Stack-Based Typed Assembly Language [⋆]

Greg Morrisett, Karl Crary, Neal Glew, and David Walker

Cornell University

Abstract. In previous work, we presented a *Typed Assembly Language* (TAL). TAL is sufficiently expressive to serve as a target language for compilers of high-level languages such as ML. This work assumed such a compiler would perform a *continuation-passing style* transform and eliminate the control stack by heap-allocating activation records. However, most compilers are based on stack allocation. This paper presents STAL, an extension of TAL with stack constructs and stack types to support the stack allocation style. We show that STAL is sufficiently expressive to support languages such as Java, Pascal, and ML; constructs such as exceptions and displays; and optimizations such as tail call elimination and callee-saves registers. This paper also formalizes the typing connection between CPS-based compilation and stack-based compilation and illustrates how STAL can formally model calling conventions by specifying them as formal translations of source function types to STAL types.

1 Introduction and Motivation

Statically typed source languages have efficiency and software engineering advantages over their dynamically typed counterparts. Modern type-directed compilers [19, 25, 7, 32, 20, 28, 12] exploit the properties of typed languages more extensively than their predecessors by preserving type information computed in the front end through a series of typed intermediate languages. These compilers use types to direct sophisticated transformations such as closure conversion [18, 31, 17, 4, 21], region inference [8], subsumption elimination [9, 11], and unboxing [19, 24, 29]. Without types these transformations are, in many cases, less effective or impossible. Furthermore, the type translation partially specifies the corresponding term translation and often captures the critical concerns in an elegant and succinct fashion. Strong type systems not only describe but also enforce many important invariants. Consequently, developers of type-based compilers may invoke a type-checker after each code transformation, and if the output fails to type-check, the developer knows that the compiler contains an internal error. Although type-checkers for decidable type systems will not catch all compiler errors, they have proven themselves valuable debugging tools in practice [22].

[⋆] This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-96-1-0317, ARPA/AF grant F30602-95-1-0047, and AASERT grant N00014-95-1-0985. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

Despite the numerous advantages of compiling with types, until recently, no compiler propagated type information through the final stages of code generation. The TIL/ML compiler, for instance, preserves types through approximately 80% of compilation but leaves the remaining 20% untyped. Many of the complex tasks of code generation including register allocation and instruction scheduling are left unchecked; types are not used to specify or explain these low-level code transformations.

These observations motivated our exploration of very low-level type systems and corresponding compiler technology. In Morrisett et al. [23], we presented a *typed assembly language* (TAL) and proved that its type system was sound with respect to an operational semantics. We demonstrated the expressiveness of this type system by sketching a type-preserving compiler from an ML-like language to TAL. The compiler ensured that well-typed source programs were always mapped to well-typed assembly language programs and that they preserved source level abstractions such as user-defined abstract data types and closures. Furthermore, we claimed that the type system of TAL did not interfere with many traditional compiler optimizations including inlining, loop-unrolling, register allocation, instruction selection, and instruction scheduling.

However, the compiler we presented was critically based on a *continuation-passing style* (CPS) transform, which eliminated the need for a control stack. In particular, activation records were represented by heap-allocated closures as in the SML of New Jersey compiler (SML/NJ) [5, 3]. For example, Figure 1 shows the TAL code our heap-based compiler would produce for the recursive factorial computation. Each function takes an additional argument which represents the control stack as a continuation closure. Instead of "returning" to the caller, a function invokes its continuation closure by jumping directly to the code of the closure, passing the environment of the closure and the result in registers.

Allocating continuation closures on the heap has many advantages over a conventional stack-based implementation. First, it is straightforward to implement control primitives such as exceptions, first-class continuations, or user-level lightweight coroutine threads [3, 31, 34]. Second, Appel and Shao [2] have shown that heap allocation of closures can have better space properties, primarily because it is easier to share environments. Third, there is a unified memory management mechanism (namely the garbage collector) for allocating and collecting all kinds of objects, including activation frames. Finally, Appel and Shao [2] have argued that, at least for SML/NJ, the locality lost by heap-allocating activation frames is negligible.

Nevertheless, there are also compelling reasons for providing support for stacks. First, Appel and Shao's work did not consider imperative languages, such as Java, where the ability to share environments is greatly reduced, nor did it consider languages that do not require garbage collection. Second, Tarditi and Diwan [14, 13] have shown that with some cache architectures, heap allocation of continuations (as in SML/NJ) can have substantial overhead due to a loss of locality. Third, stack-based activation records can have a smaller memory footprint than heap-based activation records. Finally, many machine architectures have hardware mechanisms that expect programs to behave in a stack-like fashion. For example, the Pentium Pro processor has an internal stack that it uses to predict return addresses for procedures so that instruction pre-fetching will not be stalled [16]. The internal stack is guided by the use of call/return primitives which use the standard control stack.

Clearly, compiler writers must weigh a complex set of factors before choosing stack allocation, heap allocation, or both. The target language must not constrain these design decisions. In this paper, we explore the addition of a stack to our typed assembly language in order to give compiler writers the flexibility they need. Our stack typing discipline is remarkably simple, but powerful enough to compile languages such as Pascal, Java, or ML without adding high-level primitives to the assembly language. More specifically, the typing discipline supports stack allocation of temporary variables and values that do not escape, stack allocation of procedure activation frames, exception handlers, and displays, as well as optimizations such as callee-saves registers. Unlike the JVM architecture [20], our system does not constrain the stack to have the same size at each control-flow point, nor does it require new high-level primitives for procedure call/return. Instead, our assembly language continues to have low-level RISC-like primitives such as loads, stores, and jumps. However, source-level stack allocation, general source-level stack pointers, general pointers into either the stack or heap, and some advanced optimizations cannot be typed.

A key contribution of the type structure is that it provides a unifying declarative framework for specifying procedure calling conventions regardless of the allocation strategy. In addition, the framework further elucidates the connection between a heap-based continuation-passing style compiler, and a conventional stack-based compiler. In particular, this type structure makes explicit the notion that the only differences between the two styles are that, instead of passing the continuation as a boxed, heap-allocated tuple, a stack-based compiler passes the continuation unboxed in registers and the environments for continuations are allocated on the stack. The general framework makes it easy to transfer transformations developed for one style to the other. For instance, we can easily explain the callee-saves registers of SML/NJ [5, 3, 1] and the callee-saves registers of a stack-based compiler as instances of a more general CPS transformation that is independent of the continuation representation.

2 Overview of TAL and CPS-Based Compilation

In this section, we briefly review our original proposal for typed assembly language (TAL) and sketch how a polymorphic functional language, such as ML, can be compiled to TAL in a continuation-passing style, where continuations are heap-allocated.

Figure 2 gives the syntax for TAL. Programs (P) are triples consisting of a heap, register file, and instruction sequence. Heaps map labels to heap values which are either tuples of word-sized values or code sequences. Register files map registers to word-sized values. Instruction sequences are lists of instructions

```
(H,{},I) where
H = l_fact:
        code[]{r1:⟨⟩,r2:int,r3:τ_k}.
          bneq r2,l_nonzero
          unpack [α,r3],r3               % zero branch: call k (in r3) with 1
          ld r4,r3(0)                    % project k code
          ld r1,r3(1)                    % project k environment
          mov r2,1
          jmp r4                         % jump to k
    l_nonzero:
        code[]{r1:⟨⟩,r2:int,r3:τ_k}.
          sub r4,r2,1                    % n − 1
          malloc r5[int,τ_k]             % create environment for cont in r5
          st r5(0),r2                    % store n into environment
          st r5(1),r3                    % store k into environment
          malloc r3 [∀[].{r1:⟨int^1,τ_k^1⟩,r2:int},⟨int^1,τ_k^1⟩]  % create cont closure
          mov r2,l_cont
          st r3(0),r2                    % store cont code
          st r3(1),r5                    % store environment ⟨n,k⟩
          mov r2,r4                      % arg := n − 1
          mov r3,pack [⟨int^1,τ_k^1⟩,r3] as τ_k % abstract environment type
          jmp l_fact                     % recursive call
    l_cont:
        code[]{r1:⟨int^1,τ_k^1⟩,r2:int}.   % r2 contains (n − 1)!
          ld r3,r1(0)                    % retrieve n
          ld r4,r1(1)                    % retrieve k
          mul r2,r3,r2                   % n × (n − 1)!
          unpack [α,r4],r4               % unpack k
          ld r3,r4(0)                    % project k code
          ld r1,r4(1)                    % project k environment
          jmp r3                         % jump to k
    l_halt:
        code[]{r1:⟨⟩,r2:int}.
          mov r1,r2
          halt[int]                      % halt with result in r1

and I =   malloc r1[]                    % create empty environment (⟨⟩)
          malloc r2[]                    % create empty environment
          malloc r3[∀[].{r1:⟨⟩,r2:int},⟨⟩] % create halt closure in r3
          mov r4,l_halt
          st r3(0),r4                    % store cont code
          st r3(1),r2                    % store environment ⟨⟩
          mov r2,6                       % load argument (6)
          mov r3,pack [⟨⟩,r3] as τ_k      % abstract environment type
          jmp l_fact                     % begin fact with
                                         % {r1 = ⟨⟩, r2 = 6, r3 = haltcont}
and τ_k = ∃α.⟨∀[].{r1:α,r2:int}^1, α^1⟩
```

