

The Logical Approach to Low-level Stack Reasoning

Xinyu Jiang^{1,2}

Yu Guo^{1,2}

Yiyun Chen^{1,2}

¹Department of Computer Science and Technology
University of Science and Technology of China
Hefei, China
{wewewe, guoyu}@mail.ustc.edu.cn

²The Software Security Lab, USTC
Suzhou Institute of Advanced Study, USTC
Suzhou, China
yiyun@ustc.edu.cn

Abstract—Formal verification of low-level programs often requires explicit reasoning and specification of runtime stacks. Treating stacks naively as parts of ordinary heaps can lead to very complex specifications and make proof construction more difficult. In this paper, we develop a new logic system based on the idea of memory adjacency from previous work on stack typing. It is a variant of bunched logic and it can be easily integrated into separation logic to reason about the interaction between stacks and heaps. Our logic is especially convenient for reasoning about stack operations, and it greatly simplifies both the specification and the proof of properties about stack-allocated data.

Keywords—program verification; separation logic; stack typing; hoare logic; frame rule

I. INTRODUCTION

Runtime stack in low-level system programs is often exposed to programmers and is explicitly operated on. A runtime stack is usually a thin abstraction over a continuous heap region. We need such abstraction in programming because we do not want to reason about explicit addresses all the time. Instead, stack manipulation should mostly be done by more convenient operations such as push and pop whenever possible.

Program verification makes the need of stack abstraction even more urgent. Because when we specify and write proofs about low-level programs, if we treat stacks as ordinary memory heaps, the extra cost we must pay is even higher than in programming. In the next, we first show where such cost is coming from. Note that new notations or symbols in this section will be explained as they appear.

A. Problems

Suppose we have a stack that contains three storage cells and begins with the stack pointer sp . There is a cell in the heap with the address of 1 , and it holds a pointer to the third element of the stack. We use a simple memory model here, in which the storage cells are addressed by words instead of bytes. The memory shape is shown in the picture below:



To specify such memory shapes, we usually use the assertion language in separation logic [1], where $1 \mapsto w$

describes a memory cell with label 1 and content w , and $A * B$ stands for a heap that can be split into two disjoint parts, one of which satisfies A and the other satisfies B . We write the assertion for the memory shape with these annotations:

$$sp \mapsto w_1 * sp + 1 \mapsto w_2 * sp + 2 \mapsto w_3 * 1 \mapsto sp + 2$$

There are two properties about the memory shape in the picture above. First, 1 is not equal to any address of the stack elements. Second, whenever an element in the stack with the address $addr$ has a direct successor, its successor's address must be $addr + 1$. The key benefit of using disjoint conjunction operator $*$, is that the disjoint conjunction implies the first property. Which means we need not to write additional constraints such as $sp \neq 1$, $sp + 1 \neq 1$, etc. However, we still have to explicitly specify the second property by writing down every single address we care about in the whole stack.

Like inequality constraints, the explicit locations both complicate the specification and make proofs harder. On the specification side, if we want to reason about deeper stacks, we have to keep lots of addresses in each assertion. It is very easy to abuse the commutative property of $*$ by rewriting the assertion above, and mixing the assertion for normal heaps into the middle of assertions for stack entries. This makes the resulting specification much harder to understand and manipulate:

$$sp \mapsto w_1 * 1 \mapsto sp + 2 * sp + 1 \mapsto w_2 * sp + 2 \mapsto w_3$$

On the proof side, having explicit locations requires more unnecessary arithmetic reasoning. For example, if we push a word w into the stack and rename the new stack pointer $sp - 1$ as sp' , and we would like to operate the stack with sp' , we have to first prove that $sp - 1 = sp'$ implies $sp = sp' + 1$, $sp + 1 = sp' + 2$ and $sp + 2 = sp' + 3$, then rewrite the equations in the stack assertion. This kind of trivial proof must be done each time a program does a push or pop operation. If the stack holds many values, it becomes tedious to do manual proof or semi-automatic proof with proof assistants. Even for automatic theorem provers, proving them multiple times introduces severe performance penalty.

B. A New Assertion Language

Ahmed and Walker [6] used the idea of adjacent memory and adjacent conjunction in stack typing. Informally, two non-empty memories are adjacent if there is no “gap” between them. Their adjacent conjunction is essentially a type constructor. We introduce the adjacency conjunction as a logical connective like $*$ in separation logic, written as \otimes . $A \otimes B$ specifies two adjacent memory pieces satisfying A and B respectively. \otimes implies both the first and the second property: whenever $l_1 \mapsto w_1 \otimes l_2 \mapsto w_2$ holds for a memory, we know that l_2 must equal to $l_1 + 1$. It means that we can omit either l_1 or l_2 . We introduce an operator $[w]$ to express a memory cell containing w , and we do not care about its address. And we use $l \triangleright A$ to restrict A with an exact lower bound of l in A ’s memory. Then the assertion of the picture can be written as:

$$sp \triangleright [w_1] \otimes [w_2] \otimes [w_3] * 1 \mapsto sp + 2$$

The adjacency conjunction frees us from unnecessary care of locations and their orders. We no longer have to keep all locations explicit, and it is easy to see that mixing other assertions into stack assertions is not possible now. For example, the assertion below does not equal to the original assertion.

$$sp \triangleright [w_1] * 1 \mapsto sp + 2 * [w_2] \otimes [w_3]$$

Furthermore, the burden of arithmetic proof is reduced. After pushing the value w , the assertion for the stack is like:

$$sp' \triangleright [w] \otimes [w_1] \otimes [w_2] \otimes [w_3] * 1 \mapsto sp' + 3$$

And we just have to prove that $sp + 2 = sp' + 3$.

In addition to conjunction, the adjacent version of implication can be defined in a way similar to that of separation implication. The complete definition of our new assertion language can be found in Section 2.

