J. Functional Programming **12** (1): 43–88, January 2002. © 2002 Cambridge University Press
DOI: 10.1017/S0956796801004178 Printed in the United Kingdom

Stack-based typed assembly language

GREG MORRISETT*

Department of Computer Science, Cornell University,
Ithaca, NY 14853, USA

KARL CRARY

Computer Science Department, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213, USA

NEAL GLEW

Intertrust, 4750 Patrick Henry Drive, Santa Clara,
CA 95054, USA

DAVID WALKER

Computer Science Department, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213, USA

Abstract

This paper presents STAL, a variant of Typed Assembly Language with constructs and types to support a limited form of stack allocation. As with other statically-typed low-level languages, the type system of STAL ensures that a wide class of errors cannot occur at run time, and therefore the language can be adapted for use in certifying compilers where security is a concern. Like the Java Virtual Machine Language (JVML), STAL supports stack allocation of local variables and procedure activation records, but unlike the JVML, STAL does not pre-suppose fixed notions of procedures, exceptions, or calling conventions. Rather, compiler writers can choose encodings for these high-level constructs using the more primitive RISC-like mechanisms of STAL. Consequently, some important optimizations that are impossible to perform within the JVML, such as tail call elimination or callee-saves registers, can be easily expressed within STAL.

Capsule Review

The ability to type-check low-level executable code plays an important role in ensuring safe execution of untrusted code in a secure environment, such as Web applets, mobile code, and user-provided kernel extensions. Bytecode verification in Java is a well-known example of type-checking executable code, but it applies only to a specific, rather high-level virtual machine instruction set. Typed Assembly Language (TAL), introduced by Morrisett *et al.* in 1998, extends this approach to much lower-level executable code: it provides a flexible type system for a language similar to the machine code of contemporary processors. However, one limitation of TAL is that it applies only to code compiled in continuation-passing style, that is,

* 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, ARPA grant F19628-95-C-0050, and NSF grants CCR-9317320 and CCR-9708915. 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.

with all activation records allocated on the heap. This article lifts this limitation and extends TAL with the ability to type-check code compiled in direct style, with activation records allocated on a stack. The resulting STAL type system is remarkably flexible and precise; in particular, it can handle many classic uses of the stack in optimized compiled code, including exception handling and callee-save registers. This article is therefore an important step towards making executable code verification applicable to a large class of source languages, optimizing compiler technology, and machine-level instruction sets.

1 Introduction and motivation

A certifying compiler takes high-level source code and produces low-level target code, but in addition, produces explicit evidence that the target code will not perform some ‘bad’ action when executed. By checking the evidence before executing the code, an untrusting system can verify that the target code will be well-behaved independent of the source code or compiler. Thus, to the degree that we trust the verifier, we need not trust either the code producer or the compiler.

As an example, Sun Microsystem’s `javac` compiler takes Java source code (Gosling *et al.*, 1996) and produces Java Virtual Machine Language (JVML) bytecodes (Lindholm & Yellin, 1996) which contain explicit typing annotations as evidence. An untrusting system, such as a Web browser, can verify the typing annotations against the bytecodes using a form of dataflow analysis to ensure that, when executed, the bytecodes will not violate type safety. In turn, the type safety guarantees can be used to ensure a wide class of important security properties, such as memory safety, control safety, or more generally, fault isolation.

Though a portable and successful target language for certifying compilers, the JVML type system is not without its shortcomings. In particular, the bytecodes consist of relatively high-level instructions. As such, the code must either be interpreted, which can yield poor performance, or else compiled to native code. In either case, we must introduce a software component (interpreter or compiler) into our trusted computing base, thereby increasing the probability of error. In addition, both the bytecodes and the type system are tailored for Java, and thus form a poor target for many other source languages. For example, method call and return are high-level instructions with no provision for tail calls. Consequently, with JVML it is difficult to compile functional languages, such as Scheme (Scheme, 1998), that depend upon tail calls for proper space overheads. As another example, the type system has no direct support for parametric polymorphism, making it difficult to compile languages such as SML (Milner *et al.*, 1997). Finally, the JVML type system and semantics were not designed with a formal model in mind, and thus it has been difficult to state or prove properties about the language such as type soundness, though much recent progress has been made (Freund & Mitchell, 1999; Coglio *et al.*, 1998; Goldberg, 1998; Qian, 1998; Freund & Mitchell, 1998; O’Callahan, 1999).

