Simple and Flexible Stack Types

Frances Perry[1]
frances@cs.princeton.edu

Chris Hawblitzel
chrishaw@microsoft.com

Juan Chen
juanchen@microsoft.com

June 2007

Technical Report
MSR-TR-2007-51

Typed intermediate languages and typed assembly languages for optimizing compilers require types to describe stack-allocated data. Previous type systems for stack data were either undecidable or did not treat arguments passed by reference. This paper presents a simple, sound, decidable type system expressive enough to support the Micro-CLI source language, including by-reference arguments. This type system safely expresses operations on aliased stack locations by using singleton pointers and a small subset of linear logic.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]The work by Frances Perry was done during an internship at Microsoft Research

1 Introduction

Java and C# are safe, high-level languages. The safety of Java and C# protects one program from another: safe applets cannot crash a browser, safe servlets cannot crash a server, and so on. The high level of abstraction makes programming easier, but makes compilation more challenging. Java and C# require sophisticated optimizing compilation to achieve performance competitive with programs written directly in C or assembly language.

Unfortunately, a large, complex compiler is likely to have bugs, and these bugs may cause the compiler to produce unsafe assembly language code. Proof-carrying code (PCC) [15] and typed assembly language (TAL) [14] solve this problem by verifying the safety of the assembly language code generated by the compiler, thus removing the compiler from the trusted computing base. Because the behavior of an assembly language program is undecidable in general, PCC and TAL require machine-checkable evidence to verify a program's safety. A type-preserving compiler generates this evidence by transforming a well-typed source program into a well-typed assembly language program, preserving the well-typedness of the program during each compilation phase in between the source and assembly language levels [14]. To do this, the compiler must define type systems for each intermediate language in the compilation. Java bytecode [12] and CIL [4] are well-known typed intermediate languages, but these still contain many high-level abstractions, such as single instructions for invoking virtual methods and platform-independent storage slots for local data. Below the Java bytecode and CIL levels, these abstractions break down into smaller pieces. A virtual method invocation turns into a method table lookup, instructions for pushing arguments onto a stack, a call instruction, plus prologue and epilogue code in the called method. Local data storage slots turn into machine-specific registers and stack slots. These lower-level concepts need lower-level types.

This paper describes SST (**S**imple **S**tack **T**ypes), a type system that is appropriate for type-checking stack operations in the lowest levels of a type-preserving compiler, including the final typed assembly language generated by the compiler. Previous type systems for stacks were either undecidable without explicit proof annotations [2, 9] or could not represent arguments passed and returned by reference [13]. By contrast, SST has a simple decision procedure, making it easy to use in an intermediate language. It expresses by-reference arguments, even when multiple references point to the same aliased location. It is provably type-safe, via standard preservation and progress lemmas. Finally, SST is simple and elegant enough to be a trustworthy component of a typed assembly language.

To represent stacks in the presence of aliasing, SST builds on ideas from stack-based TAL [13], alias types [18], and linear logic [6, 19]. Section 2 discusses these systems and related systems in more detail. Sections 3 and 4 introduce SST's types and instructions formally. Section 5 describes a translation from the Micro-CLI [9] source language to SST, demonstrating SST's expressiveness. Section 6 concludes.

2 Background and Related Work

Stack-based TAL (STAL) was the first TAL to support stacks. Its central idea, shared by SST, was a *stack type*, which specifies the known types of values on the stack at any point in a TAL program. For example, the STAL stack type "int :: int :: ρ" specifies that two integers live at the top of the stack, but all types deeper in the stack are unknown, specified only by the stack type variable ρ. Code blocks in STAL may be polymorphic over stack type variables.

In addition to the concatenation operator " :: ", STAL contains a compound stack type that can express some pointers into the middle of the stack. Unfortunately, STAL

$$\frac{\varsigma \Rightarrow \varsigma'}{\ell : \tau :: \varsigma \Rightarrow \ell : \tau :: \varsigma'}\ \text{s-imp-concat} \qquad \frac{\ell : \sigma \Rightarrow \ell : \sigma'}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : (\sigma' \wedge \{\ell_t : \tau\})}\ \text{s-imp-alias}$$

$$\frac{}{\varsigma \Rightarrow \varsigma}\ \text{s-imp-eq} \qquad \frac{}{\ell : (\tau :: \varsigma) \Rightarrow \ell : (\tau :: \varsigma \wedge \{\ell : \tau\})}\ \text{s-imp-add-alias}$$

$$\frac{\varsigma_1 \Rightarrow \varsigma_2 \quad \varsigma_2 \Rightarrow \varsigma_3}{\varsigma_1 \Rightarrow \varsigma_3}\ \text{s-imp-trans} \qquad \frac{}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : \sigma}\ \text{s-imp-drop-alias}$$

$$\frac{}{\ell : (\tau_1 :: \ell_q : (\sigma \wedge \{\ell_2 : \tau_2\})) \Rightarrow \ell : ((\tau_1 :: \ell_q : \sigma) \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-expand-alias}$$

$$\frac{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\}) \quad \varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_2 : \tau_2\})}{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\} \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-merge-alias}$$

Figure 1: Logical Stack Implication Rules

cannot express the possibly aliased pointers that C# compilers use to implement by-reference arguments. Consider the three C# methods below. The `swap` method takes two integer references and swaps the integers. The `f` method instantiates arguments `x` and `y` with pointers to local variables `a` and `b`, while `g` instantiates `x` and `y` with pointers to `c`:

```
void f() {
    int a = 10, b = 20;
    swap(ref a, ref b); }
void g() {
    int c = 30;
    swap(ref c, ref c); }
void swap(ref int x, ref int y) {
    int t = x;
    x = y;
    y = t; }
```

STAL cannot give a useful type to the `swap` method: even with compound types, STAL stack types must list the types of stack slots in precisely the order that they appear in memory. The STAL type for `swap` must reserve one particular stack slot for `x` and another for `y`, making it impossible for a caller to instantiate `x` and `y` with aliased pointers (as `g` does), with heap pointers (as is allowed by C#), or with two stack pointers in the opposite order. Regarding these limitations, Morrisett *et al.* say that, "it appears that this limitation could be removed by introducing a limited form of intersection type, but we have not yet explored the ramifications of this enhancement." (In fact, one subsequent TAL [2] did add intersection types, but did not explore its use for stacks. Furthermore, this type system was undecidable [2].) SST uses a form of intersection type, rather than using STAL's compound types.

A key advantage of stack allocation is the ease of stack deallocation: a program simply pops data from the top of the stack to deallocate the data. In general, popping may leave dangling pointers to popped data. STAL deals with this safely but awkwardly, applying a special validation rule before each use of any potentially dangling pointer. SST follows a more direct and flexible approach introduced by alias types [18] (although alias types handled heaps objects, not stack data). Alias types split a pointer type into two parts: the location ℓ of the data, and the type of the data at location ℓ. The pointer to the data has a singleton type $\text{Ptr}(\ell)$, which indicates that the pointer points exactly to the location ℓ, but deliberately does not specify the type of the data

at location ℓ. Instead, a separate *capability* specifies the current type at ℓ. For example, the capability $\{\ell \mapsto \text{int}\}$ specifies that ℓ currently holds an integer. Because of the separation between singleton pointer types and capabilities, the capabilities can evolve, independently of the pointer types, to track updates and deallocation.

To ensure that no two capabilities specify contradictory information about a single location, alias types impose a linearity discipline on the program's treatment of capabilities, prohibiting arbitrary duplication of the information contained in a capability. In particular, the capability $\{\ell \mapsto \text{int}\}$ is not equivalent to the capability $\{\ell \mapsto \text{int}, \ell \mapsto \text{int}\}$. However, alias types (and the similar capability calculus [3]) use non-standard operators and rules for controlling linearity. Following recent advice [20, 7, 5], SST uses operators and rules directly inspired by standard linear logic [6, 19] and separation logic [17, 8]. Linear logic and separation logic share a core of basic operators. Two are of particular interest for stacks: multiplicative conjunction "$\otimes$" (written as "$*$" in separation logic) and additive conjunction "&" (written as "$\wedge$" in separation logic). For example, to have "coffee $\otimes$ tea" is to have both coffee and tea. To have "coffee&tea" is to have a choice between coffee and tea, but not both. Ahmed and Walker observe that additive conjunction "allows us to specify different 'views' of the stack" [1] (though [1] did not explore applications of this observation); we take this observation as a starting point for representing by-reference arguments.

Jia, Spalding, Walker and Glew [9] used linear logic as the basis for a typed low-level language of stacks and heaps (we refer to this low-level language as "JSWG"). In contrast to STAL, JSWG expressed by-reference arguments. To demonstrate this, the authors also introduced the high-level "Micro-CLI" source language (modeled on the CLI intermediate format targeted by C# compilers [4]) and provided a translation from Micro-CLI programs to JSWG programs. In contrast to SST's decidable logic, JSWG's linear logic (which includes the standard linear operators $\otimes$, &, $\oplus$, $\multimap$, and !) is undecidable [11], making SST more practical than JSWG's system for a compiler intermediate language. Furthermore, JSWG expresses pointers using a heavyweight notion of "frozen" capabilities (with version numbers and "tag trees" for pointers into the stack) while SST relies solely on singleton pointer types and a minimal linear logic. Despite its smaller set of features, SST is still powerful enough to express Micro-CLI; Section 5 describes a translation of Micro-CLI programs to SST programs.

3 Simple Stack Types

Consider the STAL stack type $\text{int} :: \text{int} :: \rho$ from the Section 2. In alias type notation, each integer on the stack would have a capability $\{\ell \mapsto \text{int}\}$. In linear logic notation, the $\otimes$ operator would glue capabilities together to form a complete stack capability: $\{\ell_2 \mapsto \text{int}\} \otimes \{\ell_1 \mapsto \text{int}\} \otimes \rho$, where ℓ_2 and ℓ_1 are the locations of each of the two integers on the stack. SST takes this notation as a starting point, but makes two modifications. First, to simplify the type checking algorithm, SST replaces the commutative, associative $\otimes$ operator with the non-commutative, non-associative $::$ operator, resulting in a stack capability $\{\ell_2 \mapsto \text{int}\} :: \{\ell_1 \mapsto \text{int}\} :: \rho$. Second, rather than showing one location per stack slot, SST's notation puts stack slots in between locations, writing $\ell_2 : \text{int} :: \ell_1 : \text{int} :: \ell_0 : \rho$ to indicate that one integer falls between locations ℓ_2 and ℓ_1, and the other falls between locations ℓ_1 and ℓ_0. Note that this adds the extra location ℓ_0 to the example — for instance, the stack pointer might have type $\text{Ptr}(\ell_2)$, pointing to the top of the stack, while the frame pointer might have type $\text{Ptr}(\ell_0)$, pointing to the bottom of the frame.