C. Contributions and Paper Organization

The work in this paper can be seen as an experiment to “port” the concept of adjacency to Hoare-style program logics. Although this concept is simple, and using it in type systems is a success [2], [7], it was not clear whether traditional program logics can really benefit from it. But after we built our assertion language and used it in program logics, we were convinced that the adjacency approach can indeed make stack reasoning much easier.

We first discuss the inference rules of our language constructs in Section 2. Instead of proving their soundness syntactically, we give our assertion language a denotational semantics for a concrete memory model (Section 3) and prove these rules as lemmas. Most importantly, with a low-level program logic, we prove that the frame property of separation logic still holds for our adjacency conjunction.

Our logic shares great similarity with the original separation logic. Based on the similarity, we implement the new

assertion language as an extension to the assertion language for separation logic (Section 4). It means that we can specify programs that use both stacks and normal heaps in a single assertion language without losing the stack abstractions. Furthermore, it enables us to reason about the interaction between stacks and heaps. This feature is necessary for a useful assertion language, because for any serious low-level program, the stack is not the only memory it can operate on.

We believe that our logic does not just remove some complexity of specifications and reasoning in mind, but also greatly reduce our proof efforts and the proof checking time. Section 4 contains some simple benchmarks to illustrate this. Finally, we survey related work in Section 5 and conclude in Section 6.

II. SYNTAX AND RULES

In this section, we present the abstract syntax of our new assertion language and discuss the inference rules in our logic. We then give the assertion language a denotational semantics and show that all inference rules can be proved as lemmas using the denotations of language constructs.

A. Abstract Syntax

We reason about memories, locations and machine words. In this section, we keep words and memories abstract. For simplicity, we make locations and words have the same type. It is not a restriction of our logic, since we can always use a distinct representation for locations and define an encoding function to transform them to words.

$$\begin{array}{ll} (\text{location, word}) & l, w \in \dots \\ (\text{memory}) & \mathbb{H} \in \dots \end{array}$$

The syntax and meanings of pure formulas are like those in impredicative higher-order intuitionistic logic. We add it with the application and implication of spatial formulas, where $\mathbb{H} \Vdash A$ stands for “ \mathbb{H} satisfies A ”, and $A \Rightarrow B$ means “for all memories, B is weaker than A ”.

$$\begin{aligned} P &\triangleq \text{true} \mid \text{false} \mid P \rightarrow P \mid P \wedge P \mid P \vee P \\ &\mid \forall x. P \mid \exists x. P \mid \mathbb{H} \Vdash A \mid A \Rightarrow B \mid \text{constants} \dots \end{aligned}$$

A spatial formula can be an atom, another formula with a qualifier, or composed from other formulas by connectives. Atoms include Top and $[w]$. Top is the spatial version of true formula and holds for any memories. While $[w]$ describes any singleton heap with the content w .

We have universal and existential qualifiers for spatial formulas. For simplicity, we write them the same as qualifiers of pure formulas. Spatial \forall and \exists qualifiers are also required to be impredicative: in $\forall x. A$ and $\exists x. A$, x can be logic formulas, or other terms containing logic formulas.

Connectives are \otimes , $-\otimes$, $\otimes-$, $\&$ and \triangleright . We make \otimes , $-\otimes$, $\otimes-$ and $\&$ be right associative and in the same level of precedence, while the precedence of \triangleright is lower. \otimes

is the spatial connective of multiplication, which is non-commutative, and requires the memories specified by its left and right formulas to be adjacent. $-\otimes$ and $\otimes-$ are a pair of spatial implication connectives. $A-\otimes B$ specifies a memory that if any other memory satisfying A is adjacently appended to its right, the result will validate B . $\otimes-$ is the left-appending counterpart to $-\otimes$. $-\otimes$ and $\otimes-$ are quite like the separating implication connective in separation logic, except that we have two of them because the adjacency relation cannot exchange. $A\&B$ are lifted pure conjunction with an extra requirement that A and B must specify the same memory. $\&$ is usually mentioned as “additive conjunction” in linear logic, which is usually used in dealing with alias of pointers. It is called “additive” because in the distribution rule, its role is like the operator of addition, while \otimes is like the operator of multiplication. $1\triangleright A$ strengthens A by restricting the minimum location in the domain of A ’s memory. We can always write separation logic style formula $1\mapsto w$ of singleton heap as $1\triangleright[w]$.

$$A \triangleq \text{Top} \mid [w] \mid \forall x.A \mid \exists x.A \\ \mid A \otimes A \mid A -\otimes A \mid A \otimes -A \mid A \& A \mid 1\triangleright A$$

We also introduce some abbreviations of spatial formulas. $[_]$ specifies any single memory cell. $[w_1, \dots, w_n]$ specifies a memory block containing w_1 to w_n in order, and $\langle n \rangle$ means the memory is continuous, and its size is n . $\#A$ means A never describes an empty memory.

$$\begin{aligned} [_] &\triangleq \exists w.[w] \\ [w_1, \dots, w_n] &\triangleq [w_1] \otimes \dots \otimes [w_n] \\ \langle 0 \rangle &\triangleq !\text{true} \\ \langle n+1 \rangle &\triangleq \langle n \rangle \otimes [_] \\ \#A &\triangleq A \Rightarrow \text{Top} \otimes [_] \otimes \text{Top} \end{aligned}$$

B. Example

Our assertion language is capable of describing general stack pointers and pointer aliasing. Stack pointers require the possibility of addressing the store cells in the middle of the stack. Pointer aliasing is supported by the additive conjunction $A\&B$, which requires both the views of A and B to the stack to hold.