These observations motivated our exploration of type systems for very low-level, explicitly typed target languages with a more low-level philosophy. Our goal was to discover generic target-level type structure that could be applied for any source language and any certifying compiler. As a step towards this goal, in earlier work we presented a core *typed assembly language* (TAL) (Morrisett *et al.*, 1998a; Morrisett *et al.*, 1999a), where almost all of the instructions had a one-to-one correspondence with a conventional MIPS-like assembly language. Thus, the language had no built-in notion of high-level features such as methods, functions, or objects. The primary challenge in designing TAL was to come up with a simple but expressive type system that would allow compiler writers to efficiently encode these features.

We based the type system of TAL on a variant of the Girard-Reynolds polymorphic lambda calculus, also known as System F. Doing so had a number of benefits: First, it was easy to adapt the relatively simple rewriting model and proof techniques developed for System F to the assembly language arena. In turn, this allowed us to state and prove a type soundness theorem easily. Secondly, a number of researchers have studied how to encode high-level language features, such as abstract data types, continuations, closures and objects, using the typing facilities of System F or extensions thereof. By basing the TAL type system on System F, we could immediately transfer these results.

We demonstrated the expressiveness of TAL and its type structure by formally defining a type-preserving, certifying 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. Furthermore, we claimed that the type system of TAL did not interfere with a class of desirable compiler optimizations including inlining, loop unrolling, register allocation, common sub-expression elimination, and instruction scheduling. Our preliminary implementation experience seems to substantiate these claims.

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 Rabbit and SML of New Jersey compilers (Steele Jr., 1978; Appel & MacQueen, 1991), and we relied upon a garbage collector to reclaim these closures. There are some good reasons for implementing procedure activations in this fashion (see, for example, Appel's book (Appel, 1992)), but the approach is fairly non-standard. Indeed, almost all compilers allocate activation records on a stack and explicitly deallocate them upon procedure return (or tail call). Unfortunately, the simple typing model of TAL was unable to support this approach.

In this paper, we explore additions to the type structure of TAL that support limited forms of stack-based memory management. The resulting target language, which we call STAL, is remarkably simple, but powerful enough to compile the control aspects of languages such as Pascal, Java or ML. More specifically, the STAL typing discipline supports stack allocation of temporary variables and values that do not escape upwards; stack allocation of procedure activation frames, exception handlers, and displays; and optimizations such as tail call elimination and callee-saves registers. Unlike JVML, STAL is flexible enough that we need not add high-level procedure call/return primitives. Rather, by providing a general form of *stack polymorphism* and *polymorphic recursion*, these high-level operations can be synthesized from standard primitives such as loads, stores, and jumps.

Nevertheless, the typing discipline of STAL is not powerful enough to support all desirable optimizations. For example, the approach does not support ‘zero-overhead-try’ exception handlers, nor does it support general stack allocation of data (e.g. `alloca`). But in general, our approach seems to strike a good balance between simplicity and expressiveness. Indeed, O’Callahan suggested that the most appropriate way to model and understand important aspects of the JVML type system, such as sub-routines, was to map it to a restricted version of STAL (O’Callahan, 1999).

