ILC: A Foundation for Automated Reasoning About Pointer Programs

Limin Jia David Walker

Princeton University

{ljia,dpw}@cs.princeton.edu

Abstract

This paper presents a new program logic designed for facilitating automated reasoning about pointer programs. The program logic is directly inspired by previous work by O'Hearn, Reynolds, and Yang on separation logic, but rather than using classical bunched logic as the basis for assertions, we use Girard's intuitionistic linear logic extended with a sublogic for classical logic reasoning about arithmetic. The main contributions of the paper include the definition of a sequent calculus for our new logic, which we call ILC (Intuitionistic Linear logic with Classical arithmetic) and proof of a cut elimination. We also give a store semantics for the logic. Next, we define a simple imperative programming language with mutable references and give verification condition generation rules that produce assertions in ILC. We have proven verification condition generation is sound. Finally, we identify a fragment of ILC, ILC$^-$, that is both decidable and closed under generation of verification conditions. In other words, if loop invariants are specified in ILC$^-$, then the resulting verification conditions are also in ILC$^-$. Since verification condition generation is syntax-directed, we obtain a decidable procedure for checking properties of pointer programs.

1. Introduction

In the eighties and early nineties, formal program specification and verification was left for dead: it was too difficult, too costly, too time-consuming and completely unscalable. Perhaps the government could pay to formally specify and check parts of their most critical space shuttle infrastructure, but certainly no one else could use it to verify their own products. Amazingly, in 2005, Microsoft is now using verification technology in many of their internal projects and is currently planning to include a logical specification and checking language in their next version of Visual C.[1] This remarkable turnaround was made possible in part by moving away from complete program verification to verification of a smaller selection of simple, but useful program properties, and in part by great improvements in abstract interpretation and theorem proving technologies.

Some of the most successful recent verification projects including the Microsoft assertion language mentioned above, Leino et al.'s extended static checking project and its successors [7, 8, 3], and Necula and Lee's proof-carrying code [18, 17], to name just a few, have used conventional classical logic to specify and check

[1] Zhe Yang, Microsoft Research at Princeton Computer Science Department Colloquium, March 2005.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WXYZ '05 date, City.

Copyright © 2005 ACM supplied by printer. . . $5.00.

program properties. These conventional classical logics work exceptionally well for specifying arithmetic conditions and verifying that array accesses are in bounds. One place where there remains room for improvement is in specification and verification of programs that manipulate pointers and manage resources.

To better support verification of pointer programs, O'Hearn, Reynolds and Yang [10, 20] have advocated using the classical logic of bunched implications, extended with a collection of domain specific axioms about storage, as the assertion language for program verification. The crucial insight in this research is that the expressive connectives of the logic of bunched implications encapsulate key invariants used over and over again when reasoning about storage. For instance, the separating conjunction of bunched logic $F_1 * F_2$ says that F_1 accurately describes a portion of the store (h_1) and F_2 also accurately describes a portion of the store (h_2) *and* the two portions of the store have no location to common. In classical logic, expressing the same condition is much more verbose. One might try introducing some set theory into the logic and using a formula such as $F_1 \wedge F_2 \wedge S_1 \cap S_2 = \emptyset$ where S_1 and S_2 are the sets of program locations that F_1 and F_2 respectively depend upon. As one increases the number of separate memory chunks, the separation logic formula remains relatively simple: $F_1 * F_2 * F_3 * F_4$ represents four separate pieces of the store. On the other hand, the related classical formula becomes increasingly complex:

$$F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge S_1 \cap S_2 = \emptyset \wedge S_1 \cap S_3 = \emptyset \wedge S_1 \cap S_4 = \emptyset \wedge S_2 \cap S_3 = \emptyset \wedge S_2 \cap S_4 = \emptyset \wedge S_3 \cap S_4 = \emptyset$$

The end result is that while in theory it is not impossible to reason about memory in conventional classical logic, in practice invariants concerning linear data structures, in particular, can quickly grow to an unmanageable size. Separation logic has certainly not yet solved all memory management problems, but for many examples studied by O'Hearn et al., proofs are much more concise and manageable than they would be in classical logic. Finding logics that allow concise and intuitive specification of common program properties is clearly crucial for bringing the technology to bear on practical programming problems.

Most of the research by O'Hearn et al. so far focuses on manual construction of program proofs as opposed to automatic techniques. Two exceptions are work by Calcagno et al. [5] and Berdine et al. [4], who have identified fragments of quantifier-free separation logic that are decidable, though they have not yet shown how to use these fragments in program verification. The main purpose of this paper is to set up a foundation for automatic verification of properties of pointer programs. One natural way to pursue this agenda is to use the weakest precondition generation rules defined by O'Hearn et al. and to develop a special-purpose theorem prover to check validity of the generated assertions. However, we have not taken this road. Instead, we have developed a closely related, but new logic that combines Intuitionistic Linear logic with Classical arithmetic (ILC). We have also give a syntax-directed algorithm for generating verification conditions for a simple programming language with control flow constructs, mutable references, allocation, and deallocation.

For the purposes of automatic program verification, there are four main reasons for using this new logic as opposed to separation logic.

- First, ILC supports a strict division between the substructural part (intuitionistic linear logic), which is used to reason about memory, and the unrestricted classical part, which is used to reason about arithmetic. This separation will allow us to exploit *directly* the highly effective decision procedures for arithmetic that have been used so successfully in previous program verification efforts.
- Second, by using well-known complexity results for intuitionistic linear logic, we have identified a fragment of ILC that is decidable. Moreover, our verification condition generation algorithm is syntax-directed and closed under this fragment of ILC. In other words, if programmers write loop invariants in this fragment of ILC then the resulting verification conditions are also in this fragment and can be decided. Overall, this leads to a decidable procedure for program verification.
- Third, the proof theory for O'Hearn et al.'s separation logic is a combination of the proof theory for the logic of bunched implications plus a collection of specialized axioms. A number of the specialized axioms are dedicated to reasoning about pure formulas — those formulas that do not refer to the store. We observe that

when representing pure formulas using linear logic's unrestricted modality $!F$, the main axioms fall out for free. Consequently, it is unnecessary to develop specialized theorem proving techniques for them.

- Fourth, we have recently looked at generating proof-carrying code for programs with rich memory management invariants [2, 1, 11] and while we found encoding "single-pointer" invariants in separation logic highly effective, we were unable to find a simple encoding for general-purpose (typed) shared mutable references. Consequently, we fell back on older ideas from the work on alias types, which implicitly [22], and in newer work, explicitly [15], use linear logic's unrestricted modality as part of the encoding. Though we do not focus on this issue in this paper, it is clear that ILC can easily accommodate these encodings.

In addition to these main points, we are also simply curious to more fully understand the relative strengths and weaknesses of using intuitionistic linear logic as opposed to the logic of bunched implications as the foundation for verification of pointer programs. O'Hearn [19] explains convincingly that, in general, there are deep and important differences between the two logics, but the question of which logic is better for verifying pointer programs persists. O'Hearn, Reynolds, Yang and others have now spent approximately six years studying the use of bunched implications in this domain. However, the need for complex bunches in their proofs is rare at best. The question we ask is whether they are needed at all. This paper does not answer that question but it does lay out a foundation for using intuitionistic linear logic as opposed to bunched logic in automatic program verification. This is a starting point from which we can begin to study the relationship between the two logics in this domain in depth.

In summary, there are three central contributions of this paper. First, in Section 2, we propose ILC as opposed to bunched logic as a foundation for checking safety properties of pointer programs. We outline the proof theory for ILC as a sequent calculus, prove a cut-elimination theorem to show it is well defined, and give number of simple examples. The proof theory is sound with respect to the storage model, but not complete. Separation logic also suffers from a lack of completeness (c.f., Reynolds [21, pg. 6] comments "regrettably these [axioms] are far from complete."). This lack of completeness does not concern us as previous successful defect detectors such as ESC [7] have also been incomplete. Our second contribution (Section 3) is to define a simple imperative language with references and to give syntax-directed verification condition generation rules that use ILC as the assertion language. We prove that verification condition generation is sound with respect to our memory model and give a few examples. The third main contribution of this paper, discussed in Section 2.6, is in the definition of a very useful, but decidable fragment of the logic, ILC^-. The key property of ILC^- is that it is closed under verification condition generation. In other words, if loop invariants and pre- and post-conditions fall into ILC^- then the generated verification conditions also fall into ILC^-. All of the example programs we verify in section 3.4 use invariants in ILC^-. Overall, the decidable logic plus the syntax-directed verification condition generation gives rise to a terminating algorithm for verification of pointer programs.

Section 4 compares our research more completely with related work. Section 5 summarizes our research again and suggests directions for future work. One crucial direction for future work is the study of recursive data structures, which is beyond the scope of this initial paper.

2. Intuitionistic Linear Logic with Classical Arithmetic

In this section we introduce ILC, Intuitionistic Linear logic with Classical arithmetic. It is designed as a foundation for reasoning about state with arithmetic constraints. The basic goal is to embed classical reasoning about arithmetic into intuitionistic linear logic. We do so in a principled way by confining the classical formulas to a new modality $\bigcirc$. Our logic contains all the connectives of first-order multiplicative and additive linear logic.

This section is organized as follows: we will first familiarize the readers with the syntax, then informally discuss basic concepts behind the connectives of our logic, and the connections and differences between ILC and separation logic. The formal semantics and proof theory are introduced in Section 2.3 and Section 2.4. After discussing the properties of our logic in Section 2.5, we will present a decidable fragment, ILC^-, in Section 2.6.

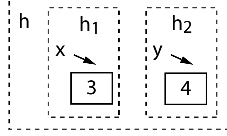


Figure 1. A sample heap

2.1 Syntax

The syntax of the logic is shown below. We use E to range over terms (integer expressions), which include variables x, constants n, and the application of function symbols $+$ and $-$. We use Pa to range over arithmetic predicates, which are equality and partial-order relationships on integers. The classical formulas A consist of classical truth, arithmetic predicates, conjunction and negation. The state predicate $(E_1 \Rightarrow E_2)$ describes a heap containing only one location, E_1 and its contents E_2. The intuitionistic formulas, F, contain the store predicate and formulas constructed from the connectives in first-order linear logic and the new modality $\bigcirc$.

Integer Terms	E	$::=$	$n \mid x \mid E + E \mid -E$
Arithmetic Predicates	Pa	$::=$	$E_1 = E_2 \mid E_1 < E_2$
Classical Formulas	A	$::=$	$\texttt{true} \mid \texttt{false} \mid Pa \mid A_1 \wedge A_2 \mid \neg A \mid A_1 \vee A_2$
State Predicates	Ps	$::=$	$(E_1 \Rightarrow E_2)$
Intuitionistic Formulas	F	$::=$	$Ps \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \multimap F_2 \mid \top \mid F_1 \,\&\, F_2 \mid \mathbf{0} \mid F_1 \oplus F_2$
			$\mid\, !\, F \mid \exists b.F \mid \forall b.F \mid \bigcirc A$

2.2 Basic Concepts

We informally discuss the semantics of the connectives and highlight the key ideas for reasoning about program states. Most of the ideas also appear in separation logic. All the examples in the section refer to Figure 1. Figure 1 shows a heap h that contains two disjoint parts: h_1 and h_2. The first part h_1 contains location x, which contains integer 3; the second part h_2 contains location y, which contains integer 4.

Emptiness. The connective **1** describes an empty heap. The counterpart in separation logic is usually written as `emp`.

Separation. Multiplicative conjunction $\otimes$ separates a linear state into two disjoint parts. For example, the heap h can be described by formula $(x \Rightarrow 3) \otimes (y \Rightarrow 4)$. Multiplicative conjunction does not allow weakening or contraction: $F_1 \otimes F_1$ is not the same as F_1. Therefore, we can uniquely identify each part in the heap and track its state changes. The multiplicative conjunction $(*)$ in separation logic has the same properties.

Update. Multiplicative implication $\multimap$ is similar to the multiplicative implication $-*$ in separation logic. Formula $F_1 \multimap F_2$ describes a heap h waiting for another piece; if given another heap h' that is described by F_1, and if h' is disjoint from h, then the union of h and h' can be described by F_2. For example, h_2 in Figure 1 can be described by $(x \Rightarrow 3) \multimap ((x \Rightarrow 3) \otimes (y \Rightarrow 4))$. Perhaps a more interesting example is that heap h in Figure 1 satisfies formula $F = (x \Rightarrow 3) \otimes ((x \Rightarrow 5) \multimap ((x \Rightarrow 5) \otimes (y \Rightarrow 4)))$. This example brings out the idea of describing store updates using multiplicative conjunction and implication. Formula F can be read as: location x is allocated on heap h (because part of h is described by $(x \Rightarrow 3)$), and if we update the contents of location x with 5 (the new piece is described by $(x \Rightarrow 5)$), then the resulting heap is described by $(x \Rightarrow 5) \otimes (y \Rightarrow 4)$. This scheme comes up very often in the preconditions we generate for the assertion language in the next section.

No information. The connective $\top$ is the unit of additive conjunction. It describes any linear state, and therefore it does not contain any specific information about the linear state it describes. The counterpart of $\top$ in separation logic is usually written `true`.

