Stack-Based Typed Assembly Language

Greg Morrisett Karl Crary David Walker Neal Glew

November 1998
CMU-CS-98-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-98-05

Abstract

In previous work, we presented *Typed Assembly Language* (TAL). TAL is sufficiently expressive to serve as a target language for compilers of high-level languages such as ML. That 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.

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, AASERT grant N00014-95-1-0985, and ARPA grant F19628-95-C-0050. 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.

Keywords: Typed compilation, typed assembly language, certified code, polymorphic recursion, stacks.

1 Introduction and Motivation

Statically typed source languages have efficiency and software engineering advantages over their dynamically typed counterparts. Modern type-directed compilers [18, 25, 7, 32, 19, 29, 11] 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 [17, 31, 16, 4, 20], region inference [8], subsumption elimination [9, 10], and unboxing [18, 22, 28]. In many cases, without types, these transformations are less effective or simply 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 typechecker after each code transformation, and if the output fails to type-check, the developer knows that the compiler contains an internal error. Although typecheckers for decidable type systems cannot catch all compiler errors, they have proven themselves valuable debugging tools in practice [21].

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 and types cannot be 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 2 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 when continuations are heap allocated [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 stack frames. Finally, Appel and Shao [2] have argued that, at least for SML/NJ, the locality lost by heap-allocating stack 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 [13, 12] 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 [15]. 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 should not constrain those 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 [19], 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

We begin with an overview of our original typed assembly language in the absence of stacks, 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.

types	τ	$::=$	$\alpha \mid int \mid \forall[\Delta].,\ \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
label assignments	Ψ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
type assignments	Δ	$::=$	$\cdot \mid \alpha, \Delta$
register assignments	,	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
registers	r	$::=$	$\texttt{r1} \mid \texttt{r2} \mid \cdots$
word values	w	$::=$	$\ell \mid i \mid ?\tau \mid w[\tau] \mid \textit{pack}\ [\tau, w]\ \textit{as}\ \tau'$
small values	v	$::=$	$r \mid w \mid v[\tau] \mid \textit{pack}\ [\tau, v]\ \textit{as}\ \tau'$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle \mid \texttt{code}[\Delta], .I$
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	ι	$::=$	$aop\ r_d, r_s, v \mid bop\ r, v \mid \texttt{ld}\ r_d, r_s(i) \mid \texttt{malloc}\ r[\vec{\tau}] \mid$
			$\texttt{mov}\ r_d, v \mid \texttt{st}\ r_d(i), r_s \mid \texttt{unpack}\ [\alpha, r_d], v \mid$
arithmetic ops	aop	$::=$	$\texttt{add} \mid \texttt{sub} \mid \texttt{mul}$
branch ops	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)

Figure 1: Syntax of TAL

Figure 1 gives the syntax for TAL. A TAL program (P) is a triple consisting of a heap, a register file, and an instruction sequence. A register file is a mapping of registers to word-sized values. A heap is a mapping of labels to heap values (values larger than a word), which are tuples and code sequences.

The instruction set consists mostly of conventional RISC-style assembly operations, including arithmetic, branches, loads, and stores. One exception, the `unpack` $[\alpha, r], v$ instruction, unpacks a value v having existential type, binding α to its hidden type in the instructions that follow, and placing the underlying value in register r. On an untyped machine, where the moving of types is immaterial, this can be implemented by a simple move instruction. The other non-standard instruction is `malloc`, which allocates memory in the heap. On a conventional machine, this instruction would be replaced by the appropriate code to allocate memory. Evaluation of TAL programs is specified as a deterministic small-step operational semantics that maps programs to programs (details appear in Morrisett *et al.* [23]).

The unusual types in TAL are for tuples and code blocks. 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 $\langle int^0, int^1 \rangle$, 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 $\langle int^1, int^1 \rangle$, 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.

Code types ($\forall[\alpha_1,\ldots,\alpha_n].,$) describe code blocks ($\texttt{code}[\alpha_1,\ldots,\alpha_n],\ .I$), which are made from instruction sequences I that expect a register file of type , . In other words, , 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 $\alpha_1,\ldots,\alpha_n$ are bound (and abstract) in , and I, and are instantiated at the call site to the function. As usual, we consider alpha-equivalent expressions to be identical; however, register names are *not* bound variables and do not alpha-vary. We also consider label assignments, register assignments, heaps, and register files equivalent when they differ only in the orderings of their fields. When Δ is empty, we often abbreviate $\forall[\Delta].,$ as simply , .

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 [20], 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] Then, after closures are introduced, the code has 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:

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

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

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

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 2 gives the CPS-based TAL code for the following ML expression which computes the factorial of 6:

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

3 Stacks

In this section, we show how to extend TAL to obtain 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.

3.1 Basic Developments

Operationally we model stacks (S) as lists of word-sized values. There are four new instructions that manipulate the stack: The `salloc` n instruction enlarges the stack by n words. The new stack slots are uninitialized, which we formalize by filling them with nonsense words (ns). On a conventional machine, assuming stacks grow toward lower addresses, an `salloc` operation would correspond to subtracting n from the stack pointer. The `sfree` n instruction removes the top n words from the stack, and corresponds to adding n to the stack pointer. The `sld` $r, \texttt{sp}(i)$ instruction loads the i^{th} word (from zero) of the stack into register r, whereas the `sst` $\texttt{sp}(i), r$ stores register r into the i^{th} word.

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

- `sfree` n and the stack does not contain at least n words, or
- `sld` $r, \texttt{sp}(i)$ or `sst` $\texttt{sp}(i), r$ and the stack does not contain at least $i+1$ words.