The basic ideas behind our stack typing discipline were presented in an earlier workshop paper (Morrisett *et al.*, 1998b). Here we extend that work by giving a more thorough and formal treatment, including a proof of type soundness. To keep the development self-contained, we begin in section 2 by giving an overview of the original core typed assembly language, and by describing how a certifying compiler for a functional language can use it as a target language. In sections 3 and 4, we motivate our extensions to support stack allocation through a series of examples involving temporary values, activation records, and exception contexts. In section 5 we formally define the static and dynamic semantics of STAL, deferring the proof of type soundness to Appendix A. We close in section 6 with a brief description of our implementation of STAL for the IA32 instruction set architecture (i.e. the Intel x86), and a discussion of related and future work.

2 Overview of TAL and CPS-based compilation

In this section, we give a brief overview of core Typed Assembly Language (TAL) and how it may be used as the target of a certifying compiler. This presentation is based on our earlier work (Morrisett *et al.*, 1999a), but simplifies the treatment of allocation and initialization (as noted below).

Like a conventional assembly language, the code in a TAL program consists of a set of labelled instruction sequences, where the labels are used as symbolic addresses for control transfers. However, TAL programs are explicitly typed, meaning that instructions and operands are decorated with enough typing information that type checking the language can be done in a syntax-directed manner.

The syntax of instructions and instruction sequences is given below and except for two of the instructions (`malloc` and `unpack`), corresponds directly to RISC-style instructions:

$$\begin{array}{llcl}
\textit{instruction sequences} & I & ::= & \texttt{jmp}\ v \mid \texttt{halt}[\tau] \mid \iota;I \\
\textit{instructions} & \iota & ::= & \textit{aop}\ r_d,r_s,v \mid \texttt{mov}\ r_d,v \mid \texttt{ld}\ r_d,r_s(i) \mid \\
 & & & \texttt{st}\ r_d(i),r_s \mid \textit{bop}\ r,v \mid \\
 & & & \texttt{malloc}\ r,\langle v_1,\ldots,v_n\rangle \mid \texttt{unpack}\ [\alpha,r_d],v \\
\textit{arithmetic ops} & \textit{aop} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \\
\textit{branch ops} & \textit{bop} & ::= & \texttt{beq} \mid \texttt{bneq} \mid \texttt{bgt} \mid \texttt{blt} \mid \texttt{bgte} \mid \texttt{blte}
\end{array}$$

An instruction sequence is a list of instructions terminated by an unconditional control transfer (either `jmp` or `halt`). In the syntax, we use r to represent a register operand and v to represent an operand that is either a register or an immediate word-sized value (e.g. an integer, code label or data label). We use the meta-variable w to range over word-sized values.

The arithmetic instructions take a source register as one operand (r_s), and either another source register or immediate value as another operand (v), perform the appropriate arithmetic on the values, and store the result in the destination register (r_d). The `mov` instruction simply moves the value of v into the destination register r_d. The instruction `ld` $r_d, r_s(i)$ loads a word of memory at an offset of i words from the base address indicated by the source operand r_s into the destination register r_d. Dually, the `st` $r_d(i), r_s$ instruction stores the contents of r_s into memory at the word offset i from the base address r_d.

The branch instructions branch to their second operand if the first operand is appropriately related to 0. For example, the instruction `bneq r2,foo` branches to the label `foo` when register `r2` is not 0. The `jmp` instruction unconditionally transfers control to the address indicated by its operand and the `halt` instruction terminates the operation of the program.

The `malloc` and `unpack` instructions do not correspond to conventional machine instructions. Rather, they are higher-level primitives that make it easier to prove code is type safe and would typically be implemented by a small sequence of machine-specific instructions. The `malloc` instruction is used to dynamically allocate and initialize memory, and returns a pointer to the memory object in the specified register. In the original formulation of TAL, we had more primitive mechanisms that separated allocation and initialization, but to simplify the presentation, we use a combined mechanism here. As in the original TAL model, we assume that heap-allocated memory is reclaimed by a garbage collector.

The `unpack` is a technical device that is used to 'open' a value that has an existential type (Mitchell & Plotkin, 1988), and is discussed more thoroughly in section 2.2.