The following grammar generates labeled stack types ς and unlabeled stack types σ (where τ indicates a single-word type, such as int):

$$\begin{array}{rrcl} \textbf{labeled stack type} & \varsigma & ::= & \ell : \sigma \\ \textbf{unlabeled stack type} & \sigma & ::= & \rho \mid \text{Empty} \mid \tau :: \varsigma \mid \sigma \wedge \{\ell : \tau\} \end{array}$$

$$\varrho ::= \rho \mid \text{Empty} \mid \tau :: \varsigma$$

$$\frac{}{\ell : \rho \overset{\{\}}{\Longrightarrow} \ell : \rho}\ \text{s-imp2-var} \qquad \frac{}{\ell : \text{Empty} \overset{\{\}}{\Longrightarrow} \ell : \text{Empty}}\ \text{s-imp2-empty}$$

$$\frac{\varsigma \overset{\phi}{\Longrightarrow} \varsigma'}{\ell : \tau :: \varsigma \overset{\phi \cup \{(\ell,\tau)\}}{\Longrightarrow} \ell : \tau :: \varsigma'}\ \text{s-imp2-concat}$$

$$\frac{\ell' : \sigma \overset{\phi}{\Longrightarrow} \ell' : \varrho}{\ell' : (\sigma \wedge \{\ell : \tau\}) \overset{\phi \cup \{(\ell,\tau)\}}{\Longrightarrow} \ell' : \varrho}\ \text{s-imp2-alias-left}$$

$$\frac{\varsigma \overset{\phi \cup \{(\ell,\tau)\}}{\Longrightarrow} \ell' : \sigma}{\varsigma \overset{\phi \cup \{(\ell,\tau)\}}{\Longrightarrow} \ell' : (\sigma \wedge \{\ell : \tau\})}\ \text{s-imp2-alias-right}$$

Figure 2: Algorithmic Stack Implication Rules

The unlabeled stack type variables ρ, empty stack Empty, and stack concatenation operator :: give SST the same expressiveness as the core of STAL, but little else. The real power of SST comes from the $\wedge$ operator, indicating aliasing. The stack type $\sigma \wedge \{\ell : \tau\}$ implies three things. First, σ holds. Second, the location ℓ resides either in the heap or in the part of the stack described by σ. Third, ℓ currently contains a word of type τ. Figure 1 shows the rules governing stack types; "$\varsigma \Rightarrow \varsigma'$" means that if ς holds, then ς' also holds. Some rules (s-imp-concat, s-imp-alias, s-imp-eq, s-imp-trans) are basic structural rules. The s-imp-add-alias and s-imp-merge-alias rules allow a program to add one or more aliases to a stack type. The s-imp-drop-alias rule lets a program drop unneeded aliases. The s-imp-expand-alias rule expands the scope of an alias, as described in more detail below.

Figure 2 shows an alternate version of the stack type rules, with a more algorithmic flavor. In fact, the alternate version is syntax directed: for any ς and ς', the syntax of ς and ς' determines which rule from Figure 2 to apply. Nevertheless, the two versions are equivalent:

Theorem 1 *$\varsigma \Rightarrow \varsigma'$ if and only if there exists some ϕ such that $\varsigma \overset{\phi}{\Longrightarrow} \varsigma'$.*

Thus, a type checker can use $\varsigma \overset{\phi}{\Longrightarrow} \varsigma'$ as a simple algorithm for deciding $\varsigma \Rightarrow \varsigma'$. The simplicity of the algorithm stems from the simplicity of the :: and $\wedge$ operators.

As an example, consider the `swap` function from Section 2. Suppose that the compiler pushes arguments to `swap` onto the stack from right-to-left, and stores the return address in a register. Upon entry to `swap`, the stack will hold the arguments `x` and `y`, each of which is a pointer to some location inside ρ:

$$\ell_2 : \text{Ptr}(\ell_x) :: \ell_1 : \text{Ptr}(\ell_y) :: \ell_0 : (\rho \wedge \{\ell_x : \text{int}\} \wedge \{\ell_y : \text{int}\})$$

Note that locations ℓ_x and ℓ_y may appear anywhere in ρ, in any order. In fact, ℓ_x and ℓ_y may be the same location. For example, suppose that just before calling `swap`, the stack has type $\ell_0 : \text{int} :: \varsigma$. Figure 1's s-imp-add-alias and s-imp-merge-alias rules prove:

$$\begin{array}{rl} & \ell_0 : \text{int} :: \varsigma \\ \Rightarrow & \ell_0 : ((\text{int} :: \varsigma) \wedge \{\ell_0 : \text{int}\} \wedge \{\ell_0 : \text{int}\}) \end{array}$$

Using this, the program can choose $\rho = (\text{int} :: \varsigma)$, choose $\ell_x = \ell_y = \ell_0$, push two pointers to ℓ_0 onto the stack, and call `swap`.

Figure 1's rules also allow reordering of aliases. For example, the s-imp-drop-alias, s-imp-alias, and s-imp-merge-alias rules prove:

$$\begin{array}{rl} & \ell_0 : (\rho \wedge \{\ell_y : \text{int}\} \wedge \{\ell_x : \text{int}\}) \\ \Rightarrow & \ell_0 : (\rho \wedge \{\ell_x : \text{int}\} \wedge \{\ell_y : \text{int}\}) \end{array}$$

Section 2 mentioned the danger of pointers left dangling after the program pops a word from the stack. The syntax $\sigma \wedge \{\ell : \tau\}$ expresses a clear scope in which ℓ remains safe to use: ℓ definitely contains type τ as long as σ remains unmodified. If the program pops a word from σ, for example, then the alias $\{\ell : \tau\}$ must be discarded (see section 4.1 for details). The rules governing this scope are simple: s-imp-expand-alias expands the scope of an alias, but there is no rule to contract the scope. Expansion is safe, and allows a caller to pass a reference on to another method. The `h` method shown below expands the scope of `c` before calling `swap`. Contraction, on the other hand, could leave unsafe dangling pointers, as shown by the illegal and unsafe C# method `illegalMethod`:

```
void h(ref int c)       { swap(ref c, ref c); }
ref int illegalMethod() { int c; return ref c; }
```

Relation to linear logic. Just as $::$ is a limited version of the linear logic $\otimes$ operator, the $\wedge$ operator is a limited version of the linear logic & operator. More specifically, the notation $\sigma \wedge \{\ell : \tau\}$ corresponds to the linear logic formula $\sigma \& (\{\ell \mapsto \tau\} \otimes \top)$, where $\top$ is the linear logic notation to indicate any resource. Intuitively, knowing $\sigma \& (\{\ell \mapsto \tau\} \otimes \top)$ means that you can choose to look at the stack in one of two ways: either consider the stack to have type σ, or consider the stack to have type $\{\ell \mapsto \tau\} \otimes \top$. The latter case tells you that the stack holds type τ at location ℓ, plus some other data represented by $\top$.

The s-imp-expand-alias rule and lack of a contraction rule also correspond to linear logic, where $A \otimes (B \& (C \otimes \top))$ implies $(A \otimes B) \& (C \otimes \top)$, but $(A \otimes B) \& (C \otimes \top)$ does not imply $A \otimes (B \& (C \otimes \top))$; linear logic can expand, but not contract, the scope of "$\&(C \otimes \top)$". Unlike JSWG [9]'s scoping via version numbers and tag trees, SST's scoping follows naturally from linear logic rules.

Locations. A location ℓ may be a location variable "η", the location of the bottom of the stack "base", the next location towards the top of the stack "next(ℓ)", or a heap location "p" (assuming an infinite supply of locations p for heap allocation):

$$\textbf{location} \quad \ell \quad ::= \quad \eta \mid \text{base} \mid \text{next}(\ell) \mid p$$

For example, the STAL type int :: int :: ρ may be written in SST as "$\text{next}^2(\eta) : \text{int} :: \text{next}(\eta) : \text{int} :: \eta : \rho$", where $\text{next}^2(\eta)$ is an abbreviation for $\text{next}(\text{next}(\eta))$. For convenience, we frequently use the following abbreviation:

$$(\tau_n \ldots \tau_1)@(\ell : \sigma) = \text{next}^n(\ell) : \tau_n :: \ldots :: \text{next}^1(\ell) : \tau_1 :: \ell : \sigma$$

With this, the STAL type int :: int :: ρ may be written in as $(\text{int}; \text{int})@(\eta : \rho)$.

4 Formalization

Types. SST supports integer type "int", nonsense type "Nonsense" for uninitialized stack slots, heap pointer type "HeapPtr(τ)" for pointers to heap values of type τ, singleton type "Ptr(ℓ)", and code type "$\forall[\Delta](\Gamma, \varsigma)$" for code blocks.

$$\textbf{type} \quad \tau \quad ::= \quad \text{int} \mid \text{Nonsense} \mid \text{HeapPtr}(\tau) \mid \text{Ptr}(\ell) \mid \forall[\Delta](\Gamma, \varsigma)$$

Type $\forall[\Delta](\Gamma, \varsigma)$ describes preconditions for code blocks. The location environment Δ is a sequence of location variables and stack type variables. The register file Γ is a partial function from registers to types. Γ and ς describe the initial register and stack state for the blocks. They may refer to the variables in Δ.

Values and Operands. A stack location d is either "base" or the next stack location "next(d)".

A word-sized value w may be an integer "i", the "nonsense" value for uninitialized stack slots, a heap location "p", a stack location "d", or instantiated values "$w[\ell]$" and "$w[\sigma]$" where w points to code blocks polymorphic over location variables and stack type variables. Contents of registers and stack slots are word-sized. As in STAL [13], word-sized values are separated from operands to prevent registers from containing registers.

$$\begin{array}{rrcl} \textbf{stack loc} & d & ::= & \text{base} \mid \text{next}(d) \\ \textbf{word value} & w & ::= & i \mid \text{nonsense} \mid p \mid d \mid w[\ell] \mid w[\sigma] \\ \textbf{operand} & o & ::= & r \mid w \mid o[\ell] \mid o[\sigma] \end{array}$$

An operand o may be a register "r", a word-sized value "w", or instantiated operands "$o[\ell]$" and "$o[\sigma]$". A special register sp is used for the stack pointer.

Instructions. Most instructions are standard. Values on the heap or stack are accessed through explicit load and store instructions.

$$\begin{array}{rrcl} \textbf{instr} & \text{ins} & ::= & \text{mov } r, o \mid \text{add } r, o \mid \text{sub } r, o \mid \text{ladd } r, i \mid \text{load } r_1, [r_2 + i] \\ & & & \mid \text{store } [r_1 + i], r_2 \mid \text{jumpif0 } r, o \mid \text{heapalloc } r = \langle o \rangle \mid (\eta, r) = \text{unpack}(o) \end{array}$$

SST uses "ladd" instructions for stack location arithmetic. The first operand points to a stack location. The second operand is a constant integer (positive or negative). A "ladd" instruction moves the stack pointer along the stack according to the integer value. The standard add and subtract instructions deal with only integer arithmetic.

The heap allocation instruction "heapalloc $r = \langle o \rangle$" allocates a word on the heap with initial value o and assigns the new heap location to r.

The unpack instruction "$(\eta, r) = \text{unpack}(o)$" coerces a heap pointer o to a heap location. It introduces a fresh location variable η for o and assigns η to r.

4.1 Type Checking Instructions

The type checker maintains a few environments. The location environment Δ and the register file Γ were explained previously. The heap environment Ψ is a partial function from heap locations to heap pointer types. A mapping "$p \mapsto \text{HeapPtr(int)}$" in Ψ means that the heap location p points to an integer on the heap. Complete semantics is shown in Appendix B.

Operand Typing Rules. The judgment $\Delta; \Psi; \Gamma \vdash o : \tau$ means that operand o has type τ under the environments. Note that a heap location can be typed in two ways: the type in the heap environment (o-p-H) or a singleton type (o-p). A stack location has a singleton type (o-d).

