ILC: A Foundation for Automated Reasoning About Pointer Programs

Limin Jia and David Walker

Princeton University, Princeton, NJ 08544, USA
{ljia,dpw}@cs.princeton.edu

Abstract. This paper shows how to use Girard's intuitionistic linear logic extended with a classical sublogic to reason about pointer programs. More specifically, first, the paper defines the proof theory for ILC (Intuitionistic Linear logic with Constraints) and shows it is well-defined via a proof of cut elimination. Second, inspired by prior work of O'Hearn, Reynolds, and Yang, the paper explains how to interpret linear logical formulas as descriptions of a program store. Third, this paper defines a simple imperative programming language with mutable references and arrays and gives verification condition generation rules that produce assertions in ILC. Finally, we identify a fragment of ILC, ILC$^-$, that is both decidable and closed under generation of verification conditions. 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. Amazingly, in 2005, Microsoft is 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 include the Microsoft assertion language mentioned above, Leino et al.'s extended static checking project and its successors [2–4], and Necula and Lee's proof-carrying code [5]. These tools have used conventional classical logic to specify and check program properties. These conventional 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 [6, 7] have advocated using separation logic, which is the classical logic of bunched implications extended with a collection of domain-specific axioms about storage. The crucial insight in this research is that the multiplicative connectives of the logic of bunched implications encapsulate "separation" invariants commonly used when reasoning about storage.

Inspired by the work of O'Hearn et al., we have begun to develop a new program logic in which the proof theory used to reason about state and resources is based on Girard's linear logic as opposed to the logic of bunched implication. There are several reasons why we decided to focus on linear logic as opposed to bunched implications as a foundation for verifying programs. First, from a practical standpoint, there are a number of tools available for our use including logic programming engines Lolli [8] and Lollimon [9], theorem provers [10] and logical frameworks such as Forum [11], LLF [12], and CLF [13]. Second, since linear logic is older than BI, more is known about it. In particular, we have been able to use known results on the complexity of various fragments of linear logic to devise a useful decidable fragment of our logic. Third, we have recently looked at generating proof-carrying code for programs with rich memory management invariants [14, 15], 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 [16], which implicitly, and in newer work [17], explicitly, 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, this paper is a starting point from which we can begin to study the relative strengths and weaknesses of using the proof theory of intuitionistic linear logic, which is based on sequents with a flat context for assumptions, as opposed to the proof theory of BI, which is based on sequents with "bunched" or tree-like contexts, as the foundation for verification of pointer programs.

To summarize, there are four central contributions of this paper. First (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 and prove a cut-elimination theorem to show that it is well defined. The proof theory is sound with respect to the storage model, but not complete. As any automated program analysis will run up against incompleteness somewhere, this lack of completeness is not an immediate practical concern for us. However, an important element of future work will be understanding the sources of the incompleteness. Logic of bunched implications does have certain completeness properties and therefore has an advantage over linear logic in this respect. 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 our verification condition generation is sound with respect to our memory model. The third main contribution (Section 2.6) is in the definition of a useful, and decidable fragment of the logic, ILC$^-$. The key property of ILC$^-$ is that it is closed under verification condition generation: if loop invariants and pre- and post-conditions fall into ILC$^-$ then the generated verification conditions also fall into ILC$^-$. The decidable logic plus the syntax-directed verification condition generation give rise to a terminating algorithm for verification of pointer programs. Fourth, we have implemented a prototype verifier for our language. Our prototype generates verification conditions in ILC. We then prove the validity of linear logic formulas in MetaPRL [18], a manual process at this point, and discharge the constraints using the CVC Lite [19] theorem prover. The examples in this paper have been verified using our implementation. Due to space considerations, we have omitted many technical details. Please see our technical report [20] for complete formal rules and additional metatheory.

2 Intuitionistic Linear Logic with Constraints

In this section we introduce ILC, Intuitionistic Linear logic with Constraints. After introducing the syntax, semantics, proof theory, and properties of ILC, we will present a decidable fragment, ILC$^-$.

2.1 Syntax

ILC formulas F include all of the first-order formulas present in multiplicative and additive intuitionistic linear logic. In addition, a modality $\bigcirc A$ encapsulates a language of classical constraints as a sublogic within ILC. For the purposes of this paper, the constraint language involves arrays and Presburger arithmetic.