Sharing. Additive conjunction $\&$ shares one linear state between two descriptions. Formula $F_1 \& F_2$ represents a state can be described by both F_1 and F_2. For example, h is described by $((x \Rightarrow 3) \otimes \top) \& ((y \Rightarrow 4) \otimes \top)$.

The additive conjunction in separation logic is written $\wedge$. The behavior of $\wedge$ is closely connected to the additive implication $\rightarrow$ and the bunched contexts, which our logic doesn't have. However, the connective $\wedge$ will behave differently from $\&$ in our logic only in the presence of the additive implication $\rightarrow$. The basic sharing properties of these two connectives are the same.

Heap Free Conditions. The unrestricted modality $!F$ describes an empty heap and asserts F is true. For instance, $!((x \Rightarrow 3) \multimap \exists y.(x \Rightarrow y))$ describes the empty heap. It says that given no initial resources, if we add a heap in which location x holds 3 then we end up with a heap in which location x holds some y. On the other hand, $!(x \Rightarrow 3)$ cannot be satisfied. The $!$ connective requires the underlying formula describe an empty heap but $(x \Rightarrow 3)$ always describes a heap with one element.

The observant reader will immediately notice that $!F$ is semantically equivalent to $F\&\mathbf{1}$. The latter formula describes the same heap in two ways, once as F and once as empty. However, as we will see in the next section, the two formulas have different proof-theoretic properties. Formula $!F$ satisfies weakening and contraction and therefore can be used as many times as we choose; $F\&\mathbf{1}$ does not satisfy this properties in general. Hence $!$ is used as a simple syntactic marker that informs the theorem prover of the structural properties to apply to the underlying formula.

The equivalent idea in separation logic is that of a "pure formula". Rather than using a connective to mark the purity attribute, a theorem prover analyzes the syntax of the formula to determine its status. Pure formulas are specially axiomatized in separation logic.

Classical Reasoning. In separation logic, negation is defined by additive implication and `false` as $\neg P \stackrel{\text{def}}{=} P \rightarrow \texttt{false}$. And law of excluded middle holds in the classical semantics. For instance, formula $(x \Rightarrow 3) \vee \neg(x \Rightarrow 3)$ is valid. However, negations of "heap-ful" conditions, such as $\neg(x \Rightarrow 3)$, appear very rare. On the other hand classical reasoning about arithmetic is ubiquitous. Consequently, we add a classical sublogic to what we have already presented. The classical formulas are confined under the modality $\bigcirc$. For example, the heap h in Figure 1 satisfies $\exists e_1.\exists e_2.(x \Rightarrow e_1) \otimes (y \Rightarrow e_2) \otimes !(\bigcirc(\neg(e_1 = e_2)))$. In separation logic we should write $\exists e_1.\exists e_2.((x \Rightarrow e_1) * (y \Rightarrow e_2)) \wedge (\neg(e_1 = e_2))$. The modality $\bigcirc$ separates the classical arithmetic reasoning from the rest of the intuitionistic linear reasoning making it possible to use an efficient off-the-shelf decision procedure for classical arithmetic.

2.3 Semantics of the Logic

Following Reynold's [21], all values are integers, some integers (an infinite collection of them) are considered heap locations. We use meta variable n when referring to integers, ℓ when referring to locations, and v when referring to values. We use stores as models for our logic. A store (heap) is a finite map from locations to values.

We also define the following operations on the store:

- $dom(h)$ denotes the domain of store h.
- $h(\ell)$ denotes the value stored at location ℓ.
- $h[\ell := v]$ denotes a store h' in which ℓ maps to v but is otherwise the same as h.
- $h_1 \uplus h_2$ denotes the union of disjoint stores. It is undefined if the stores are not disjoint.

There are three semantic judgments:

$\mathcal{M} \vDash A$	Arithmetic model $\mathcal{M}$ satisfies classical formula A
$\mathcal{M}; h \vDash F$	store h together with arithmetic model $\mathcal{M}$ satisfy formula F
$h \vDash F$	store h satisfies formula F

- $\mathcal{M}; h \vDash (E_1 \Rightarrow E_2)$ iff $dom(h) = [\![E_1]\!]$, $h([\![E_1]\!]) = [\![E_2]\!]$
- $\mathcal{M}; h \vDash \mathbf{1}$ iff $dom(h) = \emptyset$
- $\mathcal{M}; h \vDash F_1 \otimes F_2$ iff $h = h_1 \uplus h_2$, and $\mathcal{M}; h_1 \vDash F_1$, and $\mathcal{M}; h_2 \vDash F_2$.
- $\mathcal{M}; h \vDash F_1 \multimap F_2$ iff for all store h', $\mathcal{M}; h' \vDash F_1$ implies $\mathcal{M}; h \uplus h' \vDash F_2$.
- $\mathcal{M}; h \vDash \top$ it is true for all stores.
- $\mathcal{M}; h \vDash F_1 \,\&\, F_2$ iff $\mathcal{M}; h \vDash F_1$, and $\mathcal{M}; h \vDash F_2$.
- $\mathcal{M}; h \vDash \mathbf{0}$, no store satisfies $\mathbf{0}$.
- $\mathcal{M}; h \vDash F_1 \oplus F_2$ iff $\mathcal{M}; h \vDash F_1$, or $\mathcal{M}; h \vDash F_2$.
- $\mathcal{M}; h \vDash !F$ iff $dom(h) = \emptyset$, and $\mathcal{M}; h \vDash F$.
- $\mathcal{M}; h \vDash \exists x.F$ iff there exists some integer a such that $\mathcal{M}; h \vDash F[a/x]$.
- $\mathcal{M}; h \vDash \forall x.F$ iff for all integer a, $\mathcal{M}; h \vDash F[a/x]$.
- $\mathcal{M}; h \vDash \bigcirc A$ iff $dom(h) = \emptyset$, and $\mathcal{M} \vDash A$.

Figure 2. The Semantics of formulas

The arithmetic terms, functions, and predicates are interpreted in Presburger Arithmetic. We write $[\![E]\!]$ for the integer value that the closed expression E denotes to. The definition of the semantics of classical formulas are standard, and we omit it in this paper.

Because classical arithmetic constraints play an important role in our logic, we explicitly mention the arithmetic model in the semantics of the formulas. The informal semantic meanings of formulas are discussed in the previous section, and the formal definitions of $\mathcal{M}; h \vDash F$ are given in Figure 2. The semantics of the formulas is straightforward; only the classical modality needs some attention. Formula $\bigcirc A$ is valid if the store is empty and the classical formula A is valid in Presburger Arithmetic.

Lastly, a store h satisfies a formula F if and only if Presburger arithmetic together with the store satisfies F:

$$h \vDash F \text{ iff } \mathcal{M}; h \vDash F \quad \text{where } \mathcal{M} \text{ is Presburger Arithmetic.}$$

2.4 Proof Theory

In this section we formalize the sequent calculus for ILC.

Logical Contexts Our logical judgments make use of an unrestricted context Γ for classical formulas, an unrestricted context Θ for intuitionistic formulas, and a linear context Δ, also for intuitionistic formulas. The first two contexts have contraction, weakening, and exchange properties; while the last has only exchange. The context Ω contains a set of variables.

Classical Unrestricted Context	Γ	$::=$	$\cdot \mid \Gamma, A$
Intuitionistic Unrestricted Context	Θ	$::=$	$\cdot \mid \Theta, F$
Intuitionistic Linear Context	Δ	$::=$	$\cdot \mid \Delta, F$
Variable Context	Ω	$::=$	$\cdot \mid \Omega, x$

Logical Judgments There are two sequent judgments in our logic.

$\Omega \mid \Gamma \,\#\, \Gamma'$	classical sequent rules
$\Omega \mid \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F$	intuitionistic sequent rules

The sequent rules for classical logic are in the form $\Omega \mid \Gamma \,\#\, \Gamma'$, where Γ is the context for truth assumptions and Γ' is the context for false assumptions. The sequent $\Omega \mid \Gamma \,\#\, \Gamma'$ can be read as: the truth assumptions in Γ contradicts with one of the false assumptions in Γ'. The formalization is Gentzen's LK formalization, and we only give the basic *Contra* rule below.

Sequent rules for ILC: $\boxed{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F}$

$$\frac{}{\Omega \,|\, \Gamma; \Theta; F \Longrightarrow F}\ \textit{L-Init} \qquad \frac{\Omega \,|\, \Gamma; \Theta, F; \Delta, F \Longrightarrow F'}{\Omega \,|\, \Gamma; \Theta, F; \Delta \Longrightarrow F'}\ \textit{Copy}$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta_1 \Longrightarrow F_1 \quad \Omega \,|\, \Gamma; \Theta; \Delta_2 \Longrightarrow F_2}{\Omega \,|\, \Gamma; \Theta; \Delta_1, \Delta_2 \Longrightarrow F_1 \otimes F_2}\ \otimes R \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta, F_1, F_2 \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \otimes F_2 \Longrightarrow F}\ \otimes L$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \Longrightarrow F_2}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \multimap F_2}\ \multimap R \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \quad \Omega \,|\, \Gamma; \Theta; \Delta', F_2 \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, \Delta', F_1 \multimap F_2 \Longrightarrow F}\ \multimap L$$

$$\frac{}{\Omega \,|\, \Gamma; \Theta; \cdot \Longrightarrow \mathbf{1}}\ 1R \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, \mathbf{1} \Longrightarrow F}\ 1L$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \quad \Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_2}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \,\&\, F_2}\ \&R$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \,\&\, F_2 \Longrightarrow F}\ \&L1 \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta, F_2 \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \,\&\, F_2 \Longrightarrow F}\ \&L2$$

$$\frac{}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow \top}\ \top R$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \oplus F_2}\ \oplus R1 \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_2}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F_1 \oplus F_2}\ \oplus R2$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \Longrightarrow F \quad \Omega \,|\, \Gamma; \Theta; \Delta, F_2 \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \Delta, F_1 \oplus F_2 \Longrightarrow F}\ \oplus L$$

$$\frac{}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta, \mathbf{0} \Longrightarrow F}\ \mathbf{0}L$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F[t/x]}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow \exists x.F}\ \exists R \qquad \frac{\Omega, a \,|\, \Gamma; \Theta; \Delta, F[a/x] \Longrightarrow F'}{\Omega \,|\, \Gamma; \Theta; \Delta, \exists x.F \Longrightarrow F'}\ \exists L$$

$$\frac{\Omega, a \,|\, \Gamma; \Theta; \Delta \Longrightarrow F[a/x]}{\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow \forall x.F}\ \forall R \qquad \frac{\Omega \,|\, \Gamma; \Theta; \Delta, F[t/x] \Longrightarrow F'}{\Omega \,|\, \Gamma; \Theta; \Delta, \forall x.F \Longrightarrow F'}\ \forall L$$

$$\frac{\Omega \,|\, \Gamma; \Theta; \cdot \Longrightarrow F}{\Omega \,|\, \Gamma; \Theta; \cdot \Longrightarrow !F}\ !R \qquad \frac{\Omega \,|\, \Gamma; \Theta, F; \Delta \Longrightarrow F'}{\Omega \,|\, \Gamma; \Theta; \Delta, !F \Longrightarrow F'}\ !L$$

$$\frac{\Omega \,|\, \Gamma \,\#\, A}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \cdot \Longrightarrow \bigcirc A}\ \bigcirc R \qquad \frac{\Omega \,|\, \Gamma, A \,;\, \Theta \,;\, \Delta \Longrightarrow F}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta, \bigcirc A \Longrightarrow F}\ \bigcirc L \qquad \frac{\Omega \,|\, \Gamma \,\#\, \cdot}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F}\ \textit{Absurdity}$$

Figure 3. Sequent Calculus

$$\frac{}{\Omega \,|\, \Gamma, A \,\#\, A, \Gamma'}\ \mathit{Contra}$$

The intuitionistic sequent rules have the form: $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F$. An intuitive reading of the sequent is that if a state described by unrestricted assumptions in Θ, linear assumptions Δ, and satisfying all the classical arithmetic constraints in Γ, then this state can also be viewed as a state described by F. Context Ω contains all the free variables in Γ, Θ, Δ, and F. The complete set of sequent rules are listed in Figure 3

The sequent rules for multiplicative connectives, additive connectives, and quantifications are the same as those in intuitionistic linear logic except that the classical context Γ is carried around. The interesting rules are the left and right rule for the new modality $\bigcirc$ and the *absurdity* rule, which illustrates the interaction between the classical part and the intuitionistic part of the logic. The right rule for $\bigcirc$ says that if Γ contradicts the assertion "A false" (which means A is true) then we can derive $\bigcirc A$ without using any linear resources. If we read the left rule for $\bigcirc$ bottom up, it says that whenever we have $\bigcirc A$, we can put A together with other classical assumptions

in Γ. The absurdity rule is a peculiar one. The justification for this rule is that since Γ is not consistent, no state can meet the constraints imposed by Γ, therefore, any statement based on the assumption that a state satisfies those constraints is simply true.