As usual, a type safety theorem (Theorem A.1) dictates that no well-formed program can become stuck.

```
$(H, \{\}, I)$ where
$H$ = l_fact:
          code[]{r1:$\langle\rangle$,r2:$int$,r3:$\tau_k$}.
            bneq r2,l_nonzero
            unpack [$\alpha$,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:$\langle\rangle$,r2:$int$,r3:$\tau_k$}.
            sub r4,r2,1                        % $n-1$
            malloc r5[$int, \tau_k$]                 % create environment for cont in r5
            st r5(0),r2                        % store $n$ into environment
            st r5(1),r3                        % store $k$ into environment
            malloc r3 [$\forall$[].{r1:$\langle int^1, \tau_k^1\rangle$,r2:$int$}, $\langle int^1, \tau_k^1\rangle$]   % create cont closure in r3
            mov r2,l_cont
            st r3(0),r2                        % store cont code
            st r3(1),r5                        % store environment $\langle n, k\rangle$
            mov r2,r4                          % arg := $n-1$
            mov r3,$pack$ [$\langle int^1, \tau_k^1\rangle$,r3] $as$ $\tau_k$    % abstract the type of the environment
            jmp l_fact                         % recursive call
      l_cont:
          code[]{r1:$\langle int^1, \tau_k^1\rangle$,r2:$int$}.         % r2 contains $(n-1)!$
            ld r3,r1(0)                        % retrieve $n$
            ld r4,r1(1)                        % retrieve $k$
            mul r2,r3,r2                       % $n \times (n-1)!$
            unpack [$\alpha$,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:$\langle\rangle$,r2:$int$}.
            mov r1,r2
            halt[$int$]                          % halt with result in r1

and $I$ =   malloc r1[]                        % create empty environment ($\langle\rangle$)
            malloc r2[]                        % create another empty environment
            malloc r3[$\forall$[].{r1:$\langle\rangle$,r2:$int$}, $\langle\rangle$]      % create halt closure in r3
            mov r4,l_halt
            st r3(0),r4                        % store cont code
            st r3(1),r2                        % store environment $\langle\rangle$
            mov r2,6                           % load argument (6)
            mov r3,$pack$ [$\langle\rangle$,r3] $as$ $\tau_k$         % abstract the type of the environment
            jmp l_fact                         % begin fact with
                                               % {r1 = $\langle\rangle$, r2 = 6, $r3$ = haltcont}
and $\tau_k$ = $\exists\alpha.\langle\forall$[].{r1:$\alpha$,r2:$int$}$^1, \alpha^1\rangle$
```

Figure 2: Typed Assembly Code for Factorial

types	τ	$::=$	$\cdots \mid ns$
stack types	σ	$::=$	$\rho \mid nil \mid \tau{::}\sigma$
type assignments	Δ	$::=$	$\cdots \mid \rho, \Delta$
register assignments	,	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n, \mathtt{sp}{:}\sigma\}$
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	ι	$::=$	$\cdots \mid \mathtt{salloc}\ n \mid \mathtt{sfree}\ n \mid \mathtt{sld}\ r_d, \mathtt{sp}(i) \mid \mathtt{sst}\ \mathtt{sp}(i), r_s$

Figure 3: Additions to TAL for Simple Stacks

Stacks are classified by *stack types* (σ), which include *nil* and $\tau{::}\sigma$. The former describes the empty stack and the latter describes a stack of the form $w{::}S$ where w has type τ and S has type σ. Stack types also include stack type variables (ρ), which may be used to abstract the tail of a stack type. The ability to abstract stack types 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 (,) mapping registers to types. However, , 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. In the interest of clarity, from time to time we will give registers names (such as `ra` or `re`) instead of numbers.

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.

```
                            % $\sigma$
salloc  $n$                 % $ns::ns::\cdots::ns::\sigma$
sst     sp(0), $r_1$        % $\tau_1::ns::\cdots::ns::\sigma$
  ⋮
sst     sp($n-1$), $r_n$    % $\tau_1::\tau_2::\cdots::\tau_n::\sigma$
code for $e$                % $\tau_1::\tau_2::\cdots::\tau_n::\sigma$
sld     $r_1$, sp(0)        % $\tau_1::\tau_2::\cdots::\tau_n::\sigma$
  ⋮
sld     $r_n$, sp($n-1$)    % $\tau_1::\tau_2::\cdots::\tau_n::\sigma$
sfree   $n$                 % $\sigma$
```

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 `salloc` 1 followed by a store to `sp`(0), whereas the latter is a load from `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.

3.2 Stack Polymorphism

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 that there is one integer argument and one unit result, both of which are passed in register `r1`, and that 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:

$$\{\texttt{r1}{:}\mathit{int}, \mathsf{sp}{:}\sigma, \mathsf{ra}{:}\{\texttt{r1}{:}\langle\rangle, \mathsf{sp}{:}\sigma\}\}$$

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 important reasons:

- Nothing prevents the function 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.
- Such a function can only be invoked from states where the entire stack is described exactly by σ. This effectively limits invocation of the procedure to a single, pre-determined point in the execution of the program. For example, there is no way for a procedure to push its return address onto the stack and to jump to itself (*i.e.,* to recurse).

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

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

To invoke a function having 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.

This mechanism addresses the first problem 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]

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

$(H, \{\mathtt{sp} \mapsto \mathit{nil}\}, I)$ where

```
H = l_fact:
        code[$\rho$]{r1 : $\langle\rangle$, r2 : $\mathit{int}$, sp : $\rho$, ra : $\tau_\rho$}.
          bneq r2,l_nonzero[$\rho$]      % if $n = 0$ continue
          mov r1,1                       % result is 1
          jmp ra                         % return
    l_nonzero:
        code[$\rho$]{r1 : $\langle\rangle$, r2 : $\mathit{int}$, sp : $\rho$, ra : $\tau_\rho$}.
          sub r3,r2,1                    % $n - 1$
          salloc 2                       % allocate stack space for $n$ and the return address
          sst sp(0),r2                   % save $n$
          sst sp(1),ra                   % save return address
          mov r2,r3
          mov ra,l_cont[$\rho$]
          jmp l_fact[$\mathit{int}::\tau_\rho::\rho$]    % recursive call to $\mathit{fact}$ with $n - 1$,
                                         % abstracting saved data atop the stack
    l_cont:
        code[$\rho$]{r1 : $\mathit{int}$, sp : $\mathit{int}::\tau_\rho::\rho$}.
          sld r2,sp(0)                   % restore $n$
          sld ra,sp(1)                   % restore return address
          sfree 2
          mul r1,r2,r1                   % $n \times (n-1)!$
          jmp ra                         % return
    l_halt:
        code[]{r1 : $\mathit{int}$, sp : $\mathit{nil}$}.
          halt[$\mathit{int}$]

and I =   malloc r1[]                    % create empty environment
          mov r2,6                       % argument
          mov ra,l_halt                  % return address for initial call
          jmp l_fact[$\mathit{nil}$]
```

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

Figure 4: STAL Factorial Example

The second problem is also solved because the stack type variable may be instantiated in multiple 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 own return address on the stack.

