Hoare-logic rule for while loops. Furthermore, it can derive the rules for sequential composition, repeat-until loops, if-then-else statements, and many other structured control-flow constructs. Therefore, our logic can recover structured control flow, when present, in machine-language programs.

Related work on program logics for goto statements. Many researchers have also realized the difficulty of verifying properties of programs with goto statements in Hoare Logic [5, 6, 7, 8, 9]. Some of them have proposed improvements over Hoare Logic. Almost all of these works are at the level of high-level languages. They treat while loops as a separate syntactic construct and have a rule for it. In comparison, $\mathcal{L}_c$ derives rules for control-flow structures.

These previous works also differ from $\mathcal{L}_c$ in terms of the form of specification. The work by de Bruin [8] is a typical example. In his system, the judgment for a statement s is:

$$\langle L_1 : p_1, \ldots, L_n : p_n | \{p\} s \{q\} \rangle, \tag{2}$$

where $L_1, \ldots, L_n$ are labels in a program P; the assertion p_i is the invariant associated with the label L_i; the statement s is a part of the program P. Judgment (2) judges a triple $\{p\}s\{q\}$, but under all label invariants in a program. By explicitly supplying invariants for labels in the judgment, de Bruin's system can handle goto statements, and its rule for goto statements is $\langle L_1 : p_1, \ldots, L_n : p_n | \{p_i\} \textbf{goto } L_i \{\mathit{false}\} \rangle$.

Judgment (2) is sufficient for verifying properties of programs with goto statements. Typed Assembly Language (TAL [3]) by Morrisett et al. uses a similar judgment to verify type safety of assembly-language programs. However, judgment (2) assumes the availability of global information, because it judges a statement s under all label invariants of a program— $L_1 : p_1, \ldots, L_n : p_n$. Consequently, it is impossible for de Bruin's system or TAL to compose fragments with different sets of global label invariants. We believe that a better form of specification should judge s under only those label invariants associated with exits in s. This new form of specification makes fewer assumptions (fewer label invariants) about the rest of the program and is more modular.

Floyd's work [10] on program verification associates a predicate for each arc in the flowchart representation of a program. The program is correct if each statement in the program has been verified correct with respect to the predicates associates with the entry and exit arcs of the statement. In Floyd's system, however, the composition of statements is based on flowcharts and is informal, and it has no principles for eliminating intermediate arcs. Our $\mathcal{L}_c$ provides formal rules for composing statements. When verifying properties of goto statements and labels, Floyd's system also assumes the availability of the complete program.

Cardelli proposed a linking logic [11] to formalize program linking. Glew and Morrisett [12] defined a modular assembly language to perform type-safe linking. Our logic is related to these works because exit labels can be thought as imported labels in a module, and entry labels as exported labels. In some sense, we apply the idea of modular linking to verification of machine code. But since we are more concerned with program verification, we also provide a semantics for our logic, and prove it is both sound and complete.

Recent works by Benton [13], Ni and Shao [14], and Saabas and Usstalu [15] define compositional program logics for low-level machines; their systems also reason modularly about program fragments and linking. To deal with procedure calls and returns, Benton uses Hoare-style pre- and postconditions. Since our compiler uses continuation-passing style, so can our calculus; therefore our labels need only preconditons.

The rest of this paper is organized as follows. Section 2 presents the logic $\mathcal{L}_c$ on a simple imperative language that has unstructured control flow. In Section 3, we develop a semantics for the logic. The soundness and completeness theorems are then presented. In Section 4, we briefly discuss the implementation and the role of $\mathcal{L}_c$ in the Foundational Proof-Carrying Code project [4]. In Section 5, we conclude and discuss future work. A more detailed treatment of the logic, its semantics, and its applications can be found in the first author's PhD thesis [16].

2 Program Logic $\mathcal{L}_c$

We present $\mathcal{L}_c$ on a simple imperative language. Figure 1 presents the syntax of the language. Most of the syntax is self-explanatory, and we only stress a few points. First, since the particular set of primitive operators and relations does not affect the presentation, the language assumes a class of operator symbols, *OPSym*, and a class of relation symbols, *RSym*. For concreteness, *OPSym* could be $\{+, \times, 0, 1\}$ and *RSym* could be $\{=, <\}$. Second, boolean operators do not include standard constructors such as **false**, $\wedge$ and $\Rightarrow$; they can be defined by **true**, $\vee$ and $\neg$.

The language in Fig. 1 is tailored to imitate a machine language. The destination of a goto statement is unrestricted and may be a label in the middle of a loop. Furthermore, the language does not have control structures such as **if** b **then** s, and **while** b **do** s. These control structures are implemented by a sequence of primitive statements.