Example Derivation The proof tree of judgment $x, y \mid y > 1; \cdot; (x \Rightarrow y) \implies \exists z. (x \Rightarrow z) \otimes ! \bigcirc (z > 0)$ is given below. We omit the variable context in the derivation.

$$\dfrac{\dfrac{\dfrac{}{y>1;\cdot;(x\Rightarrow y)\implies(x\Rightarrow y)}\textit{L-Init}\quad \dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}}{}}{y>1 \,\#\, y>0}}{y>1;\cdot;\cdot\implies\bigcirc(y>0)}\bigcirc R}{y>1;\cdot;\cdot\implies !\bigcirc(y>0)}\,!R}{y>1;\cdot;(x\Rightarrow y)\implies(x\Rightarrow y)\otimes!\bigcirc(y>0)}\otimes R}{y>1;\cdot;(x\Rightarrow y)\implies\exists z.(x\Rightarrow z)\otimes!\bigcirc(z>0)}\exists R$$

We can see that because the classical formula is encapsulated under $\bigcirc$ and the classical context is separated from other contexts, the classical reasoning is pushed to the leaves of the derivation tree. There is a clear boundary between classical reasoning about arithmetic and the intuitionistic reasoning about states. Consequently, a decision procedure for classical arithmetic reasoning can be incorporated as a separate module in the structure of the theorem prover.

Interesting Theorems To get a better understanding of interactions between the classical and intuitionistic connect this, we present the following axioms, all of which are provable in our sequent calculus.

$$\begin{array}{rcl@{\qquad}rcl}
\bigcirc\texttt{true} & \iff & \mathbf{1} & \mathbf{0} & \iff & \bigcirc\texttt{false} \\
\bigcirc A \otimes \bigcirc B & \iff & \bigcirc(A \wedge B) & \bigcirc(A \wedge B) & \implies & \bigcirc A \,\&\, \bigcirc B \\
\bigcirc A \oplus \bigcirc B & \implies & \bigcirc(A \vee B) & & &
\end{array}$$

The first three rows illustrate formulas that are provably equivalent. The last two rows illustrate formulas for which the formula on the left implies the formula on the right but not the other way around. For instance, the intuitionistic disjunction of classical assertions implies the classical disjunction of classical formulas. Intuitively, the reason the implication does not hold in the other direction is that the classical disjunction may use the law of the excluded middle in it's proof. Naturally, if it does, there is no way to and guarantee that we can construct a proof of $\bigcirc A$ or construct a proof of $\bigcirc B$. Consequently, the intuitionistic disjunction does not hold.

It is also interesting to consider the proof theory for heap-free formulas, which we represent using Girard's unrestricted modality. The critical axioms here are the structural properties of contraction and weakening.

$$\begin{array}{rcl}
!F & \implies & \mathbf{1} \\
!F & \implies & !F \otimes !F
\end{array}$$

Tensor is also associative and commutative. In separation logic, Reynolds [21] adds specialized axioms for relating the additive conjunction of pure facts to the multiplicative conjunction of pure facts:

$$\begin{array}{rcll}
P \wedge Q & \implies & P * Q & \text{when } P \text{ or } Q \text{ is pure} \\
P * Q & \implies & P \wedge Q & \text{when } P \text{ and } Q \text{ is pure}
\end{array}$$

In our logic, we can prove $!P \otimes !Q \implies !P \,\&\, !Q$ but not the reverse. Only one of $!P$ or $!Q$ may be used in any proof assuming $!P \,\&\, !Q$. We forgo these additional axioms for practical reasons: we wish to reuse a theorem prover for first-order intuitionistic linear logic rather than building a new prover from scratch. One consequence of this choice is that programmers must write invariants consistently in the form $!P \otimes !Q$ as opposed to writing $!P \otimes !Q$ at times and writing $!P \& !Q$ at times. In separation logic, programmers do not have to be so careful as the proof theory will bridge the syntactic gap between specifications for them. It remains to be seen whether this choice actually limits our ability to verify any programs.

2.5 Properties of ILC

We have proven a cut elimination theorem of our logic (Theorem 1). We also proved that the proof theory of our logic is sound with regard to its semantics (Theorem 2).

Theorem 1 (Cut Elimination)

1. *If* $\Omega \,|\, \Gamma, A \;\#\; \Gamma'$ *and* $\Omega \,|\, \Gamma \;\#\; A, \Gamma'$ *then* $\Omega \,|\, \Gamma \;\#\; \Gamma'$.
2. *If* $\Omega \,|\, \Gamma \;\#\; A$ *and* $\Omega \,|\, \Gamma, A \,;\, \Theta \,;\, \Delta \Longrightarrow F$ *then* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F$
3. *If* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \cdot \Longrightarrow F$ *and* $\Omega \,|\, \Gamma \,;\, \Theta, F \,;\, \Delta \Longrightarrow F'$ *then* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F'$.
4. *If* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow F$ *and* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta', F \Longrightarrow F'$ *then* $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta, \Delta' \Longrightarrow F'$.

The semantic judgment for logical contexts is written as: $h \vDash \Gamma; \Theta; \Delta$. It means that store h satisfies all the arithmetic constraints in context Γ and that h contains all the unrestricted resources in context Θ and all the linear resources in context Δ. We define the semantics of the logical contexts as follows:

$h \vDash \Gamma; \Theta; \Delta$ iff $\mathcal{M} \vDash \Gamma$ and $\mathcal{M}; h \vDash (\bigotimes !\,\Theta) \otimes (\bigotimes \Delta)$
where $\mathcal{M}$ is Presburger arithmetic, and $\bigotimes \Delta$ is the resulting formula of tensoring all the formulas in Δ, and $\bigotimes !\,\Theta$ is the formula we get if we wrap $!$ around all the formulas in Θ, and tensor them together.

Theorem 2 (Soundness of Logic Deduction)

1. *If* $\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F$ *and* σ *is a substitution of integers for all the variables in* Ω*, and* $\mathcal{M} \vDash \sigma\Gamma$*, and* $\mathcal{M}; h \vDash \sigma((\bigotimes !\,\Theta) \otimes (\bigotimes \Delta))$ *then* $\mathcal{M}; h \vDash \sigma F$.
2. *If* $\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F$ *and* σ *is a substitution of integers for all the variables* Ω*, and* $h \vDash \sigma\Gamma; \sigma\Delta; \sigma\Delta$*, then* $h \vDash \sigma F$

2.6 A Decidable Fragment: ILC$^-$

Since we intend to use our logic in automated program verification, we would like to find a decidable fragment. The logic we introduced in the previous sections contains Intuitionistic Linear Logic as a sub-logic, so it is clearly undecidable. Fortunately, we have identified a fragment of our logic, ILC$^-$, which is decidable and sufficient to encode the pre- and post-conditions for many programs. One important property of ILC$^-$ is that it is closed under the verification condition generation, which we will present in the next section. In other words, if all the programmer supplied program invariants fall into this fragment, then the whole process of program verification is decidable.

Intuitionistic linear logic without $!$ is decidable since every premise of each rule is strictly smaller than its consequent (by smaller we mean the number of connectives in the sequent decreases[12]). However, we require some occurrences of $!$ and we also need classical arithmetic for our fragment to be useful. The problem with $!$ comes from the *copy* rule, which is listed below.

$$\frac{\Omega \,|\, \Gamma \,;\, \Theta, F \,;\, \Delta, F \Longrightarrow F'}{\Omega \,|\, \Gamma \,;\, \Theta, F \,;\, \Delta \Longrightarrow F'}\ \textit{Copy}$$

If we manage to exclude the *copy* rule from our logic, then we can obtain a decidable fragment. Now the question is how can we eliminate the *copy* rule and still have an expressive logic. The function of the *copy* rule is to use the assumptions in the unrestricted context. After we remove it from our logic, we lack the ability to use the assumptions in the unrestricted context Θ. The first step of solving this problem is to add another init rule, so that we can use the atomic assumptions in Θ.

$$\frac{}{\Omega \,|\, \Gamma \,;\, \Theta, P \,;\, \cdot \ \overset{-}{\Longrightarrow}\ P}\ \textit{U-Init} \qquad \frac{}{\Omega \,|\, \Gamma \,;\, \Theta, F \,;\, \cdot \ \overset{-}{\Longrightarrow}\ F}\ \textit{U-Init'}$$

We cannot add the more general rule *U-Init'*, because we won't be able to prove cut-elimination. Adding *U-Init* is not enough, because we still cannot decompose connectives in the unrestricted context. The next step is to put syntactic restrictions on logical formulas so that in this restricted fragment of ILC, we don't need rules to decompose connectives in the unrestricted context. The restriction is that all the formulas appear in the unrestricted intuitionistic context Θ are in D_u, all the formulas that appear in the linear intuitionistic context Δ are in D_l, and all the formulas that appear on the right-hand side of the sequent are in G. Each syntactic class in this decidable fragment is defined as follows:

Forms in Intuitionistic Unrestricted Ctx	D_u	$::=$	Ps
Forms in Intuitionistic Linear Ctx	D_l	$::=$	$Ps \mid {!}\,Ps \mid {!} \bigcirc A \mid \mathbf{1} \mid D_l \otimes D_l' \mid \top \mid D_l \,\&\, D_l'$
			$\mid \mathbf{0} \mid D_l \oplus D_l' \mid \exists x.D_l \mid \forall x.D_l$
Forms	G	$::=$	$Ps \mid \mathbf{1} \mid G_1 \otimes G_2 \mid D_l \multimap G \mid \top \mid G_1 \,\&\, G_2$
			$\mid \mathbf{0} \mid G_1 \oplus G_2 \mid {!}\,G \mid \exists b.G \mid \forall b.G \mid \bigcirc A$

Now the only formula that can appear in the unrestricted intuitionistic context are store predicates Ps. Consequently, the *U-Init* rule is enough to use the assumptions in the unrestricted context. Notice that we do not include $G \multimap D_l$ in D_l. That's because the function postconditions belong to this category, yet they show up both in the negative and positive position during the verification condition generation (refer to Section 3).

Lastly, we add the $! \bigcirc L$ rule to make sure that $\bigcirc A$ won't be trapped in the Θ context.

$$\frac{\Omega \,|\, \Gamma, A \,;\, \Theta \,;\, \Delta \overset{-}{\Longrightarrow} F}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta, {!} \bigcirc A \overset{-}{\Longrightarrow} F} \; {!} \bigcirc L$$

In summary, we obtain a fragment of our logic, ILC$^-$, by restricting the syntax of formulas, and replacing the *copy* rule with *U-init* and $! \bigcirc L$ rules. We proved that with the syntactic restriction, the sequent rules with *U-Init* and $! \bigcirc L$ are sound and complete with regard to the original sequent rules we developed in the previous section.

Theorem 3 (Soundness & Completeness of $\overset{-}{\Longrightarrow}$)
$\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{-}{\Longrightarrow} G$ iff $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow G$ provided that all the formulas in Γ are in A, all the formulas in Θ are in D_u, and all the formulas in Δ are in D_l.

In order to show that ILC$^-$ is decidable, we define an equivalent alternative calculus that collects classical constraints during proof search and defers them to the end. We show that the alternative calculus, which we call the linear residuation calculus, is decidable. Judgments in residuation calculus have the form $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{r}{\Longrightarrow} F \backslash Rs$, where Rs is a residuation formula:

Residuation Formulas	Rs	$::=$	$A \mid Rs_1 \wedge Rs_2 \mid \exists x.Rs \mid \forall x.Rs$

The following two theorems shows that the residuation calculus is equivalent to the original sequent calculus for ILC$^-$.

Theorem 5 (Soundness of Residuation)
If $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{r}{\Longrightarrow} F \backslash Rs$ and for any ground substitution ρ for Ω, $\mathcal{M} \vDash \rho Rs$ then $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{-}{\Longrightarrow} F$.

Theorem 6 (Completeness of Residuation)
If $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{-}{\Longrightarrow} F$, and ρ is a ground substitution for Ω then $\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta \overset{r}{\Longrightarrow} F \backslash Rs$ for some Rs and $\mathcal{M} \vDash \rho Rs$

The provability of ILC$^-$ is reduced to the validity of the residual formula and the decidability of the residuation calculus. Residuation formulas are simply formulas in Presburger Arithmetic and therefore their validity are decidable.

Lemma 8
Residuation calculus is decidable.

PROOF. By examination of the proof rules. The premise of each rule is strictly smaller than its conclusion, so there are finite number of possible proof trees altogether (a similar argument was made in Lincoln's paper on the decidability properties of propositional linear logic [12]). □

Theorem 9 (Decidability)
ILC$^-$ is Decidable

PROOF. By the result of the completeness and soundness of residuation calculus, the provability of ILC$^-$ is reduced to the validity of the residual formula and the decidability of the residuation calculus. By Lemma 8, the validity of the residual formula and the decidability of the residuation calculus are both decidable, so ILC$^-$ is Decidable. □

3. Verifying Pointer Programs

In this section, we introduce a simple imperative language with control-flow, mutable references, and functions. The language also contains assertions in ILC, which we use to define a set of syntax-directed verification condition generation rules. We present syntax-directed rules and give three examples to illustrate how to verify programs using our logic. At the end of this section, we present our main technical result, a proof of soundness of our verification condition generation.