Figure 4 gives stack-based code for the factorial program. The function is invoked by moving its environment (an empty tuple, since factorial has no free variables) 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. In so doing, the code must use stack polymorphism to account for its additions to the stack.

3.3 Calling Conventions

It is interesting to note that the stack-based code is quite similar to the heap-based code of Figure 2. In a sense, the stack-based 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 2:

$$\{\mathtt{r1}{:}\langle\rangle, \mathtt{r2}{:}\mathit{int}, \mathtt{ra}{:}\exists\alpha.\langle\{\mathtt{r1}{:}\alpha, \mathtt{r2}{:}\mathit{int}\}^1, \alpha^1\rangle\}$$

We could have used an alternative approach where the continuation closure is passed unboxed in separate registers. To do so, the function's type must perform the duty of abstracting α, since the continuation's code and environment must each still refer to the same α:

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

Now recall the type of the corresponding STAL code:

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

These types are essentially the same! Indeed, the only difference between continuation-passing execution and stack-based execution is that in stack-based execution continuations are unboxed and their environments are allocated on the stack. This connection is among the folklore of continuation-passing compilers, but the similarity of the two types in STAL summarizes the connection particularly succinctly.

The STAL types discussed above each serve the purpose of formally specifying a procedure calling convention, specifying the usage of the registers and stack on entry to and return from a procedure. In each of the above calling conventions, the environment, argument, and result are passed in registers. We also can specify that the environment, argument, return address, and the result are all passed on the stack. In this calling convention, the factorial function has type (remember that the convention for the result is given by the type of the return address):

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

These types do not constrain optimizations that respect the given calling conventions. For instance, tail-calls can be eliminated in CPS (the first two conventions) simply by forwarding the continuation to the next function. In a stack-based system (the second two), the type system similarly allows us (if necessary) to pop the current activation frame off the stack and to push 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 tail-call to self.

Types may express more complex conventions as well. For example, callee-saves registers (registers whose values must be preserved across function calls) can be handled in the same fashion as the stack pointer: A function's type abstracts the type of the callee-saves register and provides 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:

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

Alternatively, with boxed, heap-allocated closures, we would use the type:

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

This is the type that corresponds to the callee-saves protocol of Appel and Shao [1]. Again the close correspondence holds between the stack- and heap-oriented types. Indeed, either one can be obtained mechanically from the other. Thus this correspondence allows transformations developed for heap-based compilers to be used in traditional stack-based compilers and vice versa.

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 a variety of sorts 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):

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

As before, the caller could unbox the continuations:

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

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

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

Unfortunately, this calling convention uses two stack pointers, and there is only one stack. 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:

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

This type uses the notion of a *compound stack:* When σ_1 and σ_2 are stack types, the compound 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.

types	τ	$::=$	$\cdots \mid ptr(\sigma)$
stack types	σ	$::=$	$\cdots \mid \sigma_1 \circ \sigma_2$
word values	w	$::=$	$\cdots \mid ptr(i)$
instructions	ι	$::=$	$\cdots \mid$ `mov` r_d, `sp` $\mid$ `mov sp`, r_s $\mid$ `sld` $r_d, r_s(i)$ $\mid$ `sst` $r_d(i), r_s$

Figure 5: Additions to TAL for Compound Stacks

To raise an exception, the exception is placed in `r1` and control is transferred 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 `res`, 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 compound stacks is summarized in Figure 5. The type constructs $\sigma_1 \circ \sigma_2$ and $ptr(\sigma)$ 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. Of course, on a real machine, such a value would be implemented by an actual pointer. The instructions `mov` r_d, `sp` and `mov sp`, r_s save and restore the stack pointer, and the instructions `sld` $r_d, r_s(i)$ and `sst` $r_d(i), r_s$ allow for loading from and storing to pointers.