2.1 The TAL abstract machine

The dynamic semantics of TAL is specified as an abstract rewriting machine similar to the SECD (Landin, 1964) or CESK (Felleisen, 1987) machines used to model higher-level functional languages. This level of abstraction hides some machine-specific details, but makes it easier to relate the semantics to existing formalisms. The abstract machine maintains, but does not use the typing information during evaluation. This facilitates the proof of type soundness but also admits an implementation where type information is erased prior to execution.

A TAL abstract machine state M consists of three components: a heap (H), a register file (R), and a current instruction sequence (I). The heap provides a symbolic store for both code and data; the register file provides values for the registers; and the instruction sequence simulates a program counter. Evaluation is modelled by a deterministic rewriting system that maps machine states to machine states, written $M \longmapsto M'$.

Most of the rewriting rules are straightforward and directly encode the informal semantics discussed earlier. For example, there is one rule for addition using an immediate value that looks like this:

$$(H, R, \texttt{add}\ r_d, r_s, i; I) \longmapsto (H, R\{r_d \mapsto R(r_s) + i\}, I)$$

Here, we look up the value of r_s in the register file, add it to the immediate value i

(where i ranges over integer literals), and step to a new state where the register file is updated to map r_d to the sum, and where the 'program counter' has been advanced by taking the tail of the instruction sequence.

We model register files as functions from register names (r) to word-sized values (w), and we model heaps as partial functions from labels (ℓ) to heap values (discussed below). This level of abstraction hides many details, such as the relative order of heap values, which is convenient for high-level reasoning. For example, an implementation that uses a copying garbage collector is free to rearrange heap values without observably affecting the machine state.

Heap values consist of tuples of word sized values ($\langle w_1,\ldots,w_n\rangle$, $n \geqslant 0$) or typed instruction sequences ($\texttt{code}[\Delta]\Gamma.I$). It is possible to add other forms of data, including tagged unions (sums), arrays, objects, *etc.*, but in this formal treatment, we have kept data forms to a minimum to simplify the presentation.

A typed instruction sequence consists of a typing pre-condition ($[\Delta]\Gamma$) and an instruction sequence (I). Informally, Δ is a list of bound type variables that can be used as abstract types within the code, and Γ describes the types that registers must have before control can be transferred to the associated code sequence. The formal role of the typing pre-condition should become clear when we discuss the static semantics in the next section.

The syntax for machine states, register files, heap values, word values, and typed instruction sequences is summarized below:

$$\begin{array}{llcl}
\textit{machine states} & M & ::= & (H,R,I) \\
\textit{register files} & R & ::= & \{r_1 \mapsto w_1,\ldots,r_n \mapsto w_n\} \\
\textit{heaps} & H & ::= & \{\ell_1 \mapsto h_1,\ldots,\ell_n \mapsto h_n\} \\
\textit{heap values} & h & ::= & \langle w_1,\ldots,w_n\rangle \mid \texttt{code}[\Delta]\Gamma.I \\
\textit{word values} & w & ::= & \ell \mid i \mid w[\tau] \mid \textit{pack}\ [\tau,w]\ \textit{as}\ \tau' \\
\textit{operands} & v & ::= & r \mid w \mid v[\tau] \mid \textit{pack}\ [\tau,v]\ \textit{as}\ \tau'
\end{array}$$

As mentioned earlier, word values include labels (i.e. pointers) and immediate integers, but in addition, they include a number of other syntactic forms. The value $w[\tau]$ is a polymorphic type instantiation when w is a polymorphic value (i.e. a label for a polymorphic instruction sequence.) A value $\textit{pack}\ [\tau,w]\ \textit{as}\ \tau'$ is used to introduce existential types.

Like typing pre-conditions, these annotations on values are not used during evaluation but rather keep the process of type checking syntax-directed.