To simplify the presentation, the language in Fig. 1 differs from machine languages in several aspects. It uses abstract labels while machine languages use concrete addresses. This difference does not affect the results of $\mathcal{L}_c$. The language also lacks indirect jumps (jump through a variable), pc-relative jumps, and procedure calls. We will discuss in Section 4 how we deal with these features.

operator symbols	*OPSym*	op	
relation symbols	*RSym*	re	
variables	*Var*	x, y, z	
labels	*Label*	l	
primitive statements	*PrimStmt*	t	$::= x := e \mid$ **goto** $l \mid$ **if** b **goto** l
statements	*Stmt*	s	$::= t \mid l : s \mid (s_1; s_2)$
expressions	*Exp*	e	$::= x \mid op(e_1, \ldots, e_{ar(op)})$
boolean expressions	*BExp*	b	$::=$ **true** $\mid b_1 \vee b_2 \mid \neg b \mid re(e_1, \ldots, e_{ar(re)})$

Fig. 1. Language syntax, where $ar(op)$ is the arity of the symbol op

2.1 Syntax and Rules of $\mathcal{L}_c$

The syntax of $\mathcal{L}_c$ is in Fig. 2.

Program fragments. A program fragment, $l : (t) : l'$, is a primitive statement t with a start label l and an end label l'. The label l identifies the left side of t, the *normal entry*, and l' identifies the right side of t, the *normal exit.* We also use $l_1 : (s_1; s_2) : l_3$ as an abbreviation for two fragments: $l_1 : (s_1) : l_2$ and $l_2 : (s_2) : l_3$, where l_2 is a new label. We use the symbol F for a set of fragments.

fragments	*Fragment*	$f ::= l : (t) : l'$
fragment sets	*FragSet*	$F ::= \{l_1 : (t_1) : l'_1,\ \ldots,\ l_n : (t_n) : l'_n\}$
assertions	*Assertion*	$p ::= \mathbf{true} \mid p_1 \vee p_2 \mid \neg p \mid re(e_1, \ldots, e_{ar(re)}) \mid \exists x.p$
label-continuation sets	*LContSet*	$\Psi ::= \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}$

Fig. 2. $\mathcal{L}_c$: Syntax

Assertions and label continuations. Assertions are meant to describe predicates on states. $\mathcal{L}_c$ can use any assertion language. We use first-order classical logic in this presentation (see Fig. 2). This assertion language is a superset of the language of boolean expressions. We omit conjunction and universal quantifiers since they can be defined by other constructors classically.

$\mathcal{L}_c$ is parametrized over a deduction system, $\mathcal{D}$, which derives true formulas in the assertion language. We leave the rules of $\mathcal{D}$ unspecified, and assume that its judgment is $\vdash_{\mathcal{D}} p$, which is read as that p is a true formula.

A label identifies a point in a program. To associate assertions with labels, $\mathcal{L}_c$ uses the notation, $l \triangleright p$, pronounced "l with p". In Hoare Logic, when an assertion p is associated with a label l in a verified program, then whenever the control of the program reaches l, the assertion p is true on the current state. In $\mathcal{L}_c$, we interpret $l \triangleright p$ in a different way: If $l \triangleright p$ is true in a program, then whenever p is satisfied, it is safe to continue from l (or, jump to l). Therefore, we call p a *precondition* of the label l, and call $l \triangleright p$ a *label continuation.* We use the symbol Ψ for a set of label continuations.

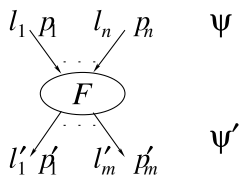
Form of specification. In $\mathcal{L}_c$, the judgment to specify properties of multiple-entry and multiple-exit program fragments has the syntax:

$$F ; \Psi' \vdash \Psi,$$

where F is a set of program fragments; Ψ' and Ψ are sets of label continuations. We next explain this judgment. Suppose

$$\Psi' = \{l'_1 \triangleright p'_1, \ldots, l'_m \triangleright p'_m\}, \text{ and } \Psi = \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}.$$

Labels $l'_1, \ldots, l'_m$ in Ψ' are exits of F, and $l_1, \ldots, l_n$ in Ψ are entries of F. The following graph depicts the relationship between F, Ψ, and Ψ':



With this relationship in mind, an informal interpretation of the judgment $F\,;\Psi' \;\vdash\; \Psi$ is as follows: for a set of fragments F, if it is safe to continue from any of the exit labels, provided that the associated assertion is true, then it is safe to continue from any of the entry labels, provided that the associated assertion is true. This interpretation draws conclusions on entry labels based on assumptions on exit labels. Note, however, this interpretation is simplified and the precise interpretation we will adopt for $F\,;\Psi' \;\vdash\; \Psi$ in Section 3 has an additional requirement: It takes at least one computation step from an entry to reach an exit. We ignore this issue for now and will come back to it.

Using this judgment, $\mathcal{L}_c$ provides rules for primitive statements. For example, Fig. 3 provides a rule for the assignment statement. In the assignment rule, the fragment, $l : (x := e) : l'$, has one entry, namely l, and one exit, namely l'. The assignment rule states that if it is safe to continue from the exit l', when p is true, then it is safe to continue from the entry l, when $p[e/x]$ is true. The reason why it is safe to continue from l can be informally established as follows: Suppose we start from l in an initial state where the next statement to execute is $x := e$ and $p[e/x]$ is true; the new state after the execution of the statement reaches the exit l', and based on the semantics of $x := e$, the assertion p is true; since we assume it is safe to continue from l', when p is true, the new state is safe to continue; hence, the initial state can safely continue from l, when $p[e/x]$ is true.