The introduction of pointers into the stack raises a delicate issue for the type 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 with sp : $\tau::\sigma$, sp $\mapsto w::S$ $(\tau \neq ns)$
mov r1,sp    % r1 : $ptr(\tau::\sigma)$
sfree 1      % sp : $\sigma$, sp $\mapsto S$
salloc 1     % sp : $ns::\sigma$, sp $\mapsto ns::S$
sld r2,r1(0) % r2 : $\tau$ but r2 $\mapsto ns$
```

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

To prevent erroneous loads of this sort, the type system requires that the pointer r_s be *valid* when used in the instructions `sld` $r_d, r_s(i)$, `sst` $r_d(i), r_s$, and `mov sp`, r_s. An invariant of the type 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 $ptr(\sigma_2)$, 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 is rejected because `r1`'s type $\tau::\sigma$ is not a tail of `sp`'s type, $ns::\sigma$.

4.3 Using Compound Stacks

Recall the type for integer to unit functions in the presence of exceptions:

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

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 subsequent 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, to allow a different view of the stack for each datum located on the stack, but we have not explored the ramifications of this enhancement.

5 Compiling to STAL

We make the discussion of the preceding chapters concrete by presenting a formal translation that compiles a high-level programming language with integer exceptions into STAL. The syntax of the source language appears in Figure 6. The static semantics of the source language is given two judgments, a type formation judgment $\Delta \vdash \tau$ type and a term formation judgment $\Delta;\,,\ \vdash e : \tau$. The rules for the former are completely standard and are omitted; the rules for the latter can be obtained by dropping the translation portion ($\rightsquigarrow C$) from the translating rules that follow. Closure conversion [20, 23] presents no interesting issues particular to this translation, so in the interest of simplicity, we assume it has already been performed. Consequently, well-typed function terms ($\mathit{fix}\ x(x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau.e$) must be closed.

In order to illustrate use of the stack, the translation uses a simple stack-oriented strategy. No register allocation is performed; all arguments and most temporaries are stored on the stack. Also, no particular effort is made to be efficient.

The translation of source types to STAL types is given below; the interest case is the calling convention for functions. The calling convention abstracts a set of type variables (Δ), and abstracts

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \alpha \mid \textit{int} \mid \forall[\Delta].(\tau_1,\ldots,\tau_n) \to \tau \mid \langle \tau_1,\ldots,\tau_n \rangle \\
\textit{terms} & e & ::= & x \mid i \mid \textit{fix}\ x(x_1{:}\tau_1,\ldots,x_n{:}\tau_n){:}\tau.e \mid e_1 e_2 \mid e[\tau] \mid \\
 & & & \quad \langle e_1,\ldots,e_n\rangle \mid \pi_i(e) \mid e_1\ p\ e_2 \mid \textit{if0}(e_1,e_2,e_3) \\
 & & & \quad \textit{raise}\,[\tau]\ e \mid \textit{try}\ e_1\ \textit{handle}\ x \Rightarrow e_2 \\
\textit{primitives} & p & ::= & + \mid - \mid \times \\
\textit{type contexts} & \Delta & ::= & \alpha_1,\ldots,\alpha_n \\
\textit{value contexts} & \Gamma & ::= & x_1{:}\tau_1,\ldots,x_n{:}\tau_n
\end{array}$$

Figure 6: Source Syntax

stack type variables representing the front (ρ_1) and back (ρ_2) of the caller's stack. The front of the stack consists of all of the caller's stack up to the enclosing exception handler, and the back consists of everything behind the enclosing exception handler. On entry to a function, the stack is to contain the function's arguments on top of the caller's stack. The exception register, `re`, and the exception stack register, `res`, contain pointers to the enclosing exception handler and its stack, respectively. Finally, the return address register, `ra`, contains a return pointer that expects the result value in `r1`, the same stack except the arguments removed, and the exception registers unchanged.[3]

$$\begin{array}{rcl}
|\alpha| & = & \alpha \\
|\textit{int}| & = & \textit{int} \\
|\langle \tau_1,\ldots,\tau_n\rangle| & = & \langle |\tau_1|^1,\ldots,|\tau_n|^1\rangle \\
|\forall[\Delta].(\tau_1,\ldots,\tau_n)\to\tau| & = & \forall[\Delta,\rho_1,\rho_2]. \\
 & & \quad \{\texttt{sp} : (|\tau_n|{::}\cdots{::}|\tau_1|{::}\rho_1 \circ \rho_2), \\
 & & \quad\ \texttt{ra} : \{\texttt{r1}{:}|\tau|, \texttt{sp}{:}\rho_1\circ\rho_2, \texttt{re}{:}\{\texttt{r1}{:}\textit{int}, \texttt{sp}{:}\rho_2\}, \texttt{res}{:}\textit{ptr}(\rho_2)\}, \\
 & & \quad\ \texttt{re} : \{\texttt{r1}{:}\textit{int}, \texttt{sp}{:}\rho_2\}, \\
 & & \quad\ \texttt{res} : \textit{ptr}(\rho_2)\}
\end{array}$$

The translation of source terms is given as a type-directed translation governed by the judgment $\Delta;\Gamma \vdash e : \tau \rightsquigarrow C$. The judgment is read as follows; in type context Δ and values context Γ, the term e has type τ and translates to a STAL code sequence C. Without the translation $\rightsquigarrow C$, this judgment specifies the static semantics of the source language. Therefore it is clear that any well-typed source term is compiled by this translation.