The basic predicates for reasoning about program state include $(E_1 \Rightarrow E_2)$, which describes a heap containing only one location, E_1, and its contents E_2; and $\mathsf{Array}(E_1, E_2, \alpha)$, which describes an array that has a starting address E_1, number of elements E_2, and list of elements α. The classical constraints use E to range over integer terms and α to range over array terms. The empty array is denoted by Nil, $\mathsf{sel}(\alpha, E)$ accesses the E_{th} element of α, and $\mathsf{upd}(\alpha, E_1, E_2)$ generates a new array with the element indexed by E_1 replaced by E_2.

$$\begin{array}{lll}
\textit{Integer Terms} & E & ::= n \mid x \mid E_1 + E_2 \mid -E \mid \mathsf{sel}(\alpha, E) \\
\textit{Array Terms} & \alpha & ::= \mathsf{Nil} \mid x \mid \mathsf{upd}(\alpha, E_1, E_2) \\
\textit{Arithmetic Predicates} & Pa & ::= E_1 = E_2 \mid E_1 < E_2 \\
\textit{Classical Formulas} & A & ::= \mathtt{true} \mid \mathtt{false} \mid Pa \mid A_1 \wedge A_2 \mid \neg A \mid A_1 \vee A_2 \\
\textit{State Predicates} & Ps & ::= (E_1 \Rightarrow E_2) \mid \mathsf{Array}(E_1, E_2, \alpha) \\
\textit{Intuitionistic Formulas} & F & ::= Ps \mid \mathbf{1} \mid F_1 \otimes F_2 \mid F_1 \multimap F_2 \mid \top \mid F_1 \,\&\, F_2 \mid \mathbf{0} \\
 & & \quad \mid F_1 \oplus F_2 \mid !\,F \mid \exists b.F \mid \forall b.F \mid \bigcirc A
\end{array}$$

2.2 Basic Concepts

We informally discuss the semantics of the connectives and highlight the key ideas for reasoning about program states. All the examples in this section refer to Figure 1, which shows a heap h containing 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 `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. 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.

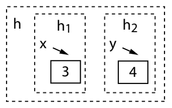


Fig. 1. A sample heap

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 can be described by $(x \Rightarrow 3) \multimap ((x \Rightarrow 3) \otimes (y \Rightarrow 4))$. A more interesting example is that h 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.

No information. The unit of additive conjunction $\top$ describes any linear state, so 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. Formula $F_1 \,\&\, F_2$ represents a state that can be described by (shared between) 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 basic sharing properties of these two connectives are the same. But the behavior of $\wedge$ is closely connected to the additive implication $\rightarrow$ and the bunched contexts, which our logic does not have.

Heap Free Conditions. The unrestricted modality $!F$ describes an empty heap and asserts F is true. For instance, $!((x \Rightarrow 3) \multimap \exists y. (x \Rightarrow y))$ 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. Note that $!F$ is semantically equivalent to $F \& \mathbf{1}$. 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 multiple times; $F \& \mathbf{1}$ does not satisfy these properties. 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, the 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 “heapful” conditions, such as $\neg (x \Rightarrow 3)$, appear very

- $\mathcal{M}; h \vDash (E_1 \Rightarrow E_2)$ iff $dom(h) = \{[\![E_1]\!]\}$, $h([\![E_1]\!]) = [\![E_2]\!]$.
- $\mathcal{M}; h \vDash \mathsf{Array}(E_1, E_2, Y)$ iff $\{[\![E_1]\!]\} = dom(h)$ and $h([\![E_1]\!]) = [\![Y]\!]_n$, where $n = [\![E_2]\!]$.
- $\mathcal{M}; h \vDash \mathbf{1}$ 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 stores h', $\mathcal{M}; h' \vDash F_1$ implies $\mathcal{M}; h \uplus h' \vDash F_2$.
- $\mathcal{M}; h \vDash \top$ is true for all stores.
- $\mathcal{M}; h \vDash F_1 \,\&\, F_2$ iff $\mathcal{M}; h \vDash F_1$, and $\mathcal{M}; h \vDash F_2$.
- $\mathcal{M}; h \vDash \mathbf{0}$ is false for all stores.
- $\mathcal{M}; h \vDash F_1 \oplus F_2$ 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 value a such that $\mathcal{M}; h \vDash F[a/x]$.
- $\mathcal{M}; h \vDash \forall x.F$ iff for all values a, $\mathcal{M}; h \vDash F[a/x]$.
- $\mathcal{M}; h \vDash \bigcirc A$ iff $dom(h) = \emptyset$, and $\mathcal{M} \vDash A$.

