New Generation Computing, 29(2011)3-29
Ohmsha, Ltd. and Springer

Weak Updates and Separation Logic

Gang TAN
Computer Science & Engineering,
Lehigh University
19 Memorial Drive West, Bethlehem,
PA 18015, USA
gtan@cse.lehigh.edu
Zhong SHAO
Department of Computer Science,
Yale University
P.O.Box 208285 New Haven,
CT 06520-8285, USA
Xinyu FENG
School of Computer Science and Technology,
University of Science and Technology of China
166 Ren'ai Road Suzhou Institute for Advanced Study, USTC
Suzhou, Jiangsu 215123, CHINA
Hongxu CAI
Google Inc., 1600 Amphitheatre Parkway
Mountain View, CA 94043, USA

Received 31 March 2010
Revised manuscript received 17 August 2010

Abstract Separation logic provides a simple but powerful technique for reasoning about low-level imperative programs that use shared data structures. Unfortunately, separation logic supports only "strong updates," in which mutation to a heap location is safe only if a unique reference is owned. This limits the applicability of separation logic when reasoning about the interaction between many high-level languages (e.g., ML, Java, C#) and low-level ones since the high-level languages do not support strong updates. Instead, they adopt the discipline of "weak updates," in which there is a global "heap type" to enforce the invariant of type-preserving heap updates. We present SL^{W}, a logic that extends separation logic with reference types and elegantly reasons about the interaction between strong and weak updates. We describe a semantic framework for reference types, which is used to prove the soundness of SL^{W}. Finally, we show how to extend SL^{W} with concurrency.

Keywords: Weak Updates, Separation Logic, Type Systems, Mutable References, Language Interoperation.

§1 Introduction

Reasoning about mutable, aliased heap data structures is essential for proving properties or checking safety of imperative programs. Two distinct approaches perform such kind of reasoning: separation logic, and a type-based approach employed by many high-level programming languages.

Extending Hoare Logic, the seminal work of separation logic[12, 17] (abbreviated to SL hereafter) is a powerful framework for proving properties of low-level imperative programs. Through its separating conjunction operator and frame rule, SL supports local reasoning about heap updates, storage allocation, and explicit storage deallocation.

SL supports "strong updates": as long as a unique reference to a heap cell is owned, the heap-update rule of SL allows the cell to be updated with any value:

$$\overline{\{(e \mapsto -) * \mathtt{p}\}[e] := e'\{(e \mapsto e') * \mathtt{p}\}} \tag{1}$$

In the above heap-update rule, there is no restriction on the new value e'. Hereafter, we refer to heaps with strong updates as *strong heaps*. Heap cells in strong heaps can hold values of different types at different times of program execution.

Most high-level programming languages (e.g., Java, C#, and ML), however, support only "weak updates." In this paradigm, programs can perform only type-preserving heap updates. There is a global "heap type" that tells the type of every allocated heap location. The contents in a location have to obey the prescribed type of the location in the heap type, at any time. Managing heaps with weak updates is a simple and type-safe mechanism for programmers to access memory. As an example, suppose an ML variable has type "τ ref" (i.e., it is a reference to a value of type τ). Then any update through this reference with a new value of type τ is type safe and does not affect other types, even in the presence of aliases and complicated points-to relations. Hereafter, we refer to heaps with weak updates as *weak heaps*.

This article is concerned with the interaction between strong and weak updates. Strong-update techniques are more precise and powerful, allowing destructive memory updates and explicit deallocation. But aliases and uniqueness have to be explicitly tracked. Weak-update techniques allow type-safe management of memory without tracking aliases, but types of memory cells can never change. A framework that mixes strong and weak updates enables a trade-off between precision and scalability.

Such a framework is also useful for reasoning about *multilingual programs*. Most real-world programs are developed in multiple programming languages. Almost all high-level languages provide foreign function interfaces for interfacing with low-level C code (for example, the OCaml/C FFI, and the Java Native Interface). Real-world programs consist of a mixture of code in both high-

level and low-level languages. A runtime state for such a program conceptually contains a union of a weak heap and a strong heap. The weak heap is managed by a high-level language (e.g., Java), accepts type-preserving heap updates, and is garbage-collected. The strong heap is managed by a low-level language, accepts strong updates, and its heap cells are manually recollected. To check the safety and correctness of multilingual programs, it is of practical value to have one framework that accommodates both strong and weak updates.

Since separation logic (SL) supports strong updates, one natural thought to mix strong and weak updates is to extend SL with types so that assertions can also describe values in weak heaps. That is, in addition to regular SL assertions, we add "$e \mapsto \tau$," which specifies a heap with a single cell and the cell holds a value of type τ. This scheme, however, would encounter two challenges.

First, when general reference types are allowed in "$e \mapsto \tau$," care must be taken to avoid unsoundness. An example demonstrating this point follows:

$$\{(x \mapsto 4) * (y \mapsto \mathsf{even\ ref})\}\ [x] := 3\ \{(x \mapsto 3) * (y \mapsto \mathsf{even\ ref})\} \tag{2}$$

The example is an instantiation of the heap-update rule in (1), assuming "$e \mapsto \tau$" is a valid assertion. The precondition states that y points to a heap cell whose contents are of type "$\mathsf{even\ ref}$". If "τ ref" is interpreted as a set of locations whose contents are of type τ, then the precondition is met on a heap where y points to x. However, the postcondition will not hold on the new heap after the update because x will point to an odd number. Therefore, the above rule is sound only if the model of "$\mathsf{even\ ref}$" does not permit y to point to x.

The second challenge of adding types to SL is how to prove its soundness with mixed SL assertions and types. Type systems are usually proved sound following a syntactic approach,[24] where types are treated as syntax. Following the tradition of Hoare Logic, SL's soundness is proved through a semantic model, and SL assertions are interpreted semantically. There is a need to resolve the differences between syntactic and semantic soundness proofs.

In this article, we propose a hybrid logic, SL^{W}, which mixes SL and a type system. Although the logic is described in a minimal language and type system, it makes a solid step toward a framework that reasons about the interaction between high-level and low-level languages. The most significant technical aspects of the logic are as follows:

- SL^{W} extends SL with a simple type system. It employs SL for reasoning about strong updates, and employs the type system for weak updates. Most interestingly, SL^{W} mixes SL assertions and types. It accommodates cross-boundary pointers (from weak to strong heaps and vice versa) by distinguishing between pointers to weak heaps and pointers to strong heaps through the type system. SL^{W} is presented in Section 2.
- To resolve the differences between syntactic types and semantic assertions, we propose a semantic model of types. Our model of reference types follows a fixed-point approach and allows us to define a denotational model of SL^{W} and prove its soundness. The model of SL^{W} is presented in Section 3.

- We extend SL^{W} with concurrency in Section 4. We add an atomic-block command and a parallel-execution command to the language, extend SL^{W} with new rules for concurrency, and adjust the semantic model for soundness. The extension to the logic demonstrates an important difference between weak and strong heaps: weak heaps can be accessed safely without synchronization, while safe access to strong heaps requires synchronization.

A preliminary version of this article was published in the Proceedings of the Seventh Asian Symposium on Programming Languages and Systems (APLAS 2009).[19] The differences between the conference version and this article is described as follows. First, we streamlined the presentation and added proofs to the most important lemmas and theorems. Second, in this article we show how to extend the syntax and semantics of SL^{W} to accommodate concurrency. The change to a concurrent setting requires some changes to the original SL^{W}. For instance, there is a need to treat variables as resources (similar to the way how memory cells are treated in separation logic). We reuse local variable types for this purpose.

§2 SL^{W}: Separation Logic With Weak Updates

We next describe SL^{W}, an extension of SL that incorporates reasoning about weak heaps. In Section 2.1, we describe a minimal language that enables us to develop SL^{W}. Rules of SL^{W} are presented in Section 2.2 and examples of using the logic in Section 2.3.

Notation convention. For a map f, we write $f[x \rightsquigarrow y]$ for a new map that agrees with f except it maps x to y. For two finite maps f_1 and f_2, we write $f_1 \perp f_2$ when their domains are disjoint. The notation $f_1 \uplus f_2$ is the union of f_1 and f_2 when $f_1 \perp f_2$, and undefined otherwise. The notation $f_1 \uplus f_2$, implicitly carries the restriction $f_1 \perp f_2$. We write $f \setminus X$ for a new map resulting from removing elements in the set X from the domain of f.

2.1 Language Syntax and Semantics

Figure 1 presents the syntax of the programming language in which we will develop SL^{W}. The language is the imperative language used by Hoare, [8] augmented with a set of commands for manipulating heap data structures. It is similar to the one used in Reynolds' presentation of SL.[17] Informally, the command "$x := [e]$" loads the contents at location e into variable x; "$[x] := e$" updates the location at x with the value of e; "$x := \mathsf{alloc}(e)$" allocates a new location, initializes it with the value of e, and assigns the new location to x;

$$
\begin{array}{rrcl}
(\textit{Command}) & c & ::= & \cdots \mid x := [e] \mid [x] := e \mid x := \mathsf{alloc}(e) \mid \mathsf{free}(e) \mid c_1; c_2 \mid \epsilon \\
(\textit{Expression}) & e & ::= & x \mid v \mid op(e_1, \ldots, e_n) \\
(\textit{Value}) & v & ::= & n
\end{array}
$$

Fig. 1 Language Syntax

"$\mathsf{free}(e)$" deallocates the location e. "$c_1; c_2$" is the usual syntax for sequencing two commands. ϵ is the empty command, used in the operational semantics to signal the termination of a program.

An expression is either a variable x, a value v, or an operator op with a list of operands. We assume there is an infinite number of variables. All values, including heap locations, are natural numbers. Treating heap locations as natural numbers allows address arithmetic (i.e., adding a number to a location). We sometimes write ℓ for a heap location, but it is just a number.

Figure 2 presents the formal operational semantics of the language. A state consists of a store s (a map from variables to values), a heap h (a map from locations to values), and a command. Commands bring one state to another state and their semantics is formally defined by a step relation $\longrightarrow$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

A state may step to a wrong state. For instance, a state whose next instruction to execute is $[x] := e$ goes to a wrong state when its store does not contain one of the variables in $\{x\} \cup \mathsf{fvar}(e)$ or the value of x is not in the domain of its heap.