In order to simplify the translation's presentation, we use code sequences that are permitted to contain address labels after `jmp` and `halt` instructions:

$$\textit{code sequences} \quad C \ ::= \ \cdot \mid \iota; C \mid \texttt{jmp}\ v;\ \ell{:}\texttt{code}[\Delta]\Gamma.C \mid \texttt{halt}[\tau];\ \ell{:}\texttt{code}[\Delta]\Gamma.C$$

These code sequences are appended together to form a conglomerate code block of the form $I; \ell_1{:}h_1; \ldots; \ell_n{:}h_n$. Such a block is converted to an official STAL program by heap allocating all but the first segment of instructions. Also in the interest of simplicity, we assume that all labels used in the translation are fresh, and we use `push` and `pop` instructions as shorthand for the appropriate allocate/store and load/free sequences.

[3]Note that this type does not protect the caller from modification of the exception register. The calling convention could be rewritten to provide this protection, but we have not done so as it would significantly complicate the presentation.

Code sequences produced by the translation assume the following preconditions: If $\Delta; , \vdash e : \tau \leadsto C$, then C has free type variables contained in Δ, has free stack type variables ρ_1, ρ_2 and ρ_3, and expects a register file with type:

$$\begin{array}{l}\{\mathtt{sp} : \rho_3 \circ |, | \circ \rho_1 \circ \rho_2, \\ \ \mathtt{fp} : \mathit{ptr}(|, | \circ \rho_1 \circ \rho_2), \\ \ \mathtt{re} : \{\mathtt{r1}{:}\mathit{int}, \mathtt{sp}{:}\rho_2\}, \\ \ \mathtt{res} : \mathit{ptr}(\rho_2)\}\end{array}$$

As discussed above, the stack contains the value variables $(|, |)$ in front of the the caller's stack $(\rho_1 \circ \rho_2)$. The stack type $|, |$ specifying the argument portion of the stack is defined by:

$$|x_1{:}\tau_1, \ldots, x_n{:}\tau_n| \quad = \quad |\tau_n|{::}\cdots{::}|\tau_1|{::}\mathit{nil}$$

Upon entry to C, the stack also contains an unknown series of temporaries specified by ρ_3. The variable ρ_3 is free in C, so appropriate substitutions for ρ_3 allow C to be used in a variety of different environments. Since the number of temporaries is unknown, C also expects a frame pointer, `fp`, to point past them to the variables. As usual, the exception registers point to the enclosing exception handler and its stack. At the end of C, the register file has the same type, with the addition that `r1` contains the term's result value of type $|\tau|$.

With these preliminaries established, we are ready to present the translation's rules. The code for a variable reference simply finds the value at an appropriate offset from the frame pointer and places it in `r1`:

$$\frac{}{\Delta; (x_{n-1}{:}\tau_{n-1}, \ldots, x_0{:}\tau_0) \vdash x_i : \tau_i \leadsto \mathtt{sld\ r1, fp}(i)} \; (0 \leq i < n)$$

A simple example of an operation that stores temporary information on the stack is arithmetic. The translation of $e_1\, p\, e_2$ computes the value of e_1 (placing it in `r1`), then pushes it onto the stack and computes the value of e_2. During the second computation, there is an additional temporary word (on top of those specified by ρ_3), so in that second computation ρ_3 is instantiated with $\mathit{int}{::}\rho_3$; this indicates that the number of temporaries is still unknown, but is one word more than externally. After computing the value of e_2, the code retrieves the first value from the stack and performs the arithmetic operation.

$$\frac{\Delta; , \vdash e_1 : \mathit{int} \leadsto C_1 \quad \Delta; , \vdash e_2 : \mathit{int} \leadsto C_2}{\begin{array}{ll}\Delta; , \vdash e_1\, p\, e_2 : \mathit{int} \leadsto & C_1 \\ & \mathtt{push\ r1} \\ & C_2[\mathit{int}{::}\rho_3/\rho_3] \\ & \mathtt{pop\ r2} \\ & \mathtt{arith}_p\ \mathtt{r1, r2, r1}\end{array}} \; \begin{pmatrix}\mathtt{arith}_+ = \mathtt{add} \\ \mathtt{arith}_- = \mathtt{sub} \\ \mathtt{arith}_\times = \mathtt{mul}\end{pmatrix}$$

Function calls are compiled (Figure 7) by evaluating the function and each of the arguments, placing their values on the stack. Then the function pointer is retrieved, the frame pointer is stored on the stack (above the arguments), a return address is loaded into `ra`, and the call is made. In the call, the front of the stack (ρ_1) is instantiated according to the current stack, which then contains the current caller's frame pointer, temporaries, and arguments, in addition to the previous caller's stack. The exception handler is unchanged so the back of the stack (ρ_2) is as well.

The code for a function (Figure 8), before executing the code for the body, must establish the body's preconditions. It does so by pushing on the stack the recursion pointer (the one value variable that

$$\frac{\Delta; ,\ \vdash e : (\tau_1, \ldots, \tau_n) \to \tau \leadsto C \quad \Delta; ,\ \vdash e_i : \tau_i \leadsto C_i}{\Delta; ,\ \vdash e(e_1, \ldots, e_n) : \tau \leadsto}$$