If an operand o has a polymorphic type $\forall[\Delta](\Gamma, \varsigma)$, $o[\ell]$ and $o[\sigma]$ instantiate the first variable in Δ with ℓ and σ respectively. The judgments $\Delta \vdash \ell$ and $\Delta \vdash \sigma$ mean that ℓ and σ are well-formed under Δ respectively.

$$\frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)}\ \text{o-reg} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash i : \text{int}}\ \text{o-int} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash \text{nonsense} : \text{Nonsense}}\ \text{o-ns}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash p : \Psi(p)}\ \text{o-p-H} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash p : \text{Ptr}(p)}\ \text{o-p} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash d : \text{Ptr}(d)}\ \text{o-d}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\eta, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \ell}{\Delta; \Psi; \Gamma \vdash o[\ell] : \forall[\Delta'](\Gamma'[\ell/\eta], \varsigma[\ell/\eta])}\ \text{o-inst-l} \qquad \frac{\Delta; \Psi; \Gamma \vdash o : \forall[\rho, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \sigma}{\Delta; \Psi; \Gamma \vdash o[\sigma] : \forall[\Delta'](\Gamma'[\sigma/\rho], \varsigma[\sigma/\rho])}\ \text{o-inst-Q}$$

The judgment $\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')$ means that assigning a value of type τ to register r results in new environments Γ' and ς'. Only Γ is changed if r is not sp. Otherwise the stack grows or shrinks according to the new value of sp.

$$\frac{r \neq \text{sp} \quad \Gamma' = \Gamma[r \mapsto \tau]}{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma)}\ \text{a-not-esp} \qquad \frac{\vdash \text{Resize}(\ell, \varsigma) = \varsigma' \quad \Gamma' = \Gamma[\text{sp} \mapsto \text{Ptr}(\ell)]}{\vdash (\Gamma, \varsigma)\{\text{sp} \leftarrow \text{Ptr}(\ell)\}(\Gamma', \varsigma')}\ \text{a-esp}$$

Stack Rules. *Resize.* When the stack grows or shrinks, SST uses the judgment $\vdash \text{Resize}(\ell, \varsigma) = \varsigma'$ to get the new stack type. The judgment means that resizing stack ς to location ℓ results in stack ς'. The location ℓ will be the top of ς'. The stack shrinks if ℓ is inside ς (s-shrink) and grows if ℓ is beyond the top of ς (s-grow). The stack drops all aliases beyond ℓ when shrinking to avoid dangling pointers.

$$\frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\vdash \text{Resize}(\ell, \varsigma) = \ell : \sigma}\ \text{s-shrink} \qquad \frac{\varsigma' = (\text{Nonsense}_n; \ldots; \text{Nonsense}_1)@(\ell : \sigma)}{\vdash \text{Resize}(\text{next}^n(\ell), \ell : \sigma) = \varsigma'}\ \text{s-grow}$$

Location Lookup. The judgment $\varsigma \vdash \ell + i = \ell'$ means that in stack ς going i slots from location ℓ leads to location ℓ'. A positive i means going toward the stack top and negative means toward the stack bottom. The notion n represents natural numbers. (The requirement $\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)$ ensures that ℓ is a stack location, not a heap location.)

$$\frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \ell + n = \text{next}^n(\ell)}\ \text{s-offset-next} \qquad \frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \text{next}^n(\ell) + (-n) = \ell}\ \text{s-offset-prev}$$

Type Lookup. The judgment $\varsigma \vdash \ell : \tau$ means that the location ℓ in stack ς has type τ. The location ℓ can be either an alias in ς, or be on the spine of ς (the stack type obtained by dropping all aliases from ς).

$$\frac{\varsigma \Rightarrow \ell' : (\sigma \wedge \{\ell : \tau\})}{\varsigma \vdash \ell : \tau}\ \text{s-lookup}$$

Stack Update. The judgment $\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma'$ means that updating the location ℓ in stack ς with type τ results in stack ς'. Weak updates do not change the stack type (s-update-weak). Strong updates change the type of ℓ and drop all aliases beyond ℓ because they may refer to the old type of ℓ (s-update-strong).

$$\frac{\varsigma \vdash \ell : \tau}{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma}\ \text{s-update-weak} \qquad \frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \tau :: \varsigma')}{\varsigma \vdash \ell \leftarrow \tau' \rightsquigarrow \overrightarrow{\tau}\ @(\ell : \tau' :: \varsigma')}\ \text{s-update-strong}$$

Instruction Typing Rules. Figure 3 lists instruction typing rules. $\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma')$ means that checking instruction "ins" changes the environments Γ and ς to new environments Γ' and ς'.

The location arithmetic instruction "ladd r, i" requires that r point to a location ℓ and i be a multiple of 4. The stack grows toward lower addresses. If i is negative, the result location is further outward from ℓ.

Loads and stores can operate on heap locations (i-load-p and i-store-p), stack locations on the spine (i-load-concat and i-store-concat), and aliases (i-load-aliased and i-store-aliased). SST supports weak updates on heap locations and aliases, and both strong and weak updates on stack locations on the spine.

The rule for heap allocation assigns a heap pointer type to the register that holds the pointer, instead of a singleton type, because the new heap location is statically unknown. The heap environment does not change after heap allocation because the rest of the program does not refer to the new heap location by name.

When control transfers, the type checker matches the current environments with those of the target. The location environment of the target should have been fully instantiated. $\Gamma \Rightarrow \Gamma'$ requires that Γ' be a subset of Γ.

$$\frac{\Delta;\Psi;\Gamma \vdash o : \tau \quad \vdash (\Gamma,\varsigma)\{r \leftarrow \tau\}(\Gamma',\varsigma')}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{mov } r, o\}(\Gamma';\varsigma')}\ \text{i-mov}$$

$$\frac{\begin{array}{c}\Gamma(r) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \vdash (\Gamma,\varsigma)\{r \leftarrow \text{Ptr}(\ell')\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{ladd } r, -4 * i\}(\Gamma';\varsigma')}\ \text{i-ladd}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{add } r, o\}(\Gamma;\varsigma)}\ \text{i-add}$$

$$\frac{\Delta;\Psi;\Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{sub } r, o\}(\Gamma;\varsigma)}\ \text{i-sub}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{HeapPtr}(\tau) \\ \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma';\varsigma')}\ \text{i-load-p}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \tau \\ \Gamma(r_1) = \text{HeapPtr}(\tau)\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma;\varsigma)}\ \text{i-store-p}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \varsigma \vdash \ell' : \tau \quad \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + (-4 * i)]\}(\Gamma';\varsigma')}\ \text{i-load-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \quad \Gamma(r_2) = \tau \\ \varsigma \vdash \ell + i = \ell' \quad \varsigma \vdash \ell' \leftarrow \tau \rightsquigarrow \varsigma'\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + (-4 * i)], r_2\}(\Gamma;\varsigma')}\ \text{i-store-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell : \tau \\ \vdash (\Gamma,\varsigma)\{r_1 \leftarrow \tau\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma';\varsigma')}\ \text{i-load-aliased}$$

$$\frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \\ \varsigma \vdash \ell : \tau \quad \Gamma(r_2) = \tau\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma;\varsigma)}\ \text{i-store-alised}$$

$$\frac{\begin{array}{c}\Delta;\Psi;\Gamma \vdash o : \tau \\ \vdash (\Gamma,\varsigma)\{r \leftarrow \text{HeapPtr}(\tau)\}(\Gamma',\varsigma')\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{heapalloc } r = \langle o \rangle\}(\Gamma';\varsigma')}\ \text{i-heapalloc}$$

$$\frac{\begin{array}{c}\Gamma(r) = \text{int} \quad \Delta;\Psi;\Gamma \vdash o : \forall[\,](\Gamma',\varsigma') \\ \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'\end{array}}{\Delta;\Psi \vdash (\Gamma;\varsigma)\{\text{jumpif0 } r, o\}(\Gamma;\varsigma)}\ \text{i-jump0}$$

Figure 3: Instruction Typing Rules

4.2 Blocks and Programs

A heap value v is either a code block "block" or a heap word "$\langle w \rangle$". A code block "$\forall[\Delta](\Gamma, \varsigma)\ b$" describes the precondition $\forall[\Delta](\Gamma, \varsigma)$ and its body b. The block body is a sequence of instructions that ends with a jump instruction. Only variables in Δ can appear free in Γ, ς, and the block body.

A program consists of a heap H, a register bank R, a stack s, and a block body as the entry point. H is a partial function from heap locations to heap values. R is a partial function from registers to word-sized values. The stack s records values on the spine. It is either the empty stack "empty" or a concatenation of a word-sized value with a stack "$w :: s$".

heap value	v	::=	block $\mid \langle w \rangle$
block	block	::=	$\forall[\Delta](\Gamma, \varsigma)\ b$
block body	b	::=	ins; $b \mid$ jump o
heap	H	::=	$p_1 \mapsto v_1, \ldots, p_n \mapsto v_n$
reg bank	R	::=	$r_1 \mapsto w_1, \ldots, r_n \mapsto w_n$
stack value	s	::=	empty $\mid w :: s$
program	P	::=	(H, R, s, b)

A program $P = (H, R, s, b)$ is well-formed (illustrated by the judgment $\vdash P$) if H matches a heap environment Ψ, R matches a register file Γ, s matches a stack type ς, and b is well-formed under Ψ, Γ, and ς. The notion "$\bullet$" means empty environments.

$$\frac{\vdash H : \Psi \quad \bullet; \Psi \vdash s : \varsigma \quad \bullet; \Psi \vdash R : \Gamma \quad \bullet; \Psi; \Gamma; \varsigma \vdash b}{\vdash (H, R, s, b)}\ \text{m-tp}$$

A heap H matches a heap environment Ψ if they have the same domain and each heap value in H has the corresponding type in Ψ (h-tp). Matching a register bank with a register file is defined similarly (g-tp).

$$\frac{\Psi = \{\ldots, p \mapsto \tau, \ldots\} \quad H = \{\ldots, p \mapsto v, \ldots\} \quad \ldots \quad \bullet; \Psi \vdash v : \tau \quad \ldots}{\vdash H : \Psi}\ \text{h-tp}$$

$$\frac{\Gamma = \{\ldots, r \mapsto \tau, \ldots\} \quad R = \{\ldots, r \mapsto w, \ldots\} \quad \ldots \quad \Delta; \Psi; \bullet \vdash w : \tau \quad \ldots}{\Delta; \Psi \vdash R : \Gamma}\ \text{g-tp}$$

A stack value s matches a stack type ς if all the locations on the spine have the corresponding type in ς (s-base and s-concat) and ς contains only aliased locations to heap pointers (s-alias) and to stack locations on the spine (s-imp).

$$\frac{}{\Delta; \Psi \vdash \text{empty} : (\text{base} : \text{Empty})}\ \text{s-base} \qquad \frac{\Delta; \Psi \vdash s : (\ell : \varsigma) \quad \Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash w :: s : (\text{next}(\ell) : \tau :: \ell : \sigma)}\ \text{s-concat}$$

$$\frac{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : \sigma)}{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : (\sigma \wedge \{p : \tau\}))}\ \text{s-alias} \qquad \frac{\Delta; \Psi \vdash s : \varsigma \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi \vdash s : \varsigma'}\ \text{s-imp}$$

To type check a block body, the checker checks the instructions in order (b-ins) until it reaches the jump instruction (b-jump).