$$
\begin{array}{lcll}
(\mathit{State}) & \sigma & ::= & (s, h, c) \\
(\mathit{Store}) & s & ::= & \{x_1 \rightsquigarrow v_1, \ldots, x_n \rightsquigarrow v_n\} \\
(\mathit{Heap}) & h & ::= & \{\ell_1 \rightsquigarrow v_1, \ldots, \ell_n \rightsquigarrow v_n\}
\end{array}
$$

$$
\frac{s(e) = \ell \in \mathrm{dom}(h)}{(s, h, x := [e]) \longrightarrow (s[x \rightsquigarrow h(\ell)], h, \epsilon)} \qquad \frac{\{x\} \cup \mathsf{fvar}(e) \not\subseteq \mathrm{dom}(s) \text{ or } s(e) \notin \mathrm{dom}(h)}{(s, h, x := [e]) \longrightarrow \mathsf{wrong}}
$$

$$
\frac{s(x) = \ell \in \mathrm{dom}(h)}{(s, h, [x] := e) \longrightarrow (s, h[\ell \rightsquigarrow s(e)], \epsilon)} \qquad \frac{\{x\} \cup \mathsf{fvar}(e) \not\subseteq \mathrm{dom}(s) \text{ or } s(x) \notin \mathrm{dom}(h)}{(s, h, [x] := e) \longrightarrow \mathsf{wrong}}
$$

$$
\frac{\ell \notin \mathrm{dom}(h)}{(s, h, x := \mathsf{alloc}(e)) \longrightarrow (s[x \rightsquigarrow \ell], h \uplus \{\ell \rightsquigarrow s(e)\}, \epsilon)}
$$

$$
\frac{\{x\} \cup \mathsf{fvar}(e) \not\subseteq \mathrm{dom}(s)}{(s, h, x := \mathsf{alloc}(e)) \longrightarrow \mathsf{wrong}}
$$

$$
\frac{s(e) = \ell \in \mathrm{dom}(h)}{(s, h, \mathsf{free}(e)) \longrightarrow (s, h \setminus \{\ell\}, \epsilon)} \qquad \frac{\mathsf{fvar}(e) \not\subseteq \mathrm{dom}(s) \text{ or } s(e) \notin \mathrm{dom}(h)}{(s, h, \mathsf{free}(e)) \longrightarrow \mathsf{wrong}}
$$

$$
\frac{(s, h, c_1) \longrightarrow (s', h', c_1')}{(s, h, c_1; c_2) \longrightarrow (s', h', c_1'; c_2)} \qquad \frac{}{(s, h, \epsilon; c_2) \longrightarrow (s, h, c_2)}
$$

$$
\frac{(s, h, c_1) \longrightarrow \mathsf{wrong}}{(s, h, c_1; c_2) \longrightarrow \mathsf{wrong}}
$$

$$
\text{where } s(e) = \begin{cases} s(x) & \text{when } e = x \text{ and } x \in \mathrm{dom}(s) \\ v & \text{when } e = v \\ op(s(e_1), \ldots, s(e_n)) & \text{when } e = op(e_1, \ldots, e_n) \end{cases}
$$

Fig. 2 Operational Semantics

2.2 The Logic SL$^{\mathrm{W}}$

Figure 3 presents assertions and types used in SL$^{\mathrm{W}}$. Assertions in SL$^{\mathrm{W}}$ include all formulas in predicate calculus (not shown in the figure), and all SL formulas. The only additional assertion form in SL$^{\mathrm{W}}$ is $\{e : \tau\}$, which denotes that e has type τ (the semantics is formally defined in Section 3).

$$
\begin{array}{rrcl}
(\mathit{Assertion}) & \mathtt{p} & ::= & \cdots \mid \mathsf{emp} \mid e_1 \mapsto e_2 \mid \mathtt{p}_1 * \mathtt{p}_2 \mid \mathtt{p}_1 -\!\!* \mathtt{p}_2 \mid e : \tau \\
(\mathit{Type}) & \tau & ::= & \mathsf{int} \mid \mathsf{wref}\ \tau \\
(\mathit{HeapType}) & \Psi & ::= & \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\
(\mathit{LocalVarType}) & \Gamma & ::= & \{x_1 : \tau_1, \ldots, x_n : \tau_n\}
\end{array}
$$

Fig. 3 Assertions and Types

SL$^{\mathrm{W}}$ is equipped with a simple type system for tracking weak-heap locations. Although the type system does not include many types in high-level languages, by including reference types it is already sufficient to show interesting interactions between strong and weak heaps. Reference types are the most common types when high-level languages interoperate with low-level languages because most data across language boundaries are passed by references.

Type int is for all numbers and it is essentially a top type since all values in SL$^{\mathrm{W}}$ are natural numbers. Type "wref τ" is for locations in a weak heap, but not in a strong heap. We have removed the ref type in the APLAS version.[19] The ref type was the type for heap locations. But since in this paper locations are treated as natural numbers, it is safe to remove ref and give int to heap locations. A heap type Ψ tells the type of every location in a weak heap; mathematically, it is a finite map from locations to types. Given a heap type Ψ, location ℓ has type "wref τ" if $\Psi(\ell)$ equals τ. A local variable type, Γ, tells the type of local variables.

Figure 4 presents typing rules for expressions, which are unsurprising. Notice that the typing rule for "wref τ" requires that the location ℓ is in the domain of the heap type Ψ and $\Psi(\ell)$ has to be the same as τ. This rule and the later weak-update rule enforce type-preserving updates on weak heaps.

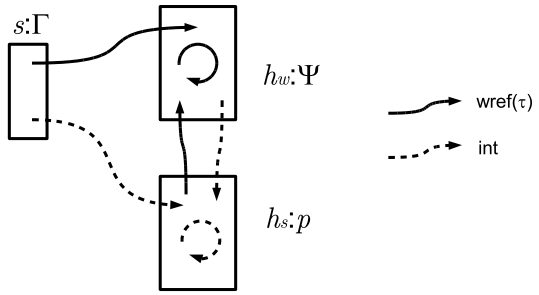
$\boxed{\Psi, \Gamma \vdash e : \tau}$

$$
\frac{x \in \mathrm{dom}(\Gamma)}{\Psi, \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Psi, \Gamma \vdash n : \mathsf{int}}
$$

$$
\frac{\Psi(\ell) = \tau}{\Psi, \Gamma \vdash \ell : \mathsf{wref}\ \tau} \qquad \frac{\forall i \in [1..n].\ \Psi, \Gamma \vdash e_i : \mathsf{int}}{\Psi, \Gamma \vdash op(e_1, \ldots, e_n) : \mathsf{int}}
$$

Fig. 4 Typing Rules for Expressions

The following schematic diagram helps to understand the relationship between weak heaps, strong heaps, local variables, assertions and various kinds of types in SL$^{\mathrm{W}}$:



As shown in the diagram, SL^{W} conceptually divides a heap into a weak heap h_w and a strong heap h_s. The weak heap is specified by a heap type Ψ, and the strong heap by SL formula $\mathtt{p}$. Pointers to weak-heap cells (in solid lines) have type "$\mathsf{wref}\ \tau$." Pointers to strong heap cells (in dotted lines) can have only type int.

Figures 5, 6, and 7 present inference rules for checking commands. These

$\boxed{\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}}$

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Psi, \Gamma \vdash e : \mathsf{int} \quad \Psi, \Gamma \vdash e' : \tau}{\Psi \vdash \{\Gamma.(e \mapsto e')\}\ x := [e]\ \{\Gamma[x \rightsquigarrow \tau].x = e' \wedge (e \mapsto x)\}} \text{ (S-LOAD)}$$
$$\text{where } x \notin \mathsf{fvar}(e) \cup \mathsf{fvar}(e')$$

$$\frac{\Psi, \Gamma \vdash x : \mathsf{int} \quad \mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma)}{\Psi \vdash \{\Gamma.(x \mapsto -)\}\ [x] := e\ \{\Gamma.(x \mapsto e)\}} \text{ (S-UPDATE)}$$

$$\frac{\{x\} \cup \mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma)}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ x := \mathsf{alloc}(e)\ \{\Gamma[x \rightsquigarrow \mathsf{int}].(x \mapsto e)\}} \text{ (S-ALLOC)}$$
$$\text{where } x \notin \mathsf{fvar}(e)$$

$$\frac{\Psi, \Gamma \vdash e : \mathsf{int}}{\Psi \vdash \{\Gamma.(e \mapsto -)\}\ \mathsf{free}(e)\ \{\Gamma.\mathsf{emp}\}} \text{ (S-FREE)}$$

Fig. 5 Rules for Heap-manipulating Commands in the World of Strong Heaps

$\boxed{\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}}$

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Psi, \Gamma \vdash e : \mathsf{wref}\ \tau}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ x := [e]\ \{\Gamma[x \rightsquigarrow \tau].\mathsf{emp}\}} \text{ (W-LOAD)}$$

$$\frac{\Psi, \Gamma \vdash x : \mathsf{wref}\ \tau \quad \Psi, \Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ [x] := e\ \{\Gamma.\mathsf{emp}\}} \text{ (W-UPDATE)}$$

$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Psi, \Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ x := \mathsf{alloc}(e)\ \{\Gamma[x \rightsquigarrow \mathsf{wref}\ \tau].\mathsf{emp}\}} \text{ (W-ALLOC)}$$

Fig. 6 Rules for Heap-manipulating Commands in the World of Weak Heaps

rules use the judgment $\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$. In this judgment, Ψ, Γ and $\mathtt{p}$ are preconditions and specify conditions on the weak heap, local variables, and the strong heap respectively. Postconditions are Γ' and $\mathtt{p}'$; they specify conditions on local variables and the strong heap of the state after executing c. Readers may wonder why there is no postcondition specification of the weak heap. As common in mutable-reference type systems, the implicit semantics of the judgment is that there exists an extended heap type $\Psi' \supseteq \Psi$ and the weak heap of the poststate should satisfy Ψ'. In terms of type checking, the particular Ψ' does not matter. The formal semantics of the judgment will be presented in Section 3.

In anticipation of the development for concurrency, rules for commands treat variables as resources. They satisfy the following lemma:

Lemma 2.1
If $\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$, then $\mathsf{fvar}(c) \subseteq \mathrm{dom}(\Gamma)$, and $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$.

That is, all variables read and written by c should be in the domain of the local-variable type Γ and Γ has the same domain as Γ'. This is the reason why the S-LOAD rule requires $x \in \mathrm{dom}(\Gamma)$. It also implicitly requires $\mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma)$, which is implied by the assumption $\Psi, \Gamma \vdash e : \mathsf{int}$. As another note, $\Gamma.\mathtt{p}$ is equivalent to false when $\mathsf{fvar}(\mathtt{p}) \not\subseteq \mathrm{dom}(\Gamma)$ (this is evident from the semantics of $\Gamma.\mathtt{p}$ in Section 3).

Rules for commands are divided into two groups. One group is for the world of strong heaps (in Fig. 5), and another for the world of weak heaps (in Fig. 6). The rules for strong heaps are almost the same as the corresponding ones in standard SL, except that they also update Γ when necessary.

The rules for weak heaps are the ones that one would usually find in

$$\frac{\Psi \vdash \{\Gamma.\mathtt{p}\}\ c_1\ \{\Gamma'.\mathtt{p}'\} \quad \Psi \vdash \{\Gamma'.\mathtt{p}'\}\ c_2\ \{\Gamma''.\mathtt{p}''\}}{\Psi \vdash \{\Gamma.\mathtt{p}\}\ c_1; c_2\ \{\Gamma''.\mathtt{p}''\}}\ (\textsc{seq})$$

$$\frac{}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ \epsilon\ \{\Gamma.\mathsf{emp}\}}\ (\textsc{emp})$$

$$\frac{\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}}{\Psi \vdash \{(\Gamma.\mathtt{p}) * (\Gamma_1.\mathtt{p}_1)\}\ c\ \{(\Gamma'.\mathtt{p}') * (\Gamma_1.\mathtt{p}_1)\}}\ (\textsc{frame})$$

where no variable occurring free in $\mathtt{p}_1$ is modified by c

$$\frac{\begin{array}{c}\Psi \vdash \{\Gamma_1.\mathtt{p}_1\}\ c\ \{\Gamma_2.\mathtt{p}_2\} \\ \vdash \{\Gamma_1'.\mathtt{p}_1'\} \Rightarrow \{\Gamma_1.\mathtt{p}_1\} \qquad \vdash \{\Gamma_2.\mathtt{p}_2\} \Rightarrow \{\Gamma_2'.\mathtt{p}_2'\}\end{array}}{\Psi \vdash \{\Gamma_1'.\mathtt{p}_1'\}\ c\ \{\Gamma_2'.\mathtt{p}_2'\}}\ (\textsc{weakening})$$

$$\boxed{\vdash \{\Gamma.\mathtt{p}\} \Rightarrow \{\Gamma'.\mathtt{p}'\}}$$

$$\frac{}{\vdash \{\Gamma.\mathtt{p}\} \Rightarrow \{\Gamma.(\mathtt{p} \wedge (x : \Gamma(x)))\}}\ (\textsc{w1}) \qquad \frac{\vdash \mathtt{p} \Rightarrow \mathtt{p}'}{\vdash \{\Gamma.\mathtt{p}\} \Rightarrow \{\Gamma.\mathtt{p}'\}}\ (\textsc{w2})$$