Fig. 2. The semantics of formulas

rarely, but classical reasoning about constraints is ubiquitous. Consequently, we add a classical sublogic to what we have already presented. The classical formulas describe constraints and are confined under the modality $\bigcirc$. For example, heap h 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 would 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 reasoning about arithmetic or other constraints from the intuitionistic linear reasoning making it possible to use an off-the-shelf theorem prover or decision procedure for the constraints.

2.3 Semantics

Our logical formulas describe program stores that map locations to values. All values are integers or integer tuples; some integers (an infinite collection of them) are considered heap locations. We use metavariable n when referring to integers, ℓ when referring to locations, and v when referring to values.

We use $dom(h)$ to denote the domain of store h, $h(\ell)$ to denote the value stored at location ℓ, $h[\,\ell := v\,]$ to denote a store h' in which ℓ maps to v but is otherwise the same as h. We write $h_1 \uplus h_2$ to denote the union of disjoint stores. The $\uplus$ operation is undefined if the stores are not disjoint. We use $|v|$ to denote the number of elements in tuple v, $v|_i$ to denote the i_{th} elements of v if $0 \leq i < |v|$, and $v[i \mapsto v']$ to denote the result of updating the i_{th} element of v with v', if $0 \leq i < |v|$.

There are three semantic judgments:

$\mathcal{M} \vDash A$	Classical formula A is valid in model $\mathcal{M}$
$\mathcal{M}; h \vDash F$	Store h together with model $\mathcal{M}$ satisfies formula F
$h \vDash F$	Store h satisfies formula F (exists a model $\mathcal{M}$ such that $\mathcal{M}; h \vDash F$)

$\mathcal{M}$ is a model for the first-order theories we consider. We write $[\![E]\!]$ for the integer value that the closed expression E denotes. The denotation of an array term $[\![\alpha]\!]_n$ is an integer tuple of length n. The semantics of classical formulas is standard, and we omit it in this paper. The formal definition of $\mathcal{M}; h \vDash F$ is given in Figure 2.

2.4 Proof Theory

Our logical judgments make use of an unrestricted context Γ for classical constraints, 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 the set of variables free in the rest of the sequent.

Our logic has two sequent judgments.

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

The sequent rules for classical logic follow the LK formalization [21]. An intuitive reading of the intuitionistic sequent is that if a state is described by unrestricted assumptions in Θ, linear assumptions Δ, and satisfies all the classical constraints in Γ, then this state can also be described by F.

Our logic has the same sequent rules 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 listed below. These rules illustrate the interaction between the classical 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.

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

Interesting Theorems The following axioms, all of which are provable in our sequent calculus, illustrate some of the interactions between the classical and intuitionistic connectives.

$$\begin{array}{lll} \bigcirc\texttt{true} \Longleftrightarrow \mathbf{1} & \bigcirc A \otimes \bigcirc B \Longleftrightarrow \bigcirc(A \wedge B) & \bigcirc(A \wedge B) \Longrightarrow \bigcirc A \,\&\, \bigcirc B \\ \bigcirc\texttt{false} \Longleftrightarrow \mathbf{0} & & \bigcirc A \oplus \bigcirc B \Longrightarrow \bigcirc(A \vee B) \end{array}$$

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: $!F \Longrightarrow \mathbf{1}$, $!F \Longrightarrow\, !F \otimes !F$. In separation logic, Reynolds [22] adds specialized axioms for relating the additive conjunction of pure facts to the multiplicative conjunction of them:

$P \wedge Q \Longrightarrow P * Q$ when P or Q is pure $\qquad P * Q \Longrightarrow P \wedge Q$ when P and Q is pure

In our logic, we can prove $!P \otimes !Q \Longrightarrow !P \,\&\, !Q$ but not the reverse. 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 $!P \otimes !Q$ instead of $!P \& !Q$. So far, we have seen no practical consequences of omitting this axiom.

2.5 Properties of ILC

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

We use the notion of semantics for logical contexts (written $h \vDash \Gamma; \Theta; \Delta$) in Theorem 2. It means that store h satisfies all the constraints in Γ and h contains all the unrestricted resources in Θ and all the linear resources in Δ.

Theorem 1 (Cut Elimination).