3.1 Syntax

The syntactic constructs of our language are listed below. We use E to range over integer expressions, B to range over boolean expressions. R ranges over condition expressions used in while loops. The condition expressions R appear a little strange. They allow conditions in while loops to reference the store. In order to generate verification conditions properly from the expressions, we require them to be in A-Normal form. Naturally, all implementations would allow programmers to write ordinary expressions and then unwind them to A-Normal form for verification. We do not show this simple unwinding process. Commands are in A normal form too. We have commands for allocation, deallocation, variable binding, dereference, assignment, function call, sequencing, while loop, if branching, skip, and return. We assume that the while loop is annotate with loop invariant I. We use C_{rt} to range over commands that can appear in function body. The major difference between ordinary commands and function bodies is that function body includes a return command as the last command. We use ι to range over condition expressions and commands. A program is composed of a sequence of mutually recursive function declarations followed by a command. Here the functions only take one argument, but it is fairly easy to extend the language to allow functions to take multiple arguments.

$$
\begin{array}{llll}
\textit{Int Exps} & E & ::= & n \mid x \mid E + E \mid -E \\
\textit{Boolean Exps} & B & ::= & \texttt{true} \mid \texttt{false} \mid E_1 = E_2 \mid E_1 < E_2 \mid B_1 \wedge B_2 \mid \neg B \mid B_1 \vee B_2 \\
\textit{Condition Exps} & R & ::= & B \mid \texttt{let}\ x\ = !E\ \texttt{in}\ R\ \texttt{end} \\
\textit{Command} & C & ::= & \texttt{let}\ x\ =\ \texttt{new}(E)\ \texttt{in}\ C\ \texttt{end} \mid \texttt{free}(E) \\
 & & & \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ C\ \texttt{end} \mid \texttt{let}\ x\ = !E\ \texttt{in}\ C\ \texttt{end} \\
 & & & \mid E_1 := E_2 \mid \texttt{let}\ x\ =\ f(\,E\,)\ \texttt{in}\ C\ \texttt{end} \mid C_1\,;\ C_2 \\
 & & & \mid \texttt{while}_{[I]}\ R\ \texttt{do}\ C \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{skip} \mid \texttt{return}\ E \\
\textit{Instructions} & \iota & ::= & C \mid R \\
\textit{Fun Dec} & fd & ::= & \texttt{fun}\ f(\,x\,)\ =\ C_{rt} \\
\textit{Program} & p & ::= & fd \ldots fd\ \texttt{in}\ C
\end{array}
$$

We also need the following runtime structures for defining operational semantics for this language.

$$
\begin{array}{llll}
\textit{Instructions} & \iota & ::= & \cdots \mid \bullet \\
\textit{Code Context} & \Psi & ::= & \cdot \mid \Psi, f \mapsto (\,a\,)\ C_{rt}\ [\Delta]\ \{Pre\}\ \{\forall ret.Post\} \\
\textit{Evaluation Context} & ctx & ::= & [\,]\,;\ C \mid \texttt{let}\ x\ =\ [\,]\ \texttt{in}\ C\ \texttt{end} \mid \texttt{while}_{[I]}\ [\,]\ R\ \texttt{do}\ C \\
\textit{Control Stack} & S & ::= & S_c \mid S_{while} \\
\textit{Command Control Stack} & S_c & ::= & S_{seq} \mid S_{funcall} \\
\textit{Seq Control Stack} & S_{seq} & ::= & \cdot \mid ([\,]\,;\ C) \triangleright S_c \\
\textit{Fun call Control Stack} & S_{funcall} & ::= & (\texttt{let}\ x\ =\ [\,]\ \texttt{in}\ C\ \texttt{end}) \triangleright S_c \\
\textit{While Control Stack} & S_{while} & ::= & (\texttt{while}_{[I]}\ [\,]\ R\ \texttt{do}\ C) \triangleright S_c
\end{array}
$$

First, we extend the instructions with a runtime empty instruction $\bullet$. It indicates the termination of certain commands. Code context Ψ maps function name to its parameter name, function body, its precondition Pre, and postcondition $\forall ret.Post$. All the free variables in Pre and $\forall ret.Post$ are in Δ. The variable ret in

$(S, h, \iota) \overset{\Psi}{\longmapsto} (S', h', \iota')$
$(S, h, \texttt{let}\ x = \texttt{new}(E)\ \texttt{in}\ C\ \texttt{end}) \overset{\Psi}{\longmapsto} (S, h \uplus (\ell \mapsto [\![E]\!]), C[\ell/x])\ \ell \notin dom(h)$
$(S, h, \texttt{let}\ x = E\ \texttt{in}\ C\ \texttt{end}) \overset{\Psi}{\longmapsto} (S, h, C[[\![E]\!]/x])$
$(S, h, E := E') \overset{\Psi}{\longmapsto} (S, h[\,[\![E]\!] := [\![E']\!]\,], \bullet)$ if $[\![E]\!] \in dom(h)$
$(S, h, \texttt{free}(E)) \overset{\Psi}{\longmapsto} (S, h', \bullet)$ where $h = h' \uplus ([\![E]\!] \mapsto v)$
$(S, h, \texttt{let}\ x = !E\ \texttt{in}\ \iota\ \texttt{end}) \overset{\Psi}{\longmapsto} (S, h, \iota[h([\![E]\!])/x])$ if $[\![E]\!] \in dom(h)$
$(S, h, \texttt{let}\ x = f(\,E\,)\ \texttt{in}\ C\ \texttt{end})$ $\overset{\Psi}{\longmapsto} (\texttt{let}\ x = [\,]\ \texttt{in}\ C\ \texttt{end} \triangleright S, h, C_{rt}[[\![E]\!]/a])$ where $\Psi(f) = (\,a\,)\ C_{rt}\ [\Delta]\ \{Pre\}\ \{\forall ret.Post\}$
$(\texttt{let}\ x = [\,]\ \texttt{in}\ C\ \texttt{end} \triangleright S, h, \texttt{return}\ E) \overset{\Psi}{\longmapsto} (S, h, C[[\![E]\!]/x])$
$(S, h, (C\,;\ C')) \overset{\Psi}{\longmapsto} (([\,]\,;\ C') \triangleright S, h, C)$
$(([\,]\,;\ C) \triangleright S, h, \bullet) \overset{\Psi}{\longmapsto} (S, h, C)$
$(S, h, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2) \overset{\Psi}{\longmapsto} (S, h, C_1)$ if $[\![B]\!] = \texttt{true}$
$(S, h, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2) \overset{\Psi}{\longmapsto} (S, h, C2)$ if $[\![B]\!] = \texttt{false}$
$(S, h, \texttt{while}_{[I]}\ R\ \texttt{do}\ C) \overset{\Psi}{\longmapsto} (\texttt{while}_{[I]}\ [\,]\ R\ \texttt{do}\ C \triangleright S, h, R)$
$((\texttt{while}_{[I]}\ [\,]\ R\ \texttt{do}\ C) \triangleright S, h, B) \overset{\Psi}{\longmapsto} (S, h, \bullet)$ if $[\![B]\!] = \texttt{false}$
$((\texttt{while}_{[I]}\ [\,]\ R\ \texttt{do}\ C) \triangleright S, h, B) \overset{\Psi}{\longmapsto} (S, h, (C\,;\ \texttt{while}_{[I]}\ R\ \texttt{do}\ C))$ if $[\![B]\!] = \texttt{true}$

Figure 4. Operational Semantics

the postcondition is a special variable referring to the return value of the function. The precondition of a function describes the state where the function is safe to be called, and the postcondition specifies the state after the function returns. The evaluation context ctx specifies the order of evaluation. The hole [ ] in an evaluation context is the place holder for the instruction currently being evaluated. An evaluation context can be a sequencing context waiting for the result of its first instruction, or a function call context waiting for the return from a function call, or a while loop context waiting for the the evaluation of the condition expression. A control stack is a sequence of evaluation contexts (also called frames). A control stack can either be a $while\ control\ stack$, or a $command\ control\ stack$. A $command\ control\ stack$ is a sequence of function call and sequence evaluation contexts. A $while\ control\ stack$ is a while evaluation context on top of a $command\ control\ stack$.

3.2 Operational Semantics

First, we define the denotation of integer expressions and boolean expressions, and they are used in the definition of the operational semantics of our language. The denotation of the integer expressions is the same as the one defined in Section 2.3. The denotation of a boolean expression B is:

$[\![B]\!] = \texttt{true}$ iff $\mathcal{M} \vDash B$ $\qquad [\![B]\!] = \texttt{false}$ iff $\mathcal{M} \nvDash B$ $\qquad$ where $\mathcal{M}$ is the arithmetic model.

A program state consists of a control stack S, a store (or a heap) h, and an instruction ι that is being evaluated[2]. We use $(S, h, \iota) \overset{\Psi}{\longmapsto} (S', h', \iota')$ to denote the small step operational semantics. The operational semantics rules are listed in Figure 4. The rules are straightforward. One thing is worth mentioning is the evaluation of function calls and returns. When we see a function call, we put the function call context on top of the control stack, look up the function body from the code context Ψ, then substitute the real argument E for a in the function body, and start evaluating the function body. Upon function return, we pop one frame off the control stack, and substitute the return value for x in the rest of the instruction, and continue the evaluation.

[2] Because the variables are bound in our language, and there is no imperative assignment to variables, we do not need a stack to map variables to values.

3.3 Verification Condition Generation

We explain the proof rules of our extended Hoare logic in this section. We assume that all the functions are annotate with preconditions and postconditions, and while loops are annotated with the loop invariants. We verify an entire program as follows: first we verify the specification of each function, assuming the assertions on all other functions are valid. Second, we verify that the command in the main program complies with its specifications. In verifying each function, we use backward-reasoning. As we go through the function body bottom up, we build a verification condition from the postcondition along the way. When we reach the first command of the function body, we need to prove that the precondition given in the function specification can logically entail the verification condition we generated. If that is the case and our verification condition generation rules are sound, then whenever the function is called in a state described by the precondition of the function, upon returning from this function, the postcondition will be established. We have not tackled the question of whether or not our verification conditions are weakest preconditions. Weakest preconditions are neither necessary (many program properties can be verified without weakest preconditions) nor sufficient (the logical proof theory must still be strong and programmers must be able to specify their requirements concisely) for practical program verification. However, it would be a nice theoretical property to have. We collect all the function specifications into the code context Ψ, and Ψ is always available when we analyze the program. There are four judgments involved in verification:

$\Delta \vdash (\exists x_1 \ldots \exists x_n, F, A)R$	There exists values for variable $x_1 \cdots x_n$ such that the precondition of executing R is F, and the core boolean expression in R is A
$\Delta \vdash^{\Psi} \{\, P \,\}\, C \,\{\, Q \,\}$	The precondition of C is P, and the postcondition is Q.
$\vdash \Psi\ \mathsf{ok}$	The code context is well formed.

Condition Expressions Intuitively, we can interpret the judgment $\Delta \vdash (\exists x_1 \ldots \exists x_n, F, A)R$ as such: if the current state satisfies F, then it is safe to execute R, and eventually R is evaluated to boolean expression A. Variables $x_1 \cdots x_n$ are free in F and A, and Δ contains all the other free variables in R, F, and A. The inference rules for condition expressions are listed below:

$$\frac{}{\Delta \vdash (\cdot, \top, B)\ B}\ \textit{boolean exp}$$

$$\frac{\Delta, x \vdash (\exists x_1 \ldots \exists x_n, F, A)\ R}{\Delta \vdash (\exists x_0.\exists x_1 \ldots \exists x_n,\ (\,(E \Rightarrow x_0)\, \otimes \top) \,\&\, F[x_0/x],\ A[x_0/x])\ \texttt{let}\ x\ = !E\ \texttt{in}\ R\ \texttt{end}}\ \textit{bind}$$

The judgment is inductively defined over the structure of the condition expression. If the condition expression is a boolean expression B already, then the precondition is $\top$, meaning no condition is needed, and the resulting boolean expression is B itself. When the condition expression R is $\texttt{let}\ x\ = !E\ \texttt{in}\ R'\ \texttt{end}$, we first derive that the condition expression R' has precondition F, its core boolean expression is A. The precondition of R has to make sure that E is a valid location in the store $(E \Rightarrow x_0)$, where x_0 is a fresh existentially quantified variable. After dereferencing location E, x is bound to the contents of E. The precondition of R also needs to guarantee that $R'[x_0/x]$ is safe to execute, in other words $F[x_0/x]$ holds. Finally, the boolean expression is $A[x_0/x]$. For example, $a \vdash (\exists x_0, (\,(a \Rightarrow x_0)\, \otimes \top)\&\top, x_0 > 0)\texttt{let}\ x\ = !a\ \texttt{in}\ x > 0\ \texttt{end}$

Commands The verification condition generation rules are backward reasoning rules. Given postconditions, we will calculate verification conditions for the commands. We explain each rule in detail. Note that all the rules are syntax directed. The pre- and post-conditions in the function specification must be in D_l (defined in Section 2.6). If each postcondition Q is in ILC$^-$ and the loop invariant is in ILC$^-$ then each rule generates a precondition in ILC$^-$. Most of the rules are identical to O'Hearn's weakest precondition generation [10] except that $*$ is replaced by $\otimes$, $-\!*$ by $\multimap$, and $\wedge$ by $\&$.

