Weak updates and separation logic

Gang Tan[1], Zhong Shao[2], Xinyu Feng[3], and Hongxu Cai[4]

[1]Lehigh University, [2]Yale University
[3]Toyota Technological Institute at Chicago, [4]Google Inc.

Abstract. Separation Logic (SL) provides a simple but powerful technique for reasoning about imperative programs that use shared data structures. Unfortunately, SL 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 SL when reasoning about the interaction between many high-level languages (e.g., ML, Java, C#) and low-level ones since these 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 SL with reference types and elegantly reasons about the interaction between strong and weak updates. We also describe a semantic framework for reference types; this framework is used to prove the soundness of SL^{W}.

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 (SL [10, 13]) 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 at 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 paper 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 heaps, one natural thought to mix strong and weak updates is to extend SL with types so that assertions can also describe 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, allowing general reference types in $\{e \mapsto \tau\}$ would make SL unsound. An example demonstrating this point is as 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) and uses the additional assertion $\{e \mapsto \tau\}$. The precondition states that y points to a heap cell whose contents are of type "even ref" (i.e., a reference to an even integer). Therefore, 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 y does not 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 [19], where types are treated as syntax. Following the tradition of Hoare Logic, SL's soundness is proved through a denotational model, and SL assertions are interpreted semantically. There is a need to resolve the conflict between syntactic and semantic soundness proofs.

In this paper, 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:

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

Fig. 1. Language syntax

- 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, and accommodates cross-boundary pointers (from weak to strong heaps and vice versa). This is achieved by statically maintaining the distinction between pointers to weak heaps and pointers to strong heaps. SL^{W} is presented in Section 2.
- To resolve the conflict 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.

2 SL^{W}: Separation logic with weak updates

We next describe SL^{W}, an extension of SL that incorporates reasoning over 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.

We first describe some common notations. 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, $f_1 \uplus f_2$ is the union of f_1 and f_2 when their domains are disjoint, and undefined otherwise. We write $f \setminus X$ for a new map after removing elements in X from the domain of f. We write $\vec{x}$ for a sequence of xs and ε for an empty sequence.

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 [6], augmented with a set of commands for manipulating heap data structures. It is similar to the one used in Reynolds' presentation of SL [13]. Informally, the command "$x := [e]$" loads the contents at location e into variable x; "$[x] := e$" updates the location at x with the value e; "$x := \mathsf{alloc}(e)$" allocates a new location, initializes it with e, and assigns the new location to x; "$\mathsf{free}(e)$" deallocates the location e.

In the syntax, we use n for integers, x for variables, ℓ for heap locations, and op for arithmetic operators. We assume there is an infinite number of variables and locations.

Figure 2 presents a formal operational semantics of the language. A state consists of a map $\mathtt{r}$ from variables to values, a heap h, and a sequence of commands. Commands bring one state to another state and their semantics is formally defined by a step relation $\longmapsto$. We write $\longmapsto^*$ for the reflexive and transitive closure of $\longmapsto$.

A state may have no next state, i.e., "getting stuck". For example, a state whose next instruction to execute is $[x] := e$ gets stuck when x does not represent a location or the

$$
\begin{array}{ll}
(\textit{State}) & s ::= (\mathtt{r}, h, \vec{c}) \\
(\textit{Locals}) & \mathtt{r} ::= \textit{Var} \to \textit{Value} \\
(\textit{Heap}) & h ::= \{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}
\end{array}
$$

$(\mathtt{r}, h, c \cdot \vec{c_1}) \longmapsto (\mathtt{r}_2, h_2, \vec{c_2})$		
if $c =$	then $(\mathtt{r}_2, h_2, \vec{c_2}) =$	
$\ldots$	$\ldots$	
$x := [e]$	$(\mathtt{r}[x \rightsquigarrow h(\ell)], h, \vec{c_1})$	when $\mathtt{r}(e) = \ell$ and $\ell \in \mathrm{dom}(h)$
$[x] := e$	$(\mathtt{r}, h[\ell \rightsquigarrow \mathtt{r}(e)], \vec{c_1})$	when $\mathtt{r}(x) = \ell$ and $\ell \in \mathrm{dom}(h)$
$x := \mathsf{alloc}(e)$	$(\mathtt{r}[x \rightsquigarrow \ell], h \uplus \{\ell \mapsto \mathtt{r}(e)\}, \vec{c_1})$	when $\ell \notin \mathrm{dom}(h)$
$\mathsf{free}(e)$	$(\mathtt{r}, h \setminus \{\ell\}, \vec{c_1})$	when $\mathtt{r}(e) = \ell$ and $\ell \in \mathrm{dom}(h)$