Suppose a stack beginning with the address 1. It contains two pointers l_1 and l_2 at its top. Each of the labels points to a cell in the middle of the stack, and the cells contains w_1 and w_2 . l_1 and l_2 may be alias pointers, which means they would point to the same cell. We care about the size and contents of neither the space between the pointers and the cells it points to, nor the space deeper in the stack than the two cells:

$$1\triangleright[l_1] \otimes [l_2] \otimes \\ (\text{Top} \otimes 1_1\triangleright[w_1] \otimes \text{Top}) \& (\text{Top} \otimes 1_2\triangleright[w_2] \otimes \text{Top})$$

The difference between \otimes and $\&$ is obvious from the assertion. $A \otimes B$ means that part of the whole memory is described by A , and the rest of it is described by B . While

$A\&B$ means the whole memory should satisfy A and B at the same time.

C. Inference Rules

As is in Fig.1, most of the rules are introduction and elimination rules of their corresponding operators. Instead of natural deduction style, we choose to specify the rules of our logic in a more proof-friendly manner. We do not use any context to gather hypotheses, and we reuse the pure formula’s properties whenever a hypothesis is needed. For example, the introduction rule of $A \Rightarrow B$ relies on the introduction rule of \rightarrow in intuitionistic logic.

The introduction and elimination rule of $A \otimes B$ and $[w]$ are not given in this section, because they are dependent to the representation of words and memories, which are still abstract.

Most of the rules of \otimes are the same as those of $*$ in separation logic except that \otimes does not have commutative property. The similarity comes with no surprise, because \otimes is also a valid multiplication connective of bunched logic. It is a restricted form of $*$.

D. The Frame Rule: An Informal Discussion

The main idea of the frame rule is that we could extend local specifications with extra memory specifications that are disjoint to the memory that local specification refers. It means that any piece of code can be reasoned locally about its own memory first, and composed with other code that operate on a different memory later. Since \otimes is just a limited version of $*$, we expect that it enjoys the same properties. They can be roughly expressed as:

$$\frac{\{A\} c \{B\} \quad C\#c}{\{A \otimes C\} c \{B \otimes C\}} \text{ (FRAME1)}$$

$$\frac{\{A\} c \{B\} \quad C\#c}{\{C \otimes A\} c \{C \otimes B\}} \text{ (FRAME2)}$$

where $\{A\} c \{B\}$ is a Hoare triple [12] with the code c , the precondition A and postcondition B . $C\#c$ means that c does not modify any free program variables in C .

The frame rules of our adjacent connectives are useful for function calls, for most of languages today are statically scoped, and any function body of their target code would not operate on its caller’s stack frame. These rules allow us to reason about functions’ local variables separately without the knowledge of invocation chains.

We give the programming language and the definition of Hoare triples in the next section, where the written form of our actual frame rules is a little different from the rules above.

III. SEMANTICS AND THE PROGRAM LOGIC

We give memories, values and our logic formulas denotations as mathematical objects. And we choose the shallow embedding technique, in which our language constructs have

$$\begin{array}{c}
\frac{\forall \mathbb{H}. \mathbb{H} \Vdash A \rightarrow \mathbb{H} \Vdash B}{A \Rightarrow B} \text{ (IMP-INTRO)} \\
\frac{\mathbb{H} \Vdash A \quad A \Rightarrow B}{\mathbb{H} \Vdash B} \text{ (IMP-ELIM)} \\
\frac{\mathbb{H} \Vdash A \otimes B \quad A \Rightarrow A'}{\mathbb{H} \Vdash A' \otimes B} \text{ (MONO1)} \\
\frac{\mathbb{H} \Vdash A \otimes B \quad B \Rightarrow B'}{\mathbb{H} \Vdash A \otimes B'} \text{ (MONO2)} \\
\frac{}{\mathbb{H} \Vdash \text{Top}} \text{ (TOP-INTRO1)} \\
\frac{\mathbb{H} \Vdash A \otimes B}{\mathbb{H} \Vdash A \otimes \text{Top}} \text{ (TOP-INTRO2)} \\
\frac{\mathbb{H} \Vdash A \otimes B}{\mathbb{H} \Vdash \text{Top} \otimes B} \text{ (TOP-INTRO3)} \\
\frac{\mathbb{H} \Vdash A \quad \mathbb{H} \Vdash B}{\mathbb{H} \Vdash A \& B} \text{ (APLUS-INTRO)} \\
\frac{\mathbb{H} \Vdash A \& B}{\mathbb{H} \Vdash A} \text{ (APLUS-ELIM1)} \\
\frac{\mathbb{H} \Vdash A \& B}{\mathbb{H} \Vdash B} \text{ (APLUS-ELIM2)}
\end{array}
\qquad
\begin{array}{c}
\frac{\mathbb{H} \Vdash (A \& B) \otimes C}{\mathbb{H} \Vdash (A \otimes C) \& (B \otimes C)} \text{ (DIST)} \\
\frac{\forall x. \mathbb{H} \Vdash F x}{\mathbb{H} \Vdash \forall x. F x} \text{ (FORALL-INTRO)} \\
\frac{\mathbb{H} \Vdash \forall x. F x}{\mathbb{H} \Vdash F y} \text{ (FORALL-ELIM)} \\
\frac{\mathbb{H} \Vdash F y}{\mathbb{H} \Vdash \exists x. F x} \text{ (EX-INTRO)} \\
\frac{\mathbb{H} \Vdash \exists x. F x \quad \forall y. F y \Rightarrow A}{\mathbb{H} \Vdash A} \text{ (EX-ELIM)} \\
\frac{\mathbb{H} \Vdash \mathbb{1} \triangleright A}{\mathbb{H} \Vdash \mathbb{1} \triangleright \mathbb{1} \triangleright A} \text{ (L-INTRO)} \\
\frac{\mathbb{H} \Vdash \mathbb{1} \triangleright A}{\mathbb{H} \Vdash A} \text{ (L-ELIM)} \\
\frac{\mathbb{H} \Vdash [\mathbf{w}] \otimes (\text{succ } \mathbb{1} \triangleright A)}{\mathbb{H} \Vdash \mathbb{1} \triangleright [\mathbf{w}] \otimes A} \text{ (L-MOVE1)} \\
\frac{\mathbb{H} \Vdash \mathbb{1} \triangleright [\mathbf{w}] \otimes A \quad \#A}{\mathbb{H} \Vdash [\mathbf{w}] \otimes (\text{succ } \mathbb{1} \triangleright A)} \text{ (L-MOVE2)}
\end{array}
\qquad
\begin{array}{c}
\frac{\mathbb{H} \Vdash \mathbb{1} \triangleright A \otimes B \quad \#A}{\mathbb{H} \Vdash (\mathbb{1} \triangleright A) \otimes B} \text{ (L-ASSOC1)} \\
\frac{\mathbb{H} \Vdash (\mathbb{1} \triangleright A) \otimes B}{\mathbb{H} \Vdash \mathbb{1} \triangleright A \otimes B} \text{ (L-ASSOC2)} \\
\frac{\mathbb{H} \Vdash A \otimes B \otimes C}{\mathbb{H} \Vdash (A \otimes B) \otimes C} \text{ (ASSOC1)} \\
\frac{\mathbb{H} \Vdash (A \otimes B) \otimes C}{\mathbb{H} \Vdash A \otimes B \otimes C} \text{ (ASSOC2)} \\
\frac{A \Rightarrow (B \otimes -C)}{(A \otimes B) \Rightarrow C} \text{ (AWAND-ELIM1)} \\
\frac{B \Rightarrow (A -\otimes C)}{(A \otimes B) \Rightarrow C} \text{ (AWAND-ELIM2)} \\
\frac{(A \otimes B) \Rightarrow C}{A \Rightarrow (B \otimes -C)} \text{ (AWAND-INTRO1)} \\
\frac{(A \otimes B) \Rightarrow C}{B \Rightarrow (A -\otimes C)} \text{ (AWAND-INTRO2)}
\end{array}$$