```
        ;; sp : $\rho_3 \circ |, | \circ \rho_1 \circ \rho_2$
        $C$
        push r1
        $C_1[\tau_{\text{fun}}::\rho_3/\rho_3]$
        push r1
        $\vdots$
        $C_n[|\tau_{n-1}|::\cdots::|\tau_1|::\tau_{\text{fun}}::\rho_3/\rho_3]$
        push r1
        ;; sp : $|\tau_n|::\cdots|\tau_1|::\tau_{\text{fun}}::\rho_3 \circ |, | \circ \rho_1 \circ \rho_2$
        sld r1, sp($n$)       ;; recover call address
        sst sp($n$), fp       ;; save frame pointer
        ;; sp : $|\tau_n|::\cdots|\tau_1|::ptr(|, | \circ \rho_1 \circ \rho_2)::\rho_3 \circ |, | \circ \rho_1 \circ \rho_2$
        mov ra, $\ell_{\text{return}}[\Delta, \rho_1, \rho_2]$
        jmp r1[$ptr(|, | \circ \rho_1 \circ \rho_2)::\rho_3 \circ |, | \circ \rho_1, \rho_2$]
    $\ell_{\text{return}}$ : code[$\Delta, \rho_1, \rho_2$]{r1 : $|\tau|$,
                           sp : $ptr(|, | \circ \rho_1 \circ \rho_2)::\rho_3 \circ |, | \circ \rho_1 \circ \rho_2$,
                           re : {r1:$int$, sp:$\rho_2$},
                           res : $ptr(\rho_2)$}.
        pop fp     ;; recover frame pointer
  (where $\tau_{\text{fun}}$  =  $|(\tau_1, \ldots, \tau_n) \to \tau|$
                    =  $\forall[\rho_1, \rho_2]$.{sp : ($|\tau_n|::\cdots::|\tau_1|::\rho_1 \circ \rho_2$),
                           ra : {r1:$|\tau|$, sp:$\rho_1 \circ \rho_2$, re:{r1:$int$, sp:$\rho_2$}, res:$ptr(\rho_2)$},
                           re : {r1:$int$, sp:$\rho_2$},
                           res : $ptr(\rho_2)$})
```

Figure 7: Function Call Compilation

$$\frac{\vec{\alpha} \vdash \tau_i \text{ type} \quad \vec{\alpha}; (x_1{:}\tau_1, \ldots, x_n{:}\tau_n, x{:}\forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau) \vdash e : \tau \leadsto C}{\begin{array}{l}
\Delta; , \vdash \mathit{fix}\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n){:}\tau.e : \forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau \leadsto \\
\qquad\qquad \texttt{jmp}\ \ell_{\text{skip}}[\Delta, \rho_1, \rho_2] \\
\qquad\quad \ell_{\text{fun}} : \texttt{code}[\vec{\alpha}, \rho_1, \rho_2]\{\texttt{sp} : |\tau_n| {::} \cdots |\tau_1| {::} \rho_1 \circ \rho_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{ra} : \tau_{\text{return}}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{re} : \{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{res} : \mathit{ptr}(\rho_2)\}. \\
\qquad\qquad \texttt{mov r1}, \ell_{\text{fun}} \\
\qquad\qquad \texttt{push r1} \qquad ;;\ \text{add recursion address to context} \\
\qquad\qquad \texttt{mov fp}, \texttt{sp} \quad\ ;;\ \text{create frame pointer} \\
\qquad\qquad \texttt{push ra} \qquad ;;\ \text{save return address} \\
\qquad\qquad ;;\ \texttt{sp} : \tau_{\text{return}} {::} |\forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau| {::} |\tau_n| {::} \cdots {::} |\tau_1| {::} \rho_1 \circ \rho_2 \\
\qquad\qquad ;;\ \texttt{fp} : \quad \mathit{ptr}(|\forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \tau| {::} |\tau_n| {::} \cdots {::} |\tau_1| {::} \rho_1 \circ \rho_2) \\
\qquad\qquad C[\tau_{\text{return}} {::} \mathit{nil}/\rho_3] \\
\qquad\qquad \texttt{pop ra} \\
\qquad\qquad \texttt{sfree}\ n+1 \\
\qquad\qquad \texttt{jmp ra} \\
\qquad\quad \ell_{\text{skip}} : \texttt{code}[\Delta, \rho_1, \rho_2]\{\texttt{sp} : \rho_3 \circ |, | \circ \rho_1 \circ \rho_2, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{fp} : \mathit{ptr}(|, | \circ \rho_1 \circ \rho_2), \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{re} : \{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{res} : \mathit{ptr}(\rho_2)\}. \\
\qquad\qquad \texttt{mov r1}, \ell_{\text{fun}} \\
\qquad (\text{where } \tau_{\text{return}} = \{\texttt{r1}{:}|\tau|, \texttt{sp}{:}\rho_1 \circ \rho_2, \texttt{re}{:}\{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \texttt{res}{:}\mathit{ptr}(\rho_2)\})
\end{array}}$$

Figure 8: Function Compilation

is not an argument), saving the return address, and creating a frame pointer. After executing the body, it retrieves the return address, frees the variables (in accordance with the calling convention), and jumps to the return address.

The remaining non-exception constructs are dealt with in a straightforward manner, and are shown in Figure 9. To raise an exception is also straightforward. After computing the exception packet (always an integer in this language), the entire front of the stack is discarded by moving the exception stack register `res` into `sp`, and then the exception handler is called. Any code following the raise is dead; the postconditions (including a "result" value of type $|\tau|$) are established by inserting a label that is never called:

$$\frac{\Delta \vdash \tau \text{ type} \quad \Delta; , \vdash e : \mathit{int} \leadsto C}{\begin{array}{ll}
\Delta; , \vdash \mathit{raise}\,[\tau]\,e : \tau \leadsto & C \\
& \texttt{mov sp}, \texttt{res} \\
& \texttt{jmp re} \\
& \ell_{\text{deadcode}} : \texttt{code}[\Delta, \rho_1, \rho_2]\{\texttt{r1} : |\tau|, \\
& \qquad\qquad\qquad\qquad\qquad \texttt{sp} : \rho_3 \circ |, | \circ \rho_1 \circ \rho_2, \\
& \qquad\qquad\qquad\qquad\qquad \texttt{fp} : \mathit{ptr}(|, | \circ \rho_1 \circ \rho_2), \\
& \qquad\qquad\qquad\qquad\qquad \texttt{re} : \{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \\
& \qquad\qquad\qquad\qquad\qquad \texttt{res} : \mathit{ptr}(\rho_2)\}.
\end{array}}$$

$$\frac{}{\Delta; ,\ \vdash i : \mathit{int} \leadsto \texttt{mov r1}, i}$$