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

Theorem 2 (Soundness of Logic Deduction).
If $\Omega \,|\, \Gamma; \Theta; \Delta \Longrightarrow F$ *and* σ *is a grounding substitution for all the variables* Ω, *and* $h \vDash \Gamma[\sigma]; \Delta[\sigma]; \Delta[\sigma]$, *then* $h \vDash F[\sigma]$.

2.6 A Decidable Fragment: ILC$^-$

We have identified a fragment of our logic, ILC$^-$, which is decidable and sufficient to encode many pre- and post-conditions for 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 annotations fall into this fragment, then the whole process of program verification is decidable.

The factors that contribute to the undecidability of ILC are that 1) it contains Intuitionistic Linear Logic as a sub-logic which is undecidable, and 2) the validity of arbitrarily quantified first-order classical formulas of equality and array theory is undecidable. In order to obtain a decidable fragment of ILC, first we replace the *copy* rule with the *U-Init* and $!\bigcirc L$ rules (we use $\overset{-}{\Longrightarrow}$ for sequents in ILC$^-$):

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

$$\frac{}{\Omega \,|\, \Gamma \,;\, \Theta, P \,;\, \cdot \;\overset{-}{\Longrightarrow}\; P}\; \mathit{U\text{-}Init} \qquad \frac{\Omega \,|\, \Gamma, A \,;\, \Theta \,;\, \Delta \;\overset{-}{\Longrightarrow}\; F}{\Omega \,|\, \Gamma \,;\, \Theta \,;\, \Delta, !\bigcirc A \;\overset{-}{\Longrightarrow}\; F}\; !\bigcirc L$$

Second, we syntactically restrict the logical formulas so that we don't need to decompose connectives in the unrestricted context. Now the two new rules have the same power as the old *copy* rule. Furthermore, we only consider Presburger Arithmetic to guarantee the decidability in classical reasoning part (any decidable system of constraints will do). Each syntactic class in this decidable fragment is defined as follows:

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

We have proven that in the above fragment the sequent rules with *U-Init* and $!\bigcirc L$ are sound and complete with regard to the original sequent rules in Section 2.4.

Theorem 3 (Soundness & Completeness of $\overset{-}{\Longrightarrow}$). *$\Omega \mid \Gamma \,;\, \Theta \,;\, \Delta \overset{-}{\Longrightarrow} G$ iff $\Omega \mid \Gamma \,;\, \Theta \,;\, \Delta \Longrightarrow G$, 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.*

The proof of the decidability of ILC$^-$ can be found in the technical report [20]. Informally, any proof search in ILC$^-$ can be reduced to two procedures: first, a proof search in the sequent calculus of intuitionistic linear logic without the *copy* rule, and second, the validity checking of Presburger Arithmetic formulas with equality. The first part is decidable since every premise of each sequent rule is strictly smaller than its consequent (by smaller we mean that the number of connectives in the sequent decreases [23]). The second part is also decidable. Therefore, the whole process is decidable.

Theorem 4 (Decidability). *ILC$^-$ is decidable.*

Discussion Notice that we only consider the decidable Presburger Arithmetic constraints in ILC$^-$. More generally, the intuitionistic linear logic part of ILC$^-$ is always decidable, and the decidability is sustained when we extend ILC$^-$ with any decidable constraint domain.

3 Verifying Pointer Programs

In this section, we show how to verify an imperative language with pointer operations using our logic. We present syntax-directed verification condition generation rules and give examples to show how they are used to verify programs.

3.1 Syntax and Operational Semantics

Now we introduce the syntax and operational semantics of an imperative language that includes control flow, mutable references, and arrays.

Syntax The syntactic constructs of our language are listed below. We use E to range over integer expressions and B to range over boolean expressions. The language has commands for allocation, deallocation, variable binding, dereference, assignment, array operations, sequencing, while loop, if branching, and skip. The while loop expression `while`$_{[I]}$ R `do` C is annotated with loop invariant I. The condition expression R computes a boolean that determines while termination. These condition expressions have special structure and scoping rules to both simplify pre-condition generation and to provide ample expressive power. The variables in the loop body of the while loop are bound by the *let* expression in the condition R. For example, in the following command $\texttt{while}_{[\top]}\ \texttt{let}\ x\ =!y\ \texttt{in}\ x > 0\ \texttt{end}\ \ \texttt{do}\ \ y\ :=\ x - 1$, variable x in the loop body is bound by the let expression. In order to generate verification conditions properly from expressions, we require them to be in A-Normal form. Naturally, an implementation would allow programmers to write ordinary expressions and then unwind them to A-Normal form for verification.