Figure 1. Inference Rules

no distinct representation, and are not interpreted to their denotational counterparts. They are directly represented as the mathematical objects, i.e. terms in the meta language.

A. The Mathematical Objects

We use well-formed terms in the Calculus of Inductive Construction (CiC) [8] as the denotational. CiC is an extension to Calculus of Constructions, which corresponds to higher-order constructive logic, and is capable to encode the constructive set theory. CiC removes the need to represent data types as λ and \forall terms by extending CC with (co)inductive definitions similar to data type definitions in functional languages, but does not break CC's consistency.

We use the annotations in [18]. We also adopt the definition of connectives \wedge , \vee and \exists as inductive type constructors. For inductive definitions, we use a more informal annotation in this section. For example, we use:

$$(nat) \quad n \triangleq O \mid S n$$

to describe the inductive type nat and its constructors O and S , rather than the annotation of $Ind()(nat : Set := O : nat, S : nat \rightarrow nat)$ in the original paper.

B. Embedding

The words are represented as integers, and the memories are defined as sets of pairs of location and word, the domain of a memory is a set of locations:

$$\begin{array}{lll}
(\text{word}) & \mathbf{w} & \triangleq \mathbf{w}_0 \mid \text{succ } \mathbf{w} \mid \text{pred } \mathbf{w} \\
(\text{memory}) & \mathbb{H} & \in \text{location} \rightarrow \text{word} \\
(\text{domain}) & \text{dom}(\mathbb{H}) & \triangleq \{ \mathbb{1} \mid \exists \mathbf{w}. \mathbb{H} \mathbb{1} = \mathbf{w} \}
\end{array}$$

The adjacency relation is represented as a predicate, and we also define the union of heaps:

$$\begin{array}{l}
\mathbb{H}_1 \circ \mathbb{H}_2 \triangleq \exists \mathbb{1}_1, \mathbb{1}_2. \mathbb{1}_2 = \text{succ } \mathbb{1}_1 \wedge \\
\quad \mathbb{1}_1 \in \text{dom}(\mathbb{H}_1) \wedge \mathbb{1}_2 \in \text{dom}(\mathbb{H}_2) \wedge \\
\quad (\forall \mathbb{1}. \mathbb{1} \in \text{dom}(\mathbb{H}_1) \rightarrow \mathbb{1} \leq \mathbb{1}_1) \wedge \\
\quad (\forall \mathbb{1}. \mathbb{1} \in \text{dom}(\mathbb{H}_2) \rightarrow \mathbb{1} \geq \mathbb{1}_2) \\
\mathbb{H}_1 \cup \mathbb{H}_2 \triangleq \lambda \mathbb{1}. \text{if}(\mathbb{H}_1 \mathbb{1} = \mathbf{w}_1) \text{ then } \mathbf{w}_1 \text{ else } \mathbb{H}_2 \mathbb{1}
\end{array}$$

Pure logic formulas are directly represented as types in the sort $Prop$. We do not give details of logic-type correspondence, and write them the same as in last section. Spatial formulas are represented as functions from memories to terms in $Prop$. Spatial application and implication are defined:

$$\begin{array}{l}
\mathbb{H} \Vdash A \triangleq A \ \mathbb{H} \\
A \Rightarrow B \triangleq \forall \mathbb{H}. \mathbb{H} \Vdash A \rightarrow \mathbb{H} \Vdash B
\end{array}$$