In Hoare Logic, the assignment rule is $\{p[e/x]\}\ x := e\ \{p\}$. This is essentially the same as the assignment rule in $\mathcal{L}_c$. In general, for any statement s that has only the normal entry and the normal exit, a Hoare triple $\{p\}s\{q\}$ has in $\mathcal{L}_c$ a corresponding judgment: $\{l : (s) : l'\}\,;\ \{l' \triangleright q\} \;\vdash\; \{l \triangleright p\}$.

But unlike Hoare triples, $F\,;\Psi' \;\vdash\; \Psi$ is a more general judgment, which is on multiple-entry and multiple-exit fragments. This capability is used in the rule for conditional-branch statements, **if** b **goto** l_1, in Fig. 3. A conditional-branch statement has two possible exits. Therefore, the if rule assumes two exit label continuations.

Composition rules. The strength of $\mathcal{L}_c$ is its composition rules. These rules can compose judgments on individual statements to form properties of the combined statement. By internalizing control flow of the combined statement, these composition rules allow modular reasoning.

Figure 3 shows $\mathcal{L}_c$'s composition rules. We illustrate these rules using the example in Fig. 4. The figure uses informal graphs, but they can be translated into formal syntax of $\mathcal{L}_c$ without much effort.

$$\boxed{F\,;\,\Psi_1 \,\vdash\, \Psi_2}$$

$$\frac{}{\{l : (x := e) : l'\}\,;\,\{l' \rhd p\} \,\vdash\, \{l \rhd p[e/x]\}}\ \textsf{assignment}$$

$$\frac{}{\{l : (\mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p\} \,\vdash\, \{l \rhd p\}}\ \textsf{goto}$$

$$\frac{}{\{l : (\mathbf{if}\ b\ \mathbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p \wedge b, l' \rhd p \wedge \neg b\} \,\vdash\, \{l \rhd p\}}\ \textsf{if}$$

$$\frac{F_1\,;\,\Psi_1' \,\vdash\, \Psi_1 \quad F_2\,;\,\Psi_2' \,\vdash\, \Psi_2}{F_1 \cup F_2\,;\,\Psi_1' \cup \Psi_2' \,\vdash\, \Psi_1 \cup \Psi_2}\ \textsf{combine} \qquad \frac{F\,;\,\Psi' \cup \{l \rhd p\} \,\vdash\, \Psi \cup \{l \rhd p\}}{F\,;\,\Psi' \,\vdash\, \Psi \cup \{l \rhd p\}}\ \textsf{discharge}$$

$$\frac{\vdash \Psi_1' \Rightarrow \Psi_2' \quad F\,;\,\Psi_2' \,\vdash\, \Psi_2 \quad \vdash \Psi_2 \Rightarrow \Psi_1}{F\,;\,\Psi_1' \,\vdash\, \Psi_1}\ \textsf{weaken}$$

$$\boxed{\vdash \Psi_1 \Rightarrow \Psi_2}$$

$$\frac{m \geq n}{\vdash \{l_1 \rhd p_1, \ldots, l_m \rhd p_m\} \Rightarrow \{l_1 \rhd p_1, \ldots, l_n \rhd p_n\}}\ \textsf{s-width}$$

$$\frac{\vdash_{\mathfrak{D}} p' \Rightarrow p}{\vdash \Psi \cup \{l \rhd p\} \Rightarrow \Psi \cup \{l \rhd p'\}}\ \textsf{s-depth}$$

Fig. 3. $\mathcal{L}_c$: Rules

Assume we already have two individual statements, depicted in the first column of Fig. 4, The first statement is an increment-by-one operation. If $x > 0$ before the statement, then after its completion $x > 0$ still holds. The second statement is **if** $x < 10$ **goto** l. It has one entry, but two exits. The entries and exits are associated with the appropriate assertions that are shown in the figure. The goal is to combine these two statements to form a property of the two-statement block. Notice that the block is effectively a repeat-until loop: it repeats incrementing x until x reaches 10. For this loop, our goal is to prove that if $x > 0$ before entering the block, then $x \geq 10$ after the completion of the block.

Figure 4 also presents the steps to derive the goal from the assumptions using $\mathcal{L}_c$'s composition rules.

In step 1, we use a rule called combine in Fig. 3. When combining two fragment sets, F_1 and F_2, the combine rule makes the union of the entries of F_1 and F_2 the entries of the combined fragment; the same goes for the exits. For the example in Fig. 4, since both statements have only one entry, we have two entries after the combine rule. Since the first statement has one exit, and the second statement has two exits, we have three exits after the combine rule.