The unpack instruction "$(\eta, r) = \text{unpack}(o)$" requires o have a heap pointer type (b-unpack). The rule introduces a fresh location variable η to Δ, assigns r a singleton type $\text{Ptr}(\eta)$, and updates the stack type to contain η.

$$\frac{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma') \quad \Delta; \Psi; \Gamma'; \varsigma' \vdash b}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{ins}; b}\ \text{b-ins} \qquad \frac{\Delta; \Psi; \Gamma \vdash o : \forall[\,](\Gamma', \varsigma') \quad \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{jump } o}\ \text{b-jump}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{HeapPtr}(\tau) \qquad r \neq \text{sp} \qquad \eta \notin \Delta \quad (\Delta; \eta); \Psi; \Gamma[r \mapsto \text{Ptr}(\eta)]; \ell : (\sigma \wedge \{\eta : \tau\}) \vdash b}{\Delta; \Psi; \Gamma; \ell : \sigma \vdash (\eta, r) = \text{unpack}(o)}\ \text{b-unpack}$$

A block is well-formed if under the heap environment and the specified precondition, the block body type-checks.

$$\frac{\Delta; \Psi; \Gamma; \varsigma \vdash b}{\Psi \vdash \forall[\Delta](\Gamma, \varsigma)\ b}\ \text{block-tp}$$

A code block has the specified precondition as its type, if the code block is well-formed and the precondition is well-formed (v-code). The heap-allocated word values have heap pointer types (v-hp).

$$\frac{\Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b \quad \Delta \vdash \forall[\Delta'](\Gamma', \varsigma')}{\Delta; \Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b : \forall[\Delta'](\Gamma', \varsigma')}\ \text{v-code} \qquad \frac{\Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash \langle w \rangle : \text{HeapPtr}(\tau)}\ \text{v-hp}$$

Evaluation The judgment $P \to P'$ means that program P evaluates to program P'. Appendix B.3 lists program evaluation rules. Evaluating a program in SST is mainly evaluating instructions in the "main" block. The heap, the register bank, and the stack might change during evaluation. When the control transfers, the body of the new code block is loaded as "main" and the evaluation continues.

The stack is represented as a sequence of word-sized values. No explicit labels are necessary for the stack locations because the label of a stack slot can be computed from the distance of the slot from the bottom of the stack.

Arithmetic on stack locations is defined as follows. $d + n$ computes the outward location n slots from d and $d - n$ computes the inward location n slots from d.

$$\begin{array}{lcl} d + 0 & = & d \\ d + (n + 1) & = & \text{next}(d) + n \\ \text{base} + (-(n + 1)) & = & \text{base} \\ \text{next}(d) + (-(n + 1)) & = & d + (-n) \end{array}$$

Stack may grow or shrink during execution of programs. The only way to grow or shrink the stack is by assigning new values to sp. We use a function "resize(d, s)" to represent growing or shrinking the stack s to location d. An auxiliary function size(s) gets the top location of s. If d is the top location of s, the stack is unchanged. If d is an outward location n slots from the top of the stack, the stack grows n slots. Otherwise, the stack throws away slots beyond d.

$$\begin{array}{lcl} \text{size}(\text{empty}) & = & \text{base} \\ \text{size}(w :: s) & = & \text{next}(\text{size}(s)) \\ \text{resize}(\text{size}(s), s) & = & s \\ \text{resize}(\text{size}(s) + (n + 1), s) & = & \text{nonsense} :: \text{resize}(\text{size}(s) + n, s) \\ \text{resize}(\text{size}(s) + (-(n + 1)), w :: s) & = & \text{resize}(\text{size}(s) + (-n), s) \end{array}$$

A function $s(d)$ returns the value at location d on s.

$$\frac{}{(w :: s)(\text{size}(w :: s)) = w}\ \text{s-lookup-top} \qquad \frac{s(d) = w}{(w' :: s)(d) = w}\ \text{s-lookup}$$

Assigning a new value to a stack slot changes the stack. The stack size does not change because each slot is word-sized. The function $s[d \leftarrow w]$ means the new stack with a new value w for the location d.

$$\frac{d = \text{size}(w :: s)}{w' :: s = (w :: s)[d \leftarrow w']}\ \text{s-assign-top} \qquad \frac{s' = s[d \leftarrow w]}{w' :: s' = (w' :: s)[d \leftarrow w]}\ \text{s-assign}$$

Operands are evaluated to word-sized values. The judgment $R \vdash o \mapsto w$ means that operand o evaluates to value w under the register bank R. Registers get their values from the register bank.

$$\frac{}{R \vdash r \mapsto R(r)}\ \text{eo-r} \qquad \frac{}{R \vdash w \mapsto w}\ \text{eo-w}$$

$$\frac{R \vdash o \mapsto w}{R \vdash o[\ell] \mapsto w[\ell]}\ \text{eo-inst-l} \qquad \frac{R \vdash o \mapsto w}{R \vdash o[\sigma] \mapsto w[\sigma]}\ \text{eo-inst-Q}$$

Assigning new values to registers may change the register bank and/or the stack. The judgment $(R, s)\{r \leftarrow w\}(R', s')$ means that assigning w to r results in new register bank R' and new stack s'. If r is not sp, only R is updated to reflect the new value of r. Otherwise, the stack needs to resize as well.

$$\frac{r \neq \text{sp} \quad R' = R[r \mapsto w]}{(R, s)\{r \leftarrow w\}(R', s)}\ \text{u-not-esp} \qquad \frac{R' = R[\text{sp} \mapsto d]}{(R, s)\{\text{sp} \leftarrow d\}(R', \text{resize}(d, s))}\ \text{u-esp}$$

The location add instruction ladd r, i deals with stack location arithmetic if r evaluates to a stack location. Each slot is 4-byte aligned. r is assigned an outward location if i is negative and an inward one if the integer is positive.

The load instruction load $r_1, [r_2{+}0]$ assigns r_1 with the value stored at heap location p if r_2 evaluates to p (e-load-p). The load instruction load $r_1, [r_2 + i]$ assigns r_1 with the value stored at stack location $d + i$ if r_2 evaluates to d (e-load-d). Similarly, the store instruction store $[r_1 + i], r_2$ changes the heap (e-store-p) or the stack (e-store-d) depending on whether r_1 evaluates to a heap location or a stack location. Loading from and storing to an alias fall into the above two cases because an alias is either a stack location on the spine of the stack or a heap location at run time.

The heap allocation instruction "heapalloc $r = \langle o \rangle$" expands the heap with a fresh heap location and assigns r with the new heap location.

The conditional jump instruction "jumpif0 r, o" falls through to the rest of the block if o evaluates to a non-zero value. Otherwise, it replaces the current block body with the target block body. The heap, the register bank, and the stack remain unchanged.

To evaluate the unpack instruction "$(\eta, r) = \text{unpack}(o)$", o must evaluate to a heap location. Then r is assigned the heap location and the rest of the block is evaluated with η replaced with the location.

We proved soundness (by standard progress and preservation theorems) and decidability of SST. The proofs can be found online [16].

Theorem 2 (Preservation) *If* $\vdash P$ *and* $P \rightarrow P'$*, then* $\vdash P'$*.*

Theorem 3 (Progress) *If* $\vdash P$*, then* $\exists P'$ *such that* $P \rightarrow P'$*.*

Theorem 4 (Decidability) *Given* Ψ *and block, there is an algorithm to decide whether "*$\Psi \vdash$ *block" holds.*

5 Source Language and Translation

As mentioned in Section 2, we translate JSWG's Micro-CLI [10] to SST. Micro-CLI supports both heap and stack allocation. A managed pointer can point to either a heap-allocated or a stack-allocated value. Managed pointers have the same constraints as those in CLI, such as they cannot be stored in objects nor returned from functions.

The syntax of Micro-CLI is restated here.

$$
\begin{array}{rlll}
\textbf{qualifiers} & q & ::= & S \mid H \\
\textbf{types} & \tau & ::= & \text{int} \mid \tau *_q \\
\\
\textbf{values} & v & ::= & n \mid x \\
\\
\textbf{program} & p & ::= & fds \;\; rb \\
\\
\textbf{function decls} & fds & ::= & \cdot \mid fd \;\; fds \\
\textbf{function decl} & fd & ::= & \tau \; f(\tau_1 \; x_1, \ldots, \tau_n \; x_n) \;\; rb \\
\\
\textbf{return block} & rb & ::= & \{lds; ss; \text{return } v\} \\
\\
\textbf{local decls} & lds & ::= & \cdot \mid ld; lds \\
\textbf{local decl} & ld & ::= & \tau \; x = v \mid \tau \; x = \text{new}_q \; v \\
\\
\textbf{statement list} & ss & ::= & \cdot \mid s; ss \\
\textbf{statement} & s & ::= & \text{if } v \text{ then } ss \text{ else } ss \mid x = v \mid x = v_1 + v_2 \mid x = v_1 - v_2 \\
 & & & \mid x = f(v_1, \ldots, v_n) \mid x = \; !v \mid v_1 := v_2
\end{array}
$$

Micro-CLI supports only the integer type and pointer types. Each pointer type is qualified by "S" (stack pointer) or "H" (heap pointer). Heap pointer types are subtypes of stack pointer types with the same referent types, that is, $\tau *_H$ is a subtype of $\tau *_S$.

A Micro-CLI program consists of a sequence of function declarations and a return block. A function declaration specifies the return type, the function name, the parameters, and the body (a return block). A return block contains a sequence of local variable declarations and a sequence of statements. A local variable declaration declares the type and the initial value of a local variable that can be used in subsequent declarations and statements.

Because SST deals with aliasing differently from JSWG, the two translations differ in rules around managed pointers which introduce aliasing. For example, if a source function has a parameter with type "pointer-to-pointer-to-int", the translation to SST creates two aliases for the pointers while the translation to JSWG uses existential types to abstract the locations and version numbers to relate the scopes. The precondition of the function in SST would have a stack type "$\text{next}(\eta) : \text{Ptr}(\eta_1) :: \eta : \{\rho \wedge \{\eta_1 : \text{Ptr}(\eta_2)\} \wedge \{\eta_2 : \text{int}\}\}$" where the function is polymorphic over η_1 and η_2.

We use the following example to show the result of translation. The "swap" function in Section 2 is rewritten into Micro-CLI syntax as follows:

$$
\begin{array}{l}
\text{int swap(int } *_S \;\; x, \text{int } *_S \;\; y)\{ \\
\quad \text{int } t = 0; \\
\quad \text{int } t' = 0; \\
\quad t = \; !x; \\
\quad t' = \; !y; \\
\quad x := \; t'; \\
\quad y := t; \\
\quad \text{return } 0; \\
\}
\end{array}
$$

Micro-CLI does not allow such syntax as "$x := \; !y$". A new variable "t'" holds the value of "$!y$" and is then assigned to x. Local variables can be initialized only by values. The local variables t and t' are initialized to 0 first and then assigned "$!x$" and "$!y$" respectively. Micro-CLI does not allow functions with no return values. The "swap" function simply returns an integer value.