Fig. 1. Typed Assembly Code for Factorial

types	$\tau ::= \alpha \mid \mathit{int} \mid \forall[\Delta].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau$
initialization flags	$\varphi ::= 0 \mid 1$
label assignments	$\Psi ::= \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
type assignments	$\Delta ::= \cdot \mid \alpha, \Delta$
register assignments	$\Gamma ::= \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
registers	$r ::= \mathtt{r1} \mid \cdots \mid \mathtt{r}k$
word values	$w ::= \ell \mid i \mid ?\tau \mid w[\tau] \mid \mathit{pack}\ [\tau, w]\ \mathit{as}\ \tau'$
small values	$v ::= r \mid w \mid v[\tau] \mid \mathit{pack}\ [\tau, v]\ \mathit{as}\ \tau'$
heap values	$h ::= \langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\Delta]\Gamma.I$
heaps	$H ::= \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	$R ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	$\iota ::= \mathit{aop}\ r_d, r_s, v \mid \mathit{bop}\ r, v \mid \mathtt{ld}\ r_d, r_s(i) \mid \mathtt{malloc}\ r[\vec{\tau}] \mid \mathtt{mov}\ r_d, v \mid \mathtt{st}\ r_d(i), r_s \mid \mathtt{unpack}\ [\alpha, r_d], v \mid$
arithmetic ops	$\mathit{aop} ::= \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul}$
branch ops	$\mathit{bop} ::= \mathtt{beq} \mid \mathtt{bneq} \mid \mathtt{bgt} \mid \mathtt{blt} \mid \mathtt{bgte} \mid \mathtt{blte}$
instruction sequences	$I ::= \iota; I \mid \mathtt{jmp}\ v \mid \mathtt{halt}\ [\tau]$
programs	$P ::= (H, R, I)$

Fig. 2. Syntax of TAL

terminated by either a `jmp` or `halt` instruction. The context Δ binds the free type variables of Γ in $\forall[\Delta].\Gamma$, and of both Γ and I in $\mathtt{code}[\Delta]\Gamma.I$. The instruction `unpack` $[\alpha, r], v$ binds α in the following instructions. We consider syntactic objects to be equivalent up to alpha-conversion, and consider label assignments, register assignments, heaps, and register files equivalent up to reordering of labels and registers. Register names do not alpha-convert. The notation $\vec{X}$ denotes a sequence of zero or more Xs, and $|\cdot|$ denotes the length of a sequence.

The instruction set consists mostly of conventional RISC-style assembly operations, including arithmetic, branches, loads, and stores. One exception, the `unpack` instruction, strips the quantifier from the type of an existentially typed value and introduces a new type variable into scope. On an untyped machine, this is implemented by an ordinary move. The other non-standard instruction is `malloc`, which is explained below. Evaluation is specified as a deterministic rewriting system that takes programs to programs (see Morrisett et al. [23] for details).

The types for TAL consist of type variables, integers, tuple types, existential types, and polymorphic code types. Tuple types contain initialization flags (either 0 or 1) that indicate whether or not components have been initialized. For example, if register r has type $\langle \mathit{int}^0, \mathit{int}^1 \rangle$, then it contains a label bound in the heap to a pair that can contain integers, where the first component may not have been initialized, but the second component has. In this context, the type system allows the second component to be loaded, but not the first. If an integer value is stored into $r(0)$ then afterwards r has the type $\langle \mathit{int}^1, \mathit{int}^1 \rangle$, reflecting the fact that the first component is now initialized. The instruction `malloc` $r[\tau_1, \ldots, \tau_n]$ heap-allocates a new tuple with uninitialized fields and places its label in register r.

TAL code types ($\forall[\alpha_1, \ldots, \alpha_n].\Gamma$) describe code blocks ($\texttt{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I$), which are instruction sequences, that expect a register file of type Γ and in which the type variables $\alpha_1, \ldots, \alpha_n$ are held abstract. In other words, Γ serves as a register file pre-condition that must hold before control may be transferred to the code block. Code blocks have no post-condition because control is either terminated via a `halt` instruction or transferred to another code block.

The type variables that are abstracted in a code block provide a means to write polymorphic code sequences. For example, the polymorphic code block

$$\begin{array}{l}
\texttt{code}[\alpha]\{\texttt{r1}{:}\alpha, \texttt{r2}{:}\forall[].\{\texttt{r1}{:}\langle\alpha^1, \alpha^1\rangle\}\}. \\
\quad \begin{array}{ll}
\texttt{malloc} & \texttt{r3}[\alpha, \alpha] \\
\texttt{st} & \texttt{r3}(0), \texttt{r1} \\
\texttt{st} & \texttt{r3}(1), \texttt{r1} \\
\texttt{mov} & \texttt{r1}, \texttt{r3} \\
\texttt{jmp} & \texttt{r2}
\end{array}
\end{array}$$

roughly corresponds to a CPS version of the ML function $\texttt{fn}\,(\texttt{x}{:}\alpha) \,\texttt{=>}\, (\texttt{x}, \texttt{x})$. The block expects upon entry that register `r1` contains a value of the abstract type α, and `r2` contains a return address (or continuation label) of type $\forall[].\{\texttt{r1} : \langle\alpha^1, \alpha^1\rangle\}$. In other words, the return address requires register `r1` to contain an initialized pair of values of type α before control can be returned to this address. The instructions of the code block allocate a tuple, store into the tuple two copies of the value in `r1`, move the pointer to the tuple into `r1` and then jump to the return address in order to "return" the tuple to the caller. If the code block is bound to a label ℓ, then it may be invoked by simultaneously instantiating the type variable and jumping to the label (*e.g.*, `jmp` $\ell[\mathit{int}]$).

Source languages like ML have nested higher-order functions that might contain free variables and thus require *closures* to represent functions. At the TAL level, we represent closures as a pair consisting of a code block label and a pointer to an environment data structure. The type of the environment must be held abstract in order to avoid typing difficulties [21], and thus we *pack* the type of the environment and the pair to form an existential type.

All functions, including continuation functions introduced during CPS conversion, are thus represented as existentials. For example, once CPS converted, a source function of type $\mathit{int} \to \langle\rangle$ has type $(\mathit{int}, (\langle\rangle \to \mathit{void})) \to \mathit{void}$.[1] After closures are introduced, the code will have type:

$$\exists\alpha_1.\langle(\alpha_1, \mathit{int}, \exists\alpha_2.\langle(\alpha_2, \langle\rangle) \to \mathit{void}, \alpha_2\rangle) \to \mathit{void}, \alpha_1\rangle$$

Finally, at the TAL level the function will be represented by a value with the type:

$$\exists\alpha_1.\langle\forall[].\{\texttt{r1}{:}\alpha_1, \texttt{r2}{:}\mathit{int}, \texttt{r3}{:}\exists\alpha_2.\langle\forall[].\{\texttt{r1}{:}\alpha_2, \texttt{r2}{:}\langle\rangle\}^1, \alpha_2^1\rangle\}^1, \alpha_1^1\rangle$$

---

[1] The *void* return types are intended to suggest the non-returning aspect of CPS code.

Here, α_1 is the abstracted type of the closure's environment. The code for the closure requires that the environment be passed in register `r1`, the integer argument in `r2`, and the continuation in `r3`. The continuation is itself a closure where α_2 is the abstracted type of its environment. The code for the continuation closure requires that the environment be passed in `r1` and the unit result of the computation in `r2`.

To apply a closure at the TAL level, we first use the `unpack` operation to open the existential package. Then the code and the environment of the closure pair are loaded into appropriate registers, along with the argument to the function. Finally, we use a jump instruction to transfer control to the closure's code.

Figure 1 gives the CPS-based TAL code for the following ML expression which computes six factorial:

```
let fun fact n = if n = 0 then 1 else n * (fact(n - 1)) in
  fact 6
end
```

3 Adding Stacks to TAL

In this section, we show how to extend TAL to achieve a Stack-based Typed Assembly Language (STAL). Figure 3 defines the new syntactic constructs for the language. In what follows, we informally discuss the dynamic and static semantics for the modified language, leaving formal treatment to Appendix A.

types	$\tau ::= \cdots \mid ns$
stack types	$\sigma ::= \rho \mid nil \mid \tau{::}\sigma$
type assignments	$\Delta ::= \cdots \mid \rho, \Delta$
register assignments	$\Gamma ::= \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n, \mathtt{sp}{:}\sigma\}$
word values	$w ::= \cdots \mid w[\sigma] \mid ns$
small values	$v ::= \cdots \mid v[\sigma]$
register files	$R ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathtt{sp} \mapsto S\}$
stacks	$S ::= nil \mid w{::}S$
instructions	$\iota ::= \cdots \mid \mathtt{salloc}\ n \mid \mathtt{sfree}\ n \mid \mathtt{sld}\ r_d, \mathtt{sp}(i) \mid \mathtt{sst}\ \mathtt{sp}(i), r_s$

Fig. 3. Additions to TAL for Simple Stacks

Operationally, we model stacks (S) as lists of word-sized values. Uninitialized stack slots are filled with nonsense (ns). Register files now include a distinguished register, `sp`, which represents the current stack. There are four new instructions that manipulate the stack. The `salloc` n instruction places n words of nonsense on the top of the stack. In a conventional machine, assuming stacks grow towards lower addresses, an `salloc` instruction would correspond to subtracting n from the current value of the stack pointer. The `sfree` n instruction removes the top n words from the stack, and corresponds to adding n to the current stack pointer. The $\texttt{sld}\ r, \texttt{sp}(i)$ instruction loads the i^{th} word of the stack into register r, whereas the $\texttt{sst}\ \texttt{sp}(i), r$ stores register r into the i^{th} word. Note, the instructions `ld` and `st` cannot be used with the stack pointer.