$$
\text{where } \mathtt{r}(e) = \begin{cases} \mathtt{r}(x) & \text{when } e = x \\ v & \text{when } e = v \\ op(\mathtt{r}(e_1), \ldots, \mathtt{r}(e_n)) & \text{when } e = op(e_1, \ldots, e_n) \end{cases}
$$

Fig. 2. Operational semantics

location is not in the domain of the state's heap. A state is a terminal state when the sequence of commands is empty.

Definition 1. *(Stuck and terminal states)*

$$
\begin{array}{ll}
\mathrm{stuck}(s) & \triangleq \neg(\exists s'.\ s \longmapsto s') \\
\mathrm{terminal}(r, h, \vec{c}) & \triangleq \vec{c} = \varepsilon
\end{array}
$$

Below we define the usual notions of safety and termination:

Definition 2. *(Safety and termination)*

$$
\begin{array}{ll}
\mathrm{safe}(s) & \triangleq \forall s'.\ \big((s \longmapsto^* s') \wedge \neg\mathrm{terminal}(s')\big) \Rightarrow \exists s''.\ s' \longmapsto s'' \\
\mathrm{terminate}(s) & \triangleq \forall s'.\ s \longmapsto^* s' \Rightarrow \exists s''.\ s' \longmapsto^* s'' \wedge \mathrm{terminal}(s'')
\end{array}
$$

2.2 The logic SL^{W}

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

SL^{W} is equipped with a simple type system that classifies integers and 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 in this setting most data are passed by references.

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

Fig. 3. Assertions and types

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

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

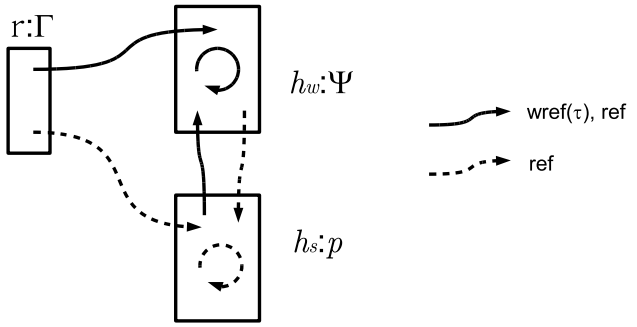
$$
\frac{\Psi(\ell) = \tau}{\Psi, \Gamma \vdash \ell : \mathsf{wref}\ \tau} \qquad \frac{\forall i \in [1..n].\ \Psi, \Gamma \vdash e_i : \mathsf{int}}{\Psi, \Gamma \vdash op(e_1, \ldots, e_n) : \mathsf{int}}
$$

Fig. 4. Typing rules for expressions

Type int is for all integers and ref for all locations. Type "wref τ" is for locations in a weak heap, but not in a strong heap. 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 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.

The following schematic diagram helps to understand the relationship between weak heaps, strong heaps, local variables, assertions and various kinds of types in SL^{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 p. Pointers to weak-heap cells (in solid lines) have type "wref τ" or ref. Pointers to strong heap cells (in dotted lines) can have only type ref.

Figure 5 presents rules for checking commands. These rules use the judgment $\Psi \vdash \{\Gamma,\ \mathtt{p}\}\ \vec{c}\ \{\Gamma',\ \mathtt{p}'\}$. In this judgment, Ψ, Γ and p are preconditions and specify conditions on the weak heap, local variables, and the strong heap respectively. Postcon-