The function is translated to the following SST function:

$$
\begin{array}{l}
\forall[\eta_x, \eta_y, \eta_0, \rho](\Gamma, \varsigma) \\
\quad \text{mov } r_{fp}, \text{sp} \\
\quad \text{mov } r_1, 0 \qquad\qquad ; r_1 = 0; \\
\quad \text{ladd sp}, -4 \\
\quad \text{store } [\text{sp} + 0], r_1 \qquad ; \text{push } r_1 \text{ (for } t') \\
\quad \text{mov } r_1, 0 \qquad\qquad ; r_1 = 0; \\
\quad \text{ladd sp}, -4 \\
\quad \text{store } [\text{sp} + 0], r_1 \qquad ; \text{push } r_1 \text{ (for } t) \\
\quad \text{load } r_1, [r_{fp} + 0] \qquad ; r_1 = x \\
\quad \text{load } r_1, [r_1 + 0] \qquad ; r_1 = [r_1] \\
\quad \text{store } [r_{fp} + (-8)], r_1 \quad ; t = r_1 \ (t = !x) \\
\quad \text{load } r_1, [r_{fp} + 4] \qquad ; r_1 = y \\
\quad \text{load } r_1, [r_1 + 0] \qquad ; r_1 = [r_1] \\
\quad \text{store } [r_{fp} + (-4)], r_1 \quad ; t' = r_1 \ (t' = !y) \\
\quad \text{load } r_1, [r_{fp} + 0] \qquad ; r_1 = x \\
\quad \text{load } r_2, [r_{fp} + (-4)] \quad ; r_2 = t' \\
\quad \text{store } [r_1 + 0], r_2 \qquad ; [r_1] = r_2 \ (x := t') \\
\quad \text{load } r_1, [r_{fp} + 4] \qquad ; r_1 = y \\
\quad \text{load } r_2, [r_{fp} + (-8)] \quad ; r_2 = t \\
\quad \text{store } [r_1 + 0], r_2 \qquad ; [r_1] = r_2 \ (y := t) \\
\quad \text{ladd sp}, 16 \qquad\qquad ; \text{pop } t, t', x, y \\
\quad \text{mov } r_1, 0 \qquad\qquad ; r_1 = 0 \\
\quad \text{ladd sp}, -4 \\
\quad \text{store } [\text{sp} + 0], r_1 \qquad ; \text{push } r_1 \\
\quad \text{jump } r_{ra} \qquad\qquad ; \text{jump } r_{ra} \\
\text{where } \Gamma = \text{sp} \mapsto \text{Ptr}(\text{next}^2(\eta_0)), \\
\qquad\qquad r_{ra} \mapsto \forall[\,](\text{sp} \mapsto \text{Ptr}(\text{next}(\eta_0)), \text{next}(\eta_0) : \text{int} :: \eta_0 : \rho) \\
\text{and } \varsigma = \text{next}^2(\eta_0) : \text{Ptr}(\eta_x) :: \text{next}(\eta_0) : \text{Ptr}(\eta_y) :: \\
\qquad\qquad \eta_0 : \{\rho \wedge \{\eta_x : \text{int}\} \wedge \{\eta_y : \text{int}\}\}
\end{array}
$$

The translation is straightforward. Many optimizations can be applied to improve the SST code, which is beyond the scope of this paper. The translation reserves register sp for the stack pointer, r_{fp} for the frame pointer, and r_{ra} for the return address. Two temporary registers r_1 and r_2 are used to hold intermediate values during the translation of a Micro-CLI instruction. Parameters and return values are passed through the stack. Local variables are allocated on the stack.

The SST function is polymorphic over four variables: η_x, η_y, η_0, and ρ. The first two represent the values of x and y. The third represents the location of the rest of the stack (abstracted by the stack type variable ρ). The parameters x and y are on the stack upon entry to the function. Section 3 explained the initial stack state. The parameters and the local variables are accessed through the frame pointer: t, t', x, and y have addresses $r_{fp} - 8$, $r_{fp} - 4$, r_{fp}, and $r_{fp} + 4$ respectively.

At the beginning of the function, the frame pointer r_{fp} is assigned sp and the initial values for t and t' are pushed onto the stack. At the end, the local variables and the parameters are popped from the stack, the return value is pushed onto the stack, and the control transfers to the return address, which is kept in register r_{ra}.

Details of the Translation. The translation rules use the following abbreviations. The map V keeps the mapping from local variables to their offsets on the stack from the frame pointer.

$$
\begin{array}{lcl}
\text{update } x & = & \text{store } [r_{fp} + (-4 * V(x))], r_1 \\
\text{push } r & = & \text{ladd sp}, -4; \text{store } [\text{sp} + 0], r \\
\text{pop } r & = & \text{load } r, [\text{sp} + 0]; \text{ladd sp}, 4
\end{array}
$$

Values are translated to SST instructions that load the values to temp registers.

$$\begin{array}{lcl} |\Gamma_s \vdash n : \text{int}|\ V\ r & = & \text{mov } r, n \\ |\Gamma_s \vdash x : \Gamma_s(x)|\ V\ r & = & \text{load } r, [r_{fp} + (-4 * V(x))] \end{array}$$

Local declarations are translated to instructions that allocate stack space and load values to the stack.

$$\frac{|\Gamma_s \vdash v : \tau|\ V\ r_1 = I}{|\Gamma_s \vdash \tau\ x = v : \Gamma_s[x : \tau]|\ V\ sz = (V[x \mapsto sz + 1], sz + 1, (I; \text{push } r_1))}\ \text{trans-ld-v}$$

$$\frac{\tau = \tau' *_S \quad |\Gamma_s \vdash v : \tau'|\ V\ r_1 = I \quad V' = V[x \mapsto sz + 2]}{|\Gamma_s \vdash \tau\ x = \text{new}_S\ v : \Gamma_s[x : \tau]|\ V\ sz = (V', sz + 2, (I; \text{push } r_1; \text{mov } r_2, \text{sp}; \text{push } r_2))}\ \text{trans-ld-s}$$

$$\frac{\tau = \tau' *_H \quad |\Gamma_s \vdash v : \tau'|\ V\ r_1 = I \quad V' = V[x \mapsto sz + 1]}{|\Gamma_s \vdash \tau\ x = \text{new}_H\ v : \Gamma_s[x : \tau]|\ V\ sz = (V', sz + 1, (I; \text{heapalloc } r_1 = \langle r_1 \rangle; \text{push } r_1))}\ \text{trans-ld-h}$$

A source language type environment keeps type environment for parameters, local variables, and function names. When translated to SST, parameters and local variables are described by stack types and function names are described by heap environments. A register bank keeps the types of the reserved registers (sp, r_{fp}, and r_{ra}).

Suppose a function f has a declaration $\tau_0\ f(x_1 : \tau_1, \ldots, x_n : \tau_n)\ rb$. Let $\tau_{ra} = \forall[\,](\text{sp} \mapsto \text{Ptr}(\text{next}(\eta)), \text{next}(\eta) : |\tau_0| :: \eta :: \rho)$

$$\frac{}{|\bullet, \_, \_, \_|\ \eta\ \rho = (\{\eta, \rho\}, \bullet, \{\text{sp} \mapsto \text{Ptr}(\eta)\}, \eta : \rho)}$$

$$\frac{|\tau_i, \varsigma, \Delta| = (\tau', \varsigma', \Delta') \qquad |\Gamma_s, V, 0, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma) \qquad \Gamma(\text{sp}) = \text{Ptr}(\ell)}{|(\Gamma_s, x_i : \tau_i), V, 0, f|\ \eta\ \rho = (\Delta', \Psi, \Gamma[\text{sp} \mapsto \text{Ptr}(\text{next}(\ell)), r_{ra} \mapsto \tau_{ra}], \text{next}(\ell) : \tau' :: \varsigma')}$$

$$\frac{\tau \neq \tau' *_S \quad |\Gamma_s, V, sz, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma) \quad \Gamma(\text{sp}) = \text{Ptr}(\ell)}{|(\Gamma_s, x : \tau), V, sz + 1, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma[\text{sp} \mapsto \text{Ptr}(\text{next}(\ell)), r_{fp} \mapsto \text{Ptr}(\eta), r_{ra} \mapsto \tau_{ra}], \text{next}(\ell) : |\tau| :: \varsigma)}$$

$$\frac{\begin{array}{c} f \text{ has local declaration } x = new_S\ v \\ |\Gamma_s, V, sz, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma) \qquad \Gamma(\text{sp}) = \text{Ptr}(\ell) \end{array}}{\begin{array}{l} |(\Gamma_s, x : \tau\ *_S), V, sz + 2, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma[\text{sp} \mapsto \text{Ptr}(\text{next}^2(\ell)), r_{fp} \mapsto \text{Ptr}(\eta), r_{ra} \mapsto \tau_{ra}], \\ \qquad \text{next}^2(\ell) : \text{Ptr}(\text{next}(\ell)) :: \text{next}(\ell) : |\tau, \Gamma, \varsigma, V, v| :: \varsigma) \end{array}}$$

$$\frac{|\Gamma_s, V, sz, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma)}{|(\Gamma_s, f : (\tau_1, \ldots, \tau_n) \to \tau), V, sz, f|\ \eta\ \rho = (\Delta, (\Psi, p_f \mapsto |(\tau_1, \ldots, \tau_n) \to \tau|), \Gamma, \varsigma)}$$

$$\frac{\tau \neq \tau' *_S}{|\tau, \varsigma, \Delta| = (|\tau|, \varsigma, \Delta)} \qquad \frac{|\tau, \varsigma, \Delta| = (\tau', \varsigma', \Delta') \quad \eta \text{ is a fresh location variable}}{|\tau *_S, \varsigma, \Delta| = (\text{Ptr}(\eta), \varsigma' \wedge \{\eta : \tau'\}, (\eta; \Delta'))}$$

Types are translated as follows. Stack pointer types are translated to singleton types. The translation needs to know which variable has the stack pointer type to get the offset of the variable from the frame pointer. When $\tau \neq \tau' *_S$, we use $|\tau|$ as a short cut for $|\tau, \_, \_, \_, \_|$.

$$\begin{array}{lcl} |\text{int}| & = & \text{int} \\ |\tau\ *_H| & = & \text{HeapPtr}(|\tau|) \\ |\tau\ *_S, \Gamma, \varsigma, V, x| & = & \tau' \\ & & \text{where } \Gamma(r_{fp}) = \text{Ptr}(\ell) \quad S \vdash \ell + V(x) : \tau' \\ |(\tau_1, \ldots, \tau_n) \to \tau| & = & \forall[\Delta](\Gamma, \varsigma) \\ & & \quad \text{where } |\{x_1 : \tau_1, \ldots, x_n : \tau_n\}, V, 0, f|\ \eta\ \rho = (\Delta, \_, \Gamma, \varsigma) \\ & & \quad \text{and } V = x_1 \mapsto 0, x_2 \mapsto -1, \ldots, x_n \mapsto -n + 1 \end{array}$$

Arguments are translated to instructions that push the arguments onto the stack. If an argument is a pointer, the translation collects locations that will be used to instantiate the aliases associated with the pointer parameter in the callee.

$$\frac{|\Gamma_s \vdash v : \tau|\ V\ r_1 = I \quad \tau \neq \tau' *_S}{|\Gamma_s \vdash v : \tau|\ V\ r_1\ \ell\ \ell_0 = ((I; \text{push } r_1), \text{next}(\ell))}$$

$$\frac{|\Gamma_s \vdash x : \Gamma_s(x)|\ V\ r_1 = I \quad |\Gamma_s(x)|\ (\tau' *_S)\ x\ V\ \ell_0\ f\ r_1 = (I', sub)}{|\Gamma_s \vdash x : \tau' *_S \mid V\ r_1\ \ell\ \ell_0 = ((I; I'; \text{push } r_1), (\text{next}(\ell); sub))}$$

$$\frac{\tau \neq \tau' *_S}{|\tau|\ \tau\ \_\ V\ \ell_0\ f\ r = (\emptyset, \emptyset)} \qquad \frac{|\tau|\ \tau'\ \_\ V\ \ell_0\ f\ r = (I', sub)}{|\tau\ *_H \mid (\tau' *_S)\ \_\ V\ \ell_0\ f\ r = (((r, \eta) = \text{unpack}(r); I'), (\eta; sub))}$$

$$\frac{|\tau|\ \tau'\ y\ V\ \ell_0\ f\ r = (I', sub) \quad f \text{ has local declaration } x = new_S\ v}{|\tau\ *_S \mid (\tau' *_S)\ x\ V\ \ell_0\ f\ r = (I', (\ell_0 + (V(x) - 1);\ sub))}$$

Translation of instructions adds into the current code block SST instructions that perform the operation. It ends the current code block and starts new ones at control transfer points.