A program becomes *stuck* if it attempts to execute:

- $\texttt{sfree}\ n$ and the stack does not contain at least n words,
- $\texttt{sld}\ r, \texttt{sp}(i)$ and the stack does not contain at least $i+1$ words or else the i^{th} word of the stack is ns, or
- $\texttt{sst}\ \texttt{sp}(i), r$ and the stack does not contain at least $i+1$ words.

As in the original TAL, the typing rules for the modified language prevent well-formed programs from becoming stuck.

Stacks are described by *stack types* (σ), which include nil and $\tau{::}\sigma$. The latter represents a stack of the form $w{::}S$ where w has type τ and S has type σ. Stack slots filled with nonsense have type ns. Stack types also include stack type variables (ρ) which may be used to abstract the tail of a stack type. The ability to abstract stacks is critical for supporting procedure calls and is discussed in detail later.

As before, the register file for the abstract machine is described by a register file type (Γ) mapping registers to types. However, Γ also maps the distinguished register `sp` to a stack type σ. Finally, code blocks and code types support polymorphic abstraction over both types and stack types.

One of the uses of the stack is to save temporary values during a computation. The general problem is to save on the stack n registers, say r_1 through r_n, of types τ_1 through τ_n, perform some computation e, and then restore the temporary values to their respective registers. This would be accomplished by the following instruction sequence where the comments (delimited by %) show the stack's type at each step of the computation.

$$\begin{array}{lll}
 & & \%\ \sigma \\
\texttt{salloc} & n & \%\ ns{::}ns{::}\cdots{::}ns{::}\sigma \\
\texttt{sst} & \texttt{sp}(0), r_1 & \%\ \tau_1{::}ns{::}\cdots{::}ns{::}\sigma \\
\vdots & & \\
\texttt{sst} & \texttt{sp}(n-1), r_n & \%\ \tau_1{::}\tau_2{::}\cdots{::}\tau_n{::}\sigma \\
\text{code for } e & & \%\ \tau_1{::}\tau_2{::}\cdots{::}\tau_n{::}\sigma \\
\texttt{sld} & r_1, \texttt{sp}(0) & \%\ \tau_1{::}\tau_2{::}\cdots{::}\tau_n{::}\sigma \\
\vdots & & \\
\texttt{sld} & r_n, \texttt{sp}(n-1) & \%\ \tau_1{::}\tau_2{::}\cdots{::}\tau_n{::}\sigma \\
\texttt{sfree} & n & \%\ \sigma
\end{array}$$

If, upon entry, r_i has type τ_i and the stack is described by σ, and if the code for e leaves the state of the stack unchanged, then this code sequence is well-typed. Furthermore, the typing discipline does not place constraints on the order in which the stores or loads are performed.

It is straightforward to model higher-level primitives, such as `push` and `pop`. The former can be seen as simply $\texttt{salloc}\ 1$ followed by a store to $\texttt{sp}(0)$, whereas the latter is a load from $\mathtt{sp}(0)$ followed by `sfree 1`. Also, a “jump-and-link” or “call” instruction which automatically moves the return address into a register or onto the stack can be synthesized from our primitives. To simplify the presentation, we did not include these instructions in STAL; a practical implementation, however, would need a full set of instructions appropriate to the architecture.

The stack is commonly used to save the current return address, and temporary values across procedure calls. Which registers to save and in what order is usually specified by a compiler-specific calling convention. Here we consider a simple calling convention where it is assumed there is one integer argument and one unit result, both of which are passed in register `r1`, and the return address is passed in the register `ra`. When invoked, a procedure may choose to place temporaries on the stack as shown above, but when it jumps to the return address, the stack should be in the same state as it was upon entry. Naively, we might expect the code for a function obeying this calling convention to have the following STAL type:

$$\forall[].\{\mathtt{r1}{:}\mathit{int}, \mathtt{sp}{:}\sigma, \mathtt{ra}{:}\forall[].\{\mathtt{r1}{:}\langle\rangle, \mathtt{sp}{:}\sigma\}\}$$

Notice that the type of the return address is constrained so that the stack must have the same shape upon return as it had upon entry. Hence, if the procedure pushes any arguments onto the stack, it must pop them off.

However, this typing is unsatisfactory for two reasons. The first problem is that there is nothing preventing the procedure from popping off values from the stack and then pushing new values (of the appropriate type) onto the stack. In other words, the caller’s stack frame is not protected from the function’s code. The second problem is much worse: such a function can only be invoked from states where the stack is exactly described by σ. This effectively prevents invocation of the procedure from two different points in the program. For example, there is no way for the procedure to push its return address on the stack and jump to itself.

The solution to both problems is to abstract the type of the stack using a stack type variable:

$$\forall[\rho].\{\mathtt{r1}{:}\mathit{int}, \mathtt{sp}{:}\rho, \mathtt{ra}{:}\forall[].\{\mathtt{r1} : \mathit{int}, \mathtt{sp}{:}\rho\}\}$$

To invoke a function with this type, the caller must instantiate the bound stack type variable ρ with the current type of the stack. As before, the function can only jump to the return address when the stack is in the same state as it was upon entry. However, the first problem above is addressed because the type checker treats ρ as an abstract stack type while checking the body of the code. Hence, the code cannot perform an `sfree`, `sld`, or `sst` on the stack. It must first allocate its own space on the stack, only this space may be accessed by the function, and the space must be freed before returning to the caller.[2] The second problem is solved because the stack type variable may be instantiated

---

[2] Some intuition on this topic may be obtained from Reynolds’ theorem on parametric polymorphism [27] but a formal proof is difficult.

in different ways. Hence multiple call sites with different stack states, including recursive calls, may now invoke the function. In fact, a recursive call will usually instantiate the stack variable with a different type than the original call because, unless it is a tail call, it will need to store its return address on the stack.

```
(H, {sp ↦ nil}, I)  where

H = l_fact:
        code[ρ]{r1 : ⟨⟩, r2 : int, sp : ρ, ra : τ_ρ}.
          bneq r2,l_nonzero[ρ]          % if n = 0 continue
          mov r1,1                      % result is 1
          jmp ra                        % return
    l_nonzero:
        code[ρ]{r1 : ⟨⟩, r2 : int, sp : ρ, ra : τ_ρ}.
          sub r3,r2,1                   % n − 1
          salloc 2                      % save n and return address to stack
          sst sp(0),r2
          sst sp(1),ra
          mov r2,r3                     % recursive call fact(n − 1)
          mov ra,l_cont[ρ]
          jmp l_fact[int::τ_ρ::ρ]
    l_cont:
        code[ρ]{r1 : int, sp : int::τ_ρ::ρ}.
          sld r2,sp(0)                  % restore n and return address
          sld ra,sp(1)
          sfree 2
          mul r1,r2,r1                  % result is n × fact(n − 1)
          jmp ra                        % return
    l_halt:
        code[]{r1 : int, sp : nil}.
          halt [int]

and I =   malloc r1[]                   % environment
          mov r2,6                      % argument
          mov ra,l_halt                 % return address for initial call
          jmp l_fact[nil]

and τ_ρ = ∀[].{r1 : int, sp : ρ}
```

Fig. 4. STAL Factorial Example

Figure 4 gives stack-based code for the factorial example of the previous section. The function is invoked by moving its environment (an empty tuple) into `r1`, the argument into `r2`, and the return address label into `ra` and jumping to the label `l_fact`. Notice that the nonzero branch must save the argument and current return address on the stack before jumping to the `fact` label in a recursive call. It is interesting to note that the stack-based code is quite similar to the heap-based code of Figure 1. Indeed, the code remains in a continuation-passing style, but instead of passing the continuation as a heap-allocated tuple, the environment of the continuation is passed in the stack pointer and the code of the continuation is passed in the return address register.

To more fully appreciate the correspondence, consider the type of the TAL version of `l_fact` from Figure 1:

$$\forall[].\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2}{:}\mathit{int}, \mathtt{r3}{:}\exists\alpha.\langle\forall[].\{\mathtt{r1}{:}\alpha, \mathtt{r2}{:}\mathit{int}\}^1, \alpha^1\rangle\}$$

We could have used an alternative approach where we pass the components of the continuation closure unboxed in separate registers. To do so, the caller must unpack the continuation and the function must abstract the type of the continuation's environment resulting in a quantifier rotation:

$$\forall[\alpha].\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2}{:}\mathit{int}, \mathtt{r3}{:}\forall[].\{\mathtt{r1}{:}\alpha, \mathtt{r2}{:}\mathit{int}\}, \mathtt{r4}{:}\alpha\}$$

Now, it is clear that the STAL code, which has type

$$\forall[\rho].\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2}{:}\mathit{int}, \mathtt{ra}{:}\forall[].\{\mathtt{sp}{:}\rho, \mathtt{r1}{:}\mathit{int}\}, \mathtt{sp}{:}\rho\}$$

is essentially the same! Indeed, the only difference between a CPS-based compiler, such as SML/NJ, and a conventional stack-based compiler, is that for the latter, continuation environments are allocated on a stack. Our type system describes this well-known connection elegantly.