- Variable Binding

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, P[E/x] \,\}\ \texttt{let}\ x\ =\ E\ \texttt{in}\ C\ \texttt{end}\ \{\, Q \,\}}\ \textit{bind}$$

Variable x is bound to E in command C, so the precondition of this bind command is the precondition of C with x substituted with E.

- Allocation

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, \forall y.\, (y \Rightarrow E) \multimap P[y/x] \,\}\, \texttt{let}\; x \;=\; \texttt{new}(E)\; \texttt{in}\; C\; \texttt{end} \,\{\, Q \,\}}\; \textit{new}$$

The *new* command allocates a new cell on the heap, stores E into the new cell, and binds x to the address of the new cell. The precondition describes a heap that is waiting for the new piece. After merging with the newly allocated piece, it satisfies the precondition of C with x substituted with the address of the new cell.

- Deallocation

$$\frac{}{\Delta \vdash \{\exists y.\, (E \Rightarrow y) \otimes Q\}\; \texttt{free}(E)\; \{Q\}}\; \textit{free}$$

The *free* command deallocates the cell whose address is E. Before executing *free*, we assert that E indeed is an allocated cell on the heap ($\exists y.\, (E \Rightarrow y)$), and is separate from the rest of the heap described by Q.

- Dereference

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, \exists y.(\, (E \Rightarrow y) \otimes \top) \,\&\, P[y/x] \,\}\, \texttt{let}\; x \;=\, !E\; \texttt{in}\; C\; \texttt{end} \,\{\, Q \,\}}\; \textit{deref}$$

The dereference command looks up the contents of address E in the heap and binds x to it. Similar to the *free* command, $(E \Rightarrow y)$ proves that E is an allocated location on the heap. After dereferencing E, we execute command C in the same heap; the additive conjunction $\&$ conveys the idea of sharing.

- Assignment

$$\frac{}{\Delta \vdash \{\, \exists y.\, (E_1 \Rightarrow y) \otimes (\, (E_1 \Rightarrow E_2) \multimap Q) \,\}\, E_1 \;:=\; E_2 \,\{\, Q \,\}}\; \textit{assignment}$$

The assignment command updates the cell at address E_1 with the value of E_2. The precondition of this command asserts that the heap comprises of two separate parts: one that contains cell E_1, the other that waits for the update.

- Sequencing

$$\frac{\Delta \vdash \{\, P \,\}\, C_1 \,\{\, P' \,\} \quad \Delta \vdash \{\, P' \,\}\, C_2 \,\{\, Q \,\}}{\Delta \vdash \{\, P \,\}\, C_1\, ;\; C_2 \,\{\, Q \,\}}\; \textit{Seq}$$

- If Statement

$$\frac{\Delta \vdash \{\, P_1 \,\}\, C_1 \,\{\, Q \,\} \quad \Delta \vdash \{\, P_2 \,\}\, C_2 \,\{\, Q \,\}}{\Delta \vdash \{\, (\,!\bigcirc B \multimap P_1) \,\&\, (\,!\bigcirc \neg B \multimap P_2) \,\}\, \texttt{if}\; B\; \texttt{then}\; C_1\; \texttt{else}\; C_2 \,\{\, Q \,\}}\; \textit{if}$$

The *if* instruction branches on boolean expression B. The precondition for *if* says that if B is true then the precondition of the true branch holds; otherwise the precondition of the false branch holds. The additive conjunction demonstrates the sharing of two possible descriptions of the same heap. Note that before the execution of the if statement either B is true or B is false, so the precondition of the branch that is not taken is proved using the absurdity rule. We will give a concrete example in the next section.

- While Loop

$$\frac{\Delta \vdash \{\, P \,\}\, C \,\{\, I \,\} \quad \Delta \vdash (\exists x_1 \ldots \exists x_n, F, B) \;\; R}{\Delta \vdash \left\{ \begin{array}{l} (\exists x_1 \ldots \exists x_n.F\&(!\bigcirc \neg B \multimap Q)\&(!\bigcirc B \multimap P)) \\ \otimes\,!\,(I \multimap (\exists x_1 \ldots \exists x_n.F\&(!\bigcirc B \multimap P)\&(!\bigcirc \neg B \multimap Q)) \end{array} \right\}\, \texttt{while}_{[I]}\; R\; \texttt{do}\; C \,\{\, Q \,\}}\; \textit{while}$$

The while loop is annotated with the loop invariant I. A while loop either executes the loop body or exits the loop depending on the condition expression R. There are two parts in the precondition of a while loop. The first part $\exists x_1 \ldots \exists x_n.F\&(!\bigcirc \neg B \multimap Q)\&(!\bigcirc B \multimap P)$ asserts that when we execute the loop for the first time, the precondition for evaluating the condition expression, F, must hold, and if the condition is not true, then the postcondition Q must hold, otherwise we will execute the loop body, so the precondition P for C must hold. The second part $!\,(I \multimap (\exists x_1 \ldots \exists x_n.F\&(!\bigcirc B \multimap P)\&(!\bigcirc \neg B \multimap Q))$ asserts that each time we re-enter the loop, the condition for entering the loop holds. Notice that the second formula is wrapped by an unrestricted connective $!$. This implies that this invariant cannot depend upon the current heap state. This is a critical criterion as the heap state may be different each time around the loop. Notice also that $!$

surrounds a formula that falls in ILC$^-$. We have not actually seen O'Hearn gave a verification condition for while loops. We assume that it would be similar to the one we present here.

- Function Call

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q) \quad \Psi(f) = (\, a \,)\, C_{rt}\, [\Delta_f]\, \{Pre\}\, \{\forall ret.Post\}}{\Delta \vdash \begin{array}{l} \{\, \exists \Delta_f.Pre[E/a] \otimes (\forall ret.Post[E/a] \multimap P[ret/x]) \,\} \\ \texttt{let}\ x\ =\ f(\, E\,)\ \texttt{in}\ C\ \texttt{end}\ \{\, Q \,\} \end{array}} \textit{ fun call}$$

The verification condition of the function call has similar formulation as the assignment command. The difference is that we are not just updating one cell on the heap; we are updating the footprint of the function. First we look up the specification of f in the code context Ψ. The specification $(\, a \,)\, C_{rt}\, [\Delta_f]\, \{Pre\}\, \{\forall ret.Post\}$ tells us that a is the parameter name, Pre is the precondition for executing the function body, $\forall ret.Post$ is the postcondition, and Δ_f contains all the free variables in Pre and $\forall ret.Post$.

The verification condition asserts that the heap consists of two separate sub-heaps. One sub-heap satisfies the precondition of the function $Pre[E/a]$, for we are calling f with real argument E. We assume that the specs of f is valid, so we believe that the postcondition $\forall ret.Post$ holds after f returns. After we update the heap with the heap described by the postcondition, the precondition for the next command C should hold.

- Function Return

$$\frac{}{\Delta \vdash \{\, Q[E/ret] \,\}\ \texttt{return}\, E\ \{\, \forall ret.Q \,\}} \textit{ return}$$

Code Context The code context is well formed if all the function specifications in the code context is well formed.

$$\frac{\begin{array}{l} \forall f \in dom(\Psi), \Psi(f) = (\, a \,)\, C_{rt}\, [\Delta]\, \{Pre\}\, \{\forall ret.Post\}, \\ \Delta \vdash \{\, Pre \,\}\, C_{rt}\, \{\forall ret.Post\,\} \quad \Delta \notin FV(C_{rt}) \end{array}}{\vdash \Psi\ \mathsf{ok}}$$

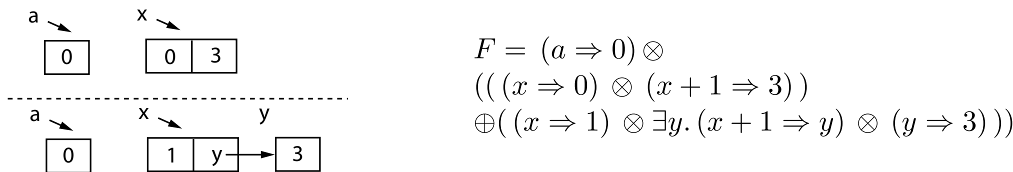
3.4 Examples

In this section, we will give three examples to demonstrate how we verify programs using the verification condition generation rules defined in the previous section. Notice that in all three examples, our invariants fall into ILC$^-$ so verification is decidable.

If Branching This example illustrates how the rules of dereference, assignment, and if branching work. It also gives hints about how to verify tagged unions, a datastructure that appears frequently in imperative code and in code generated for ML datatypes.

The store is illustrated in Figure 5. Location a contains 0 independent of the contents of x. Depending on the contents of location x, there are two possibilities of the store. If x contains integer 0, then the location next to x contains integer 3; if x contains an integer other than 0, then the location next to x contains another location y, and y contains integer 3. The first case is illustrated by the picture above the dashed line, and the second case is illustrated by the picture below the line. Formula F describes the store h. We use addictive disjunction to describe the two cases.

The following piece of code branches on the contents of x. The true branch looks up the value stored in location $x+1$, and stores it into a; the false branch looks up the value stored in location y, which is the contents of $x+1$, and stores the value into a. At the merge point of the branch, a should contains 3. The postcondition Q of this code is $(a \Rightarrow 3) \otimes \top$ where $\top$ describes other store states that we don't care about.



$$\begin{array}{l} F = (a \Rightarrow 0) \otimes \\ (((x \Rightarrow 0) \otimes (x+1 \Rightarrow 3)) \\ \oplus((x \Rightarrow 1) \otimes \exists y.(x+1 \Rightarrow y) \otimes (y \Rightarrow 3))) \end{array}$$

Figure 5. Example1

$\{F \ = (a \Rightarrow 0) \otimes (((x \Rightarrow 0) \otimes (x+1 \Rightarrow 3)) \oplus ((x \Rightarrow 1) \otimes \exists\ y. (x+1 \Rightarrow y) \otimes (y \Rightarrow 3)))\}$
`let` $t \ = !x$ `in`
`if` $(t = 0)$
`then let` $s \ = !(x+1)$ `in` $a := s$ `end`
`else let` $s \ = !(x+1)$ `in`
    `let` $r \ = !s$ `in`
    $a := r$ `end`
    `end`
`end`
$\{(a \Rightarrow 3) \otimes \ \top\}$

We annotate the code with the verification conditions we generate according to our rules.

$\{Pre \ = \ \ \exists\ z.\ ((x \Rightarrow z) \otimes \top)\ \&\ (!\bigcirc(z=0) \ \multimap \ P_1)\ \&\ (!\bigcirc\neg(z=0) \multimap \ P_2)\}$
`let` $t \ = !x$ `in`
$\{(!\bigcirc(t=0) \ \multimap \ P_1)\ \&\ (!\bigcirc\neg(t=0) \multimap \ P_2)\}$
`if` $(t=0)$
`then`
$\{\ P_1 \ = \ \exists\ u.((x+1 \Rightarrow u) \otimes \top) \& \exists\ w. (a \Rightarrow w) \otimes ((a \Rightarrow u) \multimap ((a \Rightarrow 3) \otimes \top))\}$
`let` $s \ = !(x+1)$ `in`
$\{\exists\ w. (a \Rightarrow w) \otimes ((a \Rightarrow s) \multimap ((a \Rightarrow 3) \otimes \top))\}$
$a := s$ `end`
`else`
$\{P_2 \ = \ \exists\ u.((x+1 \Rightarrow u) \otimes \top) \& \exists\ v.((u \Rightarrow v) \otimes \top) \& \exists\ w. (a \Rightarrow w) \otimes ((a \Rightarrow v) \multimap ((a \Rightarrow 3) \otimes \top))\}$
`let` $s \ = !(x+1)$ `in`
$\{\exists\ v.\ ((s \Rightarrow v) \ \otimes \ \top)\ \&\ \exists\ w.\ (a \Rightarrow w) \ \otimes \ ((a \Rightarrow v) \ \multimap \ ((a \Rightarrow 3) \otimes \ \top))\ \}$
`let` $r \ = !s$ `in`
$\{\exists\ w. (a \Rightarrow w) \otimes ((a \Rightarrow r) \multimap ((a \Rightarrow 3) \otimes \top))\}$
$a := r$ `end end`
`end`
$\{\ \ (a \Rightarrow 3) \otimes \top\}$

In order to prove that this code is correct, we need to prove that the verification condition we generated can be derived from the describing formula of the current state: $x, a \,|\, \cdot; \cdot; F \Longrightarrow Pre$.