$\boxed{\Psi \vdash \{\Gamma,\ \mathtt{p}\}\ \vec{c}\ \{\Gamma',\ \mathtt{p}'\}}$ **(*Well-formed statements*)**

$$\frac{\Psi,\Gamma \vdash e : \mathsf{ref} \quad \Psi,\Gamma \vdash y : \tau}{\Psi \vdash \{\Gamma,\ \{e \mapsto y\}\}\ x := [e]\ \{\Gamma[x \rightsquigarrow \tau],\ x = y \wedge \{e \mapsto x\}\}}\ (\textsc{s-load})$$
where $x \notin \mathsf{FV}(e)$

$$\frac{\Psi,\Gamma \vdash x : \mathsf{ref}}{\Psi \vdash \{\Gamma,\ \{x \mapsto -\}\}\ [x] := e\ \{\Gamma,\ \{x \mapsto e\}\}}\ (\textsc{s-update})$$

$$\frac{}{\Psi \vdash \{\Gamma,\ \mathsf{emp}\}\ x := \mathsf{alloc}(e)\ \{\Gamma[x \rightsquigarrow \mathsf{ref}],\ \{x \mapsto e\}\}}\ (\textsc{s-alloc})$$
where $x \notin \mathsf{FV}(e)$

$$\frac{\Psi,\Gamma \vdash e : \mathsf{ref}}{\Psi \vdash \{\Gamma,\ \{e \mapsto -\}\}\ \mathsf{free}(e)\ \{\Gamma,\ \mathsf{emp}\}}\ (\textsc{s-free})$$

$$\frac{\Psi \vdash \{\Gamma,\ \mathtt{p}\}\ \vec{c}\ \{\Gamma',\ \mathtt{p}'\}}{\Psi \vdash \{\Gamma,\ \mathtt{p} * \mathtt{p}_1\}\ \vec{c}\ \{\Gamma',\ \mathtt{p}' * \mathtt{p}_1\}}\ (\textsc{frame})$$
where no variable occurring free in $\mathtt{p}_1$ is modified by $\vec{c}$

↑ THE WORLD OF STRONG HEAPS

– –

↓ THE WORLD OF WEAK HEAPS

$$\frac{\Psi,\Gamma \vdash e : \mathsf{wref}\ \tau}{\Psi \vdash \{\Gamma,\ \mathsf{emp}\}\ x := [e]\ \{\Gamma[x \rightsquigarrow \tau],\ \mathsf{emp}\}}\ (\textsc{w-load})$$

$$\frac{\Psi,\Gamma \vdash x : \mathsf{wref}\ \tau \quad \Psi,\Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma,\ \mathsf{emp}\}\ [x] := e\ \{\Gamma,\ \mathsf{emp}\}}\ (\textsc{w-update})$$

$$\frac{\Psi,\Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma,\ \mathsf{emp}\}\ x := \mathsf{alloc}(e)\ \{\Gamma[x \rightsquigarrow \mathsf{wref}\ \tau],\ \mathsf{emp}\}}\ (\textsc{w-alloc})$$

Fig. 5. Rules for commands (Rules for assignments, conditional statements, loops, and sequencing are the same as the ones in Hoare Logic and are omitted.)

ditions are Γ' and $\mathtt{p}'$; they specify conditions on local variables and the strong heap of the state after executing $\vec{c}$. 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.

Rules in Figure 5 are divided into two groups. One group is for the world of strong heaps, and another for the world of weak heaps. 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 a type system for mutable-reference types. The weak-update rule W-UPDATE requires that the

Operational semantics: $(\mathtt{r}, h, \mathsf{s2w}(x) \cdot \vec{c}) \longmapsto (\mathtt{r}, h, \vec{c})$

Rule:
$$\frac{\Psi, \Gamma \vdash x : \mathsf{ref} \quad \Psi, \Gamma \vdash e : \tau}{\Psi \vdash \{\Gamma,\ \{x \mapsto e\}\}\ \mathsf{s2w}(x)\ \{\Gamma[x \rightsquigarrow \mathsf{wref}\ \tau],\ \mathsf{emp}\}}\ \text{S2W}$$

Fig. 6. Rule for converting a location from the strong heap to the weak heap

$$\frac{\vdash \{\Gamma_1', \mathtt{p}_1'\} \Rightarrow \{\Gamma_1, \mathtt{p}_1\} \quad \Psi \vdash \{\Gamma_1, \mathtt{p}_1\}\ \vec{c}\ \{\Gamma_2, \mathtt{p}_2\} \quad \vdash \{\Gamma_2, \mathtt{p}_2\} \Rightarrow \{\Gamma_2', \mathtt{p}_2'\}}{\Psi \vdash \{\Gamma_1', \mathtt{p}_1'\}\ \vec{c}\ \{\Gamma_2', \mathtt{p}_2'\}}\ \text{WEAKENING}$$