Int Exps	$E ::= n \mid x \mid E + E \mid -E$
Boolean Exps	$B ::= \texttt{true} \mid \texttt{false} \mid E_1 = E_2 \mid E_1 < E_2 \mid B_1 \wedge B_2 \mid \neg B \mid B_1 \vee B_2$
Condition Exps	$R ::= B \mid \texttt{let}\ x\ = !E\ \texttt{in}\ R\ \texttt{end}$
Command	$C ::= \texttt{let}\ x\ =\ \texttt{new}(E)\ \texttt{in}\ C\ \texttt{end} \mid \texttt{free}(E)$
	$\mid \texttt{let}\ x\ =\ E\ \texttt{in}\ C\ \texttt{end} \mid \texttt{let}\ x\ = !E\ \texttt{in}\ C\ \texttt{end} \mid E_1\ :=\ E_2$
	$\mid \texttt{let}\ x\ =\ \texttt{newArray}(E)\ \texttt{in}\ C\ \texttt{end} \mid \texttt{let}\ x\ =\ E_1[E_2]\ \texttt{in}\ C\ \texttt{end}$
	$\mid E_1[E_2]\ :=\ E_3 \mid \texttt{let}\ x\ =\ \texttt{Len}(E)\ \texttt{in}\ C\ \texttt{end} \mid C_1\ ;\ C_2$
	$\mid \texttt{while}_{[I]}\ R\ \ \texttt{do}\ \ C \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{skip}$

Operational Semantics A program state consists of a control stack S, a store (or a heap) h, and an instruction ι being evaluated. An instruction ι can be a command, a loop guard R followed by a command C, or the special instruction $\bullet$, which indicates the termination of certain commands. In our language, variables are bound, and there is no imperative assignment to variables. We therefore do not need a stack to map variables to values. We use $(S, h, \iota) \longmapsto (S', h', \iota')$ to denote the small step operational semantics. The control stack S is a list of evaluation contexts and is not crucial for the understanding of this paper, so we omit its definition.

3.2 Verification Condition Generation

The program to be verified is annotated with pre- and post-condition and loop invariants by the programmer. The verification condition generation scans the program bottom-up and generates a formula (from the postcondition) such that if the initial program states satisfy this formula then the program will execute safely, and if it terminates then the ending state will satisfy the specified post-condition. The computed verification condition will be satisfied by the initial state if it is logically entailed by the programmer provided precondition, which we assume will hold before the execution of the program. We have not tackled the question of whether or not our verification conditions are weakest preconditions. There are two judgments involved in verification condition generation.

$\Delta \vdash (\exists x_1 \ldots \exists x_n, F, A, \rho)R$	There exist values for variables $x_1 \cdots x_n$ such that the precondition of executing R is F, the core boolean expression in R is A, and ρ is the substitution of logical terms for the variables in R bound by *let*.
$\Delta \vdash \{\,P\,\}\ C\ \{\,Q\,\}$	The precondition of C is P, and the postcondition is Q.

$\boxed{\varDelta \vdash \{\, P \,\}\; C\; \{\, Q \,\}}$

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

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

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

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

$$\frac{\varDelta, x \vdash \{\, P \,\}\; C\; \{\, Q \,\} \quad x \notin FV(Q)}{\varDelta \vdash \begin{array}{l} \{\, !\bigcirc (\neg(E < 0)) \otimes \forall y.(\mathsf{Array}(y, E, \mathsf{Nil}) \multimap P[y/x]) \,\} \\ \texttt{let}\; x \;=\; \texttt{newArray}(E)\; \texttt{in}\; C\; \texttt{end}\; \{\, Q \,\} \end{array}}\;\textit{New Array}$$

$$\frac{\varDelta, x \vdash \{\, P \,\}\; C\; \{\, Q \,\} \quad x \notin FV(Q)}{\varDelta \vdash \begin{array}{l} \{\, \exists size.\exists \alpha.(\mathsf{Array}(E_1, size, \alpha) \otimes !\bigcirc (\neg(E_2 < 0)) \otimes !\bigcirc (E_2 < size) \otimes \top) \\ \quad \& P[\mathsf{sel}(\alpha, E_2)/x] \,\} \\ \texttt{let}\; x \;=\; E_1[E_2]\; \texttt{in}\; C\; \texttt{end}\; \{\, Q \,\} \end{array}}\;\textit{Subscript}$$

$$\frac{}{\varDelta \vdash \begin{array}{l} \{\, \exists size.\exists \alpha.\mathsf{Array}(E1, size, \alpha) \otimes !\bigcirc (\neg(E_2 < 0)) \otimes !\bigcirc (E_2 < size) \\ \otimes(\mathsf{Array}(E_1, size, \mathsf{upd}(\alpha, E_2, E_3)) \multimap Q) \,\} \\ E_1[E_2] \;:=\; E_3\; \{\, Q \,\} \end{array}}\;\textit{Array Upd}$$