Fig. 7 Sequencing, Frame, and Weakening Rules (Rules for assignments, conditional statements, and loops are similar to the ones in Hoare Logic and are omitted.)

a type system for mutable-reference types. The weak-update rule W-UPDATE requires the pointer be of type "wref τ," and the new value be of type τ. This rule enforces type-preserving updates. Once these conditions hold, Γ remains unchanged after the update. Notice in this rule there is no need to understand separation and aliases as the S-UPDATE rule does. The W-ALLOC rule does not need to extend the heap type Ψ because Ψ is only a precondition. When proving the soundness of the rule, we need to find a new Ψ' that extends Ψ and is also satisfied by the new weak heap after the allocation. Finally, there is no rule for $\mathsf{free}(e)$ in the world of weak heaps. Weak heaps should be garbage-collected.[*1]

Figure 7 presents additional rules. In the FRAME rule, the notation $(\Gamma.\mathtt{p}) * (\Gamma'.\mathtt{p}')$ stands for $\Gamma \uplus \Gamma'.(\mathtt{p} * \mathtt{p}')$. Rule W1 converts type information in Γ to information in assertion $\mathtt{p}$. This is useful since information in Γ might be overwritten due to assignments to variables. One of examples in later sections will show the use of this rule. Rule W2 uses the premise $\vdash \mathtt{p} \Rightarrow \mathtt{p}'$; any formula that is valid according to the semantics of $\vdash \mathtt{p} \Rightarrow \mathtt{p}'$ is acceptable (the semantics is defined in Sec. 3).

Figure 8 presents a rule that shows the interaction between weak and strong heaps. The command "$\epsilon_{\mathrm{s2w(x)}}$" is the empty command with a type annotation. It instructs the type checker to transform the ownership of the cell referenced by x in the strong heap to the weak heap. Notice that there is no rule for converting a location from the weak heap to the strong heap; this is similar to deallocation in weak heaps and requires the help of garbage collectors.

$$\frac{\Psi, \Gamma \vdash x : \mathsf{int} \quad \Psi, \Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma.(x \mapsto e)\}\ \epsilon_{\mathrm{s2w(x)}}\ \{\Gamma[x \rightsquigarrow \mathsf{wref}\ \tau].\mathsf{emp}\}}\ (\mathrm{S2W})$$

Fig. 8 A Rule for Converting a Location from the Strong Heap to the Weak Heap

2.3 Examples

We now show a few examples that demonstrate the use of SL^{W}. In these examples, we assume an additional type even for even integers and an obvious rule for the even type.

$$\frac{n \text{ is an even number}}{\Psi, \Gamma \vdash n : \mathsf{even}}$$

For clarity, we will also annotate the allocation instruction to indicate whether the allocation happens in the strong heap or in the weak heap. We write $x := \mathsf{alloc}_{\mathrm{s}}(e)$ for a strong-heap allocation. We write $x := \mathsf{alloc}_{\mathrm{w},\tau}(e)$ for a weak-heap allocation, and the intended type for e is τ. These annotations guide the type checking of SL^{W}.

[*1] We do not formally consider the interaction between garbage collectors and weak heaps. When considering a garbage collector, SL^{W} has to build in an extra level of indirection for cross-boundary references from strong heaps to weak heaps as objects in weak heaps may get moved (this is how the JNI implements Java references in native code). We leave this as future work.

The first example shows how the counterexample in the introduction (formula (2) on page 5) plays out in SL^{W}. The following program first initializes the heap to a form such that y points to a location of type "$\mathsf{wref\ even}$" and x points to 4, and then performs a heap update through x. The whole program is checkable in SL^{W} with respect to any heap type (remember the heap type specifies the *initial* weak heap). Below we also include conditions of the form "$\Gamma.\mathbf{p}$" between instructions.

$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}.\ \mathsf{emp}$
$z := \mathsf{alloc}_{\mathrm{w},\mathsf{even}}(2);$
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{wref\ even}\}.\ \mathsf{emp}$
$y := \mathsf{alloc}_{\mathrm{s}}(z);$
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{wref\ even}\}.\ (y \mapsto z)$ *// by rule (w1)*
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{wref\ even}\}.\ (y \mapsto z) \wedge \{z : \mathsf{wref\ even}\}$ *// by rule (w2)*
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{wref\ even}\}.\ \exists v.\ (y \mapsto v) \wedge (v : \mathsf{wref\ even})$
$z := 0;$
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}.\ \exists v.\ (y \mapsto v) \wedge (v : \mathsf{wref\ even})$
$x := \mathsf{alloc}_{\mathrm{s}}(4);$
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}.\ \exists v.\ ((y \mapsto v) \wedge (v : \mathsf{wref\ even})) * (x \mapsto 4)$
$[x] := 3$
$\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}.\ \exists v.\ ((y \mapsto v) \wedge (v : \mathsf{wref\ even})) * (x \mapsto 3)$

Different from the counterexample, the condition before "[x] := 3" limits where y can point to. In particular, y cannot point to x because (1) by the type of v, variable y must point to a weak-heap location; (2) x represents a location in the strong heap. Therefore, the update through x does not invalidate the type of v. We could easily construct an example where y indeed points to x. But in that case the type of v would be int, which would also not be affected by updates through x.

One of the motivations of SL^{W} is to reason about programs where code in high-level languages interacts with low-level code. Prior research [6,18] has shown that the interface code between high-level programs and low-level programs is error-prone. All kinds of errors may occur. One common kind of errors occurs when low-level code makes type misuses of references that point to objects in the weak heap. For instance, in the JNI, types of all references to Java objects are conflated into one type in native code—`jobject`. Consequently, there is no static checking of whether native code uses these Java references in a type-safe way. Type misuses of these Java references can result in silent memory corruption or unexpected behavior.

The first example already demonstrates how SL^{W} enables passing pointers from high-level to low-level code. In the example, the first allocation is on the weak heap and can be thought of as an operation by high-level code. Then, the location is passed to the low level by being stored in the strong heap. Unlike foreign function interfaces where types of cross-boundary references are conflated into a single type in low-level code, SL^{W} can track the accurate types of those references and enable type safety.

The next example demonstrates how low-level code can initialize a data structure in the strong heap, and then transfer that structure to the weak heap so that the structure is usable by high-level code.

$$
\begin{array}{l}
\quad \{x : \mathsf{int}, y : \mathsf{int}\}.\ \mathsf{emp} \\
x := \mathsf{alloc}_{\mathrm{s}}(4); \\
\quad \{x : \mathsf{int}, y : \mathsf{int}\}.\ (x \mapsto 4) \\
y := \mathsf{alloc}_{\mathrm{s}}(x); \\
\quad \{x : \mathsf{int}, y : \mathsf{int}\}.\ (x \mapsto 4) * (y \mapsto x) \\
\epsilon_{\mathrm{s2w}}(\mathrm{x}); \\
\quad \{x : \mathsf{wref\ even}, y : \mathsf{int}\}.\ (y \mapsto x) \\
\epsilon_{\mathrm{s2w}}(\mathrm{y}) \\
\quad \{x : \mathsf{wref\ even}, y : \mathsf{wref}\ (\mathsf{wref\ even})\}.\ \mathsf{emp}
\end{array}
$$

§3 Soundness of SL$^{\mathrm{W}}$

Soundness of SL$^{\mathrm{W}}$ is proved by a semantic approach. We first describe a semantic model for weak-reference types. Based on this model, semantics of various concepts in SL$^{\mathrm{W}}$ are defined. Every rule in SL$^{\mathrm{W}}$ is then proved as a lemma according to the semantics.

3.1 Modeling Weak-reference Types

Intuitively, a type is a set of values. This suggests that a semantic type should be a predicate of the metatype "*Value* $\rightarrow$ *Prop*." However, this idea would not support weak-reference types. To see why, let us examine a naïve model where "wref τ" in a heap h would denote a set of locations ℓ such that $h(\ell)$ is of type τ. This simple model is unfortunately unsound, which is illustrated by the following example:

1. Create a reference of type "wref even," and let the reference be named x.
2. Copy x to y. By the naïve model, a reference of type "wref even" also has type "wref int" (because an even number is also an integer). Let "wref int" be the type of y.
3. Update the reference through y with an odd integer, say 3. As y has the type "wref int," updating it with an odd integer is legal.
4. Dereference x. Alas, the dereferencing returns 3, although the type of x implies a result of an even number!

The problem with the naïve model is that, with aliases, it allows inconsistent views of memory. In the foregoing example, x and y have inconsistent views on the same memory cell. To address this problem, SL$^{\mathrm{W}}$ uses a heap type Ψ to type check a location. This follows the approach of Tofte[20] and Harper.[7] An example Ψ is as follows:

$$\Psi = \{\ell_0 : \mathsf{even},\ \ell_1 : \mathsf{int},\ \ell_2 : \mathsf{wref\ even},\ \ell_3 : \mathsf{wref\ int}\} \tag{3}$$

A heap type Ψ helps to define two related concepts, informally stated below (their formal semantic definitions will be presented in a moment):

(i) A location ℓ is of type "$\mathsf{wref}\ \tau$" if and only if $\Psi(\ell)$ equals τ.

(ii) A heap h is consistent with Ψ if for every ℓ, the value $h(\ell)$ has type $\Psi(\ell)$.

For the example Ψ, condition (ii) means that $h(\ell_0)$ should be an even number, $h(\ell_1)$ should be an integer, $h(\ell_2)$ should be of type "$\mathsf{wref\ even}$," ...

The heap type Ψ prevents aliases from having inconsistent views of the heap. Aliases have to agree on their types because the types have to agree with the type in Ψ. In particular, the example showing the unsoundness of the naïve model would not work in the above model because, in step 3 of the example, y cannot be cast from type "$\mathsf{wref\ even}$" to "$\mathsf{wref\ int}$": type "$\mathsf{wref\ even}$" implies that $\Psi(y) = \mathsf{even}$, which is a different type from int.

A subtlety of the above model is the denotation of "$\mathsf{wref}\ \tau$" depends on the heap type Ψ, but is *independent* of the heap h. A weak-reference type is connected to the heap h only indirectly, through the consistency relation between h and Ψ.

Example 3.1
Let $h = \{\ell_0 \rightsquigarrow 4,\ \ell_1 \rightsquigarrow 3,\ \ell_2 \rightsquigarrow \ell_0,\ \ell_3 \rightsquigarrow \ell_1\}$. It is consistent with the example Ψ in (3). To see this, 4 at location ℓ_0 is an even number and 3 at location ℓ_1 is an integer. At location ℓ_2, ℓ_0 is of type "$\mathsf{wref\ even}$" because, by (i), this is equivalent to $\Psi(\ell_0) = \mathsf{even}$—a true statement. Similarly, the value ℓ_1 at location ℓ_3 is of type "$\mathsf{wref\ int}$."