$$\frac{\Delta \vdash \tau' \text{ type} \quad \Delta; ,\ \vdash e : \forall[\alpha, \Delta].(\tau_1, \ldots, \tau_n) \to \tau \leadsto C}{\begin{array}{l} \Delta; ,\ \vdash e[\tau'] : \forall[\Delta]((\tau_1, \ldots, \tau_n) \to \tau)[\tau'/\alpha] \leadsto C \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{mov r1}, \texttt{r1}[|\tau'|] \end{array}}$$

$$\frac{\Delta; ,\ \vdash e_i : \tau_i \leadsto C_i}{\begin{array}{ll} \Delta; ,\ \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle \leadsto & C_1 \\ & \texttt{push r1} \\ & \vdots \\ & C_n[|\tau_{n-1}|{::}\cdots{::}|\tau_1|{::}\rho_3/\rho_3] \\ & \texttt{push r1} \\ & ;;\ \texttt{sp} : |\tau_n|{::}\cdots|\tau_1|{::}\rho_3 \circ |,\ | \circ \rho_1 \circ \rho_2 \\ & \texttt{malloc r1}[|\tau_1|, \ldots, |\tau_n|] \\ & \texttt{pop r2} \\ & \texttt{st r1}(n-1), \texttt{r2} \\ & \vdots \\ & \texttt{pop r2} \\ & \texttt{st r1}(0), \texttt{r2} \end{array}}$$

$$\frac{\Delta; ,\ \vdash e : \langle \tau_1, \ldots, \tau_n \rangle \leadsto C}{\begin{array}{l} \Delta; ,\ \vdash \pi_i(e) : \tau_i \leadsto C \\ \qquad\qquad\qquad \texttt{ld r1}, \texttt{r1}(i-1) \end{array}} \ (1 \leq i \leq n)$$

$$\frac{\Delta; ,\ \vdash e_1 : \mathit{int} \leadsto C_1 \quad \Delta; ,\ \vdash e_2 : \tau \leadsto C_2 \quad \Delta; ,\ \vdash e_3 : \tau \leadsto C_3}{\begin{array}{ll} \Delta; ,\ \vdash \mathit{if0}(e_1, e_2, e_3) : \tau \leadsto & C_1 \\ & \texttt{bneq r1}, \ell_{\text{nonzero}}[\Delta, \rho_1, \rho_2] \\ & C_2 \\ & \texttt{jmp } \ell_{\text{skip}}[\Delta, \rho_1, \rho_2] \\ & \ell_{\text{nonzero}} : \texttt{code}[\Delta, \rho_1, \rho_2]\{\texttt{sp} : \rho_3 \circ |,\ | \circ \rho_1 \circ \rho_2, \\ & \qquad\qquad\qquad\qquad \texttt{fp} : \mathit{ptr}(|,\ | \circ \rho_1 \circ \rho_2), \\ & \qquad\qquad\qquad\qquad \texttt{re} : \{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \\ & \qquad\qquad\qquad\qquad \texttt{res} : \mathit{ptr}(\rho_2)\}. \\ & C_3 \\ & \texttt{jmp } \ell_{\text{skip}}[\Delta, \rho_1, \rho_2] \\ & \ell_{\text{skip}} : \texttt{code}[\Delta, \rho_1, \rho_2]\{\texttt{r1} : |\tau|, \\ & \qquad\qquad\qquad \texttt{sp} : \rho_3 \circ |,\ | \circ \rho_1 \circ \rho_2, \\ & \qquad\qquad\qquad \texttt{fp} : \mathit{ptr}(|,\ | \circ \rho_1 \circ \rho_2), \\ & \qquad\qquad\qquad \texttt{re} : \{\texttt{r1}{:}\mathit{int}, \texttt{sp}{:}\rho_2\}, \\ & \qquad\qquad\qquad \texttt{res} : \mathit{ptr}(\rho_2)\}. \end{array}}$$

Figure 9: Integer literal, Instantiation, Tuple, and Branching Compilation

The code for exception handling (Figure 10) is long, but not complicated. First the old exception registers and frame pointer are saved on the stack, and the new exception handler is installed. The precondition for translated terms requires that the variables be in front of the handler's stack so they must be copied to the top of the stack and a new frame pointer must be created.[4] The body is then executed, with the stack type variables instantiated so that the temporaries and front are empty, and the back consists of everything except the copied variables. Either the body is finished successfully or control is transferred to the exception handler. In either case the original state is restored, and if an exception was raised, the handler executes before proceeding.

The remaining rule is the sole rule for the judgment $\vdash e$ program $\rightsquigarrow P$. This judgment states that e is a valid program in the source language, and that P is a STAL program that computes the value of e. The rule establishes the translation's precondition by installing a default exception handler and creating a frame pointer, and bundles up the resulting code as a STAL program:

$$\frac{\emptyset;\emptyset \vdash e : \mathit{int} \rightsquigarrow C}{\vdash e \text{ program} \rightsquigarrow \left|\begin{array}{l} \quad\texttt{mov re}, \ell_{\text{uncaught}} \\ \quad\texttt{mov res}, \texttt{sp} \\ \quad\texttt{mov fp}, \texttt{sp} \\ \quad C[\mathit{nil}, \mathit{nil}, \mathit{nil}/\rho_3, \rho_1, \rho_2] \\ \quad\texttt{halt}[\mathit{int}] \\ \ell_{\text{uncaught}} : \texttt{code}[\,]\{\texttt{sp}{:}\mathit{nil}, \texttt{r1}{:}\mathit{int}\}. \\ \quad\texttt{halt}[\mathit{int}] \end{array}\right|}$$

$$\text{where } |I;\, \ell_1{:}h_1\,; \ldots ;\, \ell_n{:}h_n| = (\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}, \{\texttt{sp} \mapsto \mathit{nil}\}, I)$$

Proposition 5.1 (Type Correctness) *If* $\vdash e$ program $\rightsquigarrow P$ *then* $\vdash P$.

6 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 [19]. 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 the Java Virtual Machine). In contrast, our treatment supports polymorphic stack recursion, and hence procedure calls can be encoded using existing assembly-language primitives.

More recently, O'Callahan [24] has used the mechanisms in this paper to devise an alternative, simpler type system for Java bytecodes that differs from the Java bytecode verifier's discipline [19].

[4]This is an example of when it is inconvenient that stack types specify the order in which data appear on the stack. In fact, this inefficiency can be removed using a more complicated precondition, but in the interest of clarity we have not done so.

$$\frac{\Delta; ,\ \vdash e : \tau \rightsquigarrow C \quad \Delta; ,\ , x{:}int \vdash e' : \tau \rightsquigarrow C'}{\Delta; ,\ \vdash \mathit{try}\ e\ \mathit{handle}\ x \Rightarrow e' : \tau \rightsquigarrow}$$