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

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

Fig. 3. Selected Rules for Verification Condition Generation

Commands The verification condition generation rules are backward-reasoning rules, and are syntax directed. Most of the rules are identical to O'Hearn's weakest precondition generation [6] except that $*$ is replaced by $\otimes$, $-\!\!*$ by $\multimap$ and $\wedge$ by $\&$. We explain a few key rules here. The set of selected rules is shown in Figure 3.

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 two parts: one that contains cell E_1, and another that waits for the update.

The precondition of the array allocation command first asserts that the size of the array is legal; the second part describes a heap that is waiting for the new piece described by $\mathsf{Array}(y, E, \mathsf{Nil})$. After merging with the newly allocated array, the heap satisfies the precondition of C with x substituted with the address of the new array. The precondition of the array update command first checks that E_1 indeed points to an array on the heap ($\mathsf{Array}(E1, size, \alpha)$). Then it checks that the index is in bounds ($!\bigcirc (\neg(E_2 < 0)) \otimes !\bigcirc (E_2 < size)$). The last part in the precondition describes a heap that requires the updated array to satisfy Q.

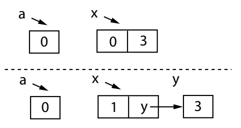
The *if* instruction branches on boolean expression B. The precondition for *if* says that if B is true then the precondition of the true branch holds; otherwise the precondition of the false branch holds. The additive conjunction is used to give two possible descriptions to the same heap. Note that the precondition of the branch that is not taken will be proven using the absurdity rule. We will give a concrete example in Section 3.3.

While loops are annotated with loop invariants. A while loop either executes the loop body or exits the loop depending on the condition expression R. There are two parts to the precondition of a while loop. The first part $(\exists x_1 \ldots \exists x_n.F \& (!\bigcirc(\neg B) \multimap Q) \& (!\bigcirc(B) \multimap P[\rho]))$ asserts that when we execute the loop for the first time, the precondition F for evaluating the condition expression must hold; if the condition is not true then the postcondition Q must hold, otherwise the precondition P for the loop body C must hold. The second part $(\,!\,(I \multimap (\exists x_1 \ldots \exists x_n.F \& (!\bigcirc(B) \multimap P[\rho]) \& (!\bigcirc(\neg B) \multimap Q)))$ 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.

3.3 Examples

In this section, we give two examples to demonstrate how we verify programs using the verification condition generation rules defined in the previous section. We prove the validity of ILC formulas in MetaPRL [18] and discharge the constraints using the CVC Lite [19] theorem prover. We do not have an automated theorem prover for ILC yet, but it is technically feasible to develop one and we are working with Frank Pfenning and Kaustuv Chaudhuri to develop the theorem prover we need.

If Branching In this example, the store is shown in Figure 4. Location a contains 0. Depending on the contents of location x, there are two possibilities for the remainder of the store. If x contains 0, then the location next to x contains 3; if x contains an integer other than 0, then the location next to x contains another location y, and y contains 3. The first case is illustrated in the figure above the dashed line, and the second case is illustrated below the dashed line. Formula F describes the store h. We use additive disjunction to describe the two cases.



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

Fig. 4. Example

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 and stores the value into a. At the merge point of the branch, a should contain 3.

```
{$F = (a \Rightarrow 0) \otimes (((x \Rightarrow 0) \otimes (x+1 \Rightarrow 3)) \oplus ((x \Rightarrow 1) \otimes \exists y.(x+1 \Rightarrow y) \otimes (y \Rightarrow 3)))$}
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
{ $(a \Rightarrow 3) \otimes \top$}
```

To show this program is memory safe and will store 3 into a in the end $((a \Rightarrow 3) \otimes \top)$, we first generate a verification condition. Next we prove that the precondition describing the initial state entails the verification condition we generated (Pre): $x, a \,|\, \cdot; \cdot; F \Longrightarrow Pre$. According to our sequent rules, one of the subgoals we need to prove is:

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

where $P_2 = \exists u.((x+1 \Rightarrow u) \otimes \top) \& \exists v.((u \Rightarrow v) \otimes \top)$

$\& \exists w.(a \Rightarrow w) \otimes ((a \Rightarrow v) \multimap ((a \Rightarrow 3) \otimes \top))$

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 are not sufficient to prove P_2, which requires $x+1$ to contain another location. However, we have a contradiction in the classical context $(\neg(0=0))$, so we prove P_2 using the absurdity rule. This is the situation where we cannot establish the precondition required by the branch that is not taken. Instead, we prove it by contradiction.

Array Copying In this example, we prove the correctness of an array copying program. The code is shown below. At the beginning of this program, there is an array x with at least 2 elements. We will allocate a new array y that has exactly one fewer element than x and copy the elements from x to y using while loop. The postcondition specifies that at the end of the program we have two arrays, that one is one element shorter than the other, and that their elements are the same up to the length of the shorter array. The loop invariant says that the loop induction variable is always between 0 and the length of the longer array, and that from the first element up to the element indexed by the loop induction variable the two arrays have the same elements.

```
{$\exists a \mathsf{Array}(x, n, a) \otimes !\bigcirc(\neg(n < 2))$}
let len = arrayLen x in
let y = newArray[len -1] in
let i = newPtr(0) in
while let j = !i in j< len-1 end
  [$\exists v.\exists n_2.\exists a.\exists b.\mathsf{Array}(x, n, a) \otimes \mathsf{Array}(y, n_2, b)$
    $\otimes !\bigcirc(n = n_2 + 1) \otimes (i \Rightarrow v) \otimes !\bigcirc(\neg(v < 0)) \otimes !\bigcirc(v < n)$
    $\otimes \forall j.!\bigcirc((\neg(j < 0) \wedge (j < v)) \supset (\mathsf{sel}(a, j) = \mathsf{sel}(b, j)))$]
do
  let z = x[j] in
  y[j] := z;
  i:= j + 1
  end ;
free(i) end end end
 {$\exists m_1.\exists m_2.\exists c.\exists d.\exists lx \exists ly.\mathsf{Array}(lx, m1, c) \otimes \mathsf{Array}(ly, m2, d)$
  $\otimes !\bigcirc(m_1 = m_2 + 1) \otimes \forall i.!\bigcirc((\neg(i < 0) \wedge (i < m_2)) \supset (\mathsf{sel}(c, i) = \mathsf{sel}(d, i)))$}
```

We remark that the proof obligations generated from this program involve existentially quantified formulas of the array theory and are not in the obviously decidable quantifier free fragment. However, CVC Lite handles them fine.

3.4 Soundness of Verification Generation

Finally, we proved that the rules for verification generation are sound with regard to the semantics of the language.

Theorem 5 (Soundenss of VC Gen). *If $\Delta \vdash \{\, P \,\}\ C\ \{\, Q \,\}$, and σ is a grounding substitution for all the variables in Δ, and $h \vDash P[\sigma]$, then*

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

4 Related Work

The most closely related work to our own is O'Hearn, Reynolds, and Yang's separation logic [6, 7]. Their key insight was the fact that a substructural logic, when used as the assertion language in a program logic, facilitates local reasoning about state. Recently, Berdine, Calcagno, and O'Hearn [24, 25] have investigated a decidable fragment of separation logic with equality, separating conjunction, and lists. An important advantage of their proof theory is that it is complete with respect to their model whereas our proof theory is incomplete. For us, this means that programmers must reason syntactically using linear logic proof rules as opposed to semantically. On the other hand, we consider a more extensive logic that, unlike Berdine et al., includes additives ($\&$, $\top$), first-order quantifiers, and an arbitrary classical sublogic, which we have instantiated with a theory of arrays and arithmetic. If the sublogic used in loop invariants is decidable then the verification conditions we generate and the overall program verification procedure is also decidable.

As researchers have been investigating new program logics, the designers of advanced type systems have been using similar techniques to check programs for safety [16, 26–28, 17]. For instance, DeLine and Fähndrich's Vault programming language [26] uses a variation of alias types [16] 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. The main difference between the program logics and the type systems is that type systems, particularly Vault, support better inferences while the logics include a wider variety of connectives and more sophisticated constraint systems, and therefore, are much more expressive.