Our techniques can be applied to other calling conventions and do not appear to inhibit most optimizations. For instance, tail calls can be eliminated in CPS simply by forwarding a continuation closure to the next function. If continuations are allocated on the stack, we have the mechanisms to pop the current activation frame off the stack and to push any arguments before performing the tail call. Furthermore, the type system is expressive enough to type this resetting and adjusting for any kind of tail call, not just a self tail call. As another example, some CISC-style conventions push the arguments, the environment, and then the return address on the stack, and return the result on the stack. With this convention, the factorial code would have type:

$$\forall[\rho].\{\mathtt{sp}{:}\forall[]\{\mathtt{sp}{:}\mathit{int}::\rho\}::\langle\rangle::\mathit{int}::\rho\}$$

Callee-saves registers (registers whose values must be preserved across function calls) can be handled in the same fashion as the stack pointer. In particular, the function holds abstract the type of the callee-saves register and requires that the register have the same type upon return. For instance, if we wish to preserve register `r3` across a call to factorial, we would use the type:

$$\forall[\rho,\alpha].\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2}{:}\mathit{int}, \mathtt{r3}{:}\alpha, \mathtt{ra}{:}\forall[]\{\mathtt{sp}{:}\rho, \mathtt{r1}{:}\mathit{int}, \mathtt{r3}{:}\alpha\}, \mathtt{sp}{:}\rho\}$$

Translating this type back in to a boxed, heap allocated closure, we obtain:

$$\forall[\alpha].\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2} : \mathit{int}, \mathtt{r3}{:}\alpha, \mathtt{ra}{:}\exists\beta.\langle\forall[]\{\mathtt{r1}{:}\beta, \mathtt{r2}{:}\mathit{int}, \mathtt{r3}{:}\alpha\}^1, \beta^1\rangle\}$$

This is the type of the callee-saves approach of Appel and Shao [1]. Thus we see how our correspondence enables transformations developed for heap-based compilers to be used in traditional stack-based compilers and vice versa. The generalization to multiple callee-saves registers and other calling conventions should be clear. Indeed, we have found that the type system of STAL provides a concise way to declaratively specify a variety of calling conventions.

4 Exceptions

We now consider how to implement exceptions in STAL. We will find that a calling convention for function calls in the presence of exceptions may be derived from the heap-based CPS calling convention, just as was the case without exceptions. However, implementing this calling convention will require that the type system be made more expressive by adding *compound* stack types. This additional expressiveness will turn out to have uses beyond exceptions, allowing most compiler-introduced uses of pointers into the midst of the stack.

4.1 Exception Calling Conventions

In a heap-based CPS framework, exceptions are implemented by passing two continuations: the usual continuation and an *exception continuation.* Code raises an exception by jumping to the latter. For an integer to unit function, this calling convention is expressed as the following TAL type (ignoring the outer closure and environment):

$$\forall[\,].\{\texttt{r1}{:}\mathit{int}, \texttt{ra}{:}\exists\alpha_1.\langle\forall[\,].\{\texttt{r1}{:}\alpha_1, \texttt{r2}{:}\langle\rangle\}^1, \alpha_1^1\rangle, \texttt{re}{:}\exists\alpha_2.\langle\forall[\,].\{\texttt{r1}{:}\alpha_2, \texttt{r2}{:}\mathit{exn}\}^1, \alpha_2^1\rangle\}$$

Again, the caller might unpack the continuations:

$$\forall[\alpha_1, \alpha_2].\{\texttt{r1}{:}\mathit{int}, \texttt{ra}{:}\forall[\,].\{\texttt{r1}{:}\alpha_1, \texttt{r2}{:}\langle\rangle\}, \texttt{ra}'{:}\alpha_1, \texttt{re}{:}\forall[\,].\{\texttt{r1}{:}\alpha_2, \texttt{r2}{:}\mathit{exn}\}, \texttt{re}'{:}\alpha_2\}$$

Then the caller might (erroneously) attempt to place the continuation environments on stacks, as before:

$$\forall[\rho_1, \rho_2].\{\texttt{r1}{:}\mathit{int}, \texttt{ra}{:}\forall[\,].\{\texttt{sp}{:}\rho_1, \texttt{r1}{:}\langle\rangle\}, \texttt{sp}{:}\rho_1, \texttt{re}{:}\forall[\,].\{\texttt{sp}{:}\rho_2, \texttt{r1}{:}\mathit{exn}\}, \texttt{sp}'{:}\rho_2\}$$

Unfortunately, this calling convention uses two stack pointers, and STAL has only one stack.[3] Observe, though, that the exception continuation's stack is necessarily a tail of the ordinary continuation's stack. This observation leads to the following calling convention for exceptions with stacks:

$$\begin{gathered}\forall[\rho_1, \rho_2].\{\,\texttt{sp}{:}\rho_1 \circ \rho_2, \texttt{r1}{:}\mathit{int}, \texttt{ra}{:}\forall[\,].\{\texttt{sp}{:}\rho_1 \circ \rho_2, \texttt{r1}{:}\langle\rangle\}, \\ \texttt{re}'{:}\mathit{ptr}(\rho_2), \texttt{re}{:}\forall[\,].\{\texttt{sp}{:}\rho_2, \texttt{r1}{:}\mathit{exn}\}\}\end{gathered}$$

---

[3] Some language implementations use a separate exception stack; with some minor modifications, this calling convention would be satisfactory for such implementations.

This type uses two new constructs we now add to STAL (see Figure 5). When σ_1 and σ_2 are stack types, the stack type $\sigma_1 \circ \sigma_2$ is the result of appending the two types. Thus, in the above type, the function is presented with a stack with type $\rho_1 \circ \rho_2$, all of which is expected by the regular continuation, but only a tail of which (ρ_2) is expected by the exception continuation. Since ρ_1 and ρ_2 are quantified, the function may still be used for any stack so long as the exception continuation accepts some tail of that stack.

To raise an exception, the exception is placed in `r1` and the control is transfered to the exception continuation. This requires cutting the actual stack down to just that expected by the exception continuation. Since the length of ρ_1 is unknown, this can not be done by `sfree`. Instead, a pointer to the desired position in the stack is supplied in $\texttt{re}'$, and is moved into `sp`. The type $ptr(\sigma)$ is the type of pointers into the stack at a position where the stack has type σ. Such pointers are obtained simply by moving `sp` into a register.

4.2 Compound Stacks

The additional syntax to support exceptions is summarized in Figure 5. The new type constructors were discussed above. The word value $ptr(i)$ is used by the operational semantics to represent pointers into the stack; the element pointed to is i words from the bottom of the stack. (See Figure 7 for details.) Of course, on a real machine, these would be implemented by actual pointers. The instructions $\texttt{mov}\ r_d, \texttt{sp}$ and $\texttt{mov sp}, r_s$ save and restore the stack pointer, and the instructions $\texttt{sld}\ r_d, r_s(i)$ and $\texttt{sst}\ r_d(i), r_s$ allow for loading from and storing to pointers.

$$\begin{array}{lll}
\textit{types} & \tau & ::= \cdots \mid ptr(\sigma) \\
\textit{stack types} & \sigma & ::= \cdots \mid \sigma_1 \circ \sigma_2 \\
\textit{word values} & w & ::= \cdots \mid ptr(i) \\
\textit{instructions} & \iota & ::= \cdots \mid \texttt{mov}\ r_d, \texttt{sp} \mid \texttt{mov sp}, r_s \mid \texttt{sld}\ r_d, r_s(i) \mid \texttt{sst}\ r_d(i), r_s
\end{array}$$

Fig. 5. Additions to TAL for Compound Stacks

The introduction of pointers into the stack raises a delicate issue for the typing system. When the stack pointer is copied into a register, changes to the stack are not reflected in the type of the copy, and can invalidate a pointer. Consider the following incorrect code:

$$\begin{array}{ll}
\multicolumn{2}{l}{\%\ \text{begin with}\ \texttt{sp} : \tau{::}\sigma,\ \texttt{sp} \mapsto w{::}S\ (\tau \neq ns)} \\
\texttt{mov r1,sp} & \%\ \texttt{r1} : ptr(\tau{::}\sigma) \\
\texttt{sfree}\ 1 & \%\ \texttt{sp} : \sigma,\ \texttt{sp} \mapsto S \\
\texttt{salloc}\ 1 & \%\ \texttt{sp} : ns{::}\sigma,\ \texttt{sp} \mapsto ns{::}S \\
\texttt{sld r2,r1}(0) & \%\ \texttt{r2} : \tau\ \text{but}\ \texttt{r2} \mapsto ns
\end{array}$$

When execution reaches the final line, `r1` still has type $\mathit{ptr}(\tau{::}\sigma)$, but this type is no longer consistent with the state of the stack; the pointer in `r1` points to ns.

To prohibit erroneous loads of this sort, the type system requires that the pointer r_s be *valid* in the instructions $\texttt{sld}\ r_d, r_s(i)$, $\texttt{sst}\ r_d(i), r_s$, and $\texttt{mov sp}, r_s$. An invariant of our system is that the type of `sp` always describes the current stack, so using a pointer into the stack will be sound if that pointer's type is consistent with `sp`'s type. Suppose `sp` has type σ_1 and r has type $\mathit{ptr}(\sigma_2)$, then r is valid if σ_2 is a tail of σ_1 (formally, if there exists some σ' such that $\sigma_1 = \sigma' \circ \sigma_2$). If a pointer is invalid, it may be neither loaded from nor moved into the stack pointer. In the above example the load will be rejected because `r1`'s type $\tau{::}\sigma$ is not a tail of $\texttt{sp}'s$ type, $\mathit{ns}{::}\sigma$.