We can now describe the semantics of the rest of the instructions. In the semantics, we use $\hat{R}$ to convert an operand to a word value as follows:

$$\begin{array}{lcl}
\hat{R}(r) & = & R(r) \\
\hat{R}(w) & = & w \\
\hat{R}(v[\tau]) & = & (\hat{R}(v))[\tau] \\
\hat{R}(\textit{pack}\ [\tau,v]\ \textit{as}\ \tau') & = & \textit{pack}\ [\tau,\hat{R}(v)]\ \textit{as}\ \tau'
\end{array}$$

The mov instruction simply moves the word value of the operand into the appropriate register:

$$(H,R,\texttt{mov}\ r_d,v;I) \longmapsto (H,R\{r_d \mapsto \hat{R}(v)\},I)$$

The `ld` instruction expects that its operand contains a label bound in the heap to a tuple. The *i*th component of the tuple is returned as the result:

$$\begin{array}{l}(H, R, \texttt{ld}\ r_d, r_s(i); I) \longmapsto \\ \quad (H, R\{r_d \mapsto w_i\}, I) \text{ when } H(\hat{R}(r_s)) = \langle w_1, \ldots, w_i, \ldots, w_n \rangle\end{array}$$

The `st` instruction is the dual:

$$\begin{array}{l}(H\{\ell \mapsto \langle w_1, \ldots, w_i, \ldots, w_n \rangle\}, R, \texttt{st}\ r_d(i), r_s; I) \longmapsto \\ \quad (H\{\ell \mapsto \langle w_1, \ldots, \hat{R}(r_s), \ldots, w_n \rangle\}, R, I) \qquad \text{when } \hat{R}(r_d) = \ell\end{array}$$

The `malloc` instruction allocates and initializes a new tuple in the heap:

$$\begin{array}{l}(H, R, \texttt{malloc}\ r_d, \langle v_1, \ldots, v_n \rangle; I) \longmapsto \\ \quad (H\{\ell \mapsto \langle \hat{R}(v_1), \ldots, \hat{R}(v_n) \rangle\}, R\{r_d \mapsto \ell\}, I) \text{ where } \ell \notin \mathit{Dom}(H)\end{array}$$

For the branch instructions, if the condition is not true, then we simply move on to the tail of the current instruction sequence. Otherwise, we 'jump' to the code specified by the operand by installing its associated code. For example:

$$\begin{array}{l}(H, R, \texttt{beq}\ r, \ell; I) \longmapsto \\ \quad (H, R, I') \qquad \text{when } R(r) = 0 \text{ and } H(\ell) = \texttt{code}[]\Gamma.I'\end{array}$$

Other control transfers, such as `jmp` behave in a similar fashion.

In general, instruction sequences can be polymorphic with respect to types. That is, they can abstract a sequence of type variables $\Delta = \alpha_1, \ldots, \alpha_n$ and these type variables may occur free within the code sequence. Before control can be transferred to a polymorphic instruction sequence, the type variables must be explicitly instantiated. Thus, the general rule for a control transfer, such as a `jmp`, requires that we supply type parameters $[\tau_1, \ldots, \tau_n]$ as in:

$$\begin{array}{l}(H, R, \texttt{jmp}\ \ell[\tau_1, \ldots, \tau_n]) \longmapsto \\ \quad (H, R, I'[\tau_i/\alpha_i]) \qquad \text{when } H(\ell) = \texttt{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I'\end{array}$$

(We write the capture-avoiding substitution of τ for α in an expression E as $E[\tau/\alpha]$.) Notice that when the control transfer is performed, we substitute the type parameter τ_i for the appropriate type variable α_i within the code sequence that abstracts the type variables. Again, the type instantiation and substitution are not necessary for evaluation, but rather make it easier to prove type soundness. In a real implementation, type information may be erased prior to execution.