Formalizing a set of semantic predicates following (i) and (ii) directly, however, would encounter difficulties because of a circularity in the model: by (ii), Ψ is a map from locations to types; by (i), the model of types takes Ψ as an argument—Ψ is necessary to decide if a location belongs to "$\mathsf{wref}\ \tau$." If defined naïvely, the model would result in inconsistent cardinality, as described by Ahmed.[1)]

We next propose a fixed-point approach. We rewrite the heap type Ψ as a recursive equation. After adding Ψ as an argument to types, the example in (3) becomes:

$$\Psi = \{\ell_0 : \mathsf{even}(\Psi),\ \ell_1 : \mathsf{int}(\Psi),\ \ell_2 : (\mathsf{wref\ even})(\Psi),\ \ell_3 : (\mathsf{wref\ int})(\Psi)\} \qquad (4)$$

Notice that Ψ appears on both the left and the right side of the equation. Once Ψ is written as a recursive equation, it follows that any fixed point of the following functional is a solution to the equation (4):

$$\lambda\Psi.\{\ell_0 : \mathsf{even}(\Psi),\ \ell_1 : \mathsf{int}(\Psi),\ \ell_2 : (\mathsf{wref\ even})(\Psi),\ \ell_3 : (\mathsf{wref\ int})(\Psi)\} \qquad (5)$$

To get a fixed point of (5), we follow the indexed model of recursive types by Appel and McAllester.[2)] We first introduce some domains:

$$\begin{array}{rcl} (\mathit{SemHeapType})\ \mathtt{F} & \in & \mathit{Loc} \rightharpoonup \mathit{SemIType} \\ (\mathit{SemIType})\ \mathtt{t} & \in & \mathit{SemHeapEnv} \to \mathit{Nat} \to \mathit{Value} \to \mathit{Prop} \\ (\mathit{SemHeapEnv})\ \phi & \in & \mathit{Loc} \rightharpoonup \mathit{Nat} \to \mathit{Value} \to \mathit{Prop} \end{array}$$

We use $\mathtt{F}$ for a semantic heap type (it is the metatype of the denotation of heap types, as we will see). It maps locations to indexed types. An important point is that from $\mathtt{F}$ we can define $\lambda\phi, \ell.\ \mathtt{F}(\ell)\ \phi$, which has the metatype $\mathit{SemHeapEnv} \to \mathit{SemHeapEnv}$. Therefore, a semantic heap type is effectively a functional similar to the one in (5), and a fixed point of $\mathtt{F}$ is of the metatype $\mathit{SemHeapEnv}$.

A semantic type $\mathtt{t}$ is a predicate over the following arguments: ϕ is a semantic heap environment; k is a natural-number index; v is a value. The heap environment $\phi \in \mathit{SemHeapEnv}$ is used in our model of $\mathrm{WRef}(\mathtt{t})$ to constrain reference types. The index k comes from the indexed model and is a technical device that enables us to define the fixed point of a semantic heap type $\mathtt{F}$.

Following the indexed model, we introduce a notion of contractiveness.

Definition 3.1 (Contractiveness)

$$\begin{array}{lcl} \mathsf{contractive}(\mathtt{F}) & \triangleq & \forall \ell \in \mathrm{dom}(\mathtt{F}).\ \mathsf{contractive}(\mathtt{F}(\ell)) \\ \mathsf{contractive}(\mathtt{t}) & \triangleq & \forall \phi, k, j \le k, v.\ (\mathtt{t}\ \phi\ j\ v) \leftrightarrow (\mathtt{t}\ (\mathsf{approx}(k, \phi))\ j\ v) \\ \mathsf{approx}(k, \phi) & \triangleq & \lambda \ell, j, v.\ j < k \wedge \phi\ l\ j\ v. \end{array}$$

We define $(\wp\mathtt{F}) = \lambda\phi, \ell.\ \mathtt{F}(\ell)\ \phi$. That is, it turns $\mathtt{F}$ into a functional of type $\mathit{SemHeapEnv} \to \mathit{SemHeapEnv}$.

Theorem 3.1
If $\mathsf{contractive}(\mathtt{F})$, then the following $\mu\mathtt{F}$ is the least fixed point[*2] of the functional $(\wp\mathtt{F})$:

$$\mu\mathtt{F} \quad \triangleq \quad \lambda \ell, k, v.\ (\wp\mathtt{F})^{k+1}(\bot)\ \ell\ k\ v,$$

where $\bot = \lambda \ell, k, v.\ \mathsf{false}$, and $(\wp\mathtt{F})^{k+1}$ applies the functional $k+1$ times.

The theorem is proved by following the indexed model of recursive types.[2)]

The following lemma is an immediate corollary of Theorem 3.1.

Lemma 3.1
For any contractive $\mathtt{F}$, any ℓ, k, v, we have $\big(\mathtt{F}(\ell)\ (\mu\mathtt{F})\ k\ v\big) \leftrightarrow \big((\mu\mathtt{F})(\ell)\ k\ v\big)$

Most of the semantic types ignore the ϕ argument. For example,

$$\mathrm{Even} \quad \triangleq \quad \lambda \phi, k, v.\ \exists u.\ v = 2 \times u.$$

We use capitalized Even to emphasize that it is a predicate, instead of the syntactic type even. The model of weak-reference types uses the argument ϕ.

Definition 3.2

$$\mathrm{WRef}(t) \quad \triangleq \quad \lambda \phi, k, \ell.\ \forall j < k, v.\ \phi\ \ell\ j\ v \leftrightarrow t\ \phi\ j\ v$$

[*2] Since $\mathtt{F}$ is contractive in the sense that "$\mathtt{F}(\ell)\ \phi\ k\ w$" performs only calls to ϕ on arguments smaller than k, it is easy to show by induction that any two fixed points of $\mathtt{F}$ are identical; therefore, the least fixed point of $\mathtt{F}$ is also its greatest fixed point.

In words, a location ℓ is of type WRef($\mathtt{t}$) under heap environment ϕ, if $\phi(\ell)$ equals $\mathtt{t}$ approximately, with index less than k.

Example 3.2
Let $\mathtt{F}_0 = \{\ell_0 : \text{Even},\ \ell_1 : \text{WRef(Even)}\}$. Then "WRef(Even) $(\mu\mathtt{F}_0)\ k\ \ell_0$" holds for any k. To see this, for any $j < k$ and v, we have

$$(\mu\mathtt{F}_0)\ \ell_0\ j\ v \leftrightarrow \mathtt{F}_0(\ell_0)(\mu\mathtt{F}_0)j\ v \leftrightarrow \text{Even}\ (\mu\mathtt{F}_0)\ j\ v$$

The first step is by lemma 3.1, and the second is by the definition of $\mathtt{F}_0$ at location ℓ_0. We can similarly show "WRef(WRef(Even)) $(\mu\mathtt{F}_0)\ k\ \ell_1$" holds.

Note that the definition of WRef($\mathtt{t}$) is more general than the "$\mathsf{wref}\ \tau$" type in SL^{W}, as τ is syntactically defined, while $\mathtt{t}$ can be any (contractive) semantic predicate.

Heap allocation. We need an additional idea to cope with heap allocation in the weak heap. Our indexed types take the fixed point of a semantic heap type $\mathtt{F}$ as an argument. But $\mathtt{F}$ changes after heap allocation. For example, from

$\mathtt{F} = \{\ell_0 : \text{Even},\ \ell_1 : \text{WRef(Even)}\}$ to $\mathtt{F}' = \{\ell_0 : \text{Even},\ \ell_1 : \text{WRef(Even)}, \ell_2 : \text{Even}\}$,

after ℓ_2 is allocated and initialized with an even number.

After a new heap location is allocated, any value that has type $\mathtt{t}$ before allocation should still have the same type after allocation. This is the monotonicity condition maintained by type systems. To model it semantically, our idea is to quantify explicitly outside of the model of types over all future semantic heap types and assert that the type in question is true over the fixed point of any future semantic heap type.

First is a notion of type-preserving heap extension from $\mathtt{F}$ to $\mathtt{F}'$:

Definition 3.3
$\mathtt{F}' \geq \mathtt{F} \triangleq \mathsf{contractive}(\mathtt{F}') \wedge \mathsf{contractive}(\mathtt{F}) \wedge$
$\forall \ell \in \text{dom}(F), \phi, k, v.\ \mathtt{F}'(\ell)\ \phi\ k\ v \leftrightarrow \mathtt{F}(\ell)\ \phi\ k\ v$

Lemma 3.2
The relation $\mathtt{F}' \geq \mathtt{F}$ is reflexive, anti-symmetric, and transitive (thus a partial order).

Next, we define the consistency relation between h and $\mathtt{F}$, and also a relation that states a value v is of type $\mathtt{t}$ under $\mathtt{F}$. Both relations quantify over all future semantic heap types, and require that the type in question be true over the fixed point of any future semantic heap type.

Definition 3.4
$\models h : \mathtt{F} \triangleq \text{dom}(h) \subseteq \text{dom}(\mathtt{F}) \wedge \forall \ell \in \text{dom}(h).\mathtt{F} \models h(\ell) : \mathtt{F}(\ell)$
$\mathtt{F} \models v : \mathtt{t} \triangleq \forall \mathtt{F}' \geq \mathtt{F}. \forall k.\ \mathtt{t}\ (\mu\mathtt{F}')\ k\ v$

With our model, the following theorem for memory operations can be proved.

Theorem 3.2

(i) (Read) If $\models h:\mathtt{F}$, and $\ell \in \mathrm{dom}(h)$, and $\mathtt{F} \models \ell:\mathrm{WRef}(\mathtt{t})$, then $\mathtt{F} \models h(\ell):\mathtt{t}$.

(ii) (Write) If $\models h:\mathtt{F}$, and $\ell \in \mathrm{dom}(h)$, and $\mathtt{F} \models \ell:\mathrm{WRef}(\mathtt{t})$, and $\mathtt{F} \models v:\mathtt{t}$, then $\models h[\ell \rightsquigarrow v]:\mathtt{F}$.

(iii) (Allocation) If $\models h:\mathtt{F}$, and $\mathtt{F} \models v:\mathtt{t}$, and $\mathsf{contractive}(\mathtt{t})$, and $\ell \notin \mathrm{dom}(\mathtt{F})$, then $\models h \uplus \{\ell \rightsquigarrow v\}:\mathtt{F} \uplus \{\ell \rightsquigarrow \mathtt{t}\}$.

Proof

We prove (i) and (iii); the proof of (ii) is similar.

(i) By the definition of $\mathtt{F} \models h(\ell):\mathtt{t}$, the goal is to prove "$\mathtt{t}\ (\mu\mathtt{F}')\ k\ (h(\ell))$," for all $\mathtt{F}' \geq \mathtt{F}$, and k.

By $\models h:\mathtt{F}$ and $\ell \in \mathrm{dom}(h)$, we have $\mathtt{F} \models h(\ell):\mathtt{F}(\ell)$. Unfold its definition and use the assumption $\mathtt{F}' \geq \mathtt{F}$, we get "$\mathtt{F}(\ell)\ (\mu\mathtt{F}')\ k\ (h(\ell))$." With this and the assumption $\mathtt{F}' \geq \mathtt{F}$, we derive "$\mathtt{F}'(\ell)\ (\mu\mathtt{F}')\ k\ (h(\ell))$", which is the same as the following by Lemma 3.1.

$$(\mu\mathtt{F}')\ \ell\ k\ (h(\ell)). \tag{6}$$

Now by the premise $\mathtt{F} \models \ell:\mathrm{WRef}(\mathtt{t})$, we derive "$(\mathrm{WRef}(\mathtt{t}))\ (\mu\mathtt{F}')\ (k+1)\ \ell$". By the definition of $\mathrm{WRef}(\mathtt{t})$, we further get

$$\forall v.\ (\mu\mathtt{F}')\ \ell\ k\ v \leftrightarrow \mathtt{t}\ (\mu\mathtt{F}')\ k\ v \tag{7}$$

By (6) and (7), we get "$\mathtt{t}\ (\mu\mathtt{F}')\ k\ (h(\ell))$," which is our goal.