4.3 Using Compound Stacks

Recall the type for a function in the presence of exceptions:

$$\begin{array}{c}\forall[\rho_1,\rho_2].\{\,\texttt{sp}{:}\rho_1\circ\rho_2, \texttt{r1}{:}\mathit{int}, \texttt{ra}{:}\forall[\,].\{\texttt{sp}{:}\rho_1\circ\rho_2, \texttt{r1}{:}\langle\rangle\},\\ \texttt{re}'{:}\mathit{ptr}(\rho_2), \texttt{re}{:}\forall[\,].\{\texttt{sp}{:}\rho_2, \texttt{r1}{:}\mathit{exn}\}\}\end{array}$$

An exception may be raised within the body of such a function by restoring the handler's stack from $\texttt{re}'$ and jumping to the handler. A new exception handler may be installed by copying the stack pointer to $\texttt{re}'$ and making forthcoming function calls with the stack type variables instantiated to nil and $\rho_1 \circ \rho_2$. Calls that do not install new exception handlers would attach their frames to ρ_1 and pass on ρ_2 unchanged.

Since exceptions are probably raised infrequently, an implementation could save a register by storing the exception continuation's code pointer on the stack, instead of in its own register. If this convention were used, functions would expect stacks with the type $\rho_1 \circ (\tau_{\text{handler}}{::}\rho_2)$ and exception pointers with the type $\mathit{ptr}(\tau_{\text{handler}}{::}\rho_2)$ where $\tau_{\text{handler}} = \forall[\,].\{\texttt{sp}{:}\rho_2, \texttt{r1}{:}\mathit{exn}\}$.

This last convention illustrates a use for compound stacks that goes beyond implementing exceptions. We have a general tool for locating data of type τ amidst the stack by using the calling convention:

$$\forall[\rho_1,\rho_2].\{\texttt{sp}{:}\rho_1\circ(\tau{::}\rho_2), \texttt{r1}{:}\mathit{ptr}(\tau{::}\rho_2), \ldots\}$$

One application of this tool would be for implementing Pascal with displays. The primary limitation of this tool is that if more than one piece of data is stored amidst the stack, although quantification may be used to avoid specifying the precise locations of that data, function calling conventions would have to specify in what *order* data appears on the stack. 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.

5 Related and Future Work

Our work is partially inspired by Reynolds [26], which uses functor categories to "replace continuations by instruction sequences and store shapes by descriptions of the structure of the run-time stack." However, Reynolds was primarily concerned with using functors to express an intermediate language of a semantics-based compiler for Algol, whereas we are primarily concerned with type structure for general-purpose target languages.

Stata and Abadi [30] formalize the Java bytecode verifier's treatment of subroutines by giving a type system for a subset of the Java Virtual Machine language. In particular, their type system ensures that for any program control point, the Java stack is of the same size each time that control point is reached during execution. Consequently, procedure call must be a primitive construct (which it is in JVML). In contrast, our treatment supports polymorphic stack recursion, and hence procedure calls can be encoded with existing assembly-language primitives.

Tofte and others [8, 33] have developed an allocation strategy involving regions. Regions are lexically scoped containers that have a LIFO ordering on their lifetimes, much like the values on a stack. As in our approach, polymorphic recursion on abstracted region variables plays a critical role. However, unlike the objects in our stacks, regions are variable-sized, and objects need not be allocated into the region which was most recently created. Furthermore, there is only one allocation mechanism in Tofte's system (the stack of regions) and no need for a garbage collector. In contrast, STAL only allows allocation at the top of the stack and assumes a garbage collector for heap-allocated values. However, the type system for STAL is considerably simpler than the type system of Tofte et al., as it requires no effect information in types.

Bailey and Davidson [6] also describe a specification language for modeling procedure calling conventions and checking that implementations respect these conventions. They are able to specify features such as a variable number of arguments that our formalism does not address. However, their model is explicitly tied to a stack-based calling convention and does not address features such as exception handlers. Furthermore, their approach does not integrate the specification of calling conventions with a general-purpose type system.

Although our type system is sufficiently expressive for compilation of a number of source languages, it falls short in several areas. First, it cannot support general pointers into the stack because of the ordering requirements; nor can stack and heap pointers be unified so that a function taking a tuple argument can be passed either a heap-allocated or a stack-allocated tuple. Second, threads and advanced mechanisms for implementing first-class continuations such as the work by Hieb et al. [15] cannot be modeled in this system without adding new primitives.

However, we claim that the framework presented here is a practical approach to compilation. To substantiate this claim, we are constructing a compiler called TALC that maps the KML language [10] to a variant of STAL described here, suitably adapted for the Intel IA32 architecture. We have found it straightforward to enrich the target language type system to include support for other type constructors, such as references, higher-order constructors, and recursive types. The compiler uses an unboxed stack allocation style of continuation passing.

Although we have discussed mechanisms for typing stacks at the assembly language level, our techniques generalize to other languages. The same mechanisms, including the use of polymorphic recursion to abstract the tail of a stack, can be used to introduce explicit stacks in higher level calculi. An intermediate language with explicit stacks would allow control over allocation at a point where more information is available to guide allocation decisions.

6 Summary

We have given a type system for a typed assembly language with both a heap and a stack. Our language is flexible enough to support the following compilation techniques: CPS using both heap allocation and stack allocation, a variety of procedure calling conventions, displays, exceptions, tail call elimination, and callee-saves registers.

A key contribution of the type system is that it makes procedure calling conventions explicit and provides a means of specifying and checking calling conventions that is grounded in language theory. The type system also makes clear the relationship between heap allocation and stack allocation of continuation closures, capturing both allocation strategies in one calculus.

References

1. Andrew Appel and Zhong Shao. Callee-saves registers in continuation-passing style. *Lisp and Symbolic Computation*, 5:189–219, 1992.
2. Andrew Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with clsoures. *Journal of Functional Programming*, 1(1), January 1993.
3. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
4. Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, January 1989.
5. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science.*
6. Mark Bailey and Jack Davidson. A formal model of procedure calling conventions. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 298–310, San Francisco, January 1995.
7. Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.
8. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.
9. Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

10. Karl Crary. *KML Reference Manual*. Department of Computer Science, Cornell University, 1996.
11. Karl Crary. Foundations for the implementation of higher-order subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.
12. Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed reprsentation transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 11–24, Amsterdam, June 1997.
13. Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1994.
14. Amer Diwan, David Tarditi, and Eliot Moss. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, August 1995.
15. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, June 1990. Published as *SIGPLAN Notices*, 25(6).
16. Intel Corporation. *Intel Architecture Optimization Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect, IL, 60056-7641, 1997.
17. David Kranz, R. Kelsey, J. Rees, P. R. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.
18. P. J. Landin. The mechanical evaluation of expressions. *Computer J.*, 6(4):308–20, 1964.
19. Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, January 1992.
20. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
21. Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, January 1996.
22. G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.
23. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651, November 1997.
24. Gregory Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995. Published as CMU Technical Report CMU-CS-95-226.
25. Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.
26. John Reynolds. Using functor categories to generate intermediate code. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 25–36, San Francisco, January 1995.

27. John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.
28. Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.
29. Zhong Shao. Flexible representation analysis. In *ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.
30. Raymie Stata and Martín Abadi. A type system for java bytecode subroutines. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, January 1998.
31. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.
32. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.
33. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, January 1994.
34. Mitchell Wand. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference*, pages 19–28, August 1980.

A Formal STAL Semantics

This appendix contains a complete technical description of our calculus, STAL. The STAL abstract machine is very similar to the TAL abstract machine (described in detail in Morrisett et al. [23]). The syntax appears in Figure 6. The operational semantics is given as a deterministic rewriting system in Figure 7. The notation $a[b/c]$ denotes capture avoiding substitution of b for c in a. The notation $a\{b \mapsto c\}$ represents map update:

$$\{b_1 \mapsto c_1, b_2 \mapsto c_2, \ldots, b_n \mapsto c_n\}\{b \mapsto c\} = \begin{cases} \{b \mapsto c, b_1 \mapsto c_1, \ldots, b_n \mapsto c_n\}, \\ \quad \text{if } b \notin \{b_1, \ldots, b_n\} \\ \{b_1 \mapsto c, b_2 \mapsto c_2, \ldots, b_n \mapsto c_n\}, \\ \quad \text{if } b = b_1 \end{cases}$$

To make the presentation simpler for the branching rules, some extra notation is used for expressing sequences of type and stack type instantiations. We introduce a new syntactic class (ψ) for type sequences:

$$\psi ::= \cdot \mid \tau, \psi \mid \sigma, \psi$$

The notation $w[\psi]$ stands for the obvious iteration of instantiations; the substitution notation $I[\psi/\Delta]$ is defined by:

$$\begin{aligned} I[\cdot/\cdot] &= I \\ I[\tau, \psi/\alpha, \Delta] &= I[\tau/\alpha][\psi/\Delta] \\ I[\sigma, \psi/\rho, \Delta] &= I[\sigma/\rho][\psi/\Delta] \end{aligned}$$

The static semantics is similar to TAL's but requires extra judgments for definitional equality of types, stack types, and register file types and uses a more compositional style for instructions. Definitional equality is needed because two stack types (such as $(\mathit{int}::\mathit{nil}) \circ (\mathit{int}::\mathit{nil})$ and $\mathit{int}::\mathit{int}::\mathit{nil}$) may be syntactically different but represent the same type. The judgments are summarized in Figure 8, the rules for type judgments appear in Figure 9, and the rules for term judgments appear in Figures 10 and 11. The notation Δ', Δ denotes appending Δ' to the front of Δ, that is:

$$\begin{aligned} \cdot, \Delta &= \Delta \\ (\alpha, \Delta'), \Delta &= \alpha, (\Delta', \Delta) \\ (\rho, \Delta'), \Delta &= \rho, (\Delta', \Delta) \end{aligned}$$

As with TAL, STAL is type sound:

Proposition A1 (Type Soundness) *If* $\vdash P$ *and* $P \longmapsto^* P'$ *then* P' *is not stuck.*

This proposition is proved using the following two lemmas.