```
            push res        ;; save old handler and frame pointer
            push re
            push fp
            ;; sp : $\sigma_{\text{handler}}$
            ;; install new handler
            mov res, sp                        ;; res : $ptr(\sigma_{\text{handler}})$
            mov re, $\ell_{\text{handle}}[\Delta, \rho_1, \rho_2]$       ;; re : $\{$r1:$int$, sp:$\sigma_{\text{handler}}\}$
            ;; to fit convention, copy arguments below the new handler's stack
            sld r1, fp$(n-1)$
            push r1
            $\vdots$
            sld r1, fp$(0)$
            push r1
            ;; sp : $|,| \circ \sigma_{\text{handler}}$
            mov fp, sp        ;; create new frame pointer
            ;; fp : $ptr(|,| \circ \sigma_{\text{handler}})$
            $C[nil, nil, \sigma_{\text{handler}}/\rho_3, \rho_1, \rho_2]$
            sfree $n$           ;; free copied arguments
            pop fp            ;; restore old handler and frame pointer
            pop re
            pop res
            jmp $\ell_{\text{skip}}[\Delta, \rho_1, \rho_2]$
        $\ell_{\text{handle}}$ : code$[\Delta, \rho_1, \rho_2]\{$r1 : $int$, sp : $\sigma_{\text{handler}}\}$
            pop fp            ;; restore old handler and frame pointer
            pop re
            pop res
            ;; sp : $\rho_3 \circ |,| \circ \rho_1 \circ \rho_2$
            $C'$
            jmp $\ell_{\text{skip}}[\Delta, \rho_1, \rho_2]$
        $\ell_{\text{skip}}$ : code$[\Delta, \rho_1, \rho_2]\{$r1 : $|\tau|$,
                                sp : $\rho_3 \circ |,| \circ \rho_1 \circ \rho_2$,
                                fp : $ptr(|,| \circ \rho_1 \circ \rho_2)$,
                                re : $\{$r1:$int$, sp:$\rho_2\}$,
                                res : $ptr(\rho_2)\}$.
```

(where $\sigma_{\text{handler}} = ptr(|,| \circ \rho_1 \circ \rho_2)::\{\texttt{r1}{:}int, \texttt{sp}{:}\rho_2\}::ptr(\rho_2)::\rho_3 \circ |,| \circ \rho_1 \circ \rho_2$
$n = sizeof(,\ ))$

Figure 10: Exception Handler Compilation

By permitting polymorphic typing of subroutines, O'Callahan's type system accepts strictly more programs while preserving safety. This type system sheds light on which of the verifier's restrictions are essential and which are not.

Tofte and others [8, 33] have developed an allocation strategy using "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 has several limitations. 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.* [14] cannot be modeled in this system without adding new primitives.

Nevertheless, 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 compiles ML to a variant of STAL described here, suitably adapted for the 32-bit Intel 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, as discussed in this paper.

Although we have discussed mechanisms for typing stacks at the assembly language level, our techniques generalize to other languages. The same mechanisms, including 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.

7 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 either heap or 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. Foundations for the implementation of higher-order subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.

[11] 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.

[12] 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.

[13] 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.

[14] 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).

[15] Intel Corporation. *Intel Architecture Optimization Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect, IL, 60056-7641, 1997.

[16] 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.

[17] P. J. Landin. The mechanical evaluation of expressions. *Computer J.*, 6(4):308–20, 1964.

[18] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, January 1992.

[19] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[20] 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.

[21] 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.

[22] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, 1995. Published as CMU Technical Report CMU-CS-95-226.

[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] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. To appear.

[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] Zhong Shao. Flexible representation analysis. In *ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[29] Zhong 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.

[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 11. The operational semantics is given as a deterministic rewriting system in Figure 12. The notation $a[b/c]$ denotes capture avoiding substitution of b for c in a. The notation $a\{b \mapsto c\}$, where a is a mapping, represents map update.

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

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

The notation $w[\psi]$ stands for the natural iteration of instantiations, and the substitution notation $I[\psi/\Delta]$ is defined to mean:

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

The static semantics is similar to TAL's but requires extra judgments for definitional equality of various forms of type. Definitional equality is needed because two stack types (such as $(int{::}nil) \circ (int{::}nil)$ and $int{::}int{::}nil$) may be syntactically different but represent the same type. The judgments are summarized in Figure 13, the rules for type judgments appear in Figure 14, and the rules for term judgments appear in Figures 15 and 16.

The principal theorem regarding the semantics is type safety:

Theorem A.1 (Type Safety) *If $\vdash P$ and $P \longmapsto^* P'$ then P' is not stuck.*

The theorem is proved using the usual Subject Reduction and Progress lemmas, each of which are proved by induction on typing derivations.

Lemma A.2 (Subject Reduction) *If $\vdash P$ and $P \longmapsto P'$ then $\vdash P'$.*

Lemma A.3 (Progress) *If $\vdash P$ then either P has the form $(H, R\{\texttt{r1} \mapsto w\}, \texttt{halt}[\tau])$ or there exists P' such that $P \longmapsto P'$.*

types	τ	$::=$	$\alpha \mid int \mid ns \mid \forall[\Delta].,\ \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau \mid ptr(\sigma)$
stack types	σ	$::=$	$\rho \mid nil \mid \tau{::}\sigma \mid \sigma_1 \circ \sigma_2$
initialization flags	φ	$::=$	$0 \mid 1$
label assignments	Ψ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
type assignments	Δ	$::=$	$\cdot \mid \alpha, \Delta \mid \rho, \Delta$
register assignments	,	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n, \texttt{sp}{:}\sigma\}$
registers	r	$::=$	$\texttt{r1} \mid \texttt{r2} \mid \cdots$
word values	w	$::=$	$\ell \mid i \mid ns \mid ?\tau \mid w[\tau] \mid w[\sigma] \mid pack\ [\tau, w]\ as\ \tau' \mid ptr(i)$
small values	v	$::=$	$r \mid w \mid v[\tau] \mid v[\sigma] \mid pack\ [\tau, v]\ as\ \tau'$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle \mid \texttt{code}[\Delta], .I$
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	$::=$	$nil \mid w{::}S$
instructions	ι	$::=$	$aop\ r_d, r_s, v \mid bop\ r, v \mid \texttt{ld}\ r_d, r_s(i) \mid \texttt{malloc}\ r[\vec{\tau}] \mid$
			$\texttt{mov}\ r_d, v \mid \texttt{mov}\ \texttt{sp}, r_s \mid \texttt{mov}\ r_d, \texttt{sp} \mid \texttt{salloc}\ n \mid$
			$\texttt{sfree}\ n \mid \texttt{sld}\ r_d, \texttt{sp}(i) \mid \texttt{sld}\ r_d, r_s(i) \mid$
			$\texttt{sst}\ \texttt{sp}(i), r_s \mid \texttt{sst}\ r_d(i), r_s \mid \texttt{st}\ r_d(i), r_s \mid$
			$\texttt{unpack}\ [\alpha, r_d], v$
arithmetic ops	aop	$::=$	$\texttt{add} \mid \texttt{sub} \mid \texttt{mul}$
branch ops	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)

Figure 11: Syntax of STAL

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

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

Figure 12: 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)
$\Delta \vdash ,$	$,$ 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
$\Delta \vdash ,_1 = ,_2$	$,_1$ and $,_2$ are equal register file types
$\Delta \vdash \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash ,_1 \leq ,_2$	$,_1$ is a register file subtype of $,_2$
$\vdash H : \Psi$	the heap H has type Ψ
$\Psi \vdash S : \sigma$	the stack S has type σ
$\Psi \vdash R : ,$	the register file R has type $,$
$\Psi \vdash h : \tau$ hval	the heap value h has type τ
$\Psi; \Delta \vdash w : \tau$ wval	the word value w has type τ
$\Psi; \Delta \vdash w : \tau^{\varphi}$	the word value w has flagged type τ^{φ}
	($i.e.$, w has type τ or w is $?\tau$ and φ is 0)
$\Psi; \Delta; , \vdash v : \tau$	the small value v has type τ
$\Psi; \Delta; , \vdash \iota \Rightarrow \Delta'; ,'$	instruction ι requires a context of type $\Psi; \Delta; ,$
	and produces a context of type $\Psi; \Delta'; ,'$
$\Psi; \Delta; , \vdash I$	I is a valid sequence of instructions
$\vdash P$	P is a valid program

Figure 13: Static Semantics of STAL (judgments)

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

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

$$\boxed{\Delta \vdash \tau_1 = \tau_2 \quad \Delta \vdash \sigma_1 = \sigma_2 \quad \Delta \vdash ,_1 = ,_2}$$

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

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

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

$$\frac{\Delta', \Delta \vdash ,_1 = ,_2}{\Delta \vdash \forall[\Delta'].,_1 = \forall[\Delta'].,_2} \qquad \frac{\Delta \vdash \tau_i = \tau_i'}{\Delta \vdash \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle = \langle \tau_1'^{\varphi_1}, \ldots, \tau_n'^{\varphi_n} \rangle}$$

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

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

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

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

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

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

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

$$\boxed{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash ,_1 \leq ,_2}$$

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

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

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

Figure 14: Static Semantics of STAL, Judgments for Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Psi; \Delta; , \vdash \iota \Rightarrow \Delta'; , ' \quad \Psi; \Delta'; , ' \vdash I}{\Psi; \Delta; , \vdash \iota; I} \qquad \frac{\Delta \vdash ,_1 \le ,_2 \quad \Psi; \Delta; ,_1 \vdash v : \forall[].,_2}{\Psi; \Delta; ,_1 \vdash \texttt{jmp}\ v}$$

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

Figure 15: STAL Static Semantics, Term Constructs except Instructions

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

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

$$\frac{\Psi;\Delta;,_1 \vdash r : \mathit{int} \quad \Psi;\Delta;,_1 \vdash v : \forall[].,_2 \quad \Delta \vdash ,_1 \leq ,_2}{\Psi;\Delta;,_1 \vdash \mathit{bop}\ \ r, v \Rightarrow \Delta;,_1}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 16: STAL Static Semantics, Instructions