(iii) Let $h' = h \uplus \{\ell \rightsquigarrow v\}$, and $\mathtt{F}' = \mathtt{F} \uplus \{\ell \rightsquigarrow \mathtt{t}\}$. To show $\models h':\mathtt{F}'$, we need to show

$$\mathrm{dom}(h') \subseteq \mathrm{dom}(\mathtt{F}'), \tag{8}$$

$$\forall \ell' \in \mathrm{dom}(h').\ \mathtt{F}' \models h'(\ell'):\mathtt{F}'(\ell'). \tag{9}$$

(8) is immediate from $\mathrm{dom}(h) \subseteq \mathrm{dom}(\mathtt{F})$. To prove (9), we show that for all $\ell' \in \mathrm{dom}(h')$, for all $\mathtt{F}'' \geq \mathtt{F}'$, for all k, "$\mathtt{F}'(\ell')\ (\mu\mathtt{F}'')\ k\ (h'(\ell'))$". We prove it by a case analysis of ℓ'.

(a) $\ell' \neq \ell$. The goal becomes "$\mathtt{F}(\ell')\ (\mu\mathtt{F}'')\ k\ (h(\ell'))$". First, it is easy to show $\mathtt{F}' \geq \mathtt{F}$. By this and $\mathtt{F}'' \geq \mathtt{F}'$, we get $\mathtt{F}'' \geq \mathtt{F}$. From $\models h:\mathtt{F}$, we get $\mathtt{F} \models h(\ell'):\mathtt{F}(\ell')$. By this and $\mathtt{F}'' \geq \mathtt{F}$, we derive the goal.

(b) $\ell' = \ell$. The goal becomes "$\mathtt{t}\ (\mu\mathtt{F}'')\ k\ v$". This is immediate from $\mathtt{F} \models v:\mathtt{t}$ and $\mathtt{F}'' \geq \mathtt{F}$.

■

3.2 Semantic Model of SL$^{\mathrm{W}}$

To show the soundness of SL$^{\mathrm{W}}$, we define semantics for judgments in SL$^{\mathrm{W}}$ and then prove each rule as a lemma according to the semantics. Figure 9 presents definitions that are used in the semantics.

The semantics of types is unsurprising. In particular, the semantics of $[\![\mathsf{wref}\ \tau]\!]$ is defined in terms of the predicate $\mathrm{WRef}(t)$ in Definition 3.2. All these

$\boxed{[\![\tau]\!] \in \mathit{SemIType}}$ $\quad [\![\mathsf{int}]\!] \triangleq \lambda\phi, k, v.\ \mathrm{True}. \quad [\![\mathsf{wref}\ \tau]\!] \triangleq \mathrm{WRef}([\![\tau]\!])$

$\boxed{[\![\Psi]\!] \in \mathit{Loc} \rightharpoonup \mathit{SemIType}}$ $\quad [\![\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}]\!] \triangleq \{\ell_1 : [\![\tau_1]\!], \ldots, \ell_n : [\![\tau_n]\!]\}$

$\boxed{[\![\Gamma]\!] \in \mathit{Var} \rightharpoonup \mathit{SemIType}}$ $\quad [\![\{x_1 : \tau_1, \ldots, x_n : \tau_n\}]\!] \triangleq \{x_1 : [\![\tau_1]\!], \ldots, x_n : [\![\tau_n]\!]\}$

$\boxed{\mathtt{F}, s, h \models \mathtt{p}}$

$$
\begin{array}{l}
\mathtt{F}, s, h \models \{e : \tau\} \triangleq \mathtt{F} \models s(e) : [\![\tau]\!] \\
\mathtt{F}, s, h \models \mathsf{emp} \triangleq \mathrm{dom}(h) = \emptyset \\
\mathtt{F}, s, h \models (e_1 \mapsto e_2) \triangleq \mathrm{dom}(h) = s(e_1) \ \wedge\ h(s(e_1)) = s(e_2) \\
\mathtt{F}, s, h \models \mathtt{p}_1 * \mathtt{p}_2 \triangleq \exists h_1, h_2, s_1, s_2. \\
\quad (h = h_1 \uplus h_2) \ \wedge\ (s = s_1 \uplus s_2) \ \wedge\ (\mathtt{F}, s_1, h_1 \models \mathtt{p}_1) \ \wedge\ (\mathtt{F}, s_2, h_2 \models \mathtt{p}_2) \\
\mathtt{F}, s, h \models \mathtt{p}_1 -\!\!* \mathtt{p}_2 \triangleq \forall h_1.\ ((h_1 \perp h) \ \wedge\ (\{\}, s, h_1 \models \mathtt{p}_1)) \Rightarrow (\mathtt{F}, s, h_1 \uplus h \models \mathtt{p}_2) \\
\hline
\mathtt{F} \models s : \Gamma \triangleq \mathrm{dom}(s) = \mathrm{dom}(\Gamma) \ \wedge\ \forall x \in \mathrm{dom}(\Gamma).\ \mathtt{F} \models s(x) : [\![\Gamma(x)]\!] \\
\mathtt{F}, s, h \models \Gamma.\mathtt{p} \triangleq \mathsf{fvar}(\mathtt{p}) \subseteq \mathrm{dom}(\Gamma) \ \wedge\ (\mathtt{F} \models s : \Gamma) \ \wedge\ (\mathtt{F}, s, h \models \mathtt{p}) \\
\models (s, h)\ \mathrm{sat}\ (\mathtt{F}, \Gamma.\mathtt{p}) \triangleq \\
\qquad \exists h_1, h_2.\ h = h_1 \uplus h_2 \ \wedge\ \mathrm{dom}(h_1) = \mathrm{dom}(\mathtt{F}) \ \wedge\ (\models h_1 : \mathtt{F}) \ \wedge\ (\mathtt{F}, s, h_2 \models \Gamma.\mathtt{p})
\end{array}
$$

Fig. 9 Semantic Definitions

types are contractive. The semantics of Ψ and Γ is just the point-wise extension of the semantics of types.

The predicate "$\mathtt{F}, s, h \models \mathtt{p}$" interprets the truth of assertion $\mathtt{p}$. When $\mathtt{p}$ is a SL formula, the interpretation is similar to the one in SL. When $\mathtt{p}$ is $\{e : \tau\}$, the interpretation depends on $\mathtt{F}$. Notice that the interpretation of $\{e : \tau\}$ is independent of the heap; it is a pure assertion (that is, it does not depend on the strong heap).

The definition of $\mathtt{F} \models s : \Gamma$ requires the domains of s and Γ match and every value in s has the specified type in Γ. The definition of "$\models (s, h)$ sat $(\mathtt{F}, \Gamma.\mathtt{p})$" splits the heap into two parts. One for the weak heap, which should satisfy $\mathtt{F}$, and the other for the strong heap, which is specified by $\mathtt{p}$.

With the above definitions, we are ready to define the semantics of the judgments in SL^{W}. The following definitions interpret "$\Psi, \Gamma \vdash e : \tau$", "$\vdash \mathtt{p} \Rightarrow \mathtt{p}'$," and "$\vdash \{\Gamma.\mathtt{p}\} \Rightarrow \{\Gamma'.\mathtt{p}'\}$."

Definition 3.5

$$
\begin{array}{l}
\Psi, \Gamma \models e : \tau \triangleq \mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma) \ \wedge\ \forall \mathtt{F} \geq [\![\Psi]\!].\ \forall s.\ \mathtt{F} \models s : \Gamma \Rightarrow \mathtt{F} \models s(e) : [\![\tau]\!]. \\
\models \mathtt{p} \Rightarrow \mathtt{p}' \triangleq \forall \mathtt{F}, s, h.\ (\mathtt{F}, s, h \models \mathtt{p}) \Rightarrow (\mathtt{F}, s, h \models \mathtt{p}') \\
\models \{\Gamma.\mathtt{p}\} \Rightarrow \{\Gamma'.\mathtt{p}'\} \triangleq \forall \mathtt{F}, s, h.\ (\mathtt{F}, s, h \models \Gamma.\mathtt{p}) \Rightarrow (\mathtt{F}, s, h \models \Gamma'.\mathtt{p}')
\end{array}
$$

Now we are ready to interpret $\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$. Below we present a partial-correctness interpretation.

Definition 3.6 (Partial correctness)

$$\begin{array}{l}\Psi \models \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\} \triangleq \\ \quad \forall \mathtt{F} \geq [\![\Psi]\!], s, h.\ \big(\models (s,h) \text{ sat } (\mathtt{F}, \Gamma.\mathtt{p})\big) \Rightarrow \\ \quad\quad \neg((s,h,c) \longrightarrow^* \mathsf{wrong}) \wedge \\ \quad\quad \big(\forall s', h'.\ (s,h,c) \longrightarrow^* (s',h',\epsilon) \Rightarrow \exists \mathtt{F}' \geq \mathtt{F}.\ \models (s',h') \text{ sat } (\mathtt{F}', \Gamma'.\mathtt{p}')\big)\end{array}$$

In the interpretation, it assumes $\mathtt{F}$ and a state that satisfies the condition $(\mathtt{F}, \Gamma.\mathtt{p})$ and requires that the state does not go wrong. In addition, it requires that, for any terminal state after the execution of c, we must be able to find a new semantic heap type $\mathtt{F}'$ so that $\mathtt{F}' \geq \mathtt{F}$ and the new state satisfies $(\mathtt{F}', \Gamma'.\mathtt{p}')$. Note that $\mathtt{F}'$ may be larger than $\mathtt{F}$ due to allocations in c.

Theorem 3.3 (Soundness)
If $\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$, then $\Psi \models \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

Proof
The proof is by induction over the derivation of the assumption. We show the cases S-LOAD and W-ALLOC. Most other cases are similar. The case of the FRAME rule uses Lemma 3.3.

(i)
$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Psi, \Gamma \vdash e : \mathsf{int} \quad \Psi, \Gamma \vdash e' : \tau}{\Psi \vdash \{\Gamma.(e \mapsto e')\}\ x := [e]\ \{\Gamma[x \rightsquigarrow \tau].x = e' \wedge (e \mapsto x)\}}\ (\text{S-LOAD})$$
$$\text{where } x \notin \mathsf{fvar}(e) \cup \mathsf{fvar}(e')$$

Pick $\mathtt{F} \geq [\![\Psi]\!], s, h$ and assume $\models (s,h)$ sat $(\mathtt{F}, \Gamma.(e \mapsto e'))$. Therefore, for some h_1 and h_2, we have

$$h = h_1 \uplus h_2 \ \wedge\ \mathrm{dom}(h_1) = \mathrm{dom}(\mathtt{F}) \wedge\ \models h_1 : \mathtt{F} \ \wedge\ \mathtt{F}, s, h_2 \models \Gamma.(e \mapsto e') \qquad (10)$$

The first subgoal is to show $(s, h, (x := [e]))$ does not step to a wrong state. It is sufficient to show "$\{x\} \cup \mathsf{fvar}(e) \subseteq \mathrm{dom}(s)$" and "$s(e) \in \mathrm{dom}(h)$". The first is true because "$\{x\} \cup \mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma)$" (by the assumptions of the S-LOAD rule), and "$\mathrm{dom}(s) = \mathrm{dom}(\Gamma)$" (by $\mathtt{F}, s, h_2 \models \Gamma.(e \mapsto e')$ in (10)).