$$\frac{\Gamma_s \vdash x : \tau \quad |\Gamma_s \vdash v : \tau|\ V\ r_1 = I_v \quad \tau \neq \tau' *_S}{|\Gamma_s \vdash x = v|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_0, \Psi_1, cb, (I; I_v; \text{update } x))}\ \text{trans-mov}$$

$$\frac{\Gamma_s \vdash x : \text{int} \quad |\Gamma_s \vdash v_1 : \text{int}|\ V\ r_1 = I_1 \quad |\Gamma_s \vdash v_2 : \text{int}|\ V\ r_2 = I_2}{|\Gamma_s \vdash x = v_1 + v_2|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_0, \Psi_1, cb, (I; I_1; I_2; \text{add } r_1, r_2; \text{update } x))}\ \text{trans-add}$$

$$\frac{\Gamma_s \vdash x : \text{int} \quad |\Gamma_s \vdash v_1 : \text{int}|\ V\ r_1 = I_1; \quad |\Gamma_s \vdash v_2 : V|\ \text{int}\ r_2 = I_2}{|\Gamma_s \vdash x = v_1 - v_2|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_0, \Psi_1, cb, (I; I_1; I_2; \text{sub } r_1, r_2; \text{update } x))}\ \text{trans-sub}$$

$$\frac{\Gamma_s \vdash x : \tau \quad |\Gamma_s \vdash v : \tau\ *_q \mid V\ r_1 = I_v \quad \tau \neq \tau' *_S}{|\Gamma_s \vdash x = !v|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_0, \Psi_1, cb, (I; I_v; \text{load } r_1, [r_1 + 0]; \text{update } x))}\ \text{trans-deref}$$

$$\frac{|\Gamma_s \vdash v_1 : \tau\ *_q \mid V\ r_1 = I_1 \quad |\Gamma_s \vdash v_2 : \tau|\ V\ r_2 = I_2 \quad \tau \neq \tau' *_S}{|\Gamma_s \vdash v_1 := v_2|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_0, \Psi_1, cb, (I; I_1; I_2; \text{store } [r_1 + 0], r_2))}\ \text{trans-assign}$$

$$\frac{\begin{array}{l}|\Gamma_s \vdash v : \text{int}|\ V\ r_1 = I_v \qquad |\Gamma_s, V, sz, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma) \\ |\Gamma_s \vdash sts_1|\ V\ sz\ f\ C_0\ (\Psi_1, cb_t \mapsto \forall[\Delta](\Gamma, \varsigma))\ cb_t\ \emptyset = (C^t, \Psi^t, cb^t, I^t) \\ |\Gamma_s \vdash sts_2|\ V\ sz\ f\ C_0\ (\Psi_1, cb_f \mapsto \forall[\Delta](\Gamma, \varsigma))\ cb_f\ \emptyset = (C^f, \Psi^f, cb^f, I^f)\end{array}}{|\Gamma_s \vdash \text{if } v \text{ then } sts_1 \text{ else } sts_2|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_1, \Psi_2, cb_{cont}, \emptyset)}\ \text{trans-if}$$

$$\text{where} \quad \begin{array}{l} C_1 = (C^t \cup C^f), cb^t \mapsto \forall[\Delta](\Gamma, \varsigma)(I^t; \text{jump } cb_{cont}[\Delta]), cb^f \mapsto \forall[\Delta](\Gamma, \varsigma)(I^f; \text{jump } cb_{cont}[\Delta]), \\ \qquad\qquad cb \mapsto \forall[\Delta_1](\Gamma_1, \varsigma_1)(I; I_v; \text{jumpif0 } r_1, cb_f[\Delta]; \text{jump } cb_t[\Delta]) \\ \Psi_2 = (\Psi^t \cup \Psi^f), cb_{cont} \mapsto \forall[\Delta](\Gamma, \varsigma) \end{array}$$

$$\frac{\begin{array}{l}\Gamma_s(f') = (\tau s) \to \tau \quad \tau s = \tau_1, \ldots, \tau_n \qquad \Gamma_s \vdash x : \tau \quad \tau \neq \tau'' *_S \\ |\Gamma_s, V, sz, f|\ \eta\ \rho = (\Delta, \Psi, \Gamma, \varsigma) \qquad \Gamma(\text{sp}) = \text{Ptr}(\ell) \\ |\Gamma_s \vdash vs : \tau s|\ V\ r_1\ \ell\ (\ell - sz) = (Is, sub)\ \forall\ 1 \leq i \leq n \qquad \Psi_1(cb) = \forall[\Delta_1](\Gamma_1, \varsigma_1)\end{array}}{|\Gamma_s \vdash x = f'(vs)|\ V\ sz\ f\ C_0\ \Psi_1\ cb\ I = (C_1, \Psi_2, cb_{cont}, (\text{pop } r_1; \text{update } x; \text{pop } r_{ra}; \text{pop } r_{fp}))}\ \text{trans-call}$$

$$\text{where} \quad \begin{array}{l} C_1 = C_0, cb \mapsto \forall[\Delta_1](\Gamma_1, \varsigma_1)(I; \text{push } r_{fp}; \text{push } r_{ra}; Is; \\ \qquad\qquad\qquad\qquad \text{mov } r_{ra}, cb_{cont}[\Delta]; \text{jump } p_{f'}[sub, \text{next}^2(\ell), \sigma]) \\ \sigma = \Gamma(r_{ra}) :: \text{next}(\ell) : \Gamma(r_{fp}) :: \varsigma \\ \Psi_2 = \Psi_1, cb_{cont} \mapsto \forall[\Delta](\Gamma[\text{sp} \mapsto \text{Ptr}(\text{next}^3(\ell))], \text{next}^3(\ell) : |\tau| :: \text{next}^2(\ell) : \sigma) \end{array}$$

Translation of basic blocks consists of translating local declarations and instructions. At the end of the block, arguments are popped from the stack and the return value is pushed.

$$\frac{\begin{array}{l}\Gamma_s(f) = (\tau_1, \ldots, \tau_n) \to \tau \quad |\Gamma_s \vdash lds : \Gamma'_s|\ V\ 0 = (V', sz', I_{ld}) \\ |\Gamma'_s \vdash sts|\ V'\ sz'\ f\ C_0\ \Psi_1\ p_f\ (\text{mov } r_{fp}, \text{sp}; I_{ld}) = (C'_0, \Psi_2, cb', I') \\ |\Gamma'_s \vdash v : \tau|\ V'\ r_1 = I_v \quad \Psi_2(cb') = \forall[\Delta_2](\Gamma_2, \varsigma_2)\end{array}}{|\Gamma_s \vdash^{\tau} lds; sts; \text{return } v|\ f\ C_0\ \Psi_1 = ((C'_0, cb' \mapsto b'), \Psi_2)}\ \text{trans-b}$$

where $b' = \forall[\Delta_2](\Gamma_2, \varsigma_2)\{I'; I_v; \text{ladd sp}, -4 * (sz' + n); \text{push } r_1; \text{jump } r_{ra}\}$,
and f has declaration $\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ rb$
and $V = \{x_1 \mapsto 0, x_2 \mapsto -1, \ldots, x_n \mapsto -n + 1\}$

Translation of function declaration is mainly translating the basic block with parameters in the source type environment.

$$\frac{\begin{array}{c}\Gamma'_s = \Gamma_s[f : (\tau_1, \ldots, \tau_n) \to \tau, x_1 : \tau_1, \ldots, x_n : \tau_n] \\ |\Gamma'_s \vdash^{\tau} rb|\ f\ C\ \Psi[p_f \mapsto |(\tau_1, \ldots, \tau_n) \to \tau|] = (C', \Psi')\end{array}}{|\Gamma_s \vdash \tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ rb : \Gamma_s[f : (\tau_1, \ldots, \tau_n) \to \tau]|\ C\ \Psi = (C', \Psi')}\ \text{trans-f}$$

Translation of programs uses a "halt" block as the return address of the main body. The "halt" block expects the return value from the main body.

$$\frac{\begin{array}{c}|\bullet \vdash fds : \Gamma_s|\ \{p_{halt} \mapsto \text{block}_{halt}\}\ \{p_{halt} \mapsto \tau_{halt}\} = (C, \Psi) \\ |\Gamma_s \vdash^{\tau} rb|\ main\ C\ (\Psi, p_{main} \mapsto \tau_{main}) = (C', \Psi')\end{array}}{|\vdash (fd_1, \ldots, fd_n, rb)| = (C', \Psi', (\text{mov } r_{ra}, p_{halt}; \text{jump } p_{main}[\text{base}, \text{empty}]))}\ \text{trans-p}$$

where $\tau_{halt} = \forall[\,](\text{sp} \mapsto \text{Ptr}(\text{next}(\text{base})), \text{next}(\text{base}) : \text{int} :: \text{base} : \text{Empty})$
and $\tau_{main} = \forall[\eta, \rho](\{\text{sp} \mapsto \text{Ptr}(\eta),$
$r_{ra} \mapsto \forall[\,](\text{sp} \mapsto \text{Ptr}(\text{next}(\eta)), \text{next}(\eta) : \text{int} :: \eta : \rho)\}, \eta : \rho)$

We proved the type-preservation theorem of the translation:

Theorem 5 (Type-preserving Translation) *Well-typed Micro-CLI programs translate to well-typed SST programs.*

6 Conclusions

With a simple stack type ς, SST safely supports many low-level idioms: stack pointers, frame pointers, by-value arguments, and by-reference arguments, where by-reference arguments may point to both stack data and heap data.

This paper presented one particular type system built around the stack type ς, but many variations are possible. For example, we treated the stack pointer register as a special register to safely accomodate kernel-mode code in the presence of interrupts, but some other settings could treat the stack pointer as an ordinary register. For GC safety, we allowed pointer arithmetic on stack pointers but disallowed pointer arithmetic on heap pointers. For simplicity, we assumed infinite stack space to grow in, but a type checker based on SST could also verify stack overflow checks (perhaps in cooperation with virtual-memory-based overflow checks). Also for simplicity, our heap consisted of one-word objects, but this extends naturally to objects with multiple fields. Finally, to ensure simple, efficient type checking, we used a small, restricted linear logic, but we could trade efficiency for expressiveness by varying the linear logic, without abandoning the basic SST approach.

References

[1] Amal Ahmed and David Walker. The logical approach to stack typing. In *2003 ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003.

[2] Karl Crary. Toward a foundational typed assembly language. In *Symposium on Principles of Programming Languages*, 2003.

[3] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.

[4] ECMA. *Standard ECMA-335 Common Language Infrastructure (CLI)*. 2006.

[5] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *15th European Symposium on Programming (ESOP'06)*, 2006.

[6] Jean-Yves Girard. Linear logic. In *Theoretical Computer Science*, 1987.

[7] Chris Hawblitzel. Linear types for aliased resources (extended version). Technical Report MSR-TR-2005-141, Microsoft Research, 2005.

[8] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[9] Limin Jia, Frances Spalding [Perry], David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 407–416, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Limin Jia, Frances Spalding [Perry], David Walker, and Neal Glew. Certifying compilation for a language with stack allocation. Technical Report TR-724-05, Princeton University, 2005.

[11] Patrick Lincoln, John C. Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. *Ann. Pure Appl. Logic*, 56(1-3):239–311, 1992.

[12] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[13] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 13(5):957–959, 2003.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 21, pages 527–568. ACM Press, 1999.

[15] George Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[16] Frances Perry, Chris Hawblitzel, and Juan Chen. Proofs for SST, 2007. http://research.microsoft.com/users/juanchen/stack.

[17] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, 2002.

[18] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.

[19] P. L. Wadler. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY, 1993. Springer-Verlag.

[20] David Walker. Mechanical reasoning about low-level programs. lecture notes, http://www.cs.cmu.edu/~dpw/papers.html, 2001.