Lemma 1 (Subject Reduction). *If* $\vdash P$ *and* $P \longmapsto P'$ *then* $\vdash P'$.

A well-formed terminal state has the form $(H, R\{\mathtt{r1} \mapsto w\}, \mathtt{halt}\,[\tau])$ where there exists a Ψ such that $\vdash H : \Psi$ and $\Psi; \cdot \vdash w : \tau$ wval.

Lemma 2 (Progress). *If* $\vdash P$ *then either* P *is a well-formed terminal state or there exists* P' *such that* $P \longmapsto P'$.

types	$\tau ::= \alpha \mid \mathit{int} \mid \mathit{ns} \mid \forall[\Delta].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau \mid \mathit{ptr}(\sigma)$
stack types	$\sigma ::= \rho \mid \mathit{nil} \mid \tau{::}\sigma \mid \sigma_1 \circ \sigma_2$
initialization flags	$\varphi ::= 0 \mid 1$
label assignments	$\Psi ::= \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
type assignments	$\Delta ::= \cdot \mid \alpha, \Delta \mid \rho, \Delta$
register assignments	$\Gamma ::= \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n, \texttt{sp}{:}\sigma\}$

registers	$r ::= \texttt{r1} \mid \cdots \mid \texttt{r}k$
word values	$w ::= \ell \mid i \mid \mathit{ns} \mid ?\tau \mid w[\tau] \mid w[\sigma] \mid \mathit{pack}\ [\tau, w]\ \mathit{as}\ \tau' \mid \mathit{ptr}(i)$
small values	$v ::= r \mid w \mid v[\tau] \mid v[\sigma] \mid \mathit{pack}\ [\tau, v]\ \mathit{as}\ \tau'$
heap values	$h ::= \langle w_1, \ldots, w_n \rangle \mid \texttt{code}[\Delta]\Gamma.I$
heaps	$H ::= \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	$R ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \texttt{sp} \mapsto S\}$
stacks	$S ::= \mathit{nil} \mid w{::}S$

instructions	$\iota ::= \mathit{aop}\ r_d, r_s, v \mid \mathit{bop}\ r, v \mid \texttt{ld}\ r_d, r_s(i) \mid \texttt{malloc}\ r[\vec{\tau}] \mid$ $\texttt{mov}\ r_d, v \mid \texttt{mov sp}, r_s \mid \texttt{mov}\ r_d, \texttt{sp} \mid \texttt{salloc}\ n \mid$ $\texttt{sfree}\ n \mid \texttt{sld}\ r_d, \texttt{sp}(i) \mid \texttt{sld}\ r_d, r_s(i) \mid$ $\texttt{sst sp}(i), r_s \mid \texttt{sst}\ r_d(i), r_s \mid \texttt{st}\ r_d(i), r_s \mid$ $\texttt{unpack}\ [\alpha, r_d], v$
arithmetic ops	$\mathit{aop} ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul}$
branch ops	$\mathit{bop} ::= \texttt{beq} \mid \texttt{bneq} \mid \texttt{bgt} \mid \texttt{blt} \mid \texttt{bgte} \mid \texttt{blte}$
instruction sequences	$I ::= \iota; I \mid \texttt{jmp}\ v \mid \texttt{halt}\ [\tau]$
programs	$P ::= (H, R, I)$

Fig. 6. Syntax of STAL

$(H,R,I) \longmapsto P$ where	
if $I =$	then $P =$
$\mathtt{add}\ r_d, r_s, v; I'$	$(H, R\{r_d \mapsto R(r_s) + \hat{R}(v)\}, I')$ and similarly for `mul` and `sub`
$\mathtt{beq}\ r, v; I'$ when $R(r) \neq 0$	(H, R, I') and similarly for `bneq`, `blt`, *etc.*
$\mathtt{beq}\ r, v; I'$ when $R(r) = 0$	$(H, R, I''[\psi/\Delta])$ where $\hat{R}(v) = \ell[\psi]$ and $H(\ell) = \mathtt{code}[\Delta]\Gamma.I''$ and similarly for `bneq`, `blt`, *etc.*
$\mathtt{jmp}\ v$	$(H, R, I'[\psi/\Delta])$ where $\hat{R}(v) = \ell[\psi]$ and $H(\ell) = \mathtt{code}[\Delta]\Gamma.I'$
$\mathtt{ld}\ r_d, r_s(i); I'$	$(H, R\{r_d \mapsto w_i\}, I')$ where $R(r_s) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$ and $0 \leq i < n$
$\mathtt{malloc}\ r_d[\tau_1, \ldots, \tau_n]; I'$	$(H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}, R\{r_d \mapsto \ell\}, I')$ where $\ell \notin H$
$\mathtt{mov}\ r_d, v; I'$	$(H, R\{r_d \mapsto \hat{R}(v)\}, I')$
$\mathtt{mov}\ r_d, \mathtt{sp}; I'$	$(H, R\{r_d \mapsto ptr(\lvert S \rvert)\}, I')$
$\mathtt{mov}\ \mathtt{sp}, r_s; I'$	$(H, R\{\mathtt{sp} \mapsto w_j{::}\cdots{::}w_1{::}nil\}, I')$ where $R(\mathtt{sp}) = w_n{::}\cdots{::}w_1{::}nil$ and $R(r_s) = ptr(j)$ with $0 \leq j \leq n$
$\mathtt{salloc}\ n; I'$	$(H, R\{\mathtt{sp} \mapsto \underbrace{ns{::}\cdots{::}ns}_{n}{::}R(\mathtt{sp})\}, I')$
$\mathtt{sfree}\ n; I'$	$(H, R\{\mathtt{sp} \mapsto S\}, I')$ where $R(\mathtt{sp}) = w_1{::}\cdots{::}w_n{::}S$
$\mathtt{sld}\ r_d, \mathtt{sp}(i); I'$	$(H, R\{r_d \mapsto w_i\}, I')$ where $R(\mathtt{sp}) = w_0{::}\cdots{::}w_{n-1}{::}nil$ and $0 \leq i < n$
$\mathtt{sld}\ r_d, r_s(i); I'$	$(H, R\{r_d \mapsto w_{j-i}\}, I')$ where $R(r_s) = ptr(j)$ and $R(\mathtt{sp}) = w_n{::}\cdots{::}w_1{::}nil$ and $0 \leq i < j \leq n$
$\mathtt{sst}\ \mathtt{sp}(i), r_s; I'$	$(H, R\{\mathtt{sp} \mapsto w_0{::}\cdots{::}w_{i-1}{::}R(r_s){::}S\}, I')$ where $R(\mathtt{sp}) = w_0{::}\cdots{::}w_i{::}S$ and $0 \leq i$
$\mathtt{sst}\ r_d(i), r_s; I'$	$(H, R\{\mathtt{sp} \mapsto w_n{::}\cdots{::}w_{j-i+1}{::}R(r_s){::}w_{j-i-1}{::}\cdots{::}w_1{::}nil\}, I')$ where $R(r_d) = ptr(j)$ and $R(\mathtt{sp}) = w_n{::}\cdots{::}w_1{::}nil$ and $0 \leq i < j \leq n$
$\mathtt{st}\ r_d(i), r_s; I'$	$(H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1} \rangle\}, R, I')$ where $R(r_d) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$ and $0 \leq i < n$
$\mathtt{unpack}\ [\alpha, r_d], v; I'$	$(H, R\{r_d \mapsto w\}, I'[\tau/\alpha])$ where $\hat{R}(v) = pack\ [\tau, w]\ as\ \tau'$

$$\text{Where } \hat{R}(v) = \begin{cases} R(r) & \text{when } v = r \\ w & \text{when } v = w \\ \hat{R}(v')[\tau] & \text{when } v = v'[\tau] \\ pack\ [\tau, \hat{R}(v')]\ as\ \tau' & \text{when } v = pack\ [\tau, v']\ as\ \tau' \end{cases}$$