The second subgoal is to show $\exists \mathtt{F}' \geq \mathtt{F}.\ \models (s', h)$ sat $(\mathtt{F}', \Gamma'.\mathtt{p}')$, where $s' = s[x \rightsquigarrow h(s(e))]$, $\Gamma' = \Gamma[x \rightsquigarrow \tau]$, and $\mathtt{p}' = (x = e') \wedge (e \mapsto x)$. Let $\mathtt{F}' = \mathtt{F}$. Then by (10), we only need to show $\mathtt{F}, s', h_2 \models \Gamma'.\mathtt{p}'$, which is the same as

$$(\mathsf{fvar}(\mathtt{p}') \subseteq \mathrm{dom}(\Gamma')) \wedge (\mathtt{F} \models s' : \Gamma') \wedge (\mathtt{F}, s', h_2 \models \mathtt{p}')$$

All the above can be proved easily using the assumptions of the S-LOAD rule.

(ii)
$$\frac{x \in \mathrm{dom}(\Gamma) \quad \Psi, \Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma.\mathsf{emp}\}\ x := \mathsf{alloc}(e)\ \{\Gamma[x \rightsquigarrow \mathsf{wref}\ \tau].\mathsf{emp}\}}\ (\text{W-ALLOC})$$

Pick $\mathtt{F} \geq [\![\Psi]\!], s, h$ and assume $\models (s,h)$ sat $(\mathtt{F}, \Gamma.\mathsf{emp})$. Therefore, we have

$$\mathrm{dom}(h) = \mathrm{dom}(\mathtt{F}) \wedge\ \models h : \mathtt{F} \ \wedge\ \mathtt{F}, s, \{\} \models \Gamma.\mathsf{emp} \qquad (11)$$

The first subgoal is to show $(s, h, x := \mathsf{alloc}(e))$ does not go to a wrong state. It is sufficient to show "$\{x\} \cup \mathsf{fvar}(e) \subseteq \mathrm{dom}(s)$". It is true because $x \in \mathrm{dom}(\Gamma)$, $\mathsf{fvar}(e) \subseteq \mathrm{dom}(\Gamma)$ (from "$\Psi, \Gamma \models e : \tau$"), and "$\mathrm{dom}(s) = \mathrm{dom}(\Gamma)$".

The second subgoal is to show $\exists \mathtt{F}' \geq \mathtt{F}.\ \models (s', h')$ sat $(\mathtt{F}', \Gamma'.\mathsf{emp})$, where $s' = s[x \rightsquigarrow \ell]$, $h' = h \uplus \{\ell \rightsquigarrow s(e)\}$, and $\Gamma' = \Gamma[x \rightsquigarrow \mathsf{wref}\ \tau]$.

Let $\mathtt{F}' = \mathtt{F} \uplus \{\ell \rightsquigarrow [\![\tau]\!]\}$; then $\mathtt{F}' \geq \mathtt{F}$. By Theorem 3.2(iii) and $\mathtt{F} \models s(e) : [\![\tau]\!]$ (from "$\Psi, \Gamma \models e : \tau$"), we can show $\models h' : \mathtt{F}'$. It is also easy to show $\mathtt{F}', s', \{\} \models \Gamma'.\mathsf{emp}$. All these give us $\models (s', h')$ sat $(\mathtt{F}', \Gamma'.\mathsf{emp})$. ■

Lemma 3.3 (Locality)
If $s = s_1 \uplus s_2$, $h = h_1 \uplus h_2$, $\neg(s_1, h_1, c) \longrightarrow \mathsf{wrong}$, then

1. $\neg(s, h, c) \longrightarrow \mathsf{wrong}$; and
2. for all s' and h', if $(s, h, c) \longrightarrow (s', h', c')$, then there exist s_1' and h_1' such that $s' = s_1' \uplus s_2$, $h' = h_1' \uplus h_2$, and $(s_1, h_1, c) \longrightarrow (s_1', h_1', c')$.

§4 Concurrency

In this section, we extend SL^{W} with concurrency. Figure 10 presents the syntax of extra commands for concurrency and their operational semantics. The atomic-block command $\mathsf{atomic}\{c\}$ executes c atomically. Vafeiadis and Parkinson[23] pointed out that such a command can be used to model synchronizations provided by locks or transactional memory. The operational semantics of $\mathsf{atomic}\{c\}$ executes c in one step and is not interrupted by other threads. The parallel-composition command "$c_1 \| c_2$" has the usual semantics of running c_1 and c_2 in parallel. It makes a step when either c_1 makes a step or c_2 makes a step. When both c_1 and c_2 reach the end (i.e., ϵ), the command "$c_1 \| c_2$" terminates.

The judgment for checking commands is changed to

$$\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}.$$

Compared to the sequential version, the concurrent one has an extra $\Gamma_{\mathtt{G}}.\mathtt{I}$ for specifying invariants in a shared strong heap. We use the symbol $\mathtt{I}$ for a global invariant that should always hold unless in an atomic block; invariant $\mathtt{I}$ is a separation-logic assertion and is required to be a *precise* assertion.[13] The following diagram depicts the relationship between various heaps and their specifications from the point view of one particular thread.

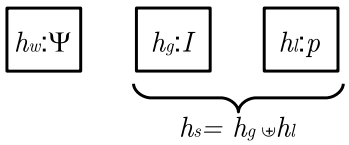
$$(\textit{Command})\quad c ::= \ldots \mid \mathsf{atomic}\{c\} \mid (c_1 \| c_2)$$

$$\frac{(s,h,c) \longrightarrow^* (s',h',\epsilon)}{(s,h,\mathsf{atomic}\{c\}) \longrightarrow (s',h',\epsilon)} \qquad \frac{(s,h,c) \longrightarrow^* \mathsf{wrong}}{(s,h,\mathsf{atomic}\{c\}) \longrightarrow \mathsf{wrong}}$$

$$\frac{(s,h,c_1) \longrightarrow (s',h',c_1')}{(s,h,c_1 \| c_2) \longrightarrow (s',h',c_1' \| c_2)} \qquad \frac{(s,h,c_2) \longrightarrow (s',h',c_2')}{(s,h,c_1 \| c_2) \longrightarrow (s',h',c_1 \| c_2')}$$

$$\frac{(s,h,c_1) \longrightarrow \mathsf{wrong}}{(s,h,c_1 \| c_2) \longrightarrow \mathsf{wrong}} \qquad \frac{(s,h,c_2) \longrightarrow \mathsf{wrong}}{(s,h,c_1 \| c_2) \longrightarrow \mathsf{wrong}} \qquad \frac{}{(s,h,\epsilon \| \epsilon) \longrightarrow (s,h,\epsilon)}$$

Fig. 10 Syntax and Semantics of Commands for Concurrency



The weak heap h_w remains the same as that in the sequential setting and is specified by a heap type Ψ. The strong heap h_s, however, is conceptually split into a global one (h_g), shared by all threads, and a local one (h_l), private to one thread. The global one is specified by the global invariant $\mathtt{I}$, and the local one by $\mathtt{p}$.

Figure 11 presents inference rules for checking commands. The ATOMIC rule states that the global strong heap with invariant $\mathtt{I}$ is accessible for the command c in an atomic block. The command can temporarily break the invariant, but has to restore the invariant when it is finished. Since it is inside an atomic block, its temporary breaking of the invariant will not interfere with other threads.

The PAR rule conceptually breaks the store and the local strong heap into two disjoint parts, specified by $\Gamma_1.\mathtt{p}_1$ and $\Gamma_2.\mathtt{p}_2$, respectively. It then checks c_1 with $\Gamma_1.\mathtt{p}_1$ as the precondition and c_2 with $\Gamma_2.\mathtt{p}_2$. For simplicity, the rule does not allow concurrent reads of variables or heap locations. This could be enabled by fractional permissions.[5)]

In addition, rules in Fig. 5, 6, and 7 have to be adjusted: every judgment of the form $\Psi \vdash \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$ is changed to $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi\ \vdash_{\mathrm{c}}\ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

It is worth mentioning that the rules allow the weak heap to be accessed concurrently without synchronization (in contrast to the strong heap). Although it is partly because there are only single-cell references and because access to these cells is atomic in the operational semantics, this demonstrates an important difference between strong and weak heaps in the concurrent setting.

$\boxed{\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi\ \vdash_{\mathrm{c}}\ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}}$

$$\frac{\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma') \qquad \{\}.\mathsf{emp}, \Psi\ \vdash_{\mathrm{c}}\ \{(\Gamma_{\mathtt{G}}.I) * (\Gamma.\mathtt{p})\}\ c\ \{(\Gamma_{\mathtt{G}}.I) * (\Gamma'.\mathtt{p}')\}}{\Gamma_{\mathtt{G}}.I, \Psi\ \vdash_{\mathrm{c}}\ \{\Gamma.\mathtt{p}\}\ \mathsf{atomic}\{c\}\ \{\Gamma'.\mathtt{p}'\}}\ (\textsc{atomic})$$

$$\frac{\forall i \in [1,2].\ \Gamma_{\mathtt{G}}.\mathtt{I}, \Psi\ \vdash_{\mathrm{c}}\ \{\Gamma_i.\mathtt{p}_i\}\ c_i\ \{\Gamma'_i.\mathtt{p}'_i\}}{\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi\ \vdash_{\mathrm{c}}\ \{(\Gamma_1.\mathtt{p}_1) * (\Gamma_2.\mathtt{p}_2)\}\ c_1 \| c_2\ \{(\Gamma'_1.\mathtt{p}'_1) * (\Gamma'_2.\mathtt{p}'_2)\}}\ (\textsc{par})$$

Fig. 11 Checking Commands in the Presence of Concurrency

An example This example computes the largest Fibonacci number that is less than 101376. Below is the initialization code for the example.

$$r1 := \mathsf{alloc}_{\mathrm{w,int}}(0);\ r2 := r1;\ y := \mathsf{alloc}_{\mathrm{s}}(1);\ z := \mathsf{alloc}_{\mathrm{s}}(1)$$

The example simulates the situation where a high-level program invokes some low-level program for performing the actual computation. The first operation

in the initialization allocates a weak-heap cell for storing the final result; this is to simulate an operation by the high-level program. The low-level program receives the reference $r1$, makes a copy ($r2 := r1$), and then initializes two cells in the strong heap (pointed to by y and z) for storing two adjacent numbers in the Fibonacci sequence. The two cells are initialized to be 1.