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

$$\frac{}{\vdash \{\Gamma, \mathtt{p}\} \Rightarrow \{\Gamma, \mathtt{p} \wedge \{x : \Gamma(x)\}\}}\ \text{W1} \qquad \frac{\vdash \mathtt{p} \Rightarrow \mathtt{p}'}{\vdash \{\Gamma, \mathtt{p}\} \Rightarrow \{\Gamma, \mathtt{p}'\}}\ \text{W2}$$

Fig. 7. Weakening rules

pointer be of type "wref τ", and that 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]

Figures 6 and 7 present some rules that show the interaction between weak and strong heaps. Figure 6 adds a new instruction "s2w(x)" for converting a location from a strong heap to a weak heap. Operationally, this instruction is a no-op (so it is an annotation, rather than a "real" instruction). Its typing rule, however, involves transforming the ownership in the strong heap to a pointer of weak-reference types. 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.

Figure 7 presents weakening rules. Rule W1 converts type information in Γ to information in assertion 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 valid SL formula $\mathtt{p} \Rightarrow \mathtt{p}'$ is acceptable.

2.3 Examples

We now show a few examples that demonstrate the use of SL$^{\mathrm{W}}$. In these examples, we assume an additional type even for even integers. For clarity, we will also annotate

[1] We do not formally consider the interaction between garbage collectors and weak heaps. When considering a garbage collector, SL$^{\mathrm{W}}$ 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 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 help in guiding the type checking of SL^{W}.

The first example shows how the counterexample in the introduction (formula (2) on page 2) plays out in SL^{W}. The following program first initializes the heap to a form such that y points to a location of type "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 "Γ, p" between instructions.

$$
\begin{array}{ll}
 & \{\},\ \mathsf{emp} \\
z := \mathsf{alloc}_{\mathrm{w},\mathsf{even}}(2) & \\
 & \{z : \mathsf{wref\ even}\},\ \mathsf{emp} \\
y := \mathsf{alloc}_{\mathrm{s}}(z) & \\
 & \{y : \mathsf{ref}, z : \mathsf{wref\ even}\},\ \{y \mapsto z\} \qquad \text{by rule (w1)} \\
 & \{y : \mathsf{ref}, z : \mathsf{wref\ even}\},\ \{y \mapsto z\} \wedge \{z : \mathsf{wref\ even}\} \qquad \text{by rule (w2)} \\
 & \{y : \mathsf{ref}, z : \mathsf{wref\ even}\},\ \exists v.\ \{y \mapsto v\} \wedge \{v : \mathsf{wref\ even}\} \\
z := 0 & \\
 & \{y : \mathsf{ref}, z : \mathsf{int}\},\ \exists v.\ \{y \mapsto v\} \wedge \{v : \mathsf{wref\ even}\} \\
x := \mathsf{alloc}_{\mathrm{s}}(4) & \\
 & \{x : \mathsf{ref}, y : \mathsf{ref}, z : \mathsf{int}\},\ \exists v.\ (\{y \mapsto v\} \wedge \{v : \mathsf{wref\ even}\}) * \{x \mapsto 4\} \\
[x] := 3 & \\
 & \{x : \mathsf{ref}, y : \mathsf{ref}, z : \mathsf{int}\},\ \exists v.\ (\{y \mapsto v\} \wedge \{v : \mathsf{wref\ even}\}) * \{x \mapsto 3\}
\end{array}
$$

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 ref, 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 [4, 14] has shown that it is error prone when high-level code interoperates with low-level code. 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.

$$
\begin{array}{l}
\quad \{\},\ \mathsf{emp} \\
x := \mathsf{alloc_s}(4) \\
\quad \{x : \mathsf{ref}\},\ \{x \mapsto 4\} \\
y := \mathsf{alloc_s}(x) \\
\quad \{x : \mathsf{ref}, y : \mathsf{ref}\},\ \{x \mapsto 4\} * \{y \mapsto x\} \\
\mathsf{s2w}(x) \\
\quad \{x : \mathsf{wref\ even}, y : \mathsf{ref}\},\ \{y \mapsto x\} \\
\mathsf{s2w}(y) \\
\quad \{x : \mathsf{wref\ even}, y : \mathsf{wref}\ (\mathsf{wref\ even})\},\ \mathsf{emp}
\end{array}
$$