The definition of spatial formula atoms and connectives are as follows:

$$\begin{array}{l}
\text{Top} \triangleq \lambda \mathbb{H}. \text{true} \\
[\mathbf{w}] \triangleq \lambda \mathbb{H}. \exists \mathbb{1}. \mathbb{H} = \{(\mathbb{1}, \mathbf{w})\} \\
A \otimes B \triangleq \lambda \mathbb{H}. \exists \mathbb{H}_1, \mathbb{H}_2. \mathbb{H} \Vdash A \wedge \mathbb{H} \Vdash B \wedge \\
\quad \mathbb{H}_1 \circ \mathbb{H}_2 \wedge \mathbb{H} = \mathbb{H}_1 \cup \mathbb{H}_2 \\
A -\otimes B \triangleq \lambda \mathbb{H}_1. \forall \mathbb{H}_2. \mathbb{H}_1 \circ \mathbb{H}_2 \rightarrow \mathbb{H}_2 \Vdash A \rightarrow \\
\quad (\mathbb{H}_1 \cup \mathbb{H}_2) \Vdash B \\
A \otimes -B \triangleq \lambda \mathbb{H}_2. \forall \mathbb{H}_1. \mathbb{H}_1 \circ \mathbb{H}_2 \rightarrow \mathbb{H}_1 \Vdash A \rightarrow \\
\quad (\mathbb{H}_1 \cup \mathbb{H}_2) \Vdash B \\
A \& B \triangleq \lambda \mathbb{H}. \mathbb{H} \Vdash A \wedge \mathbb{H} \Vdash B \\
\mathbb{1} \triangleright A \triangleq \lambda \mathbb{H}. \mathbb{H} \Vdash A \wedge \mathbb{1} \in \text{dom}(\mathbb{H}) \wedge \\
\quad \forall \mathbb{1}'. \mathbb{1}' \in \text{dom}(\mathbb{H}) \rightarrow \mathbb{1} \leq \mathbb{1}'
\end{array}$$

C. Rules as Lemmas

With the straightforward semantics above, We do not have to do induction over our inference rules to prove their soundness. The rules are now all represented as lemmas in the meta language. For example, the rule of ASSOC-1 can be written as the following lemma:

$$\forall \mathbb{H}. \mathbb{H} \Vdash A \otimes B \otimes C \rightarrow \mathbb{H} \Vdash (A \otimes B) \otimes C$$

We can prove the rules following CiC's deduction style.

The introduction and elimination rule of $A \otimes B$ and $[w]$ are given below, for we have the memory model now.

$$\frac{\mathbb{H}_1 \Vdash A \quad \mathbb{H}_2 \Vdash B \quad \mathbb{H}_1 \circ \mathbb{H}_2}{(\mathbb{H}_1 \cup \mathbb{H}_2) \Vdash A \otimes B} \text{ (ASTAR-INTRO)}$$

$$\frac{\mathbb{H} \Vdash A \otimes B}{\exists \mathbb{H}' . \mathbb{H}' \subseteq \mathbb{H} \wedge \mathbb{H}' \Vdash A} \text{ (ASTAR-ELIM1)}$$

$$\frac{\mathbb{H} \Vdash A \otimes B}{\exists \mathbb{H}' . \mathbb{H}' \subseteq \mathbb{H} \wedge \mathbb{H}' \Vdash B} \text{ (ASTAR-ELIM2)}$$

$$\frac{\mathbb{H} \mathbf{1} = w}{\mathbb{H} \Vdash A \otimes \mathbf{1} \triangleright [w] \otimes B} \text{ (HAS-INTRO)}$$

$$\frac{\mathbb{H} \Vdash A \otimes \mathbf{1} \triangleright [w] \otimes B}{\mathbb{H} \mathbf{1} = w} \text{ (HAS-ELIM)}$$

D. Interaction with Heap Predicates

We can also embed the separation logic formulas for normal heaps in CiC. We first define disjointness of memory:

$$\mathbb{H}_1 \perp \mathbb{H}_2 \triangleq \text{dom}(\mathbb{H}_1) \cap \text{dom}(\mathbb{H}_2) = \emptyset$$

The separating conjunction and implication, as well as the lifting of pure logic formulas, are embedded in this way:

$$\begin{aligned} A * B &\triangleq \lambda \mathbb{H}. \exists \mathbb{H}_1, \mathbb{H}_2. \mathbb{H} \Vdash A \wedge \mathbb{H} \Vdash B \wedge \\ &\quad \mathbb{H}_1 \perp \mathbb{H}_2 \wedge \mathbb{H} = \mathbb{H}_1 \cup \mathbb{H}_2 \\ A \multimap B &\triangleq \lambda \mathbb{H}_1. \forall \mathbb{H}_2. \mathbb{H}_1 \perp \mathbb{H}_2 \rightarrow \mathbb{H}_2 \Vdash A \rightarrow \\ &\quad (\mathbb{H}_1 \cup \mathbb{H}_2) \Vdash B \\ !P &\triangleq \lambda \mathbb{H}. P \end{aligned}$$

It is now possible for us to write complex invariants over the interaction between stacks and heaps. For example, we can specify multiple stack-like structures in a heap. It is quite important for programs with explicit continuations, coroutines, or user threads, and use stacks for function call.

E. The Program Logic

As the title shows, we focus on low-level reasoning. The tradition shape of Hoare triples are defined with higher-level programming languages, and with support of named variables and native control structures(loops, function calls). They do not work well with low-level language features like jumps.

Here we adopt the verification framework of certified programming language(CAP) [9] as our programming and reasoning model. CAP is a Hoare style verification framework for assembly code. It uses CPS reasoning, which only

makes use of preconditions, and postconditions are just preconditions for continuations. We make two significant modifications to CAP. First, we change the machine model of the original paper to x86-style. Second, we merge our assertion language into the inference rules for stack-related instructions.