After initialization, the low-level code uses two threads for performing the computation: $c_1 \| c_2$. The program c_1, presented below, contains a loop that retrieves the two numbers stored in the global strong heap, terminates if the second is greater than or equal to 101376, and computes the next two numbers in the Fibonacci sequence otherwise. We define c_2 to be the same as c_1 except that $r1$, t_{d1}, t_{y1}, and t_{z1} are replaced by $r2$, t_{d2}, t_{y2}, and t_{z2}, respectively.

$$
\begin{array}{l}
t_{d1} := 0; \\
\mathsf{while}\ (t_{d1} = 0)\ \mathsf{do}\ \{ \\
\quad \mathsf{atomic}\ \{ \\
\quad\quad t_{y1} := [y];\ t_{z1} := [z]; \\
\quad\quad \mathsf{if}\ (t_{z1} \geq 101376)\ \{t_{d1} := 1\} \\
\quad\quad \mathsf{else}\ \{[y] := t_{z1};\ [z] := t_{y1} + t_{z1}\} \\
\quad \} \\
\}; \\
[r1] := t_{y1}
\end{array}
$$

After initialization, the heap satisfies the specification $(\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma_1.\mathsf{emp}) * (\Gamma_2.\mathsf{emp})$, where $\Gamma_{\mathtt{G}} = \{y : \mathsf{int},\ z : \mathsf{int}\}$, $\mathtt{I} = (y \mapsto -) * (z \mapsto -)$, $\Gamma_1 = \{r1 : \mathsf{wref\ int},\ t_{d1} : \mathsf{int},\ t_{y1} : \mathsf{int},\ t_{z1} : \mathsf{int}\}$, and $\Gamma_2 = \{r2 : \mathsf{wref\ int},\ t_{d2} : \mathsf{int},\ t_{y2} : \mathsf{int},\ t_{z2} : \mathsf{int}\}$. It is easy to show that $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma_1.\mathsf{emp}\}\ c_1\ \{\Gamma_1.\mathsf{emp}\}$ holds for any Ψ, assuming the obvious rules for while and if:

$$\frac{\Gamma_{\mathtt{G}}.I, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p} \wedge b\}\ c\ \{\Gamma.\mathtt{p}\}}{\Gamma_{\mathtt{G}}.I, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p}\}\ \mathsf{while}\ (b)\ \mathsf{do}\ c\ \{\Gamma.\mathtt{p} \wedge \neg b\}}\ (\textsc{while})$$

$$\frac{\Gamma_{\mathtt{G}}.I, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p} \wedge b\}\ c_1\ \{\Gamma'.\mathtt{p}'\} \qquad \Gamma_{\mathtt{G}}.I, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p} \wedge \neg b\}\ c_2\ \{\Gamma'.\mathtt{p}'\}}{\Gamma_{\mathtt{G}}.I, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p}\}\ \mathsf{if}\ (b)\ c_1\ \mathsf{else}\ c_2\ \{\Gamma'.\mathtt{p}'\}}\ (\textsc{if})$$

Similarly, $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma_2.\mathsf{emp}\}\ c_2\ \{\Gamma_2.\mathsf{emp}\}$ holds. Therefore, by the parallel-composition rule, we have

$$\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma_1.\mathsf{emp} * \Gamma_2.\mathsf{emp}\}\ c_1 \| c_2\ \{\Gamma_1.\mathsf{emp} * \Gamma_2.\mathsf{emp}\}$$

Soundness We next prove the soundness of the concurrent version of SL^{W}. We first define the semantics of the judgment for commands.

Definition 4.1 (Partial correctness)

$$\begin{array}{l}\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \models_{\mathrm{c}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\} \ \triangleq \\ \quad \forall \mathtt{F} \geq [\![\Psi]\!], s, h.\ \big(\models\ (s,h) \text{ sat } (\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma.\mathtt{p}))\big) \Rightarrow \\ \quad\quad \neg((s,h,c) \longrightarrow^{*} \mathsf{wrong}) \wedge \\ \quad\quad \big(\forall s', h'.\ (s,h,c) \longrightarrow^{*} (s',h',\epsilon) \Rightarrow \\ \quad\quad\quad \exists \mathtt{F}' \geq \mathtt{F}.\ \models\ (s',h') \text{ sat } (\mathtt{F}', (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma'.\mathtt{p}'))\end{array}$$

Theorem 4.1 (Soundness)
If $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \vdash_{\mathrm{c}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$, then $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \models_{\mathrm{c}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

Proving the soundness theorem directly based on the induction over the derivation of the assumption would not go through for the case of the PAR rule. We need a stronger induction principle given by Definition 4.3. Then the soundness follows from the following Lemma 4.2 and Lemma 4.3.

Definition 4.2
$\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models^{0}_{\mathrm{c}} (s,h,c) : \Gamma.\mathtt{p}$ always holds; $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models^{k+1}_{\mathrm{c}} (s,h,c) : \Gamma.\mathtt{p}$ holds iff

1. $\neg((s,h,c) \longrightarrow \mathsf{wrong})$;
2. if $c = \epsilon$, then $\models\ (s,h)$ sat $(\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma.\mathtt{p}))$.
3. for all s', h', and c', if $(s,h,c) \longrightarrow (s',h',c')$, then there exists $\mathtt{F}'$ such that $\mathtt{F}' \geq \mathtt{F}$ and $\forall j \leq k.\ \Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F}' \models^{j}_{\mathrm{c}} (s',h',c') : \Gamma.\mathtt{p}$;
4. there exist h_1, h_2, s_1 and s_2 such that $h = h_1 \uplus h_2$, $s = s_1 \uplus s_2$, and
   a. $\models\ (s_1, h_1)$ sat $(\mathtt{F}, \Gamma_{\mathtt{G}}.\mathtt{I})$;
   b. for all h_1', s_1' and $\mathtt{F}'$, if $h_1' \perp h_2$, $s_1' \perp s_2$, $\mathtt{F}' \geq \mathtt{F}$, and $\models\ (s_1', h_1')$ sat $(\mathtt{F}', \Gamma_{\mathtt{G}}.\mathtt{I})$, then, $\forall j \leq k.\ \Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F}' \models^{j}_{\mathrm{c}} (s_1' \uplus s_2, h_1' \uplus h_2, c) : \Gamma.\mathtt{p}$.

We also define $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s,h,c) : \Gamma.\mathtt{p}$ as $\forall k.\ \Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models^{k}_{\mathrm{c}} (s,h,c) : \Gamma.\mathtt{p}$.

Lemma 4.1
If $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s,h,c) : \Gamma.\mathtt{p}$, then (1) $\neg((s,h,c) \longrightarrow^{*} \mathsf{wrong})$, and,
(2) for all s', h', if $(s,h,c) \longrightarrow^{*} (s',h',\epsilon)$, then there exists $\mathtt{F}'$ such that $\mathtt{F}' \geq \mathtt{F}$ and $\models\ (s',h')$ sat $(\mathtt{F}', (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma.\mathtt{p}))$.

Proof
By induction over the number of execution steps. ■

Definition 4.3

$$\begin{array}{l}\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \models_{\mathrm{sc}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\} \ \triangleq \\ \quad \forall \mathtt{F} \geq [\![\Psi]\!], s, h.\ \big(\models\ (s,h) \text{ sat } (\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma.\mathtt{p}))\big) \Rightarrow (\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s,h,c) : \Gamma'.\mathtt{p}')\end{array}$$

Lemma 4.2
If $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \models_{\mathrm{sc}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$, then $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \ \models_{\mathrm{c}} \ \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

Proof
Immediate from Lemma 4.1. ■

Lemma 4.3
If $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$, then $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \models_{\mathrm{sc}} \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

Proof
By induction over the derivation of $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma.\mathtt{p}\}\ c\ \{\Gamma'.\mathtt{p}'\}$.

Case 1 The PAR rule. We know $c = c_1 \| c_2$ and there exist Γ_1, Γ_2, Γ_1', Γ_2', $\mathtt{p}_1$, $\mathtt{p}_2$, $\mathtt{p}_1'$ and $\mathtt{p}_2'$ such that $\Gamma.\mathtt{p} = \Gamma_1.\mathtt{p}_1 * \Gamma_2.\mathtt{p}_2$, $\Gamma'.\mathtt{p}' = \Gamma_1'.\mathtt{p}_1' * \Gamma_2'.\mathtt{p}_2'$, $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma_1.\mathtt{p}_1\}\ c_1\ \{\Gamma_1'.\mathtt{p}_1'\}$, and $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \vdash_{\mathrm{c}} \{\Gamma_2.\mathtt{p}_2\}\ c_2\ \{\Gamma_2'.\mathtt{p}_2'\}$.

For all s, h and $\mathtt{F}$, if $\mathtt{F} \geq [\![\Psi]\!]$ and $\models (s,h)$ sat $(\mathtt{F}, \Gamma_{\mathtt{G}}.\mathtt{I} * \Gamma.\mathtt{p})$, show $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s,h,c) : \Gamma'.\mathtt{p}'$. We know there exist s_0, s_1, s_2, h_0, h_1 and h_2 such that $s = s_0 \uplus s_1 \uplus s_2$, $h = h_0 \uplus h_1 \uplus h_2$, $\models (s_0, h_0)$ sat $(\mathtt{F}, \Gamma_{\mathtt{G}}.\mathtt{I})$, $\mathtt{F}, s_1, h_1 \models \Gamma_1.\mathtt{p}_1$, and $\mathtt{F}, s_2, h_2 \models \Gamma_2.\mathtt{p}_2$.

By the induction hypothesis we know $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \models_{\mathrm{sc}} \{\Gamma_1.\mathtt{p}_1\}\ c_1\ \{\Gamma_1'.\mathtt{p}_1'\}$, and $\Gamma_{\mathtt{G}}.\mathtt{I}, \Psi \models_{\mathrm{sc}} \{\Gamma_2.\mathtt{p}_2\}\ c_2\ \{\Gamma_2'.\mathtt{p}_2'\}$. Then we know

(1) $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s_0 \uplus s_1, h_0 \uplus h_1, c_1) : \Gamma_1'.\mathtt{p}_1'$; and

(2) $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s_0 \uplus s_2, h_0 \uplus h_2, c_2) : \Gamma_2'.\mathtt{p}_2'$.

Our final goal $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}} (s,h,c) : \Gamma'.\mathtt{p}'$ can be derived from Lemma 4.4 below.
Other cases: Proof is similar and is omitted here. ■

Lemma 4.4
For all k, if $s = s_0 \uplus s_1 \uplus s_2$, $h = h_0 \uplus h_1 \uplus h_2$, $\models (s_0, h_0)$ sat $(\mathtt{F}, \Gamma_{\mathtt{G}}.\mathtt{I})$, $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{k} (s_0 \uplus s_1, h_0 \uplus h_1, c_1) : \Gamma_1.\mathtt{p}_1$, and $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{k} (s_0 \uplus s_2, h_0 \uplus h_2, c_2) : \Gamma_2.\mathtt{p}_2$, then we have $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{k} (s,h,c_1 \| c_2) : (\Gamma_1.\mathtt{p}_1) * (\Gamma_2.\mathtt{p}_2)$.

Proof
By induction over k. It is trivial when $k = 0$. Suppose $k = i+1$. We need to prove the four sub-goals in Definition 4.2.

The sub-goals 1, 2 and 4.a are trivial. 4.b can be proved based on the induction hypothesis. Now we prove the sub-goal 3.

Suppose $(s,h,c_1 \| c_2) \longrightarrow (s',h',c')$ for some s', h' and c'. There are three cases.

Case 1 $c_1 = c_2 = c' = \epsilon$.
Then we know $s = s'$, and $h = h'$. By $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{i+1} (s_0 \uplus s_1, h_0 \uplus h_1, c_1) : \Gamma_1.\mathtt{p}_1$, we know $\models (s_0 \uplus s_1, h_0 \uplus h_1)$ sat $(\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma_1.\mathtt{p}_1))$. Similarly, we can prove $\models (s_0 \uplus s_2, h_0 \uplus h_2)$ sat $(\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma_2.\mathtt{p}_2))$. Since $\models (s_0, h_0)$ sat $(\mathtt{F}, \Gamma_{\mathtt{G}}.\mathtt{I})$ and $\mathtt{I}$ is precise, we have $\models (s,h)$ sat $(\mathtt{F}, (\Gamma_{\mathtt{G}}.\mathtt{I}) * (\Gamma_1.\mathtt{p}_1) * (\Gamma_2.\mathtt{p}_2))$. By Definition 4.2 it is easy to prove $\forall j \leq i.\, \Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{j} (s',h',c') : (\Gamma_1.\mathtt{p}_1) * (\Gamma_2.\mathtt{p}_2)$.
Case 2 $c' = c_1' \| c_2$ and $(s,h,c_1) \longrightarrow (s',h',c_1')$.
By Lemma 3.3, we know there exist s'', and h'' such that $s' = s'' \uplus s_2$, $h' = h'' \uplus h_2$, and $(s_0 \uplus s_1, h_0 \uplus h_1, c_1) \longrightarrow (s'', h'', c_1')$. By $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{i+1} (s_0 \uplus s_1, h_0 \uplus h_1, c_1) : \Gamma_1.\mathtt{p}_1$ we know there exists $\mathtt{F}' \geq \mathtt{F}$ such that for all $j \leq i$, $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F}' \models_{\mathrm{c}}^{i} (s'', h'', c_1') : \Gamma_1.\mathtt{p}_1$ (following 3 of Definition 4.2). Then by 4.a of Definition 4.2 we know there exist s_0', s_1', h_0' and h_1' such that $s'' = s_0' \uplus s_1'$, $h'' = h_0' \uplus h_1'$, and $\models (s_0', h_0')$ sat $(\mathtt{F}', \Gamma_{\mathtt{G}}.\mathtt{I})$.