After combining fragments, there may be some label that is both an entry and an exit. For example, the label l after the step 1 in Fig. 4 is both an entry and an exit. Furthermore, the entry and the exit for l carry the same assertion: $x > 0$. In such a case, the discharge rule in Fig. 3 can eliminate the label l as an exit. Formally, the discharge rule states that if some $l \rhd p$ appears on both the left and the right of the $\vdash$, then it can be removed from the left; Remember exits are on the left, so this rule removes an exit. The label l_1 is also both an entry and an

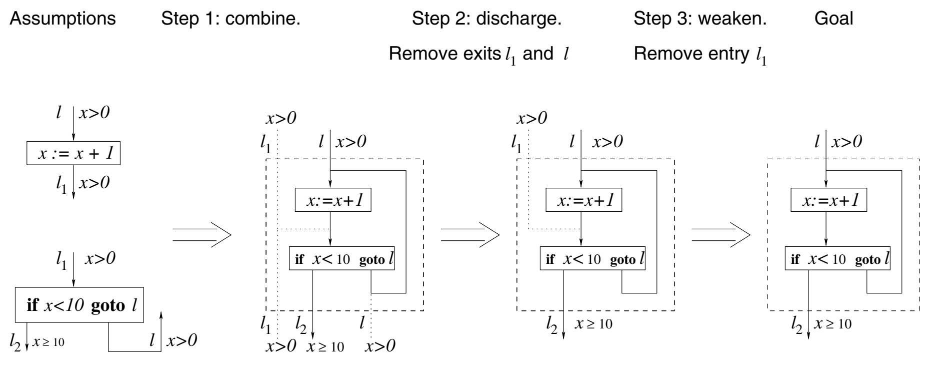


Fig. 4. An example to illustrate $\mathcal{L}_c$'s composition rules

exit, and the entry and the exit carry the same assertion. The discharge rule can remove l_1 as an exit as well. Therefore, the step 2 in Fig. 4 applies the discharge rule twice to remove both l and l_1 as exits. After this step, only one exit is left.

In the last step, we remove l_1 as an entry using the weaken rule. The weaken rule uses a relation between two sets of label continuations: $\vdash \Psi_1 \Rightarrow \Psi_2$, which is read as Ψ_1 is a stronger set of label continuations than Ψ_2.

The rule s-width in Fig. 3 states that a set of label continuations is stronger than its subset. Therefore, $\vdash \{l_1 \triangleright (x > 0), l \triangleright (x > 0)\} \Rightarrow \{l \triangleright (x > 0)\}$ is derivable. Using this result and the weaken rule, the step 3 in Fig. 4 removes the label l_1 as an entry.

After these steps, we have one entry and one exit left for the repeat-until loop, and we have proved the desired property for the loop.

One natural question to ask is which labels the logic should keep as entries. The example eliminates l_1 as an entry, while l remains. The reason is that the final goal tells what should be entries. In other scenarios, we may want to keep l_1 as an entry; for example, in cases when other fragments need to jump to l_1. This is possible in unstructured control flow even though l_1 points to the middle of a loop. In general, the logic $\mathcal{L}_c$ itself does not decide which entries to keep and needs extra information.

The example in Fig. 4 has used almost all composition rules, except for the s-depth rule. The s-depth rule states that a label continuation with a weaker precondition is stronger than a continuation with a stronger precondition. The rule is contravariant over the preconditions. An example of using this rule and the weaken rule is to derive $F\,;\,\Psi' \vdash \{l \triangleright p \wedge q\}$ from $F\,;\,\Psi' \vdash \{l \triangleright p\}$.

Deriving Hoare-logic rules. The composition rules in $\mathcal{L}_c$ can derive all Hoare-logic rules for common control-flow structures. We next show the derivation of the while rule. Assume that a while loop is implemented by the sequence in Equation (1) on page 81, which will be abbreviated by "**while** b **do** s". As we have mentioned, a Hoare triple $\{p\}s\{q\}$ corresponds to $\{l : (s) : l'\}\,;\,\{l' \triangleright q\} \vdash \{l \triangleright p\}$ in $\mathcal{L}_c$. With this correspondence, the derivation of the rule is:

$$\cfrac{\cfrac{\cfrac{\overline{(1)} \quad \{l_1 : (s) : l_2\}\,;\, \{l_2 \rhd p\} \;\vdash\; \{l_1 \rhd p \wedge b\} \quad \overline{(2)}\;\textsf{goto}}{\begin{array}{l}\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p,\ l' \rhd p \wedge \neg b\} \\ \quad \vdash\ \{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p\}\end{array}}\;\textsf{combine}}{\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l' \rhd p \wedge \neg b\} \;\vdash\; \{l \rhd p,\ l_1 \rhd p \wedge b,\ l_2 \rhd p\}}\;\textsf{discharge}}{\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l' \rhd p \wedge \neg b\} \;\vdash\; \{l \rhd p\}}\;\textsf{weaken}$$

where $(1) = \{l : (\textbf{if}\ \neg b\ \textbf{goto}\ l') : l_1\}\,;\, \{l' \rhd p \wedge \neg b,\ l_1 \rhd p \wedge b\} \;\vdash\; \{l \rhd p\}$[1].
and $\quad (2) = \{l_2 : (\textbf{goto}\ l) : l'\}\,;\, \{l \rhd p\} \;\vdash\; \{l_2 \rhd p\}$

In the same spirit, $\mathcal{L}_c$ can derive rules for many other control-flow structures, including sequential composition, repeat-until loops, if-then-else statements. More examples are in the thesis [16–chapter 2].

3 Semantics of $\mathcal{L}_c$

In this section, we develop a semantics for $\mathcal{L}_c$. We will show that a semantics based on pure continuations does not work. We adopt a semantics based on continuations together with approximations of counting computation steps.