The abstract machine's terminal states are of the form $(H, R[\texttt{r1} \mapsto v], \texttt{halt}[\tau])$, where v is a value of type τ. We say that a non-terminal machine state is *stuck* if there is no valid transition to a new machine state. For example, the machine becomes stuck if it attempts to perform a load or store on an integer operand, or attempts to jump to an operand that is not a code label. The goal of the type system, discussed in the next section, is to ensure that well-formed programs never become stuck.

2.2 Overview of the TAL type system

The abstract syntax for the types of the TAL abstract machine is given below:

$$\begin{array}{lcll} \textit{types} & \tau & ::= & \alpha \mid \textit{int} \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\Delta].\Gamma \mid \exists\alpha.\tau \\ \textit{type assignments} & \Delta & ::= & \cdot \mid \Delta, \alpha \\ \textit{register assignments} & \Gamma & ::= & \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \\ \textit{label assignments} & \Psi & ::= & \{\ell_1 : \tau_1, \ldots \ell_n : \tau_n\} \end{array}$$

Types include type variables, *int*, tuple types, polymorphic code types, and existential types. Code types ($\forall[\Delta].\Gamma$), similar to polymorphic function types, are used to classify pointers to heap-allocated instruction sequences. As discussed above, control can only be transferred to a value with a code type when we supply types for the type parameters, and ensure that our current register state has a typing that satisfies the pre-condition of the instruction sequence. Code types have no post-condition or return type because control is either terminated via `halt` or transferred to another code block. When the set of abstracted type variables is empty, we often omit the '$\forall[\Delta]$'.

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

```
code[α]{r1:α, r2:∀[].{r1:⟨α, α⟩}}.
        malloc  r3, ⟨r1, r1⟩
        mov     r1, r3
        jmp     r2
```

roughly corresponds to a CPS version of the ML function `fn(x:α)=>(x,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 $\forall[].\{\texttt{r1}:\langle\alpha,\alpha\rangle\}$. In other words, the return address requires register `r1` to contain a pointer to a pair of values of type α before control can be returned to this address. The instructions of the code block allocate and initialize a tuple using 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` $\ell[\textit{int}]$).

Existential types are used to represent a form of first-class abstract data types. When a value v has type $\tau_1[\tau_2/\alpha]$ we can abstract the type τ_2 by packing the value into the type $\exists\alpha.\tau_1$. For example, if v has type $\langle \textit{int}, \{\texttt{r1}:\langle \textit{int}, \textit{int}\rangle\}\rangle$, then we can abstract some of the occurrences of *int* by writing *pack* $[\textit{int}, v]$ *as* $\exists\alpha.\langle\alpha, \{\texttt{r1}:\langle\alpha, \textit{int}\rangle\}\rangle$.

The instruction `unpack` $[\beta, r_d], v$ opens a value v of existential type, placing a copy of the underlying packed value in r_d. In addition, it introduces a local type variable (β) to name the abstracted type. The scope of the type variable extends to the end of the enclosing instruction sequence. As usual, the type variable must be different from any other variable in the context to avoid unsoundness, but this can always be accomplished by alpha-converting the code sequence. The following code block gives a simple example of using an existential:

```
code[]{r1:∃α.⟨α,{r1:α}⟩}.
        unpack  [β,r2],r1  %  r2:⟨β,{r1:β}⟩
        ld      r1,r2(0)   %  r1:β
        ld      r3,r2(1)   %  r3:{r1:β}
        jmp     r3
```

Here, the code expects a tuple value in register r1, where the type of the first component is abstract (α), and the second component is a code label expecting a value of type α to be passed in register r1. The sequence begins by unpacking the value into register r2, introducing a unique, local name β for the abstract type. Register r2 is thus assigned the type $\langle\beta, \{\mathtt{r1}{:}\beta\}\rangle$. We then load the abstract value and the code label into registers r1 and r3 respectively, and jump to the code label. In the next section, we give a similar but more realistic example showing how existential types can be used to implement closures.