Then, by $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F} \models_{\mathrm{c}}^{i+1} (s_0 \uplus s_2, h_0 \uplus h_2, c_2) : \Gamma_2.\mathtt{p}_2$ and 4.b of Definition 4.2 we know for all $j \leq i$, $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F}' \models_{\mathrm{c}}^{j} (s_0' \uplus s_2, h_0' \uplus h_2, c_2) : \Gamma_2.\mathtt{p}_2$.

By the induction hypothesis, we know for all $j \leq i$, $\Gamma_{\mathtt{G}}.\mathtt{I}, \mathtt{F}' \models_{\mathrm{c}}^{j} (s', h', c') : (\Gamma_1.\mathtt{p}_1) * (\Gamma_2.\mathtt{p}_2)$.

Case 3 $c' = c_1 \| c_2'$ and $(s, h, c_2) \longrightarrow (s', h', c_2')$.
The proof is similar to the case above. ■

§5 Related Work

We discuss related work in three categories: (1) work related to language interoperation; (2) work related to integrating SL with type systems; and (3) work related to semantic models of types.

Most work in language interoperation focuses on the design and implementation of foreign function interfaces. Examples are plenty. Given a multilingual program, one natural question is how to reason about the program as a whole. This kind of reasoning requires models, program analyzers, and program logics that can work across language boundaries. Previous work has addressed the question of how to model the interoperation between dynamically typed languages and statically typed languages,[11] and the interoperation between two safe languages when they have different systems of computational effects.[22] By integrating SL and type systems, SL^{W} can elegantly reason about properties of heaps that are shared by high-level and low-level code.

Previous systems of integrating SL with type systems [10, 14] assume that programs are well-typed according to a syntactic type system, and SL is then used as an add-on to reason about more properties of programs. Honda *et al*'s program logic [9, 25] for higher-order languages supports reference types but also requires a separate type system (in addition to the Hoare assertions); Reus *et al* [16] presented an extension of separation logic for supporting higher-order store (i.e., references to higher-order functions), but their logic does not support weak heaps which we believe embodies the key feature of reference types (i.e., the ability to perform safe updates without knowing the exact aliasing relation). Compared to previous systems, SL^{W} targets the interoperation between high-level and low-level code. It allows cross-boundary references and mixes SL formulas and types.

Pottier showed how to use an anti-frame rule to encode weak references in separation logic.[15] His encoding relies heavily on the information-hiding aspect. A client of a weak reference is supplied with a pair of a getter and a setter function and the very existence of a reference cell is hidden by the anti-frame rule. Most type systems (and also this article), in contrast, use a global heap type to ensure the soundness of weak references. The benefit is that a client of a weak heap can get direct references to objects in the weak heap. This style matches the setting of foreign function interfaces where references to weak-heap objects are passed to foreign code. The plus side of Pottier's encoding is that a single logic suffices while we mix separation logic with a type system.

The soundness of SL^W is justified by defining a semantic model, notably for types. Ahmed[1] and Appel *et al.*[3] presented a powerful index-based semantic model for a rich type system with ML-style references. They rely on constructing a "dependently typed" global heap type to break the circularity discussed in Section 3. Our current work, in contrast, simply takes a fixed point of the recursively defined heap type predicate and avoids building any dependently typed data structures. Our work also differs from theirs in that we are reasoning about reference types in a program logic. Appel *et al.*[3] can also support impredicative polymorphism, which is not addressed in our current work. Birkedal *et al.*[4] recently presented a category-theoretic model that accommodates reference types as well as impredicative polymorphism. Similar to our model, their model also finds a fixed point and there is no need to work with approximation information. On the other hand, it appears that an implementation of their model requires a stratification of types and the use of dependent types, which our model avoids.

§6 Discussion and Future Work

This work aims toward a framework for reasoning about language interoperation, but a lot remains to be done. A realistic high-level language contains many more language features and types. We do not foresee much difficulty in incorporating language features and types at the logic level as their modeling is largely independent from the interaction between weak and strong heaps. One technical concern is how to extend our semantic model to cover a complicated type system, including function types and OO classes.

SL^W does not formally consider the effect of a garbage collector. A garbage collector would break the crucial monotonicity condition of the weak heap that our semantic model relies on. We believe a possible way to overcome this problem is to use a region-based type system.[21] A garbage collector would also imply that there cannot be direct references from strong heaps to weak heaps; an extra level of indirection has to be added.

§7 Conclusion

In his survey paper of separation logic,[17] Reynolds asked "*whether the dividing line between types and assertions can be erased.*" This article adds evidence that the type-based approach has its unique place when ensuring safety in weak heaps and when reasoning about the interaction between weak and strong heaps. The combination of types and SL provides a powerful framework for checking safety and verifying properties of multilingual programs.

Acknowledgements

We thank Hongseok Yang for pointing out the connection between our work and Pottier's work. We thank anonymous referees for suggestions and comments on an earlier version of this article. Gang Tan is supported in part by NSF grant CCF-0915157. Zhong Shao is supported in part by a gift from Microsoft and NSF grants CCF-0524545 and CCF-0811665. Xinyu Feng is supported in

part by NSF grant CCF-0524545 and National Natural Science Foundation of China (grant No. 90818019).

References

1) Ahmed, A. J., "Semantics of Types for Mutable State," Ph. D. thesis, Princeton University, 2004.
2) Appel, A. W. and McAllester, D., "An indexed model of recursive types for foundational proof-carrying code," *ACM Trans. on Prog. Lang. and Sys., 23, 5*, pp. 657-683, 2001.
3) Appel, A. W., Mellies, P.-A., Richards, C. D. and Vouillon, J., "A very modal model of a modern, major, general type system," in *Proc. of 34th ACM Symp. on Principles of Prog. Lang.*, ACM Press, pp. 109-122, Jan. 2007.
4) Birkedal, L., Støvring, K. and Thamsborg, J., "Realizability semantics of parametric polymorphism, references, and recursive types," in *FoSSaCS*, Springer-Verlag, pp. 456-470, April 2009.
5) Bornat, R., Calcagno, C., O'Hearn, P. and Parkinson, M., "Permission accounting in separation logic," in *Proc. 32nd ACM Symp. on Principles of Prog. Lang.*, pp. 259-270, 2005.
6) Furr, M. and Foster, J. S., "Checking type safety of foreign function calls," *ACM Trans. Program. Lang. Syst., 30, 4*, pp. 1-63, 2008.
7) Harper, R., "A simplified account of polymorphic references," *Information Processing Letters, 57, 1*, pp. 15-16, 1996.
8) Hoare, C. A. R., "An axiomatic basis for computer programming," *Commun. ACM, 12, 10*, pp. 578-580, October 1969.
9) Honda, K., Yoshida, N. and Berger, M., "An observationally complete program logic for imperative higher-order frame rules," in *Proc. 20th IEEE Symposium on Logic in Computer Science*, pp. 270-279, June 2005.
10) Krishnaswami, N., Birkedal, L., Aldrich, J. and Reynolds, J., "Idealized ML and its separation logic," Unpublished manuscript, July 2007.
11) Matthews, J. and Findler, R. B., "Operational semantics for multi-language programs," in *Proc. 34th ACM Symp. on Principles of Prog. Lang.*, pp. 3-10, 2007.
12) O'Hearn, P. W., Reynolds, J. C. and Yang, H., "Local reasoning about programs that alter data structures," in *Computer Science Logic*, pp. 1-19, 2001.
13) O'Hearn, P. W., Yang, H. and Reynolds, J. C., "Separation and information hiding," in *Proc. 31th ACM Symp. on Principles of Prog. Lang.*, pp. 268-280, Venice, Italy, Jan. 2004.
14) Parkinson, M., "Local reasoning for Java," Ph.D. thesis, *Tech Report UCAM-CL-TR-654*, University of Cambridge Computer Laboratory, Oxford, Nov. 2005.
15) Pottier, F., "Hiding local state in direct style: a higher-order anti-frame rule," in *Proc. 23rd IEEE Symposium on Logic in Computer Science*, pp. 331-340, June 2008.
16) Reus, B. and Schwinghammer, J., "Separation logic for higher-order store," in *20th International Workshop on Computer Science Logic (CSL)*, pp. 575-590, 2006.

17) Reynolds, J. C., "Separation logic: A logic for shared mutable data structures," in *Proc. 17th IEEE Symposium on Logic in Computer Science*, pp. 55-74, July 2002.

18) Tan, G. and Croft, J., "An empirical security study of the native code in the JDK," in *17th Usenix Security Symposium*, pp. 365-377, 2008.

19) Tan, G., Shao, Z., Feng, X. and Cai, H., "Weak updates and separation logic," in *Proc. of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09)*, pp. 178-193, 2009.

20) Tofte, M., "Type inference for polymorphic references," *Inf. and Comp., 89, 1*, pp. 1-34, 1990.

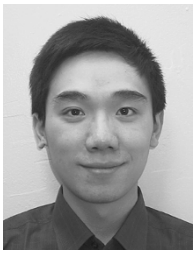
21) Tofte, M. and Talpin, J.-P., "Region-based memory management," *Information and Computation, 132, 2*, pp. 109-176, 1997.

22) Trifonov, V. and Shao, Z., "Safe and principled language interoperation," in *8th European Symposium on Programming (ESOP)*, pp. 128-146, 1999.

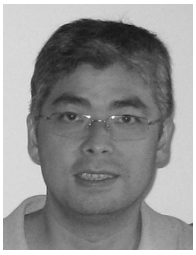
23) Vafeiadis, V. and Parkinson, M. J., "A marriage of rely/guarantee and separation logic," in *CONCUR*, pp. 256-271, 2007.

24) Wright, A. K. and Felleisen, M., "A syntactic approach to type soundness," *Information and Computation, 115, 1*, pp. 38-94, 1994.

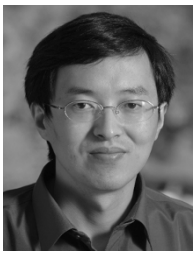
25) Yoshida, N., Honda, K. and Berge, M., "Logical reasoning for higher-order functions with local state," in *FoSSaCS* (Seidl, H. ed.), pp. 361-377, March 2007.



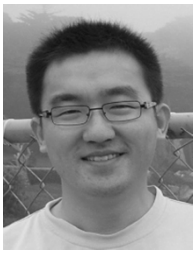
Gang Tan: He is an Assistant Professor of Computer Science and Engineering at Lehigh University, USA. His research interests include software security and programming languages. He leads Lehigh's Security of Software (SOS) group, which aims to create secure and reliable software systems.



Zhong Shao: He is a Professor of Computer Science at Yale University. His research interests include certified code, language-based security, programming languages and compilers, and operating systems. He currently leads the Yale FLINT group to build a practical infrastructure for constructing large-scale certified systems software.



Xinyu Feng: He is a professor of School of Computer Science and Technology, University of Science and Technology of China. His research interests are in the area of programming languages and formal methods. In particular, he is interested in developing theories, programming languages and tools to build formally certified system software, with rigorous guarantees of safety and correctness.



Hongxu Cai: He is a software engineer at Google Inc., Mountain View, California. He received his Doctoral degree in computer science from Tsinghua University, China.