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 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 dereference 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 [16] and Harper [5]. 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 "wref τ" 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, it means that $h(\ell_0)$ should be an even number, $h(\ell_1)$ should be an integer, $h(\ell_2)$ should be of type "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 "wref even" to "wref int": type "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 "wref τ" 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. Let $h = \{\ell_0 \mapsto 4,\ \ell_1 \mapsto 3,\ \ell_2 \mapsto \ell_0,\ \ell_3 \mapsto \ell_1\}$. 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 "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 "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 "wref τ". 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:

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

We use 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 F we can define $\lambda\phi,\ell.\ \mathtt{F}(\ell)\ \phi$, which has the metatype *SemHeapEnv* → *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 *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 4. *(Contractiveness)*

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

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 5. *If* $\mathsf{contractive}(F)$*, then the following* μF *is the least fixed point*[2] *of the functional* $(\wp F)$*:*

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

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

The theorem is proved by following the indexed model of recursive types [2]. We present the proof in our technical report [15].

The following lemma is an immediate corollary of Theorem 5.

Lemma 6. *For any contractive* F*, any* ℓ, k, v*, we have* $\big(F(\ell)\ (\mu F)\ k\ v\big) \leftrightarrow \big((\mu F)(\ell)\ k\ v\big)$

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

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

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

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

Example 8. Let $\mathtt{F}_0 = \{\ell_0 : \mathrm{Even},\ \ell_1 : \mathrm{WRef}(\mathrm{Even})\}$. Then "$\mathrm{WRef}(\mathrm{Even})\ (\mu \mathtt{F}_0)\ k\ \ell_0$" holds for any k. To see this, for any $j < k$ and v, we have

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

The first step is by lemma 6, and the second is by the definition of $\mathtt{F}_0$ at location ℓ_0. We can similarly show "$\mathrm{WRef}(\mathrm{WRef}(\mathrm{Even}))\ (\mu \mathtt{F}_0)\ k\ \ell_1$" holds. □

Note that the definition of $\mathrm{WRef}(\mathtt{t})$ 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.

[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.

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

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

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

Definition 9. $F' \geq F \triangleq$
$\mathsf{contractive}(F') \wedge \mathsf{contractive}(F) \wedge \forall \ell \in \mathrm{dom}(F), \phi, k, v.\ F'(\ell)\ \phi\ k\ v \leftrightarrow F(\ell)\ \phi\ k\ v$

Lemma 10. *The relation $F' \geq F$ 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 11. $\models h : F \triangleq \mathrm{dom}(h) \subseteq \mathrm{dom}(F) \wedge \forall \ell \in \mathrm{dom}(h).F \models h(\ell) : F(\ell)$
$F \models v : t \triangleq \forall F' \geq F. \forall k.\ t\ (\mu F')\ k\ v$

With our model, the following theorem for memory operations can be proved (please see our technical report [15] for proofs).

Theorem 12.

(i) (Read) If $\models h : F$, and $\ell \in \mathrm{dom}(h)$, and $F \models \ell : \text{WRef}(t)$, then $F \models h(\ell) : t$.
(ii) (Write) If $\models h : F$, and $\ell \in \mathrm{dom}(h)$, and $F \models \ell : \text{WRef}(t)$, and $F \models v : t$, then $\models h[\ell \rightsquigarrow v] : F$.
(iii) (Allocation) If $\models h : F$, and $F \models v : t$, and $\mathsf{contractive}(t)$, and $\ell \notin \mathrm{dom}(F)$, then $\models h \uplus \{\ell \mapsto v\} : F \uplus \{\ell \mapsto t\}$.

3.2 Semantic model of SL$^{\mathbf{W}}$

To show the soundness of SL$^{\mathbf{W}}$, we define semantics for judgments in SL$^{\mathbf{W}}$ and then prove each rule as a lemma according to the semantics. Figure 8 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 $\text{WRef}(t)$ in Definition 7. All these types are contractive. The semantics of Ψ and Γ is just the point-wise extension of the semantics of types.