Fig. 7. Operational Semantics of STAL

Judgement	Meaning
$\Delta \vdash \tau$	τ is a valid type
$\Delta \vdash \sigma$	σ is a valid stack type
$\vdash \Psi$	Ψ is a valid heap type (no context is used because heap types must be closed)
$\Delta \vdash \Gamma$	Γ is a valid register file type
$\Delta \vdash \tau_1 = \tau_2$	τ_1 and τ_2 are equal types
$\Delta \vdash \sigma_1 = \sigma_2$	σ_1 and σ_2 are equal stack types
$\Delta \vdash \Gamma_1 = \Gamma_2$	Γ_1 and Γ_2 are equal register file types
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash \Gamma_1 \leq \Gamma_2$	Γ_1 is a register file subtype of Γ_2
$\vdash H : \Psi$	the heap H has type Ψ
$\Psi \vdash S : \sigma$	the stack S has type σ
$\Psi \vdash R : \Gamma$	the register file R has type Γ
$\Psi \vdash h : \tau$ hval	the heap value h has type τ
$\Psi; \Delta \vdash w : \tau$ wval	the word value w has type τ
$\Psi; \Delta \vdash w : \tau^{\varphi}$ fwval	the word value w has flagged type τ^{φ} ($i.e.$, w has type τ or w is $?\tau$ and φ is 0)
$\Psi; \Delta; \Gamma \vdash v : \tau$	the small value v has type τ
$\Psi; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma'$	given a context of type $\Psi; \Delta; \Gamma$, ι is a well formed instruction and produces a context of type $\Psi; \Delta'; \Gamma'$
$\Psi; \Delta; \Gamma \vdash I$	I is a valid sequence of instructions
$\vdash P$	P is a valid program

Fig. 8. Static Semantics of STAL (judgments)

$$\boxed{\Delta \vdash \tau \quad \Delta \vdash \sigma \quad \vdash \Psi \quad \Delta \vdash \Gamma}$$

$$\frac{\Delta \vdash \tau = \tau}{\Delta \vdash \tau} \qquad \frac{\Delta \vdash \sigma = \sigma}{\Delta \vdash \sigma} \qquad \frac{\cdot \vdash \tau_i}{\vdash \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}} \qquad \frac{\Delta \vdash \Gamma = \Gamma}{\Delta \vdash \Gamma}$$

$$\boxed{\Delta \vdash \tau_1 = \tau_2 \quad \Delta \vdash \sigma_1 = \sigma_2 \quad \Delta \vdash \Gamma_1 = \Gamma_2}$$

$$\frac{\Delta \vdash \tau_2 = \tau_1}{\Delta \vdash \tau_1 = \tau_2} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 \quad \Delta \vdash \tau_2 = \tau_3}{\Delta \vdash \tau_1 = \tau_3}$$

$$\frac{\Delta \vdash \sigma_2 = \sigma_1}{\Delta \vdash \sigma_1 = \sigma_2} \qquad \frac{\Delta \vdash \sigma_1 = \sigma_2 \quad \Delta \vdash \sigma_2 = \sigma_3}{\Delta \vdash \sigma_1 = \sigma_3}$$

$$\frac{}{\Delta \vdash \alpha = \alpha}\ (\alpha \in \Delta) \qquad \frac{}{\Delta \vdash \mathit{int} = \mathit{int}}$$

$$\frac{\Delta', \Delta \vdash \Gamma_1 = \Gamma_2}{\Delta \vdash \forall[\Delta'].\Gamma_1 = \forall[\Delta'].\Gamma_2} \qquad \frac{\Delta \vdash \tau_i = \tau'_i}{\Delta \vdash \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle = \langle {\tau'_1}^{\varphi_1}, \ldots, {\tau'_n}^{\varphi_n} \rangle}$$

$$\frac{\alpha, \Delta \vdash \tau_1 = \tau_2}{\Delta \vdash \exists \alpha.\tau_1 = \exists \alpha.\tau_2} \qquad \frac{}{\Delta \vdash \mathit{ns} = \mathit{ns}} \qquad \frac{\Delta \vdash \sigma_1 = \sigma_2}{\Delta \vdash \mathit{ptr}(\sigma_1) = \mathit{ptr}(\sigma_2)}$$

$$\frac{}{\Delta \vdash \rho = \rho}\ (\rho \in \Delta) \qquad \frac{}{\Delta \vdash \mathit{nil} = \mathit{nil}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 \quad \Delta \vdash \sigma_1 = \sigma_2}{\Delta \vdash \tau_1{::}\sigma_1 = \tau_2{::}\sigma_2} \qquad \frac{\Delta \vdash \sigma_1 = \sigma'_1 \quad \Delta \vdash \sigma_2 = \sigma'_2}{\Delta \vdash \sigma_1 \circ \sigma_2 = \sigma'_1 \circ \sigma'_2}$$

$$\frac{\Delta \vdash \sigma}{\Delta \vdash \mathit{nil} \circ \sigma = \sigma} \qquad \frac{\Delta \vdash \sigma}{\Delta \vdash \sigma \circ \mathit{nil} = \sigma}$$

$$\frac{\Delta \vdash \tau \quad \Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash (\tau{::}\sigma_1) \circ \sigma_2 = \tau{::}(\sigma_1 \circ \sigma_2)}$$

$$\frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2 \quad \Delta \vdash \sigma_3}{\Delta \vdash (\sigma_1 \circ \sigma_2) \circ \sigma_3 = \sigma_1 \circ (\sigma_2 \circ \sigma_3)}$$

$$\frac{\Delta \vdash \sigma = \sigma' \quad \Delta \vdash \tau_i = \tau'_i}{\Delta \vdash \{\mathtt{sp}{:}\sigma, r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n\} = \{\mathtt{sp}{:}\sigma', r_1{:}\tau'_1, \ldots, r_n{:}\tau'_n\}}$$

$$\boxed{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash \Gamma_1 \leq \Gamma_2}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2}{\Delta \vdash \tau_1 \leq \tau_2} \qquad \frac{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash \tau_2 \leq \tau_3}{\Delta \vdash \tau_1 \leq \tau_3}$$

$$\frac{\Delta \vdash \tau_i}{\Delta \vdash \langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle \leq \langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^0, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle}$$

$$\frac{\Delta \vdash \sigma = \sigma' \quad \Delta \vdash \tau_i = \tau'_i \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash \tau_i \quad (\text{for } n < i \leq m)}{\Delta \vdash \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_m{:}\tau_m\} \leq \{\mathtt{sp}{:}\sigma', r_1{:}\tau'_1, \ldots, r_n{:}\tau'_n\}}\ (m \geq n)$$

Fig. 9. Static Semantics of STAL, Judgments for Types

$$\boxed{\vdash P \quad \vdash H : \Psi \quad \Psi \vdash S : \sigma \quad \Psi \vdash R : \Gamma}$$

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \cdot; \Gamma \vdash I}{\vdash (H, R, I)}$$

$$\frac{\vdash \Psi \quad \Psi \vdash h_i : \tau_i \text{ hval}}{\vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} : \Psi} \ (\Psi = \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\})$$

$$\frac{}{\Psi \vdash \mathit{nil} : \mathit{nil}} \qquad \frac{\Psi; \cdot \vdash w : \tau \text{ wval} \quad \Psi \vdash S : \sigma}{\Psi \vdash w{::}S : \tau{::}\sigma}$$

$$\frac{\Psi \vdash S : \sigma \quad \Psi; \cdot \vdash w_i : \tau_i \text{ wval} \quad (\text{for } 1 \leq i \leq n)}{\Psi \vdash \{\mathtt{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_m \mapsto w_m\} : \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}} \ (m \geq n)$$

$$\boxed{\Psi \vdash h : \tau \text{ hval} \quad \Psi; \Delta \vdash w : \tau \text{ wval} \quad \Psi; \Delta \vdash w : \tau^{\varphi} \text{ fwval} \quad \Psi; \Delta; \Gamma \vdash v : \tau}$$

$$\frac{\Psi; \cdot \vdash w_i : \tau_i^{\varphi_i} \text{ fwval}}{\Psi \vdash \langle w_1, \ldots, w_n \rangle : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \text{ hval}} \qquad \frac{\Delta \vdash \Gamma \quad \Psi; \Delta; \Gamma \vdash I}{\Psi \vdash \mathtt{code}[\Delta]\Gamma.I : \forall[\Delta].\Gamma \text{ hval}}$$

$$\frac{\Delta \vdash \tau_1 \leq \tau_2}{\Psi; \Delta \vdash \ell : \tau_2 \text{ wval}} \ (\Psi(\ell) = \tau_1) \qquad \frac{}{\Psi; \Delta \vdash i : \mathit{int} \text{ wval}}$$

$$\frac{\Delta \vdash \tau \quad \Psi; \Delta \vdash w : \forall[\alpha, \Delta'].\Gamma \text{ wval}}{\Psi; \Delta \vdash w[\tau] : \forall[\Delta'].\Gamma[\tau/\alpha] \text{ wval}} \qquad \frac{\Delta \vdash \sigma \quad \Psi; \Delta \vdash w : \forall[\rho, \Delta'].\Gamma \text{ wval}}{\Psi; \Delta \vdash w[\sigma] : \forall[\Delta'].\Gamma[\sigma/\rho] \text{ wval}}$$

$$\frac{\Delta \vdash \tau \quad \Psi; \Delta \vdash w : \tau'[\tau/\alpha] \text{ wval}}{\Psi; \Delta \vdash \mathit{pack}\ [\tau, w]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau' \text{ wval}} \qquad \frac{}{\Psi; \Delta \vdash \mathit{ns} : \mathit{ns} \text{ wval}}$$