The counterpart of a Hoare triple in CAP is a judgment of well-formed instruction sequence:

$$\Psi \vdash \{a\} \mathbb{I}$$

where \mathbb{I} is an instruction sequence of assembly instructions ending with a non-conditional jump. a is a predicate over program states, and Ψ is a code heap specification, which is isomorphic to the code heap, except that it contains predicates instead of instruction sequences. Their definitions are given as:

$$\begin{aligned} (\text{CdHpSpec}) \quad \Psi &\in \text{location} \rightarrow \text{Pred} \\ (\text{Pred}) \quad a &\triangleq \text{State} \rightarrow \text{Prop} \\ (\text{State}) \quad \mathbb{S} &\triangleq (\mathbb{H}, \mathbb{R}) \\ (\text{Regfile}) \quad \mathbb{R} &\triangleq \{r \rightsquigarrow w\}^* \\ (\text{Register}) \quad r &\triangleq ax \mid bx \mid cx \mid dx \\ &\quad \mid sp \mid bp \mid si \mid di \\ (\text{ISeq}) \quad \mathbb{I} &\triangleq i; \mathbb{I} \mid \text{call } f \mid \text{retw} \mid \dots \\ (\text{Instr}) \quad i &\triangleq \text{pushw } r \mid \text{popw } r \mid \dots \end{aligned}$$

We only list some instructions that are relevant to stack operations. The operational semantics of the machine model is omitted here, and can be found in [19]. We overload the implication and spatial conjunction for state predicates. Their definition is in Fig.3, which also contains some definitions useful for the frame rule.

The inference rules of instruction sequence with instruction `pushw`, `popw`, `call` and `retw` are in Fig.3. For simplicity and clarity, we abbreviate $\lambda(\mathbb{H}, \mathbb{R}). a(\mathbb{H}, \mathbb{R})$ as a in these rules, and write $f \ x_1 \ x_2 \ \dots$ as $f(x_1, x_2, \dots)$. The rules for the whole program and code heaps are the same as the original paper, and we do not list them out.

The inference rules for `pushw` and `popw` are quite similar to axiomatic semantics of Hoare logic. Notice that the rules for `call` and `retw` are a little strange when first looking. They roughly mean that: the precondition in the address of the jump destination is weaker than the precondition after the stack is mutated.

With the machine model and program logic, we are now able to make the the frame rules concrete and then prove them. Some auxiliary definitions, such as the monotonicity of state predicate, the locality of code heap specification, and the precision of spatial asserts are in Fig.3. The informal explanation of these definitions can be found in [13], [17]. We give the frame properties for CAP below. They can be proved by induction over the instruction sequence \mathbb{I} .

$$\begin{aligned}
a \Rightarrow a' &\triangleq \forall \mathbb{S}. a \mathbb{S} \rightarrow a' \mathbb{S} \\
\text{precise}(A) &\triangleq \forall \mathbb{H}, \mathbb{H}', \mathbb{H}'' . \mathbb{H}' \subseteq \mathbb{H} \rightarrow \mathbb{H}'' \subseteq \mathbb{H} \rightarrow \mathbb{H}' \Vdash A \rightarrow \mathbb{H}'' \Vdash A \rightarrow \mathbb{H}' = \mathbb{H}'' \\
\text{mono}(a) &\triangleq \forall \mathbb{H}, \mathbb{H}', \mathbb{R}. a(\mathbb{H}, \mathbb{R}) \rightarrow \mathbb{H} \subseteq \mathbb{H}' \rightarrow a(\mathbb{H}', \mathbb{R}) \\
\text{local}(\Psi) &\triangleq \forall (\mathbf{f}, a) \in \Psi . \text{mono}(a) \\
a * A &\triangleq \lambda(\mathbb{H}, \mathbb{R}) . \exists \mathbb{H}', \mathbb{H}'' . \mathbb{H} = \mathbb{H}' \cup \mathbb{H}'' \wedge \mathbb{H}' \perp \mathbb{H}'' \wedge a(\mathbb{H}', \mathbb{R}) \wedge \mathbb{H}'' \Vdash A \\
a \otimes A &\triangleq \lambda(\mathbb{H}, \mathbb{R}) . \exists \mathbb{H}', \mathbb{H}'' . \mathbb{H} = \mathbb{H}' \cup \mathbb{H}'' \wedge \mathbb{H}' \circ \mathbb{H}'' \wedge a(\mathbb{H}', \mathbb{R}) \wedge \mathbb{H}'' \Vdash A \\
A \otimes a &\triangleq \lambda(\mathbb{H}, \mathbb{R}) . \exists \mathbb{H}', \mathbb{H}'' . \mathbb{H} = \mathbb{H}' \cup \mathbb{H}'' \wedge \mathbb{H}' \circ \mathbb{H}'' \wedge a(\mathbb{H}'', \mathbb{R}) \wedge \mathbb{H}' \Vdash A
\end{aligned}$$

Figure 2. Auxiliary Definitions

$$\begin{aligned}
&\frac{a \Rightarrow a' \quad \Psi \vdash \{a'\} \mathbb{I}}{\Psi \vdash \{a\} \mathbb{I}} \text{ (WEAKEN)} \\
&\frac{\Psi \vdash \{\mathbb{R}(sp) \triangleright [\mathbf{w}] \otimes a * a' * !\mathbb{R}(r) = \mathbf{w}\} \mathbb{I} \quad \#a}{\Psi \vdash \{[_]\} \otimes \mathbb{R}(sp) \triangleright a * a' * !\mathbb{R}(r) = \mathbf{w}\} \text{pushw } r; \mathbb{I}} \text{ (PUSH)} \\
&\frac{\Psi \vdash \{[_] \otimes \mathbb{R}(sp) \triangleright a * a' * !\mathbb{R}(r) = \mathbf{w}\} \mathbb{I}}{\Psi \vdash \{\mathbb{R}(sp) \triangleright [\mathbf{w}] \otimes a * a'\} \text{popw } r; \mathbb{I}} \text{ (POP)} \\
&\frac{\mathbb{R}(sp) \triangleright [\mathbf{f}_{ret}] \otimes a * a' \Rightarrow \Psi(\mathbf{f}) \quad \#a}{\Psi \vdash \{[_] \otimes \mathbb{R}(sp) \triangleright a * a'\} \text{call } \mathbf{f}; \mathbf{f}_{ret} : \mathbb{I}} \text{ (CALL)} \\
&\frac{[_] \otimes \mathbb{R}(sp) \triangleright a * a' \Rightarrow \Psi(\mathbf{f}_{ret})}{\Psi \vdash \{\mathbb{R}(sp) \triangleright [\mathbf{f}_{ret}] \otimes a * a'\} \text{retw}} \text{ (RET)}
\end{aligned}$$