3.1 Operational Semantics for the Language

First, we present an operational semantics for the imperative language in Fig. 1. The operational semantics assumes an interpretation $\oint$ of the primitive symbols in *OPSym* and *RSym* in the following way: *Val* is a nonempty domain; for each *op* in *OPSym*, its semantics, $\underline{op}$, is a function in $(Val^{ar(op)} \to Val)$; for each *re* in *RSym*, $\underline{re}$ is a relation $\subset Val^{ar(re)}$, where $ar(op)$ is the arity of the operator.

A machine state is a triple, (pc, π, m): a program counter pc, which is an address; an instruction memory π, which maps addresses to primitive statements or to an **illegal** statement; a data memory m, which maps variables to values. Figure 5 lists the relevant semantic domains.

Before presenting the operational semantics, we introduce some notation. For a state σ, the notation control(σ), i_of(σ), and m_of(σ) projects σ into its program counter, instruction memory, and data memory, respectively. For a mapping m, the notation $m[x \mapsto v]$ denotes a new mapping that maps x to v and leaves other slots unchanged.

The operational semantics for the language is presented in Fig. 6 as a step relation $\sigma \mapsto_\theta \sigma'$ that executes the statement pointed by the program counter. The operational semantics is conventional, except that it is parametrized over a label map $\theta \in LMap$, which maps abstract labels to concrete addresses. When the next statement to execute is **goto** l, the control is changed to $\theta(l)$.

In the operational semantics, if the current statement in a state σ is an **illegal** statement, then σ has no next state to step to; such a state is called a stuck state.

[1] The judgment is derived from the if rule and the weaken rule, assuming that $\vdash_{\mathfrak{D}} p \wedge \neg\neg b \Rightarrow p \wedge b$.

Name	Domain Construction
values, v	*Val* is a nonempty domain
addresses, n	$\mathit{Addr} = \mathbb{N}$
instr. memories, π	$\mathit{IM} = \mathit{Addr} \to \mathit{PrimStmt} \cup \{\mathbf{illegal}\}$
data memories, m	$\mathit{DM} = \mathit{Var} \to \mathit{Val}$
states, σ	$\Sigma = \mathit{Addr} \times \mathit{IM} \times \mathit{DM}$
label maps, θ	$\mathit{LMap} = \mathit{Label} \to \mathit{Addr}$

where $\mathbb{N}$ is the domain of natural numbers.

Fig. 5. Semantic domains

$(\mathrm{pc}, \pi, m) \mapsto_\theta \sigma$ where	
if $\pi(\mathrm{pc}) =$	then $\sigma =$
$x := e$	$(\mathrm{pc}+1, \pi, m[x \mapsto \mathcal{V}\,[\![e]\!]\,m])$
goto l	$(\theta(l), \pi, m)$
if b **goto** l	$\begin{cases} (\theta(l), \pi, m) & \text{if } \mathcal{B}\,[\![b]\!]\,m = \mathrm{tt} \\ (\mathrm{pc}+1, \pi, m) & \text{otherwise} \end{cases}$

where $\mathcal{V} : \mathit{Exp} \to \mathit{DM} \to \mathit{Val}$, and $\mathcal{B} : \mathit{BExp} \to \mathit{DM} \to \{\mathrm{tt}, \mathrm{ff}\}$.

Their definitions are

$$\mathcal{V}\,[\![x]\!]\,m \triangleq m\,[x] \quad \mathcal{V}\,[\![op(e_1, \ldots, e_{ar(op)})]\!]\,m \triangleq \underline{op}(\mathcal{V}\,[\![e_1]\!]\,m, \ldots, \mathcal{V}\,[\![e_{ar(op)}]\!]\,m).$$

$$\mathcal{B}\,[\![\mathbf{true}]\!]\,m \triangleq \mathrm{tt} \quad \mathcal{B}\,[\![b_1 \vee b_2]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}\,[\![b_1]\!]\,m = \mathrm{tt} \text{ or } \mathcal{B}\,[\![b_2]\!]\,m = \mathrm{tt} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$

$$\mathcal{B}\,[\![\neg b]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}\,[\![b]\!]\,m = \mathrm{ff} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$

$$\mathcal{B}\,[\![re(e_1, \ldots, e_{ar(re)})]\!]\,m \triangleq \begin{cases} \mathrm{tt} \text{ if } \langle \mathcal{V}\,[\![e_1]\!]\,m, \ldots, \mathcal{V}\,[\![e_{ar(re)}]\!]\,m \rangle \in \underline{re} \\ \mathrm{ff} \text{ otherwise} \end{cases}$$

Fig. 6. Operational semantics of the language in Fig. 1

If a state σ will not reach a stuck state within k steps, it is *safe for k steps*:

$$\mathrm{safe\_state}(\sigma, k) \;\triangleq\; \forall \sigma' \in \Sigma. \forall j < k.\;\; \sigma \mapsto_\theta^j \sigma' \;\Rightarrow\; \exists \sigma''.\; \sigma' \mapsto_\theta \sigma'',$$

where $\mapsto_\theta^j$ denotes j steps being taken.