$$\frac{\Delta \vdash \sigma}{\Psi; \Delta \vdash \mathit{ptr}(i) : \mathit{ptr}(\sigma) \text{ wval}} \ (|\sigma| = i) \qquad \frac{\Delta \vdash \tau}{\Psi; \Delta \vdash ?\tau : \tau^0 \text{ fwval}}$$

$$\frac{\Psi; \Delta \vdash w : \tau \text{ wval}}{\Psi; \Delta \vdash w : \tau^{\varphi} \text{ fwval}} \qquad \frac{}{\Psi; \Delta; \Gamma \vdash r : \tau} \ (\Gamma(r) = \tau) \qquad \frac{\Psi; \Delta \vdash w : \tau \text{ wval}}{\Psi; \Delta; \Gamma \vdash w : \tau}$$

$$\frac{\Delta \vdash \tau \quad \Psi; \Delta; \Gamma \vdash v : \forall[\alpha, \Delta'].\Gamma'}{\Psi; \Delta; \Gamma \vdash v[\tau] : \forall[\Delta'].\Gamma'[\tau/\alpha]} \qquad \frac{\Delta \vdash \sigma \quad \Psi; \Delta; \Gamma \vdash v : \forall[\rho, \Delta'].\Gamma'}{\Psi; \Delta; \Gamma \vdash v[\sigma] : \forall[\Delta'].\Gamma'[\sigma/\rho]}$$

$$\frac{\Delta \vdash \tau \quad \Psi; \Delta; \Gamma \vdash v : \tau'[\tau/\alpha]}{\Psi; \Delta; \Gamma \vdash \mathit{pack}\ [\tau, v]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}$$

$$\frac{\cdot \vdash \tau_1 = \tau_2 \quad \Psi \vdash h : \tau_2 \text{ hval}}{\Psi \vdash h : \tau_1 \text{ hval}} \qquad \frac{\Delta \vdash \tau_1 = \tau_2 \quad \Psi; \Delta \vdash w : \tau_2 \text{ wval}}{\Psi; \Delta \vdash w : \tau_1 \text{ wval}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_2 \quad \Psi; \Delta; \Gamma \vdash v : \tau_2}{\Psi; \Delta; \Gamma \vdash v : \tau_1}$$

$$\boxed{\Psi; \Delta; \Gamma \vdash I}$$

$$\frac{\Psi; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma' \quad \Psi; \Delta'; \Gamma' \vdash I}{\Psi; \Delta; \Gamma \vdash \iota; I} \qquad \frac{\Delta \vdash \Gamma_1 \leq \Gamma_2 \quad \Psi; \Delta; \Gamma_1 \vdash v : \forall[\,].\Gamma_2}{\Psi; \Delta; \Gamma_1 \vdash \mathtt{jmp}\ v}$$

$$\frac{\Delta \vdash \tau \quad \Psi; \Delta; \Gamma \vdash \mathtt{r1} : \tau}{\Psi; \Delta; \Gamma \vdash \mathtt{halt}\ [\tau]}$$

Fig. 10. STAL Static Semantics, Term Constructs except Instructions

$$\boxed{\Psi; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma'}$$

$$\frac{\Psi; \Delta; \Gamma \vdash r_s : \mathit{int} \quad \Psi; \Delta; \Gamma \vdash v : \mathit{int}}{\Psi; \Delta; \Gamma \vdash \mathit{aop}\ \ r_d, r_s, v \Rightarrow \Delta; \Gamma\{r_d{:}\mathit{int}\}}$$

$$\frac{\Psi; \Delta; \Gamma_1 \vdash r : \mathit{int} \quad \Psi; \Delta; \Gamma_1 \vdash v : \forall[\,].\Gamma_2 \quad \Delta \vdash \Gamma_1 \leq \Gamma_2}{\Psi; \Delta; \Gamma_1 \vdash \mathit{bop}\ \ r, v \Rightarrow \Delta; \Gamma_1}$$

$$\frac{\Psi; \Delta; \Gamma \vdash r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle}{\Psi; \Delta; \Gamma \vdash \mathtt{ld}\ \ r_d, r_s(i) \Rightarrow \Delta; \Gamma\{r_d{:}\tau_i\}}\ (\varphi_i = 1 \wedge 0 \leq i < n)$$

$$\frac{\Delta \vdash \tau_i}{\Psi; \Delta; \Gamma \vdash \mathtt{malloc}\ \ r[\tau_1, \ldots, \tau_n] \Rightarrow \Delta; \Gamma\{r{:}\langle \tau_1^0, \ldots, \tau_n^0 \rangle\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau}{\Psi; \Delta; \Gamma \vdash \mathtt{mov}\ \ r_d, v \Rightarrow \Delta; \Gamma\{r_d{:}\tau\}}$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash \mathtt{mov}\ \ r_d, \mathtt{sp} \Rightarrow \Delta; \Gamma\{r_d{:}\mathit{ptr}(\sigma)\}}\ (\Gamma(\mathtt{sp}) = \sigma)$$

$$\frac{\Psi; \Delta; \Gamma \vdash r_s : \mathit{ptr}(\sigma_2) \quad \Delta \vdash \sigma_1 = \sigma_3 \circ \sigma_2}{\Psi; \Delta; \Gamma \vdash \mathtt{mov}\ \mathtt{sp}, r_s \Rightarrow \Delta; \Gamma\{\mathtt{sp}{:}\sigma_2\}}\ (\Gamma(\mathtt{sp}) = \sigma_1)$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash \mathtt{salloc}\ \ n \Rightarrow \Delta; \Gamma\{\mathtt{sp}{:}\underbrace{ns{::}\cdots{::}ns}_{n}{::}\sigma\}}\ (\Gamma(\mathtt{sp}) = \sigma)$$

$$\frac{\Delta \vdash \sigma_1 = \tau_0{::}\cdots{::}\tau_{n-1}{::}\sigma_2}{\Psi; \Delta; \Gamma \vdash \mathtt{sfree}\ \ n \Rightarrow \Delta; \Gamma\{\mathtt{sp}{:}\sigma_2\}}\ (\Gamma(\mathtt{sp}) = \sigma_1)$$

$$\frac{\Delta \vdash \sigma_1 = \tau_0{::}\cdots{::}\tau_i{::}\sigma_2}{\Psi; \Delta; \Gamma \vdash \mathtt{sld}\ \ r_d, \mathtt{sp}(i) \Rightarrow \Delta; \Gamma\{r_d{:}\tau_i\}}\ (\Gamma(\mathtt{sp}) = \sigma_1 \wedge 0 \leq i)$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash r_s : \mathit{ptr}(\sigma_3) \qquad \Delta \vdash \sigma_1 = \sigma_2 \circ \sigma_3 \\ \Delta \vdash \sigma_3 = \tau_0{::}\cdots{::}\tau_i{::}\sigma_4\end{array}}{\Psi; \Delta; \Gamma \vdash \mathtt{sld}\ \ r_d, r_s(i) \Rightarrow \Delta; \Gamma\{r_d{:}\tau_i\}}\ (\Gamma(\mathtt{sp}) = \sigma_1 \wedge 0 \leq i)$$

$$\frac{\Delta \vdash \sigma_1 = \tau_0{::}\cdots{::}\tau_i{::}\sigma_2 \quad \Psi; \Delta; \Gamma \vdash r_s : \tau}{\Psi; \Delta; \Gamma \vdash \mathtt{sst}\ \ \mathtt{sp}(i), r_s \Rightarrow \Delta; \Gamma\{\mathtt{sp}{:}\tau_0{::}\cdots{::}\tau_{i-1}{::}\tau{::}\sigma_2\}}\ (\Gamma(\mathtt{sp}) = \sigma_1 \wedge 0 \leq i)$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash r_d : \mathit{ptr}(\sigma_3) \qquad \Psi; \Delta; \Gamma \vdash r_s : \tau \\ \Delta \vdash \sigma_1 = \sigma_2 \circ \sigma_3 \qquad \Delta \vdash \sigma_3 = \tau_0{::}\cdots{::}\tau_i{::}\sigma_4 \\ \Delta \vdash \sigma_5 = \tau_0{::}\cdots{::}\tau_{i-1}{::}\tau{::}\sigma_4\end{array}}{\Psi; \Delta; \Gamma \vdash \mathtt{sst}\ \ r_d(i), r_s \Rightarrow \Delta; \Gamma\{\mathtt{sp}{:}\sigma_2 \circ \sigma_5, r_d{:}\mathit{ptr}(\sigma_5)\}}\ (\Gamma(\mathtt{sp}) = \sigma_1 \wedge 0 \leq i)$$

$$\frac{\Psi; \Delta; \Gamma \vdash r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \quad \Psi; \Delta; \Gamma \vdash r_s : \tau_i}{\Psi; \Delta; \Gamma \vdash \mathtt{st}\ \ r_d(i), r_s \Rightarrow \Delta; \Gamma\{r_d{:}\langle \tau_0^{\varphi_0}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\}}\ (0 \leq i < n)$$

$$\frac{\Psi; \Delta; \Gamma \vdash v : \exists \alpha.\tau}{\Psi; \Delta; \Gamma \vdash \mathtt{unpack}\ \ [\alpha, r_d], v \Rightarrow \alpha, \Delta; \Gamma\{r_d{:}\tau\}}\ (\alpha \notin \Delta)$$

Fig. 11. STAL Static Semantics, Instructions