Figure 3. Inference Rules for Instruction Sequence

$$\begin{aligned}
&\frac{\Psi \vdash \{a\} \mathbb{I} \quad \text{precise}(A) \quad \text{local}(\Psi)}{\Psi \vdash \{a * A\} \mathbb{I}} \text{ (FRAME-SEP)} \\
&\frac{\Psi \vdash \{a\} \mathbb{I} \quad \text{precise}(A) \quad \text{local}(\Psi)}{\Psi \vdash \{a \otimes A\} \mathbb{I}} \text{ (FRAME-ADJ1)} \\
&\frac{\Psi \vdash \{a\} \mathbb{I} \quad \text{precise}(A) \quad \text{local}(\Psi)}{\Psi \vdash \{A \otimes a\} \mathbb{I}} \text{ (FRAME-ADJ2)}
\end{aligned}$$

The frame rules above are in fact first-order ones. The higher-order frame rules exist for several CAP-like logic frameworks [13], [14], we expect all of them to hold for the adjacency connectives.

IV. IMPLEMENTATION

All work in this paper, including the shallow embedding of our assertion language, the proof of inference rules as lemmas, the program logic with its frame rules, and some experiments using our logic, are implemented in the Coq proof assistant. Fortunately we were able to start with a solid code base consisting of a library of lemmas and tactics for separation logic assertions, and the mechanized program logic CAP. The implementation of the assertion language and its inference rules is an extension to the separation logic assertion language library. Our program logic is not just an extension to the original SCAP implementation, because our logic modifies some of CAP's inference rules.

We implement some automatic tactics on top of our inference rules. The tactics include: finding explicit labels

in spatial formulas, moving labels by arbitrary number of steps, and the splitting/merging of unused stack space.

Our implementation consists of approximately 10k lines of Coq, including the code and proof scripts. Most of our code is reusable, and we distribute it as a library [19].

Based on the new library for our assertion language and the adapted program logic, we did some benchmark by proving the same program twice: once using only the abstractions provided by the original separation logic, and once using our new abstractions for stacks. We choose a small sequential program [19] calculating the dot product of two vectors as the example. We try to make the results more illustrative by removing proof code that is irrelevant to stack operations, like the proof scripts that constructs and destructs the predicates in the meta language. We compared their length of specification and proof measured with lines of code, their proof checking time in seconds, and the size of their compiled proof terms in kbytes. Our benchmark is done using Coq 8.1, on two 1.6GHz quad core Xeon CPUs with 4GB RAM.

	Pure seplogic	New abstraction
LOC	1671	1214
time(sec)	15.6	5.7
size(kbytes)	881.6	501.4

The benchmark above is not quite reliable, and its main purpose is to show the idea, because the example code is short and is not a typical low-level program. We are interested to see benchmarks of reasoning larger and more realistic low-level examples such as OS kernels, garbage collectors and virtual machines. Unfortunately, the program logic in this paper is rather simple, and does not scale well for complex programs. There are already much proof of low-level programs [20], [21], using more expressive program logics [10], [11]. Adding our new abstractions to these program logics can make such benchmarks possible. This will be the main direction of our future work.

V. RELATED WORK

Although the motivation of our logic was directly from our practise in program verification, our work borrow ideas from bunched implications(BI) [4] and typed systems with linear logic[16] concepts. We discuss the related work mainly in these two domains.

A. Logic of Bunched Implications

Separation logic suggested by John Reynolds [1] is an instance of BI which facilitates reasoning about the shape of the store, including information hiding in the presence of pointers. It treats all data heaps with no difference, and does not focus on how to simplify stack-based memory management.

The most exciting and distinguishing features of separation logic is the frame rule [1] and its higher-order forms [17], which improve the modularity of reasoning. We proved that the frame rule holds for the our newly introduced adjacent connective.

Istiaq and O’Hearn [5] used another form of BI to deal with the compatibility of spatial connectives with classical logic. BI in their paper is similar to separation logic. They did not keep the difference between stacks and normal heaps as well. In O’Hearn’s work on bunched typing [3], the Curry-Howard cousin to bunched logic, he briefly talked about the adjacent connectives as “Non-commutative connectives”, but he did not discuss them in detail.

The logic approach to stack typing proposed by David Walker [6] provides more integration by supporting interaction between stacks and normal heaps. It is capable to type pointers to cells on the stack. It is more of a logic system than a type system in the common sense, for its “singleton types” is a form of dependent type, and roughly equals to the singleton heap predicates of separation logic. It does not investigate how to eliminate unnecessary explicit labels and the reasoning about them.

B. Type Systems with Linear Logic

The earliest idea of commutative adjacency annotation can be found in the product type $\tau_1 \times \tau_2$ in ML-like languages, since they usually allocate a pair’s first and second element in adjacent memory spaces. The non-commutative linear logic [23] formalized the concept of the assertion order, but the order does not imply adjacency unless we give it a proper semantic model.

Stack-based typed assembly language by Morrisett *et al.* [2] is the first to use both of the ideas above to type stacks. Its type constructor $::$ makes non-commutative product types and enforces adjacency. It lacks the integration of stack type pointers and heap pointers, and does not handle general pointers that may be aliased into the stack.