According to our sequent rules, after we apply left rules, we need to prove the following two subgoals:

$$x, a \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow Pre$$
$$x, a, y \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 1),\ (x+1 \Rightarrow y),\ (y \Rightarrow 3) \ \Longrightarrow Pre$$

Let us examine the proof of first subgoal, the second one is very similar.

$$x, a \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow \exists z.((x \Rightarrow z) \otimes \top) \& (!\bigcirc(z=0) \multimap P_1) \& (!\bigcirc\neg(z=0) \multimap P_2)$$

It is obvious that the existential variable z is instantiated by 0. After we apply the $\&R$ rule twice, we obtain the following three subgoals:

$$x, a \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow ((x \Rightarrow 0) \otimes \top)$$
$$x, a \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow (!\bigcirc(0=0)) \multimap P_1$$
$$x, a \,|\, \cdot; \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow (!\bigcirc\neg(0=0)) \multimap P_2$$

The interesting one is the last one. After applying $!\bigcirc L$ rule, we have

$$x, a \,|\, \neg(0=0); \cdot;\ (a \Rightarrow 0),\ (x \Rightarrow 0),\ (x+1 \Rightarrow 3) \ \Longrightarrow P_2$$

Obviously, the resources in the linear context is not enough to prove P_2, which requires $x + 1$ to contain another location. However, we have a contradiction in the classical context $\neg(0 = 0)$. We prove the above subgoal using the absurdity rule. This is the situation where the false branch is not taken, so we cannot establish the precondition required by that branch. Instead, we prove the precondition of the false branch by contradiction.

While Loop The second example involves a simple while loop that computes the sum between integer 1 and 5 inclusive. The sum is stored in location s, and the loop induction variable is stored in location a. The post condition of this code is $(s \Rightarrow 15) \otimes (a \Rightarrow 0)$. The precondition for this program requires that s and a are allocated on the heap: $F = \exists u.\exists v.(a \Rightarrow u) \otimes (s \Rightarrow v)$. The loop invariant I is $\exists x.\exists y.(s \Rightarrow x) \otimes (a \Rightarrow y) \otimes ! \bigcirc (x = \sum_{i=y+1}^{5} i) \otimes (! \bigcirc (y \geq 0)) \otimes (! \bigcirc (y \leq 5))$.

```
while[I] (let x = !a in x > 0 end)
do  let x = !a
    in let y = !s
       in s := y + x
       end;
       a := x - 1
    end
```

Function Interfaces The last example involves function calls. In the previous examples, the heap is the only state we keep track of. In this example, we show that we can keep track of other linear resources such as file handles and sockets as well. This example comes from Vault [6]. Vault uses a linear type system to keep track of program states. The basic idea of Vault is that each resource is guarded with a unique key, and the state of the key is tracked linearly. Here, we use Hoare Logic to verify safety properties of the program as opposed to a type system. We take the headers of the functions in the socket library and annotate them with pre/postconditions, and we show how to verify programs that call into the library. We don't consider how to verify the library implementation in this example. In order to write this example, we extend our logic with the following syntactic constructs:

Keys	K	$::=$	$K_1 \mid \cdots$
Types	τ	$::=$	$\textbf{socket}(K)$
Socket States	st	$::=$	$\texttt{raw} \mid \texttt{named} \mid \texttt{listening} \mid \texttt{ready}$
State Predicates	Ps	$::=$	$\cdots \mid K@st$
Heap Free Predicates	P	$::=$	$E : \textbf{socket}(K)$

We assume that there is a pool of unique keys to guard the resources. We use K to range over the keys. We use τ to range over types. For this example sockets are the only type of resource. Each socket is guarded by a key K. st ranges over socket states. We add a new state predicate $K@st$, which describes that key K currently has state st. We also add a new heap free predicate $(E : \textbf{socket}(K))$, which means that expression E is a socket and guarded by key K. The theorem prover need not be informed in advance the special semantics properties of $(E : \textbf{socket}(K))$ (i.e. the fact it is heap free). It is syntactically obvious since it is surrounded by $!$.

This is a simplified example. We focus on the state changes of the sockets and omit details like error handling. We annotate the functions as below. All the free variables in the pre- and post-condition are in Δ. Function $socket$ creates and returns a new raw socket. Function $bind$, $listen$ changes the state of a socket from "raw" to "named", from "named" to "listening" respectively. Function $accept$ creates a new socket at "ready" states. Function $receive$ receives data from a socket in "ready" state. Function $close$ disposes the key guarding the socket, so it can never be used again.

Δ	Precondition	fun name, args	Postcondition
$[\cdot]$	$\{\mathbf{1}\}$	$socket(d, c, i)$	$\{\forall ret.\exists K.K@\texttt{raw} \otimes !(ret : \textbf{socket}(K))\}$
$[K]$	$\{K@\texttt{raw}\}$	$bind(sk, sa)$	$\{K@\texttt{named}\}$
$[K]$	$\{K@\texttt{named} \otimes !(sk : \textbf{socket}(S))\}$	$listen(sk, i)$	$\{K@\texttt{listening}\}$
$[K]$	$\{K@\texttt{listening} \otimes !(sk : \textbf{socket}(K))\}$	$accept(sk, sa)$	$\{\forall ret.\exists N.!(ret : \textbf{socket}(N))$ $\otimes N@\texttt{ready} \otimes K@\texttt{listening}\}$
$[K]$	$\{K@\texttt{ready} \otimes !(sk : \textbf{socket}(K))\}$	$receive(sk, b)$	$\{K@\texttt{ready}\}$
$[K, s]$	$\{K@s \otimes !(sk : \textbf{socket}(K))\}$	$close(sk)$	$\{\mathbf{1}\}$

The following code is not correct because we cannot accept connections on a raw socket. We can detect this error by computing (decidable) verification condition and finding it unprovable.

```
let mysocket = socket('UNIX, 'INET, 0) in
let y = listen(mysocket, 0) in ...
```

Let's try to verify this code. The postcondition is $\exists k.\,!\,(mysocket : \mathbf{socket}(k)) \otimes k@\texttt{listening}$.

$\{Pre \;=\; 1 \;\otimes (\forall\; ret.((\exists\; k''.(k''@\texttt{raw}) \otimes !\,(ret : \mathbf{socket}(k''))) \multimap F[ret/mysocket]))\}$
`let mysocket = socket('UNIX, 'INET, 0) in`
$\{F = \exists\; k'.\; (k'@\texttt{named} \;\otimes\; !\,(mysocket : \mathbf{socket}(k'))) \otimes \;\; (k'@\texttt{listening} \multimap \;\; Q)\}$
`let y =listen(mysocket, 0) in`
$\{Q = \exists\; k.\;\; !\,(mysocket : \mathbf{socket}(k)) \;\;\otimes\;\; k@\texttt{listening}\}$

When attempting to prove $\cdot \,|\, \cdot; \cdot; \mathbf{1} \Longrightarrow Pre$. The proof will fail because the following goal is unprovable.

$$ret, k \,|\, \cdot; ret : \mathbf{socket}(k); k@\texttt{raw} \Longrightarrow k@\texttt{listening}$$

.

3.5 Soundness of Verification Generation

We say a control stack S and a heap h is safe for an instruction ι for n steps, if the program state (S, h, ι) can take n steps:

(S, h) is safe for ι for n steps iff there exist S', h', and ι' such that $(S, h, \iota) \overset{\Psi}{\longmapsto}{}^{n} (S', h', \iota')$.

We proved the monotonicity lemma and the frame lemma. The Monotonicity lemma means that if an instruction is safe on a smaller heap, then it is safe on a larger heap. The Frame Property shows that a program's footprint is local.

Lemma 12 (Monotonicity)
For all n, $n \geq 0$, if (S, h) is safe for ι for n steps, then $(S, h \uplus h')$ is safe for ι for n steps.

Lemma 13 (Frame Property)
For all n, $n \geq 0$, if (S, h) is safe for ι for n steps, and $(S, h \uplus h_1, \iota) \overset{\Psi}{\longmapsto}{}^{n} (S', h', \iota')$, then $h' = h'' \uplus h_1$, and $(S, h, \iota) \overset{\Psi}{\longmapsto}{}^{n} (S', h'', \iota')$.

Finally, we proved the Safety Theorem. It shows that the rules for verification generation are sound with regard to the semantics of the language. In order to prove this theorem, we need verification condition for the internal state of the machine. This is defined by a judgment with the form: $\Delta \vdash \{\, P \,\}\, S\, \mathsf{prop}$, where $\mathsf{prop} = \;\mathsf{seq} \mid\; \mathsf{funcall}$ which is simple and is omitted from the main text.

Corollary 18 (Safety)
If $\vdash \Psi$ ok, $\Delta \vdash \{\, P \,\}\, C\, \{\, Q \,\}$, and σ is a substitution of integers for all the variables in Δ, and $h \vDash \sigma P$, then

- *either for all $n \geq 0$, there exist S', h', and ι such that $(\cdot, h, C) \overset{\Psi}{\longmapsto}{}^{n} (S', h', \iota)$.*
- *or there exists a $k \geq 0$ such that $(\cdot, h, C) \overset{\Psi}{\longmapsto}{}^{k} (\cdot, h', \bullet)$.*

4. Related Work

The most closely related work to our own is O'Hearn, Reynolds and Yang's separation logic [10, 20]. The idea of using a general-purpose substructural logic as the assertion language in a Hoare logic was theirs. Moreover, most of our Hoare rules are derived directly from O'Hearn's. One contribution of the current paper is the observation that ILC, a combination of intuitionistic linear logic and classical arithmetic, can serve as a foundation for reasoning about pointers instead of O'Hearn and Pym's bunched implications (BI). So far research by O'Hearn

et al. has focused exclusively on using BI as the underlying logic. As O'Hearn [19] effectively explains, despite their similarities, BI and linear logic are also different in many nontrivial ways. We wish to more fully understand the strengths and weaknesses of using each logic to reason about pointer programs automatically.

In this paper, we have exploited past research on the complexity of different fragments of linear logic. For propositional linear logic, Lincoln et al. [12] give a nice survey of results. Later, Lincoln and Scedrov [13] prove first-order linear logic (without !) is nondeterministic exponential time hard. Our idea for the decidable fragment of ILC comes directly from these results.

From early on in the development of separation logic, Calcagno et al. [5] began to study the complexity of validity for various fragments. They found that the validity problem for a simple first-order logic (equality, points to, false, classical $\supset$, $\forall$) was undecidable. On the other hand, removing first-order quantifiers and adding back separating conjunction and linear implication gives a logic in which validity is decidable. Unfortunately, from the perspective of program verification, this result did not lead to any immediate gains as it was difficult to see how to do without first-order quantifiers.

More recently, Berdine, Calcagno and O'Hearn [4] have investigated another fragment of separation logic, this time with equality, separating conjunction and lists. They show that validity for this fragment is decidable. It is interesting to note that Berdine's fragment is separated into two parts, the pure formulas and the heap formulas. Therefore, at least superficially, it appears that Berdine et al. are moving away from the logic of bunched implications as a foundation for logical reasoning and towards linear logic with its tell-tale dual zones, though they do not call out this fact in their paper. Another interesting point is that they restrict negation to appearing over pure formulas as we do: allowing a more general form of negation increases the complexity of their decision procedure to PSpace from linear time. The main contribution of the current paper over this previous work is to lay out the syntactic proof theory surrounding a closely related fragment: we do not deal with list predicates, but we do consider the proof theory for the additives (conjunction and disjunction), classical arithmetic, and first-order quantifiers. In addition, we show how to exploit our fragment of the logic for program verification. Berdine et al. are also currently investigating how to use their fragment of the logic for program verification — a nontrivial task as it does not have quantifiers — but as far as we are aware, their results are not publicly available. on the other hand, Berdine's decision procedure is complete with respect to the storage model. Our proof theory is not complete with respect to the storage model. Therefore, programmers must reason syntactically in our system.

At the same time as these researchers have been investigating new program logics, the designers of advanced type systems have been using similar techniques to check programs for safety [22, 6, 9, 14, 15]. For instance, DeLine and Fähndrich's Vault programming language [6] uses a variation of alias types [22] to reason about memory management and software protocols for device drivers. Alias types very much resemble the fragment of separation logic containing the empty formula, the points-to predicate and separating conjunction. In addition, alias types have a second points-to predicate that can be used to represent shared parts of the heap, an idea that is not directly present in separation logic. We believe it is straightforward to add this second form of points-to predicate to ILC and include it under Girard's modality. In fact, the main reason that we separated ! from $\circ$ in the logic was to allow this extension. This a second predicate could be included in separation logic, if it was declared another form of "pure formula." The main difference between the program logics and the type systems is that the type systems do not include arithmetic and therefore, overall, are much less expressive.

More recently, Zhu and Xi [25] have shown how to blend the idea of alias types with Xi's previous work on Dependent ML [24] to produce a type system with "stateful views." The common link between this work and our own is that they both allow a mixture of linear and unrestricted reasoning. There are also many differences. Zhu and Xi define a type system to check for safety whereas we define a program logic with verification condition generation. Zhu and Xi's type checking algorithm appears to require quite a number of annotations — in general, when a programmer gets or sets a reference, they must bind a new proof variable, though in some cases these annotations can be inferred.[3] On the other hand, Zhu and Xi define facilities for handling recursive

[3] The full type checking algorithm is not presented in their paper, so a precise assessment is diffi cult.

data structures, something we do not attempt in this paper. The definition of the underlying logic used by Zhu and Xi is closely tied to the programming language and its data types. Consequently, it is difficult to extract the logic alone and compare it in detail to our own. Zhu and Xi do not investigate meta-theoretic properties of their logic such as cut elimination nor do they discuss decidability.