3.2 Semantics of $\mathcal{L}_c$

The semantics of $\mathcal{L}_c$ is centered on an interpretation of the judgment $F\,;\,\Psi' \vdash \Psi$. We have discussed an informal interpretation: for the set of fragments F, if Ψ' is true, then Ψ is true; a label-continuation set Ψ being true means it is safe to continue from any label in Ψ, provided that the associated assertion is true. However, this interpretation is too naive, since it cannot justify the discharge rule. When both Ψ' and Ψ in the discharge rule are empty sets, the rule becomes

$$\frac{F\,;\,\{l \triangleright p\} \vdash \{l \triangleright p\}}{F\,;\,\emptyset \vdash \{l \triangleright p\}}$$

According to the informal interpretation, the above rule is like stating "from $l \rhd p \Rightarrow l \rhd p$, derive $l \rhd p$", which is clearly unsound.

The problem is not that $\mathcal{L}_c$ is intrinsically unsound, but that the interpretation is too weak to utilize invariants implicitly in $\mathcal{L}_c$. The interpretation that we adopt is a stronger one. The basic idea is based on a notion of label continuations being *approximately* true. The judgment $F ; \Psi' \vdash \Psi$ is interpreted as, by assuming the truth of Ψ' at a lower approximation, Ψ is true at a higher approximation. In this inductive interpretation, Ψ' and Ψ are treated differently, and it allows the discharge rule to be justified by induction.

Appel and McAllester proposed the indexed model [17], where all predicates are approximated by *counting computation steps*. Our own work [18] used the indexed model to construct a semantic model for a typed assembly language. Next, we will adopt the idea of approximation by counting computation steps from the indexed model to develop a semantics for $\mathcal{L}_c$.

Label continuations being approximately true. We first introduce a semantic function, $\mathcal{A} : \mathit{Assertion} \to \mathit{DM} \to \{\mathrm{tt}, \mathrm{ff}\}$, which gives a meaning to assertions:

$$\mathcal{A}\,[\![\exists x.p]\!]\,m \triangleq \begin{cases} \mathrm{tt} \text{ if } \exists d \in \mathit{Val}.\ \mathcal{A}\,[\![p[d/x]]\!]\,m = \mathrm{tt} \\ \mathrm{ff} \text{ otherwise.} \end{cases}$$

The definition of "$\mathcal{A}\,[\![p]\!]\,m$" on other cases of p is the same as the definition of $\mathcal{B}$ (in Fig. 6) except every occurrence of $\mathcal{B}$ is replaced by $\mathcal{A}$.

Next, we present a notion, $\sigma; \theta \models_k l \rhd p$, to mean that a label continuation $l \rhd p$ is k-approximately true in state σ relative to a label map θ:

$$\begin{aligned} &\sigma; \theta \models_k l \rhd p \ \triangleq \\ &\quad \forall \sigma' \in \Sigma.\ \sigma \mapsto_\theta^* \sigma' \ \wedge\ \mathrm{control}(\sigma') = \theta(l) \ \wedge\ \mathcal{A}\,[\![p]\!]\,(\mathrm{m\_of}(\sigma')) = \mathrm{tt} \\ &\qquad\qquad \Rightarrow\ \mathrm{safe\_state}(\sigma', k) \end{aligned} \tag{3}$$

where $\mapsto_\theta^*$ denotes multiple steps being taken.

There are several points that need to be clarified about the definition. First, by this definition, $l \rhd p$ being a true label continuation in σ to approximation k means that the state is safe to execute for k steps. In other words, the state will not get stuck within k steps.

Second, the definition is relative to a label map θ, which is used to translate the abstract label l to its concrete address.

Last, the definition quantifies over all future states σ' that σ can step to (including σ itself). The reason is that if $\sigma; \theta \models_k l \rhd p$, provided that p is satisfied, it should be safe to continue from location l, not just now, but also in the future. In other words, if $l \rhd p$ is true in the current state, it should also be true in all future states. Therefore, the definition of $\sigma; \theta \models_k l \rhd p$ has to satisfy the following lemma:

Lemma 1. *If $\sigma \mapsto_\theta^* \sigma'$, and $\sigma; \theta \models_k l \rhd p$, then $\sigma'; \theta \models_k l \rhd p$.*

By quantifying over all future states, the definition of $\sigma; \theta \models_k l \rhd p$ satisfies the above lemma. On this aspect, the semantics of $\sigma; \theta \models_k l \rhd p$ is similar to the

Kripke model [19–Ch 2.5] of intuitionistic logic: Knowledge is preserved from current states to future states.

The semantics of a single label continuation is then extended to a set of label continuations: $\sigma; \theta \models_k \Psi \triangleq \forall (l \triangleright p) \in \Psi.\ \sigma; \theta \models_k l \triangleright p$

Loading statements. The predicate $\mathrm{loaded}(F, \pi, \theta)$ describes the loading of a fragment set F into an instruction memory π with respect to a label mapping θ:

$$\mathrm{loaded}(F, \pi, \theta) \triangleq \forall (l : (t) : l') \in F.\ \pi(\theta(l)) = t \ \wedge\ \theta(l') = \theta(l) + 1.$$

Note that some θ are not valid with respect to F. For example, if $F = \{l : (x := 1) : l'\}$, and θ maps l to address 100, then to be consistent θ has to map l' to the address 101. This is the reason why the definition requires[2] that $\theta(l') = \theta(l) + 1$.