More recently, Zhu and Xi [29] have shown how to blend the idea of alias types with Xi's previous work on Dependent ML [30] 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. On the other hand, Zhu and Xi define facilities for handling recursive data structures, something we do not attempt in this paper.

5 Conclusions

We have developed a sequent calculus for ILC, linear logic with constraints, and proved a cut elimination theorem. We have also defined a collection of sound, syntax-directed verification condition generation rules for a simple imperative language that produce assertions in ILC. Lastly, we have identified a fragment of ILC, ILC$^-$, that is both decidable and closed under generation of verification conditions. If loop invariants and pre-/post-conditions 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.

Acknowledgements We would like to thank Kaustuv Chaudhuri, Manuel Fähndrich, Peter O'Hearn, and Frank Pfenning for fruitful discussions about this research. We are grateful to Jason Hickey and Aleksey Nogin for helping us with MetaPrl, the proof assistant we used to check our linear logical proofs.

References

1. Yang, Z.: Putting program analysis to work at Microsoft (2005) Princeton Computer Science Department Colloquium.
2. Detlefs, D.L.: An overview of the extended static checking system. In: The First Workshop on Formal Methods in Software Practice. (1996)
3. Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxes, J., Stata, R.: Extended static checking for java. In: ACM Conference on Programming Language Design and Implementation. (2002)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: CASSIS 2004. Number 3362 in LNCS (2004) 49–69
5. Necula, G.: Proof-carrying code. In: Twenty-Fourth ACM Symposium on Principles of Programming Languages, Paris (1997) 106–119
6. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: 28th ACM Symposium on Principles of Programming Languages. (2001)
7. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic. Number 2142 in LNCS (2001)

8. Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and Computation **110** (1994)
9. Lopez, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: PPDP. (2005)
10. Chaudhuri, K., Pfenning, F.: A focusing inverse method prover for first-order linear logic. In: CADE-20. (2005)
11. Miller, D.: A multiple-conclusion meta-logic. In: Ninth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press (1994) 272–281
12. Cervesato, I., Pfenning, F.: A linear logical framework. In: Information and Computation. (2000)
13. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework: The propositional fragment. In: Types for Proofs and Programs. (2004)
14. Ahmed, A., Jia, L., Walker, D.: Reasoning about hierarchical storage. In: IEEE Symposium on Logic in Computer Science. (2003)
15. Jia, L., Spalding, F., Walker, D., Glew, N.: Certifying compilation for a language with stack allocation. In: IEEE Symposium on Logic in Computer Science. (2005)
16. Smith, F., Walker, D., Morrisett, G.: Alias types. In: European Symposium on Programming, Berlin (2000) 366–381
17. Morrisett, G., Ahmed, A., Fluet, M.: L^3: A linear language with locations. In: 7th International Conference on Typed Lambda Calculi and Applications. (2005)
18. Hickey, Nogin, Constable, Aydemir, Barzilay, Bryukhov, Eaton, Granicz, Kopylov, Kreitz, Krupski, Lorigo, Schmitt, Witty, Yu: MetaPRL – A modular logical environment. In: IWHOLTP, LNCS (2003)
19. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Proceedings of the 16^{th} International Conference on Computer Aided Verification (CAV '04). (2004)
20. Jia, L., Walker, D.: ILC: A foundation for automated reasoning about pointer programs. Technical Report TR-738-05, Princeton University (2005)
21. Gentzen, G.: The Collected Papers of Gerhard Gentzen. North Holland (1969) Edited by M. E. Szabo.
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. (2002)
23. Lincoln, P., Mitchell, J., Scedrov, A., Shankar, N.: Decision problems for propositional linear logic. Annals of Pure and Applied Logic **56** (1992) 239–311
24. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FST TCS 2004. Number 3328 in LNCS (2004)
25. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Asian Symposium on Programming Languages and Systems. Number 3780 in LNCS (2005) 52–68
26. Deline, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: ACM Conference on Programming Language Design and Implementation. (2001)
27. Foster, J., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: ACM Conference on Programming Language Design and Implementation. (2002)
28. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: International conference on functional programming. (2003)
29. Zhu, D., Xi, H.: Safe Programming with Pointers through Stateful Views. In: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages, Springer-Verlag LNCS vol. 3350 (2005)
30. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: ACM Conference on Programming Language Design and Implementation, Montreal (1998) 249–257