A SST Syntax

location	ℓ	::=	$\eta \mid \text{base} \mid \text{next}(\ell) \mid p$
labeled stack type	ς	::=	$\ell : \sigma$
unlabeled stack type	σ	::=	$\rho \mid \text{Empty} \mid \tau :: \varsigma \mid \sigma \wedge \{\ell : \tau\}$
type	τ	::=	$\text{int} \mid \text{Nonsense} \mid \text{Ptr}(\ell) \mid \text{HeapPtr}(\tau) \mid \forall[\Delta](\Gamma, \varsigma)$
stack loc	d	::=	$\text{base} \mid \text{next}(d)$
word value	w	::=	$i \mid \text{nonsense} \mid p \mid d \mid w[\ell] \mid w[\sigma]$
operand	o	::=	$r \mid w \mid o[\ell] \mid o[\sigma]$
instr	ins	::=	$\text{mov } r, o \mid \text{add } r, o \mid \text{sub } r, o \mid \text{ladd } r, i$
			$\mid \text{load } r_1, [r_2 + i] \mid \text{store } [r_1 + i], r_2$
			$\mid \text{jumpif0 } r, o \mid \text{heapalloc } r = \langle o \rangle$
			$\mid (\eta, r) = \text{unpack}(o)$
heap value	v	::=	$\text{block} \mid \langle w \rangle$
block	block	::=	$\forall[\Delta](\Gamma, \varsigma)\ b$
block body	b	::=	$\text{ins}; b \mid \text{jump } o$
loc env	Δ	::=	$\bullet \mid \eta; \Delta \mid \rho; \Delta$
heap	H	::=	$p_1 \mapsto v_1, \ldots, p_n \mapsto v_n$
heap env	Ψ	::=	$p_1 \mapsto \tau_1, \ldots, p_n \mapsto \tau_n$
reg bank	R	::=	$r_1 \mapsto w_1, \ldots, r_n \mapsto w_n$
reg file	Γ	::=	$r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n$
stack value	s	::=	$\text{empty} \mid w :: s$
program	P	::=	(H, R, s, b)

We use the following abbreviation:

$$(\tau_n \ldots \tau_1)@(\ell : \sigma) = \text{next}^n(\ell) : \tau_n :: \ldots :: \text{next}^1(\ell) : \tau_1 :: \ell : \sigma$$

B SST Semantics

B.1 Well-formedness

$\boxed{\Delta \vdash \ell}$

$$\frac{}{\{\ldots, \eta, \ldots\} \vdash \eta}\ \text{wf-l-var} \qquad \frac{}{\Delta \vdash \text{base}}\ \text{wf-l-base}$$

$$\frac{\Delta \vdash \ell}{\Delta \vdash \text{next}(\ell)}\ \text{wf-l-next} \qquad \frac{}{\Delta \vdash p}\ \text{wf-l-p}$$

$\boxed{\Delta \vdash \varsigma}$

$$\frac{\Delta \vdash \ell}{\Delta \vdash \ell : \text{Empty}}\ \text{wf-S-empty} \qquad \frac{\Delta \vdash \ell \quad \rho \in \Delta}{\Delta \vdash \ell : \rho}\ \text{wf-S-P}$$

$$\frac{\begin{array}{c}\Delta \vdash \ell \qquad \Delta \vdash \tau \qquad \Delta \vdash \ell_q : \sigma \\ \forall\ \ell'_q, \tau', \sigma' : \tau = \tau' \text{ if } \ell_q : \sigma \Rightarrow \ell'_q : (\sigma' \wedge \{\ell : \tau'\})\end{array}}{\Delta \vdash \ell_q : (\sigma \wedge \{\ell : \tau\})}\ \text{wf-S-alias}$$

$$\frac{\begin{array}{c}\Delta \vdash \ell \qquad \Delta \vdash \tau \qquad \Delta \vdash \varsigma \\ \forall\ \ell'_q, \ell', \tau', \sigma' : \ell \neq \ell' \text{ if } \varsigma \Rightarrow \ell'_q : (\sigma' \wedge \{\ell' : \tau'\})\end{array}}{\Delta \vdash \ell : (\tau :: \varsigma)}\ \text{wf-S-concat}$$

$\boxed{\Delta \vdash \tau}$

$$\frac{}{\Delta \vdash \text{int}}\ \text{wf-t-int} \qquad \frac{}{\Delta \vdash \text{Nonsense}}\ \text{wf-t-ns} \qquad \frac{\Delta \vdash \tau}{\Delta \vdash \text{HeapPtr}(\tau)}\ \text{wf-t-hp}$$

$$\frac{\Delta \vdash \ell}{\Delta \vdash \text{Ptr}(\ell)}\ \text{wf-t-single} \qquad \frac{\begin{array}{c}\Delta, \Delta' \ \vdash\ \Gamma' \quad \Delta, \Delta' \vdash \varsigma' \\ \Delta \cap \Delta' = \{\}\end{array}}{\Delta \vdash \forall[\Delta'](\Gamma', \varsigma')}\ \text{wf-t-code}$$

$\boxed{\Delta \vdash \Gamma}$

$$\frac{\ldots\ \Delta \vdash \tau\ \ldots}{\Delta \vdash \{\ldots, r \mapsto \tau, \ldots\}}\ \text{wf-G}$$

B.2 Static Semantics

$\boxed{\Delta; \Psi; \Gamma \vdash o : \tau}$

$$\frac{}{\Delta; \Psi; \Gamma \vdash r : \Gamma(r)}\ \text{o-reg} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash i : \text{int}}\ \text{o-int} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash \text{nonsense} : \text{Nonsense}}\ \text{o-ns}$$

$$\frac{}{\Delta; \Psi; \Gamma \vdash p : \Psi(p)}\ \text{o-p-H} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash p : \text{Ptr}(p)}\ \text{o-p} \qquad \frac{}{\Delta; \Psi; \Gamma \vdash d : \text{Ptr}(d)}\ \text{o-d}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \forall[\eta, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \ell}{\Delta; \Psi; \Gamma \vdash o[\ell] : \forall[\Delta'](\Gamma'[\ell/\eta], \varsigma[\ell/\eta])}\ \text{o-inst-l} \qquad \frac{\Delta; \Psi; \Gamma \vdash o : \forall[\rho, \Delta'](\Gamma', \varsigma) \quad \Delta \vdash \sigma}{\Delta; \Psi; \Gamma \vdash o[\sigma] : \forall[\Delta'](\Gamma'[\sigma/\rho], \varsigma[\sigma/\rho])}\ \text{o-inst-Q}$$

$\boxed{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')}$

$$\frac{r \neq \text{sp} \quad \Gamma' = \Gamma[r \mapsto \tau]}{\vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma)}\ \text{a-not-esp} \qquad \frac{\vdash \text{Resize}(\ell, \varsigma) = \varsigma' \quad \Gamma' = \Gamma[\text{sp} \mapsto \text{Ptr}(\ell)]}{\vdash (\Gamma, \varsigma)\{\text{sp} \leftarrow \text{Ptr}(\ell)\}(\Gamma', \varsigma')}\ \text{a-esp}$$

$\boxed{\vdash \text{Resize}(\ell, \varsigma) = \varsigma'}$

$$\frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\vdash \text{Resize}(\ell, \varsigma) = \ell : \sigma}\ \text{s-shrink} \qquad \frac{\varsigma' = (\text{Nonsense}_n; \ldots; \text{Nonsense}_1)@(\ell : \sigma)}{\vdash \text{Resize}(\text{next}^n(\ell), \ell : \sigma) = \varsigma'}\ \text{s-grow}$$

$\boxed{\varsigma \vdash \ell + i = \ell'}$

$$\frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \ell + n = \text{next}^n(\ell)}\ \text{s-offset-next} \qquad \frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \sigma)}{\varsigma \vdash \text{next}^n(\ell) + (-n) = \ell}\ \text{s-offset-prev}$$

$\boxed{\varsigma \vdash \ell : \tau}$

$$\frac{\varsigma \Rightarrow \ell' : (\sigma \wedge \{\ell : \tau\})}{\varsigma \vdash \ell : \tau}\ \text{s-lookup}$$

$\boxed{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma'}$

$$\frac{\varsigma \vdash \ell : \tau}{\varsigma \vdash \ell \leftarrow \tau \rightsquigarrow \varsigma}\ \text{s-update-weak} \qquad \frac{\varsigma \Rightarrow \overrightarrow{\tau}\ @(\ell : \tau :: \varsigma')}{\varsigma \vdash \ell \leftarrow \tau' \rightsquigarrow \overrightarrow{\tau}\ @(\ell : \tau' :: \varsigma')}\ \text{s-update-strong}$$

$\boxed{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma')}$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \tau \quad \vdash (\Gamma, \varsigma)\{r \leftarrow \tau\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{mov } r, o\}(\Gamma'; \varsigma')}\ \text{i-mov}$$

$$\frac{\Gamma(r) = \text{Ptr}(\ell)\ \varsigma \vdash \ell + i = \ell' \ \vdash (\Gamma, \varsigma)\{r \leftarrow \text{Ptr}(\ell')\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ladd } r, -4 * i\}(\Gamma'; \varsigma')}\ \text{i-ladd}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{add } r, o\}(\Gamma; \varsigma)}\ \text{i-add}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{int} \quad r \neq \text{sp} \quad \Gamma(r) = \text{int}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{sub } r, o\}(\Gamma; \varsigma)}\ \text{i-sub}$$

$$\frac{\Gamma(r_2) = \text{HeapPtr}(\tau) \quad \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma'; \varsigma')}\ \text{i-load-p}$$

$$\frac{\Gamma(r_2) = \tau \quad \Gamma(r_1) = \text{HeapPtr}(\tau)}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma; \varsigma)}\ \text{i-store-p}$$

$$\frac{\begin{array}{c}\Gamma(r_2) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell + i = \ell' \\ \varsigma \vdash \ell' : \tau \quad \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load } r_1, [r_2 + (-4 * i)]\}(\Gamma'; \varsigma')}\ \text{i-load-concat}$$

$$\frac{\begin{array}{c}\Gamma(r_1) = \text{Ptr}(\ell) \quad \Gamma(r_2) = \tau \\ \varsigma \vdash \ell + i = \ell' \quad \varsigma \vdash \ell' \leftarrow \tau \rightsquigarrow \varsigma'\end{array}}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store } [r_1 + (-4 * i)], r_2\}(\Gamma; \varsigma')}\ \text{i-store-concat}$$

$$\frac{\Gamma(r_2) = \text{Ptr}(\ell)\ \varsigma \vdash \ell : \tau \ \vdash (\Gamma, \varsigma)\{r_1 \leftarrow \tau\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{load } r_1, [r_2 + 0]\}(\Gamma'; \varsigma')}\ \text{i-load-aliased}$$

$$\frac{\Gamma(r_1) = \text{Ptr}(\ell) \quad \varsigma \vdash \ell : \tau \quad \Gamma(r_2) = \tau}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{store } [r_1 + 0], r_2\}(\Gamma; \varsigma)}\ \text{i-store-aliased}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \tau \ \vdash (\Gamma, \varsigma)\{r \leftarrow \text{HeapPtr}(\tau)\}(\Gamma', \varsigma')}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{heapalloc } r = \langle o \rangle\}(\Gamma'; \varsigma')}\ \text{i-heapalloc}$$

$$\frac{\Gamma(r) = \text{int}\ \Delta; \Psi; \Gamma \vdash o : \forall[\,](\Gamma', \varsigma')\ \Gamma \Rightarrow \Gamma'\ \varsigma \Rightarrow \varsigma'}{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{jumpif0 } r, o\}(\Gamma; \varsigma)}\ \text{i-jump0}$$