The predicate "$\mathtt{F}, \mathtt{r}, h \models \mathtt{p}$" interprets the truth of assertion $\mathtt{p}$. When $\mathtt{p}$ is a standard SL formula, the interpretation is the same as the one in SL. When $\mathtt{p}$ is $\{e : \tau\}$, the

$\boxed{[\![\tau]\!] \in \mathit{SemIType}}$

$[\![\mathsf{int}]\!] \triangleq \lambda\phi, k, v.\ \exists n.v = n.$ $\quad [\![\mathsf{ref}]\!] \triangleq \lambda\phi, k, v.\ \exists \ell.v = \ell.$ $\quad [\![\mathsf{wref}\ \tau]\!] \triangleq \mathrm{WRef}([\![\tau]\!])$

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

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

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

$\mathtt{F}, \mathtt{r}, h \models \{e : \tau\} \triangleq \mathtt{F} \models \mathtt{r}(e) : [\![\tau]\!]$
$\mathtt{F}, \mathtt{r}, h \models \mathsf{emp} \triangleq \mathrm{dom}(h) = \emptyset$
$\mathtt{F}, \mathtt{r}, h \models \{e_1 \mapsto e_2\} \triangleq \mathrm{dom}(h) = \mathtt{r}(e_1) \wedge h(\mathtt{r}(e_1)) = \mathtt{r}(e_2)$
$\mathtt{F}, \mathtt{r}, h \models \mathtt{p}_1 * \mathtt{p}_2 \triangleq \exists h_1, h_2.\ (h = h_1 \uplus h_2) \wedge (\mathtt{F}, \mathtt{r}, h_1 \models \mathtt{p}_1) \wedge (\mathtt{F}, \mathtt{r}, h_2 \models \mathtt{p}_2)$
$\mathtt{F}, \mathtt{r}, h \models \mathtt{p}_1 -\!\!* \mathtt{p}_2 \triangleq \forall h_1.\ ((\mathrm{dom}(h_1) \cap \mathrm{dom}(h) = \emptyset) \wedge (\mathtt{F}, \mathtt{r}, h_1 \models \mathtt{p}_1)) \Rightarrow (\mathtt{F}, \mathtt{r}, h_1 \uplus h \models \mathtt{p}_2)$

$\mathtt{F} \models \mathtt{r} : \Gamma \quad \triangleq \forall x \in \mathrm{dom}(\Gamma).\ \mathtt{F} \models \mathtt{r}(x) : [\![\Gamma(x)]\!]$
$\mathtt{r}, h \models \mathtt{F} * \mathtt{p} \triangleq \exists h_1, h_2.\ (h = h_1 \uplus h_2) \wedge (\mathrm{dom}(h_1) = \mathrm{dom}(\mathtt{F})) \wedge (\models h_1 : \mathtt{F}) \wedge (\mathtt{F}, \mathtt{r}, h_2 \models \mathtt{p})$

Fig. 8. Semantic definitions

---

interpretation depends on 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 $\mathtt{F} \models \mathtt{r} : \Gamma$ is the point-wise extension of $\mathtt{F} \models v : t$ to local variable types. The definition of "$\mathtt{r}, h \models \mathtt{F} * \mathtt{p}$" splits the heap into two parts. One for the weak heap, which should satisfy F, and the other for the strong heap, which is specified by 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 13.

$$\Psi, \Gamma \models e : \tau \triangleq \forall F \geq [\![\Psi]\!].\ \forall r.\ F \models r : \Gamma \Rightarrow F \models r(e) : [\![\tau]\!].$$

$$\models p \Rightarrow p' \triangleq \forall F, r, h.\ (F, r, h \models p) \Rightarrow (F, r, h \models p')$$

$$\models \{\Gamma, p\} \Rightarrow \{\Gamma', p'\} \triangleq$$
$$\forall F, r, h.\ (F \models r : \Gamma \wedge r, h \models F * p) \Rightarrow (F \models r : \Gamma' \wedge r, h \models F * p')$$

Now we are ready to interpret $\Psi \vdash \{\Gamma, \mathtt{p}\}\ \vec{c}\ \{\Gamma', \mathtt{p}'\}$. Following Hoare Logic, we define both partial and total correctness:

Definition 14. *(Partial and total correctness)*

$$\begin{array}{l}\Psi \models_{\mathrm{p}} \{\Gamma,\ p\}\ \vec{c}\ \{\Gamma',\ p'\} \triangleq \\ \quad \forall F \geq [\![\Psi]\!], r, h.\ \big((F \models r:\Gamma) \wedge (r,h \models F * p)\big) \Rightarrow \\ \quad\quad \mathrm{safe}(r,h,\vec{c}) \wedge \\ \quad\quad \big(\forall r', h'.\ (r,h,\vec{c}) \longmapsto^{*} (r',h',\varepsilon) \Rightarrow \exists F' \geq F.\ (F' \models r':\Gamma') \wedge (r',h' \models F' * p')\big)\end{array}$$

$$\begin{array}{l}\Psi \models_{\mathrm{t}} \{\Gamma,\ p\}\ \vec{c}\ \{\Gamma',\ p'\} \triangleq \\ \quad (\Psi \models_{\mathrm{p}} \{\Gamma,\ p\}\ \vec{c}\ \{\Gamma',\ p'\}) \wedge \\ \quad \big(\forall F \geq [\![\Psi]\!], r, h.\ \big((F \models r:\Gamma) \wedge (r,h \models F * p)\big) \Rightarrow \mathrm{terminate}(r,h,\vec{c})\big)\end{array}$$

In the partial-correctness interpretation, it assumes a state that satisfies the condition $\{\Gamma, \mathrm{p}\}$ and requires that the state be safe (see Definition 2 on page 4 for safety). In addition, it requires that, for any terminal state after the execution of $\vec{c}$, we must be able to find a new semantic heap type F' so that $\mathrm{F}' \geq \mathrm{F}$ and the new state satisfies $\{\Gamma', \mathrm{p}'\}$. Note that F' may be larger than F due to allocations in $\vec{c}$. The total-correctness interpretation requires termination in addition to the requirements of partial correctness.

Theorem 15. *All rules in Figures 5, 6 and 7 are sound for both partial and total correctness.*

The proof uses Theorem 12. It is largely straightforward and omitted. We refer interested readers to our technical report [15] for the proof.

4 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 [9], and the interoperation between two safe languages when they have different systems of computational effects [18]. 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 [11, 8] 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 [7, 20] for higher-order languages supports reference types but also requires a separate type system (in addition to the Hoare assertions); Reus *et al* [12] 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.

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.

5 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 [17]. 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.

6 Conclusion

In his survey paper of Separation Logic [13], Reynolds asked *"whether the dividing line between types and assertions can be erased"*. This paper 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.

Acknowledgments

We thank anonymous referees for suggestions and comments on an earlier version of this paper. 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] A. J. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[2] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. on Prog. Lang. and Sys.*, 23(5):657–683, 2001.

[3] A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL '07*, pages 109–122. ACM Press, Jan. 2007.

[4] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 30(4):1–63, 2008.

[5] R. Harper. A simplified account of polymorphic references. *Information Processing Letters*, 57(1):15–16, 1996.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):578–580, October 1969.

[7] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order frame rules. In *LICS '05*, pages 270–279, June 2005.

[8] N. Krishnaswami, L. Birkedal, J. Aldrich, and J. Reynolds. Idealized ML and its separation logic. July 2007.

[9] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proc. 34th ACM Symp. on Principles of Prog. Lang.*, pages 3–10, 2007.

[10] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19, 2001.

[11] M. Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge Computer Laboratory, Oxford, Nov. 2005. Tech Report UCAM-CL-TR-654.

[12] B. Reus and J. Schwinghammer. Separation logic for higher-order store. In *20th International Workshop on Computer Science Logic (CSL)*, pages 575–590, 2006.

[13] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.

[14] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *17th Usenix Security Symposium*, pages 365–377, 2008.

[15] G. Tan, Z. Shao, X. Feng, and H. Cai. Weak updates and separation logic. `http://www.cse.lehigh.edu/~gtan/paper/WUSL-tr.pdf`, June 2009.

[16] M. Tofte. Type inference for polymorphic references. *Inf. and Comp.*, 89(1):1–34, 1990.

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

[18] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *8th European Symposium on Programming (ESOP)*, pages 128–146, 1999.

[19] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[20] N. Yoshida, K. Honda, and M. Berge. Logical reasoning for higher-order functions with local state. In *FoSSaCS*, pages 361–377, March 2007.