Perry *et al.* [7] introduced a type system that is more expressive than Stack-based typed assembly language, while retains the decidability of type checking. Their multiplication connective is like ours, but their addition connective is different. The type $\sigma \wedge \{1 : \tau\}$ in their system cannot be directly rewritten as $A \& (1 \triangleright [w])$ in our assertion language, but is approximately equal to $A \& (\text{Top} \otimes (1 \triangleright [w] \otimes \text{Top}))$.

The most important difference between these type systems and our logic, is that they work on different abstract layers and handle different grades of safety policies. These type

systems enjoy fully decidable type checking and strong correspondence to their source languages’ type systems. But they are not able to capture complicated invariants for handwritten assembly or those generated by some aggressive optimizing compilers. While our logic sacrifices the decidability and automation, since explicit proof annotations are required. But it could express a relatively larger area of program properties.

C. Other Work

Our program logic is an adapted CAP [9]. CAP uses a syntactic inference rule like traditional typing rules in type systems, can be seen as a logical counterpart to TAL. There are many extensions to CAP [10], [11] that support stack-like control structures, general code pointers, etc. Their current inference rules directly use the machine model’s operational semantics, which is rather primitive. It is possible to integrate our assertion language to their inference rules to make them higher level and easier to use.

Peterson *et al.* [22] proposed a programming language with a linear type system to support memory allocation and data layout. Their “ordered multiplicative” is in fact an adjacency type constructor. They did not show how to support commutative linear logic and ordered logic simultaneously, and they focused on higher-level languages.

Smallfoot built by Berdine *et al.* [15] is an automatic verification tool that checks separation logic specifications of sequential and concurrent programs via symbolic execution. The distributive version of smallfoot is for a higher-level language, but the main idea of symbolic execution can be used for the automation of reasoning about low-level programs. Our work can be used to improve the performance of symbolic execution with stacks.

VI. CONCLUSION

We proposed a logic for reasoning about low-level programs that operate on stacks based on the concept of adjacency. Like separation logic, our logic is a variant of bunched logic. Our logic can be easily merged into separation logic as an extension. The inference rules of the assertion language and the program logic have all been proved sound in the Coq proof assistant.

There are mainly three future directions for our work. The first is to apply our new abstractions and assertion languages to more expressive program logics. Second, we believe that more degrees of automation can be achieved, since our assertion language makes the assertions’ shape more orderly, and tactics can make use of these shape invariants. Third, we are quite sure that higher-order frame rules should also hold for adjacent connectives and we plan to implement them and prove their soundness in Coq.

ACKNOWLEDGEMENTS

The authors are very grateful to Zhong Shao and Xinyu Feng and anonymous referees for their constructive comments. This research is supported in part by grants from National Natural Science Foundation of China (under grant No. 90718026), China Postdoctoral Science Foundation (under grant No. 20080430770) and Natural Science Foundation of Jiangsu Province, China (under grant No. BK2008181). Any opinions, findings and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

REFERENCES

- [1] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science*, 2002.
- [2] G. Morrisett, K. Crary, N. Glew and D. Walker. Stack-Based Typed Assembly Language. In *Workshop on Types in Compilation*, Kyoto, Japan, March 1998.
- [3] P. O'Hearn. *On bunched typing*. Journal of Functional Programming, 13(4), 747796, 2003.
- [4] P. O'Hearn and D. Pym. *The Logic of Bunched Implications*. Bulletin of Symbolic Logic 5(2), 215-244, 1999.
- [5] S. Istiaq and P. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *POPL'01*, pages 14-26, London, UK, January 2001.
- [6] Amal Ahmed and David Walker, *The Logical Approach to Stack Typing*. In *TLDI'03*, New Orleans, Louisiana, USA, January 2003.
- [7] F. Perry, C. Howblitzel and J. Chen. Simple and Flexible Stack Types. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, July 2007.
- [8] F. Pfenning and C. Mohring. *Inductively defined types in the Calculus of Constructions*. Lecture Notes in Computer Science, Volume 442, 1990.
- [9] D. Yu, N. Hamid, and Z. Shao, Building Certified Libraries for PCC: Dynamic Storage Allocation. In *ESOP'03*, Warsaw, Poland, April 2003.
- [10] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *PLDI'06*, Ottawa, Canada, pages 401-414, June 2006.
- [11] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *POPL'06*, Charleston, South Carolina, pages 320-333, January 2006.
- [12] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576580,583 October 1969.
- [13] X. Feng and Z. Shao. Local Reasoning and Information Hiding in SCAP. Unpublished manuscript.
- [14] H. Cai, X. Feng, Z. Shao and G. Tan. Towards Logic Reasoning about Code Pointers. Unpublished manuscript.
- [15] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO 2006*.
- [16] J. Girard. *Linear Logic*. Theoretical Computer Science, 50:1-102, 1987.
- [17] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL'04*, Venice, Italy, January 2004.
- [18] The Coq Development Team. The Coq Proof Assistant Reference manual. <http://pauillac.inria.fr/coq/doc/main.html>, 2006.
- [19] Software Security Lab, USTC. Coq Implementation of The Logic for Stack Reasoning. <http://ssg.ustcsz.edu.cn/~cat/lassert-code.tar.gz>, 2009.
- [20] X. Feng, Z. Shao, Y. Dong and Y. Guo. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads. In *PLDI'08*, Tucson, Arizona, pages 170-182, June 2008.
- [21] A. McCreight, Z. Shao, C. Lin, and L. Li. A General Framework for Certifying Garbage Collectors and Their Mutators. In *PLDI'07*, San Diego, CA, pages 468-479, June 2007.
- [22] L. Petersen, R. Harper, K. Crary, F. Pfenning. A type theory for memory allocation and data layout. In *POPL'03*, 2003.
- [23] J. Polakow and F. Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA)*, 1999.