$\boxed{\Gamma \Rightarrow \Gamma'}$

$$\frac{\Gamma' \subseteq \Gamma}{\Gamma \Rightarrow \Gamma'}\ \text{G-imp}$$

$\boxed{\varsigma \Rightarrow \varsigma'}$

$$\frac{\varsigma \Rightarrow \varsigma'}{\ell : \tau :: \varsigma \Rightarrow \ell : \tau :: \varsigma'}\ \text{s-imp-concat} \qquad \frac{\ell : \sigma \Rightarrow \ell : \sigma'}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : (\sigma' \wedge \{\ell_t : \tau\})}\ \text{s-imp-alias}$$

$$\frac{}{\varsigma \Rightarrow \varsigma}\ \text{s-imp-eq} \qquad \frac{}{\ell : (\tau :: \varsigma) \Rightarrow \ell : (\tau :: \varsigma \wedge \{\ell : \tau\})}\ \text{s-imp-add-alias}$$

$$\frac{\varsigma_1 \Rightarrow \varsigma_2 \quad \varsigma_2 \Rightarrow \varsigma_3}{\varsigma_1 \Rightarrow \varsigma_3}\ \text{s-imp-trans} \qquad \frac{}{\ell : (\sigma \wedge \{\ell_t : \tau\}) \Rightarrow \ell : \sigma}\ \text{s-imp-drop-alias}$$

$$\frac{}{\ell : (\tau_1 :: \ell_q : (\sigma \wedge \{\ell_2 : \tau_2\})) \Rightarrow \ell : ((\tau_1 :: \ell_q : \sigma) \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-expand-alias}$$

$$\frac{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\}) \quad \varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_2 : \tau_2\})}{\varsigma \Rightarrow \ell : (\sigma \wedge \{\ell_1 : \tau_1\} \wedge \{\ell_2 : \tau_2\})}\ \text{s-imp-merge-alias}$$

Figure 4: Stack Implication Rules

$\boxed{\vdash P}$

$$\frac{\vdash H : \Psi \quad \bullet; \Psi \vdash s : \varsigma \quad \bullet; \Psi \vdash R : \Gamma \quad \bullet; \Psi; \Gamma; \varsigma \vdash b}{\vdash (H, R, s, b)}\ \text{m-tp}$$

$\boxed{\vdash H : \Psi}$

$$\frac{\Psi = \{\ldots, p \mapsto \tau, \ldots\}\ H = \{\ldots, p \mapsto v, \ldots\}\ \ldots\ \ \bullet; \Psi \vdash v : \tau\ \ \ldots}{\vdash H : \Psi}\ \text{h-tp}$$

$\boxed{\Delta; \Psi \vdash R : \Gamma}$

$$\frac{\Gamma = \{\ldots, r \mapsto \tau, \ldots\}\ R = \{\ldots, r \mapsto w, \ldots\}\ \ldots\ \ \Delta; \Psi; \bullet \vdash w : \tau\ \ \ldots}{\Delta; \Psi \vdash R : \Gamma}\ \text{g-tp}$$

$\boxed{\Delta; \Psi \vdash s : \varsigma}$

$$\frac{}{\Delta; \Psi \vdash \text{empty} : (\text{base} : \text{Empty})}\ \text{s-base} \qquad \frac{\Delta; \Psi \vdash s : (\ell : \sigma) \quad \Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash w :: s : (\text{next}(\ell) : \tau :: \ell : \sigma)}\ \text{s-concat}$$

$$\frac{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : \sigma)}{\Delta; \Psi, \{p \mapsto \text{HeapPtr}(\tau)\} \vdash s : (\ell : (\sigma \wedge \{p : \tau\}))}\ \text{s-alias} \qquad \frac{\Delta; \Psi \vdash s : \varsigma \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi \vdash s : \varsigma'}\ \text{s-imp}$$

$\boxed{\Delta; \Psi; \Gamma; \varsigma \vdash b}$

$$\frac{\Delta; \Psi \vdash (\Gamma; \varsigma)\{\text{ins}\}(\Gamma'; \varsigma') \quad \Delta; \Psi; \Gamma'; \varsigma' \vdash b}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{ins}; b}\ \text{b-ins} \qquad \frac{\Delta; \Psi; \Gamma \vdash o : \forall[\,](\Gamma', \varsigma') \quad \Gamma \Rightarrow \Gamma' \quad \varsigma \Rightarrow \varsigma'}{\Delta; \Psi; \Gamma; \varsigma \vdash \text{jump } o}\ \text{b-jump}$$

$$\frac{\Delta; \Psi; \Gamma \vdash o : \text{HeapPtr}(\tau) \qquad r \neq \text{sp} \qquad \eta \notin \Delta \quad (\Delta; \eta); \Psi; \Gamma[r \mapsto \text{Ptr}(\eta)]; \ell : (\sigma \wedge \{\eta : \tau\}) \vdash b}{\Delta; \Psi; \Gamma; \ell : \sigma \vdash (\eta, r) = \text{unpack}(o)}\ \text{b-unpack}$$

$\boxed{\Psi \vdash \text{block}}$

$$\frac{\Delta; \Psi; \Gamma; \varsigma \vdash b}{\Psi \vdash \forall[\Delta](\Gamma, \varsigma)\ b}\ \text{block-tp}$$

$\boxed{\Delta; \Psi \vdash v : \tau}$

$$\frac{\Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b \quad \Delta \vdash \forall[\Delta'](\Gamma', \varsigma')}{\Delta; \Psi \vdash \forall[\Delta'](\Gamma', \varsigma')\ b : \forall[\Delta'](\Gamma', \varsigma')}\ \text{v-code} \qquad \frac{\Delta; \Psi; \bullet \vdash w : \tau}{\Delta; \Psi \vdash \langle w \rangle : \text{HeapPtr}(\tau)}\ \text{v-hp}$$

B.3 Dynamic Semantics

$\boxed{d + i = d'}$

$$\begin{array}{lcl} d + 0 & = & d \\ d + (n+1) & = & \text{next}(d) + n \\ \text{base} + (-(n+1)) & = & \text{base} \\ \text{next}(d) + (-(n+1)) & = & d + (-n) \end{array}$$

$\boxed{\text{size}(s) = d}$

$$\begin{array}{lcl} \text{size}(\text{empty}) & = & \text{base} \\ \text{size}(w :: s) & = & \text{next}(\text{size}(s)) \end{array}$$

$\boxed{\text{resize}(d, s) = s'}$

$$\begin{array}{l} \text{resize}(\text{size}(s), s) \ = \ s \\ \text{resize}(\text{size}(s) + (n+1), s) \ = \ \text{nonsense} :: \text{resize}(\text{size}(s) + n, s) \\ \text{resize}(\text{size}(s) + (-(n+1)), w :: s) \ = \ \text{resize}(\text{size}(s) + (-n), s) \end{array}$$

$\boxed{s(d) = w}$

$$\frac{}{(w :: s)(\text{size}(w :: s)) = w}\ \text{s-lookup-top} \qquad \frac{s(d) = w}{(w' :: s)(d) = w}\ \text{s-lookup}$$

$\boxed{s' = s[d \leftarrow w]}$

$$\frac{d = \text{size}(w :: s)}{w' :: s = (w :: s)[d \leftarrow w']}\ \text{s-assign-top} \qquad \frac{s' = s[d \leftarrow w]}{w' :: s' = (w' :: s)[d \leftarrow w]}\ \text{s-assign}$$

$\boxed{R \vdash o \mapsto w}$

$$\frac{}{R \vdash r \mapsto R(r)}\ \text{eo-r} \qquad \frac{}{R \vdash w \mapsto w}\ \text{eo-w}$$

$$\frac{R \vdash o \mapsto w}{R \vdash o[\ell] \mapsto w[\ell]}\ \text{eo-inst-l} \qquad \frac{R \vdash o \mapsto w}{R \vdash o[\sigma] \mapsto w[\sigma]}\ \text{eo-inst-Q}$$

$\boxed{(R, s)\{r \leftarrow w\}(R', s')}$

$$\frac{r \neq \text{sp} \quad R' = R[r \mapsto w]}{(R, s)\{r \leftarrow w\}(R', s)}\ \text{u-not-esp} \qquad \frac{R' = R[\text{sp} \mapsto d]}{(R, s)\{\text{sp} \leftarrow d\}(R', \text{resize}(d, s))}\ \text{u-esp}$$

$\boxed{P \to P'}$

$$\frac{R \vdash o \mapsto w \quad (R,s)\{r \leftarrow w\}(R',s')}{(H,R,s,(\text{mov } r,o;\ b)) \to (H,R',s',b)}\ \text{e-mov}$$

$$\frac{R \vdash r \mapsto d \quad (R,s)\{r \leftarrow d+i\}(R',s')}{(H,R,s,(\text{ladd } r,-4*i;\ b)) \to (H,R',s',b)}\ \text{e-ladd}$$

$$\frac{R \vdash r \mapsto i_1 \quad R \vdash o \mapsto i_2 \quad (R,s)\{r \leftarrow i_1+i_2\}(R',s')}{(H,R,s,(\text{add } r,o;\ b)) \to (H,R',s',b)}\ \text{e-add}$$

$$\frac{R \vdash r \mapsto i_1 \quad R \vdash o \mapsto i_2 \quad (R,s)\{r \leftarrow i_1-i_2\}(R',s')}{(H,R,s,(\text{sub } r,o;\ b)) \to (H,R',s',b)}\ \text{e-sub}$$

$$\frac{R \vdash r_2 \mapsto p \quad H(p) = \langle w \rangle \quad (R,s)\{r_1 \leftarrow w\}(R',s')}{(H,R,s,(\text{load } r_1,[r_2+0];b)) \to (H,R',s',b)}\ \text{e-load-p}$$

$$\frac{R \vdash r_2 \mapsto d \quad s(d+i) = w \quad (R,s)\{r_1 \leftarrow w\}(R',s')}{(H,R,s,(\text{load } r_1,[r_2+(-4*i)];b)) \to (H,R',s',b)}\ \text{e-load-d}$$

$$\frac{R \vdash r_1 \mapsto p \quad H(p) = \langle w \rangle \quad R \vdash r_2 \mapsto w'}{(H,R,s,(\text{store } [r_1+0],r_2;b)) \to (H[p \leftarrow \langle w' \rangle],R,s,b)}\ \text{e-store-p}$$

$$\frac{R \vdash r_1 \mapsto d \quad R \vdash r_2 \mapsto w \quad s' = s[d+i \leftarrow w]}{(H,R,s,(\text{store } [r_1+(-4*i)],r_2;b)) \to (H,R,s',b)}\ \text{e-store-d}$$

$$\frac{R \vdash o \mapsto w \quad p \notin \text{domain}(H) \quad H' = H, p \mapsto \langle w \rangle \quad (R,s)\{r \leftarrow p\}(R',s')}{(H,R,s,(\text{heapalloc } r = \langle o \rangle;b)) \to (H',R',s',b)}\ \text{e-heapalloc}$$

$$\frac{R \vdash r \mapsto i \quad i \neq 0}{(H,R,s,(\text{jumpif0 } r,o;b)) \to (H,R,s,b)}\ \text{e-jump0-false}$$

$$\frac{R \vdash r \mapsto 0 \quad R \vdash o \mapsto p[\text{subst}] \quad H(p) = \forall[\Delta](\Gamma,\varsigma)\ b_2}{(H,R,s,(\text{jumpif0 } r,o;b_1)) \to (H,R,s,b_2[\text{subst}/\Delta])}\ \text{e-jump0-true}$$

$$\frac{R \vdash o \mapsto p \quad (R,s)\{r \leftarrow p\}(R',s')}{(H,R,s,((\eta,r) = \text{unpack}(o);b)) \to (H,R',s',b[p/\eta])}\ \text{e-unpack}$$

$$\frac{R \vdash o \mapsto p[\text{subst}] \quad H(p) = \forall[\Delta](\Gamma,\varsigma)\ b}{(H,R,s,\text{jump } o) \to (H,R,s,b[\text{subst}/\Delta])}\ \text{e-jump}$$