Semantics of the judgment $F\,;\Psi' \vdash \Psi$*.* We define a relation, $F\,;\Psi' \models \Psi$, which is the semantic modeling of $F\,;\Psi' \vdash \Psi$.

$$\begin{array}{l} F\,;\Psi' \models \Psi \triangleq \\ \quad \forall \sigma \in \Sigma, \theta \in \mathit{LMap}.\ \mathrm{loaded}(F, \mathrm{i\_of}(\sigma), \theta) \Rightarrow \\ \qquad \forall k \in \mathbb{N}.\ \big(\sigma; \theta \models_k \Psi' \ \Rightarrow\ \sigma; \theta \models_{k+1} \Psi\big). \end{array}$$

The definition quantifies over all label maps θ and all states σ such that F is loaded in the state with respect to θ. It derives the truth of Ψ to approximation $k+1$, from the truth of Ψ' to approximation k. In other words, if it is safe to continue from any of the labels in Ψ', provided that the associated assertion is true, for some number k of computation steps, then it is safe to continue from any of the labels in Ψ, provided that the associated assertion is true, for $k+1$ computation steps. This inductive definition allows the discharge rule to be proved by induction over k.

We have given $F\,;\Psi' \models \Psi$ a strong definition. But the question is what about rules other than the discharge rule. Do they support such a strong semantics? The answer is yes for $\mathcal{L}_c$, because of one implicit invariant—for any judgment $F\,;\Psi' \vdash \Psi$ that is derivable, it takes at least one computation step from labels in Ψ to reach labels in Ψ'. Or, it takes at least one step from entries of F to reach an exit of F. Because of this invariant, although it is safe to continue from exit labels only for k steps, we can still show that it is safe to continue from entry labels for $k+1$ steps.

Finally, since $\mathcal{L}_c$ also contains rules for deriving $\vdash \Psi \Rightarrow \Psi'$ and $\vdash_{\mathfrak{D}} p$, we define relations, $\models \Psi \Rightarrow \Psi'$ and $\models p$, to model their meanings, respectively.

$$\begin{array}{ll} \models \Psi \Rightarrow \Psi' & \triangleq \forall \sigma \in \Sigma, \theta \in \mathit{LMap}, k \in \mathbb{N}.\ (\sigma; \theta \models_k \Psi) \Rightarrow (\sigma; \theta \models_k \Psi') \\ \models p & \triangleq \forall m \in \mathit{DM}.\ \mathcal{A}\,[\![p]\!]\,m = \mathrm{tt} \end{array}$$

[2] There is a simplification. The definition in the thesis [16] also requires that θ does not map exit labels to addresses occupied by F; otherwise, the exit label would not be a “true” exit label.

Soundness and completeness. Based on the semantics we have developed, we next present soundness and completeness theorems for $\mathcal{L}_c$. Due to space limit, we only informally discuss related concepts and cannot present detailed proofs; they can be found in the thesis [16].

As a start, since $\mathcal{L}_c$ is parametrized by a deduction system $\mathcal{D}$, which derives formulas in the assertion language, it is necessary to assume properties of $\mathcal{D}$ before proving properties of $\mathcal{L}_c$: If $\vdash_{\mathcal{D}} p \;\Rightarrow\; \models p$, for any p, then $\mathcal{D}$ is *sound*; if $\models p \;\Rightarrow\; \vdash_{\mathcal{D}} p$, for any p, then $\mathcal{D}$ is *complete*.

Next, we present the soundness and completeness theorems of $\mathcal{L}_c$.

Theorem 1. *(Soundness) Assume $\mathcal{D}$ is sound. If $F\,;\,\Psi \vdash \Psi'$, then $F\,;\,\Psi \models \Psi'$.*

The proof is by induction over the derivation of $F\,;\,\Psi \vdash \Psi'$. The most interesting case is the proof of the discharge rule, which is proved by induction over the number of future computation steps k.

Theorem 2. *(Completeness.)*
Assume $\mathcal{D}$ is complete and Assertion is expressive relative to $\oint$. Assume Assertion is negatively testable by the statement language. Assume (F, Ψ', Ψ) is normal. If $F\,;\,\Psi' \models \Psi$, then $F\,;\,\Psi' \vdash \Psi$.

We informally explain the meanings of expressiveness, *Assertion* being negatively testable, and (F, Ψ', Ψ) being normal below; their precise definitions are in the thesis [16]. As pointed out by Cook [20], a program logic can fail to be complete, if the assertion language is not powerful enough to express invariants for loops in a program. Therefore, the completeness theorem assumes that the assertion language is *expressive.* Also, the theorem assumes that the assertion language is *negatively testable* by the statement language. It means that for any assertion p, there is a sequence of statements that terminates when p is false and diverges when p is true. The triple (F, Ψ', Ψ) being normal means that any label is defined in F at most once; it includes also other sanity requirements on Ψ' and Ψ.