Together with Ahmed, Glew and Spalding [2, 1, 11], we have recently been exploring ways to use substructural logics to create richer forms of Typed Assembly Language. We are particularly interested in reasoning about the safety of assembly language programs that explicitly allocate and deallocate data on the stack, in the heap, or in user-defined memory regions [23]. None of the type systems we have defined have syntax-directed typing rules. In theory, when dealing with Typed Assembly Language [16] or Proof-Carrying Code [17], this is irrelevant — the compiler can generate explicit proofs or typing derivations when necessary. In practice, however, it is very difficult and time-consuming to implement a certifying compiler without automated type- or proof-reconstruction. We believe the current research brings us a step closer to implementing a certifying compiler for these memory management properties.

5. Future Work and Conclusions

In future work, we plan to implement ILC by extending an existing linear logic theorem prover with decision procedures for classical arithmetic. We are interested in using the theorem prover both to support source-level reasoning about safety properties of pointer programs and to support generation of Typed Assembly Language.

Also high on our list of priorities is support for recursive data structures. There are many ways to approach this problem. On the one hand, we could proceed as O'Hearn et al. do and simply add general-purpose recursive definitions to the logic. This approach would immediately lead to an undecidable system. On the other hand, to retain decidable checking, we believe we can take the approach used in Vault [6] and other type systems, which do not reason precisely about data structures with complex shapes and do not reason about properties of individual cells in a list or tree. We believe there may also be a range of interesting choices in between the two extremes.

To conclude, we developed a sequent calculus for a new logic, which we call ILC (Intuitionistic Linear logic with Classical arithmetic) and proved a cut elimination theorem for our logic. We also defined verification condition generation rules for a simple imperative language that produce assertions in ILC. We have proven the soundness of verification condition generation. Finally, we identify a fragment of ILC, ILC^-, that is both decidable and closed under generation of verification conditions. If loop invariants are specified in ILC^-, then the resulting verification conditions are also in ILC^-. Since verification condition generation is syntax-directed, we obtain a decidable procedure for checking properties of pointer programs.

Acknowledgments

We would like to thank Frank Pfenning for fruitful discussions about this research and for helping us work out the reading of our sequents.

References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science*, pages 33–44, Ottawa, Canada, June 2003.

[2] A. Ahmed and D. Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, New Orleans, Jan. 2003.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS 2004*, number 3362 in LNCS, pages 49–69, 2004.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FST TCS 2004: Foundations of Software Technology and Theoretical Computer Science*, number 3328 in LNCS, pages 97–109, 2004.

[5] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119, Berlin, 2001. Springer-Verlag.

[6] R. Deline and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM Press.

[7] D. L. Detlefs. An overview of the extended static checking system. In *The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM(SIGSOFT), Jan. 1996.

[8] E. S. C. for Java. Cormac Flanagan and Rustan Leino and Mark Lillibridge and Greg Nelson and James Saxes and Raymie Stata. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[9] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifi ers. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[10] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, Jan. 2001.

[11] L. Jia, F. Spalding, D. Walker, and N. Glew. Certifying compilation for a language with stack allocation. In *IEEE Symposium on Logic in Computer Science*, June 2005. To appear.

[12] P. Lincoln, J. Mitchell, A. Scedrov, and N. Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, 1992.

[13] P. Lincoln and A. Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.

[14] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refi nements. In *ACM SIGPLAN International conference on functional programming*, Aug. 2003.

[15] G. Morrisett, A. Ahmed, and M. Fluet. L^3: A linear language with locations. In *Seventh International Conference on Typed Lambda Calculi and Applications*, Apr. 2005. To appear.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 3(21):528–569, May 1999.

[17] G. Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[18] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[19] P. O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, July 2003.

[20] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, number 2142 in LNCS, pages 1–19, Paris, 2001.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[22] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, Mar. 2000.

[23] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[24] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[25] D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS vol. 3350.

A. Appendix

A.1 Sequent Calculus Rules

Sequent rules for classical logic: $\boxed{\Gamma \# \Gamma'}$

$$\frac{}{\Gamma, A \# A, \Gamma'}\; Contra$$

$$\frac{\Gamma \# A, A \wedge B, \Gamma' \quad \Gamma \# B, A \wedge B, \Gamma'}{\Gamma \# A \wedge B, \Gamma'}\; \wedge F$$

$$\frac{\Gamma, A, A \wedge B \# \Gamma'}{\Gamma, A \wedge B \# \Gamma'}\; \wedge T1 \qquad \frac{\Gamma, B, A \wedge B \# \Gamma'}{\Gamma, A \wedge B \# \Gamma'}\; \wedge T2$$

$$\frac{}{\Gamma \# \mathtt{true}, \Gamma'}\; \mathtt{true}F$$

$$\frac{\Gamma \# A, A \vee B, \Gamma'}{\Gamma \# A \vee B, \Gamma'}\; \vee F1 \qquad \frac{\Gamma \# B, A \vee B, \Gamma'}{\Gamma \# A \vee B, \Gamma'}\; \vee F2$$

$$\frac{\Gamma, A, A \vee B \# \Gamma' \quad \Gamma, B, A \vee B \# \Gamma'}{\Gamma, A \vee B \# \Gamma'}\; \vee T$$

$$\frac{}{\Gamma, \mathtt{false} \# \Gamma'}\; \mathtt{false}T$$

$$\frac{\Gamma, A \# \neg A, \Gamma'}{\Gamma \# \neg A, \Gamma'}\; \neg F \qquad \frac{\Gamma, \neg A \# A, \Gamma'}{\Gamma, \neg A \# \Gamma'}\; \neg T$$

A.2 Decidability of ILC^{-}

The definitions and proofs in the section are very similar to the residuation calculus in Chapter 4 in Frank Pfenning's Notes on Automated Theorem Proving[4]. We use Rs to denote the residuation formulas.

$$\textit{Residuation Formulas} \quad Rs \;::=\; A \mid Rs_1 \wedge Rs_2 \mid \exists x.Rs \mid \forall x.Rs$$

Proof rules

$$\frac{}{\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \cdot \;\stackrel{r}{\Longrightarrow}\; \bigcirc A \backslash (\bigwedge \Gamma \supset A)}\; \bigcirc R \qquad \frac{}{\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \Delta \;\stackrel{r}{\Longrightarrow}\; F \backslash (\bigwedge \Gamma \supset \mathtt{false})}\; \textit{absurdity}$$

Claim 4

The validity of residual formulas is decidable.

PROOF. The residual formulas are Presburger Arithmetic. □

Theorem 5 (Soundness of Residuation)

If $\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \Delta \;\stackrel{r}{\Longrightarrow}\; F \backslash Rs$ *and for any ground substitution* ρ *for* Ω, $\mathcal{M} \vDash \rho Rs$ *then* $\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \Delta \;\stackrel{-}{\Longrightarrow}\; F$.

PROOF. By induction on the depth of derivation $\Gamma\, ;\, \Theta\, ;\, \Delta \;\stackrel{r}{\Longrightarrow}\; F \backslash Rs$.

case $\dfrac{}{\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \cdot \;\stackrel{r}{\Longrightarrow}\; \bigcirc A \backslash (\bigwedge \Gamma \supset A)}\; \bigcirc R$

(1) $\mathcal{M} \vDash \rho(\bigwedge \Gamma \supset A)$	Premise
(2) $\Omega \,|\, \Gamma \# A$	(1)
(3) $\Omega \,|\, \Gamma\, ;\, \Theta\, ;\, \Delta \;\stackrel{-}{\Longrightarrow}\; \bigcirc A$	(2) and $\bigcirc R$

case The absurdity case is similar to the $\bigcirc R$ case

□

[4] Available at http://www.cs.cmu.edu/ fp/coursed/atp

Lemma 6 (Completeness of Residuation)
If $\Omega \mid \Gamma\,;\,\Theta\,;\,\Delta \stackrel{-}{\Longrightarrow} F$ *where* $F = \sigma F'$, $\Gamma = \sigma\Gamma'$, $\Theta = \sigma\Theta'$, $\Delta = \sigma\Delta'$ *and* ρ *is a ground substitution for* Ω *then* $\Omega \mid \Gamma'\,;\,\Theta'\,;\,\Delta' \stackrel{r}{\Longrightarrow} F' \backslash Rs$ *for some* Rs *and* $\mathcal{M} \vDash \rho\sigma Rs$

PROOF. By induction on the depth of $\Gamma\,;\,\Theta\,;\,\Delta \Longrightarrow F$.

$$\frac{\Omega \mid \Gamma \# A}{\Omega \mid \Gamma\,;\,\Theta\,;\,\cdot \stackrel{-}{\Longrightarrow} \bigcirc A}\ \bigcirc R$$

case $\Omega \mid \Gamma\,;\,\Theta\,;\,\cdot \stackrel{-}{\Longrightarrow} \bigcirc A$

(1) $\Omega \mid \Gamma'\,;\,\Theta'\,;\,\cdot \stackrel{r}{\Longrightarrow} \bigcirc A' \backslash (\bigwedge \Gamma' \supset A')$ — $\bigcirc$*R*(in the residuation seq rules)
(2) $\Omega \mid \Gamma \# A$ — by premise
(3) $F = \sigma F'$, $\Gamma = \sigma\Gamma'$, $\Theta = \sigma\Theta'$ — Premise
(4) $\vDash \rho\sigma(\bigwedge \Gamma' \supset A')$ — by (2) and (3)

case The absurdity case is similar to the $\bigcirc$*R* case

□

Theorem 7 (Completeness of Residuation)
If $\Omega \mid \Gamma\,;\,\Theta\,;\,\Delta \stackrel{-}{\Longrightarrow} F$, *and* ρ *is a ground substitution for* Ω *then* $\Omega \mid \Gamma\,;\,\Theta\,;\,\Delta \stackrel{r}{\Longrightarrow} F \backslash Rs$ *for some* Rs *and* $\mathcal{M} \vDash \rho Rs$

PROOF. By Lemma 6 □

Lemma 8
Residuation calculus is decidable.

PROOF. By examination of the proof rules. The premise of each rule is strictly smaller than its conclusion, so there are finite number of possible proof trees altogether (a similar argument was made in Lincoln's paper on the decidability properties of propositional linear logic [12]). □

Theorem 9 (Decidability)
ILC$^-$ *is Decidable*

PROOF. By the result of the completeness and soundness of residuation calculus, the provability of ILC$^-$ is reduced to the validity of the residual formula and the decidability of the residuation calculus. By Claim 4 and Lemma 8, the validity of the residual formula and the decidability of the residuation calculus are both decidable, so ILC$^-$ is Decidable. □

A.3 Verification Condition Generation Rules

$\boxed{\Delta \vdash (\exists x_1 \ldots \exists x_n, F_1 \& \ldots \& F_n, A) R}$

$$\frac{}{\Delta \vdash (\cdot, \top, B)\ \ B}\ \textit{boolean exp}$$

$$\frac{\Delta, x \vdash (\exists x_1 \ldots \exists x_n, F, A)\ \ R}{\Delta \vdash (\exists x_0.\exists x_1 \ldots \exists x_n,\ ((E \Rightarrow x_0) \otimes \top) \,\&\, F[x_0/x],\ A[x_0/x])\ \ \texttt{let}\ x\ = !E\ \texttt{in}\ R\ \texttt{end}}\ \textit{bind}$$

$\boxed{\Delta \vdash \{\, P \,\}\, C \,\{\, Q \,\}}$

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, P[E/x] \,\}\ \texttt{let}\ x\ =\ E\ \texttt{in}\ C\ \texttt{end}\ \{\, Q \,\}}\ \textit{bind}$$

$$\frac{\Delta, x \vdash \{\, P \,\}\, C \,\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, \forall y.\,(y \Rightarrow E)\ \multimap P[y/x] \,\}\ \texttt{let}\ x\ =\ \texttt{new}(E)\ \texttt{in}\ C\ \texttt{end}\ \{\, Q \,\}}\ \textit{new}$$

$$\frac{}{\Delta \vdash \{\exists y.\,(E \Rightarrow y)\ \otimes Q\}\ \ \texttt{free}(E)\ \ \{Q\}}\ \textit{free}$$

$$\frac{\Delta, x \vdash \{\, P \,\}\; C \;\{\, Q \,\} \quad x \notin FV(Q)}{\Delta \vdash \{\, \exists y.(\,(E \Rightarrow y)\, \otimes \top\,) \,\&\, P[y/x] \,\}\; \texttt{let}\; x \;=\! !E \;\texttt{in}\; C \;\texttt{end}\; \{\, Q \,\}}\ \textit{deref}$$

$$\frac{}{\Delta \vdash \{\, \exists x.\,(E_1 \Rightarrow x)\, \otimes (\,(E_1 \Rightarrow E_2)\, \multimap Q) \,\}\; E_1 \;:=\; E_2 \;\{\, Q \,\}}\ \textit{assignment}$$

$$\frac{\Delta, x \vdash \{\, P \,\}\; C \;\{\, Q \,\} \quad x \notin FV(Q) \quad \Psi(f) = (\,a\,)\; C_{rt}\; [\Delta]\; \{Pre\}\; \{[ret]Post\}}{\Delta \vdash \begin{array}{l} \{\, \exists \Delta.Pre[E/a] \otimes (\forall ret.Post[E/a] \multimap P[ret/x]) \,\} \\ \texttt{let}\; x \;=\; f(\,E\,)\; \texttt{in}\; C\; \texttt{end}\; \{\, Q \,\} \end{array}}\ \textit{fun call}$$

$$\frac{\Delta \vdash \{\, P \,\}\; C_1 \;\{\, P' \,\} \quad \Delta \vdash \{\, P' \,\}\; C_2 \;\{\, Q \,\}}{\Delta \vdash \{\, P \,\}\; C_1\,;\; C_2 \;\{\, Q \,\}}\ \textit{Seq}$$

$$\frac{\Delta \vdash \{\, P_1 \,\}\; C_1 \;\{\, Q \,\} \quad \Delta \vdash \{\, P_2 \,\}\; C_2 \;\{\, Q \,\}}{\Delta \vdash \{\, (\,!\bigcirc B \multimap P_1) \,\&\, (\,!\bigcirc \neg B \multimap P_2) \,\}\; \texttt{if}\; B\; \texttt{then}\; C_1\; \texttt{else}\; C_2 \;\{\, Q \,\}}\ \textit{if}$$

$$\frac{\Delta \vdash \{\, P \,\}\; C \;\{\, I \,\} \quad \Delta \vdash (\exists x_1 \ldots \exists x_n, F, B)\;\; R}{\Delta \vdash \{\; \begin{array}{l} (\exists x_1 \ldots \exists x_n.F \& (!\bigcirc \neg B \multimap Q) \& (!\bigcirc B \multimap P)) \\ \otimes\, !\,(I \multimap (\exists x_1 \ldots \exists x_n.F \& (!\bigcirc B \multimap P) \& (!\bigcirc \neg B \multimap Q)) \end{array} \;\}\; \texttt{while}_{[I]}\; R\; \texttt{do}\; C \;\{\, Q \,\}}\ \textit{while}$$

$$\frac{}{\Delta \vdash \{\, Q[E/ret] \,\}\; \texttt{return}\, E \;\{\, \forall ret.Q \,\}}\ \textit{return}$$

$\boxed{\vdash \{\, P \,\}\; S\; \mathsf{prop}, \;\; \text{where} \;\; \mathsf{prop} = \;\; \mathsf{seq} \mid \;\; \mathsf{funcall}}$

$$\frac{}{\Delta \vdash \{\, Q \,\}\; \cdot \;\; \mathsf{seq}}$$

$$\frac{\Delta \vdash \{\, P \,\}\; C \;\{\, Q \,\} \quad \Delta \vdash \{\, Q \,\}\; S_c \;\mathsf{seq}}{\Delta \vdash \{\, P \,\}\; [\,]\,;\; C \triangleright S_c \;\; \mathsf{seq}}$$

$$\frac{\Delta \vdash \{\, P \,\}\; C_{rt} \;\{\, Q \,\} \quad \Delta \vdash \{\, Q \,\}\; S_c \;\mathsf{funcall}}{\Delta \vdash \{\, P \,\}\; [\,]\,;\; C_{rt} \triangleright S_c \;\; \mathsf{seq}}$$

$$\frac{\Delta, x \vdash \{\, P[x/ret] \,\}\; C \;\{\, Q \,\} \quad \Delta \vdash \{\, Q \,\}\; S_c \;\mathsf{seq}}{\Delta \vdash \{\forall ret.P\}\; \texttt{let}\; x \;=\; [\,]\; \texttt{in}\; C\; \texttt{end} \triangleright S_c \;\; \mathsf{funcall}}$$

$$\frac{\Delta, x \vdash \{\, P[x/ret] \,\}\; C_{rt} \;\{\, Q \,\} \quad \Delta \vdash \{\, Q \,\}\; S_c \;\mathsf{funcall}}{\Delta \vdash \{\forall ret.P\}\; \texttt{let}\; x \;=\; [\,]\; \texttt{in}\; C_{rt}\; \texttt{end} \triangleright S_c \;\; \mathsf{funcall}}$$

$\boxed{\vdash \Psi\; \mathsf{ok}}$

$$\frac{\begin{array}{l} \forall f \in dom(\Psi), \Psi(f) = (\,a\,)\; C_{rt}\; [\Delta]\; \{Pre\}\; \{\forall ret.Post\}, \\ \Delta \vdash^{\Psi} \{\, Pre \,\}\; C_{rt} \;\{\, \forall ret.Post \,\} \quad \Delta \notin FV(C_{rt}) \end{array}}{\vdash \Psi\; \mathsf{ok}}$$

A.4 Soundness of Verification Condition Generation

Here C refers to the ordinary commands that don't contain return command.

Definition

- $\vDash^n \{\, P \,\}\; C \;\{\, Q \,\}$ iff for all h, Ψ, such that $h \vDash P$, and $\vDash \Psi$ ok,
  - either there exists k, $0 \le k \le n$ such that $(S, h, C) \stackrel{\Psi}{\longmapsto}^{k} (S, h', \bullet)$, and $h' \vDash Q$
  - or there exists S', h', and ι, and $(S, h, C) \stackrel{\Psi}{\longmapsto}^{n} (S', h', \iota)$
- $\vDash^n \{\, P \,\}\; C_{rt} \;\{\, \forall ret.Q \,\}$ iff for all h, Ψ, such that $h \vDash P$, and $\vDash \Psi$ ok,
  - either there exists k, $0 \le k \le n$ such that $(S, h, C_{rt}) \stackrel{\Psi}{\longmapsto}^{k} (S, h', \texttt{return}\, E)$, and $h' \vDash Q[E/ret]$

- or there exists S', h', and ι, and $(S, h, C_{rt}) \stackrel{\Psi}{\longmapsto}^n (S', h', \iota)$

- $\models \{ P \} C \{ Q \}$ iff for all $n \geq 0$, $\models^n \{ P \} C \{ Q \}$
- $\models \{ P \} C_{rt} \{ \forall ret.Q \}$ iff for all $n \geq 0$, $\models^n \{ P \} C_{rt} \{ \forall ret.Q \}$
- $\models \Psi$ ok iff $\forall f \in dom(\Psi), \Psi(f) = (a)\, C_{rt}\, [\Delta]\, \{Pre\}\, \{\forall ret.Post\}$, and for all substitution σ for Δ, and σ_a for a, $\models \{ \sigma\sigma_a Pre \}\ \sigma\sigma_a C_{rt}\ \{ \sigma\sigma_a \forall ret.Post \}$
- $\models \{ Q \}\ S\ seq$ iff for all Ψ, C, and h, such that $\models \Psi$ ok, $\models \{ P \} C \{ Q \}$, and $h \models P$,
  - either for all $n \geq 0$, there exist S', h', and ι such that $(S, h, C) \stackrel{\Psi}{\longmapsto}^n (S', h', \iota)$.
  - or exists $k \geq 0$, h' such that $(S, h, C) \stackrel{\Psi}{\longmapsto}^k (\cdot, h', \bullet)$.
- $\models \{ \forall ret.Q \}\ S\ funcall$ iff for all Ψ, C_{rt}, and h, such that $\models \Psi$ ok, $\models \{ P \} C_{rt} \{ \forall ret.Q \}$, and $h \models P$,
  - either for all $n \geq 0$, there exist S', h', and ι such that $(S, h, C_{rt}) \stackrel{\Psi}{\longmapsto}^n (S', h', \iota)$.
  - or exists $k \geq 0$, h' such that $(S, h, C_{rt}) \stackrel{\Psi}{\longmapsto}^k (\cdot, h', \bullet)$.

Lemma 10
if $\Delta \vdash (\exists x_1 \cdots \exists x_n, F_1 \& \ldots \& F_n, A)R$, and σ_x is a substitution for $x_1, \cdots x_n$, and σ is a substitution for Δ, and $h \models \sigma\sigma_x(F_1 \& \ldots \& F_n)$, then $(S, h, R) \stackrel{\Psi}{\longmapsto}^n (S, h, B)$, such that $\cdot \models !\bigcirc (\sigma\sigma_x A)$ iff $[\![B]\!] = \texttt{true}$, and $\cdot \models !\bigcirc \neg(\sigma\sigma_x A)$ iff $[\![B]\!] = \texttt{false}$.

PROOF. By induction on the structure of R □

Defs

- (S, h) is safe for ι for n steps iff $(S, h, \iota) \stackrel{\Psi}{\longmapsto}^n (S', h', \iota')$
- $dangle(h)$ is the set of dangling pointers that heap h points to.
- $\hat{f}(x) = f(x) if x \in dom(f) x otherwise$
- We treat $\hat{f}$ as a substitution of the location values in h to location values in h_1.
- $(S, h, \iota) \stackrel{f}{\sim} (S_1, h_1, \iota_1)$ iff f is a bijection from $dom(h) \cup dangle(h)$ to $dom(h_1) \cup dangle(h_1)$, and $\forall \ell \in dom(h) \cup dangle(h)$, $\hat{f}(S) = S_1$, $\hat{f}(\iota) = \iota_1$, and for all $\ell \in dom(h)$, $\hat{f}(h(\ell)) = h_1(f(\ell))$.

Lemma 11 (Monotonicity)

1. *If $(S, h, \iota) \stackrel{f}{\sim} (S_1, h_1, \iota_1)$, and $(S, h, \iota) \stackrel{\Psi}{\longmapsto} (S', h', \iota')$, and $(S_1, h_1 \uplus h_2, \iota_1) \stackrel{\Psi}{\longmapsto} (S'_1, h'_1 \uplus h_2, \iota'_1)$, then $(S', h', \iota') \stackrel{f}{\sim} (S'_1, h'_1, \iota'_1)$.*
2. *For all n, $n \geq 0$, if (S, h) is safe for ι for n steps, then $(S, h \uplus h')$ is safe for ι for n steps.*

PROOF. 1. By examine all the cases in operational semantics.
2. Follows 1.

□

Lemma 12 (Frame Property)

For all n, $n \geq 0$, if (S, h) is safe for ι for n steps, and $(S, h \uplus h_1, \iota) \stackrel{\Psi}{\longmapsto}^n (S', h', \iota')$, then $h' = h'' \uplus h_1$, and $(S, h, \iota) \stackrel{\Psi}{\longmapsto}^n (S', h'', \iota')$.

PROOF. By indution on n. □

Lemma 13
If $\Delta \vdash^{\Psi} \{ P \} C \{ Q \}$ then for all substitution σ for Δ, for all $n \geq 0$, $\models^n \{ \sigma P \}\ \sigma C\ \{ \sigma Q \}$.

PROOF. By induction on n, call upon Lemma 10, Lemma 14, and Lemma 11. $\square$

Lemma 14
If $\Delta \vdash^{\Psi} \{ P \} C \{ \forall ret.Q \}$ *then for all substitution* σ *for* Δ*, for all* $n \geq 0$, $\models^{n} \{ \sigma P \} \sigma C \{ \sigma \forall ret.Q \}$.

PROOF. By induction on n, call upon Lemma 13, and Lemma 11. $\square$

Lemma 15
If $\Delta \vdash \{ P \} S_c\ prop$ *then for all substitution* σ *for* Δ, $\models \{ \sigma P \} \sigma S_c\ prop$.

PROOF. By induction on the structure of S_c, and call upon Lemma 13 and Lemma 14. $\square$

Lemma 16
if $\vdash \Psi$ ok *then* $\models \Psi$ ok

PROOF. By Lemma 14. $\square$

Theorem 17 (Safety)
If $\vdash \Psi$ ok, $\Delta \vdash \{ P \} C \{ Q \}$*, and* $\Delta \vdash \{ Q \} S$*, and* σ *is a substitution for* Δ*, and* $h \models \sigma P$*, then*

- *either for all* $n \geq 0$*, there exist* S'*,* h'*, and* ι *such that* $(S, h, C) \overset{\Psi}{\longmapsto}{}^{n} (S', h', \iota)$.
- *or there exists a* $k \geq 0$ *such that* $(S, h, C) \overset{\Psi}{\longmapsto}{}^{k} (\cdot, h', \bullet)$.

PROOF. By Lemma 13, Lemma 15, and Lemma 16 $\square$

Corollary 18 (Safety)
If $\vdash \Psi$ ok, $\Delta \vdash \{ P \} C \{ Q \}$*, and* σ *is a substitution for* Δ*, and* $h \models \sigma P$*, then*

- *either for all* $n \geq 0$*, there exist* S'*,* h'*, and* ι *such that* $(\cdot, h, C) \overset{\Psi}{\longmapsto}{}^{n} (S', h', \iota)$.
- *or there exists a* $k \geq 0$ *such that* $(\cdot, h, C) \overset{\Psi}{\longmapsto}{}^{k} (\cdot, h', \bullet)$.

PROOF. Follows from Theorem 17. $\square$