4 Implementation in FPCC

This work is a part of the Foundational Proof-Carrying Code (FPCC) project [4] at Princeton. FPCC verifies memory safety of machine code from the smallest possible set of axioms—machine semantics plus logic. The safety proof is developed in two stages. First, we design a type system at the machine-code level. Machine code is type checked in the type system and thus a typing derivation is a safety witness. In the second stage, we prove the soundness theorem for the type system: If machine code type checks, it is memory safe. This proof is developed with respect to machine semantics plus logic, and is machine checked. The typing derivation composed with the soundness proof is the safety proof of the machine code. The major research problem of the FPCC project is to prove the soundness of our type system—a low-level typed assembly language (LTAL [21]). LTAL can check the memory-safety of SPARC machine code that is generated from our ML compiler.

When proving the soundness of LTAL, we found it is easier to have an intermediate calculus to aid the proving process, because having a simple soundness proof was not LTAL's design goal. We first prove the intermediate calculus is sound from logic plus machine semantics. Then we prove LTAL is sound based on the lemmas provided by the intermediate calculus.

The intermediate calculus in the FPCC project is $\mathcal{L}_c$, together with a type theory [22] as the assertion language. By encoding on top of SPARC machine the semantics of $\mathcal{L}_c$, which we have presented, we have proved that $\mathcal{L}_c$ is sound with machine-checked proofs in Twelf. Then, we prove that LTAL is sound from the lemmas provided by $\mathcal{L}_c$.

The first author's thesis [16–chapter 3] covers the step from $\mathcal{L}_c$ to LTAL in great detail. Here we only discuss a point about control-flow structures in machine code. The simple language in Section 2 on which we presented $\mathcal{L}_c$ lacks indirect jumps, pc-relative jumps, and procedure calls. Our implementation on SPARC handles pc-relative jumps, and handles indirect jumps using first-class continuation types in the assertion language. These features will not affect the soundness result of $\mathcal{L}_c$, as our implementation has shown. However, we have not investigated the impact of indirect jumps to the completeness result, which is unnecessary for the FPCC project. We have not modeled procedure call-and-return—since our compiler uses continuation-passing style, continuation calls and continuation-passing suffice. Procedure calls, if needed, could be handled by following the work of Benton [13].

5 Conclusion and Future Work

Previous program logics for goto statements are too weak to modularly reason about program fragments with multiple entries and multiple exits. We have presented $\mathcal{L}_c$, which needs only local information to reason about a program fragment and compose program fragments in an elegant way. $\mathcal{L}_c$ is not only useful for reasoning about unstructured control flow in machine languages, but also useful for deriving rules for common control-flow structures. We have also presented for $\mathcal{L}_c$ a semantics, based on which the soundness and completeness theorems are formally proved. We have implemented $\mathcal{L}_c$ on top of SPARC machine language. The implementation has been embedded into the Foundational Proof-Carrying Code Project to produce memory-safety proofs for machine-language programs.

One possible future extension is to combine this work with modules, to produce a module system with simple composition rules, and with a semantics based on counting computation steps.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the Association for Computing Machinery **12** (1969) 578–580
2. Necula, G.: Proof-carrying code. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, ACM Press (1997) 106–119

3. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. on Programming Languages and Systems **21** (1999) 527–568
4. Appel, A.W.: Foundational proof-carrying code. In: Symposium on Logic in Computer Science (LICS '01), IEEE (2001) 247–258
5. Clint, M., Hoare, C.A.R.: Program proving: Jumps and functions. Acta Informatica (1972) 214–224
6. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta Informatica **7** (1977) 357–360
7. Arbib, M., Alagic, S.: Proof rules for gotos. Acta Informatica **11** (1979) 139–148
8. de Bruin, A.: Goto statements: Semantics and deduction systems. Acta Informatica **15** (1981) 385–424
9. O'Donnell, M.J.: A critique of the foundations of hoare style programming logics. Communications of the Association for Computing Machinery **25** (1982) 927–935
10. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposia in Applied Mathematics, Providence, Rhode Island (1967) 19–32
11. Cardelli, L.: Program fragments, linking, and modularization. In: 24th ACM Symposium on Principles of Programming Languages. (1997) 266–277
12. Glew, N., Morrisett, G.: Type-safe linking and modular assembly language. In: 26th ACM Symposium on Principles of Programming Languages. (1999) 250–261
13. Benton, N.: A typed, compositional logic for a stack-based abstract machine. In: 3rd Asian Symposium on Programming Languages and Systems. (2005)
14. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: 33rd ACM Symposium on Principles of Programming Languages. (2006) To appear.
15. Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. In: Proceedings of the Second Workshop on Structured Operational Semantics (SOS'05). (2005)
16. Tan, G.: A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code. PhD thesis, Princeton University (2005)
17. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. on Programming Languages and Systems **23** (2001) 657–683
18. Tan, G., Appel, A.W., Swadi, K.N., Wu, D.: Construction of a semantic model for a typed assembly language. In: Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04). (2004) 30–43
19. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Available as DIKU Rapport 98/14 (1998)
20. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing **7** (1978) 70–90
21. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound TAL for back-end optimization. In: ACM Conference on Programming Language Design and Implementation. (2003) 208–219
22. Swadi, K.N.: Typed Machine Language. PhD thesis, Princeton University (2003)