Finally, label assignments (Ψ) are used to give types to heaps during evaluation. A heap $H = \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$ has type $\Psi = \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$ when, under the assumption that $H{:}\Psi$, we can show that $h_i{:}\tau_i$ for $1 \leqslant i \leqslant n$. Thus, the typing rule for heaps is similar to a 'letrec' construct in a conventional functional language as it allows heap values to indirectly refer to one another via their labels.

The major typing judgment for the abstract machine requires that we be able to assign a type Ψ to the heap H, assign a type Γ to the register file R, and check that the current instruction sequence I is well-formed under the assumptions of Ψ and Γ:

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

We determine that an instruction sequence is well-formed by checking that it uses registers and labels in a type-consistent manner. For instance, the add instruction requires that both of its operands have type *int*, whereas the jmp instruction requires that its operand have a code type. After checking that an instruction uses operands in a type-consistent manner, we produce a typing post-condition which is used as the pre-condition of the next instruction in the sequence. For example, the typing rule for the add instruction looks similar to this:

$$\frac{\Gamma(r_s) = \mathit{int} \quad \Psi;\Delta;\Gamma \vdash v : \mathit{int}}{\Psi;\Delta;\Gamma \vdash \mathtt{add}\ r_d, r_s, v \Rightarrow \Delta;\Gamma\{r_d : \mathit{int}\}}$$

In this particular case, the post condition is the same as the pre-condition, except that register r_d is assigned the type *int*.

In general, we need to keep track of the types of labels and registers, and the set of type variables that are in scope. Thus, the judgment for instructions is parameterized by a label assignment Ψ, a type assignment Δ, and a register assignment Γ. With respect to typing, most instructions only affect the register file. However, the unpack instruction introduces new type variables that can be used in subsequent instructions in a sequence:

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

Consequently, the post-condition for an instruction includes a new type assignment as well as an updated register assignment.

Notice that the post-condition does not affect the label assignment. This is because, unlike registers, we do not allow labels to change types under evaluation. The issue here is that, unlike registers, labels are first-class values. Because each copy is checked independently, it is difficult to ensure that the different occurrences are treated consistently without explicitly tracking which values are (potentially) aliases to a given label. Therefore, we require that the type of a label remain constant through evaluation, similar to the treatment of state used in other typing disciplines (for instance, Harper (1994)).

Though the type of a given label cannot change, the heap (and thus its type) can be *extended* via `malloc`. At type-checking time, we do not know what label will be used when the `malloc` is executed so we cannot calculate a precise type for the heap. Rather, we rely upon the fact that the heap grows monotonically and consequently, the compile time type of the heap will always be a super-type of the heap at any point during evaluation. Thus, the typing rule for `malloc` uses only the information available at type-checking time:

$$\frac{\Psi;\Delta;\Gamma \vdash v_i : \tau_i \ (1 \leqslant i \leqslant n)}{\Psi;\Delta;\Gamma \vdash \texttt{malloc}\ r_d, \langle v_1,\ldots,v_n\rangle \Rightarrow \Delta;\Gamma[r_d \mapsto \langle \tau_1,\ldots,\tau_n\rangle]}$$

We string sequences of instructions together by using the post-condition of one instruction as the pre-condition of the next in the style of Hoare-logic:

$$\frac{\Psi;\Delta;\Gamma \vdash \iota \Rightarrow \Delta';\Gamma' \quad \Psi;\Delta';\Gamma' \vdash I}{\Psi;\Delta;\Gamma \vdash \iota;I}$$

A control transfer, such as a `jmp`, requires that the operand have a code type and that the current register typing is a sub-type of the destination's typing pre-condition:

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall[].\Gamma' \quad \Gamma \leqslant \Gamma'}{\Psi;\Delta;\Gamma \vdash \texttt{jmp}\ v}$$

We choose to treat subtyping on register file types similar to 'width' sub-typing on records:

$$\{r_1{:}\tau_1,\ldots,r_n{:}\tau_n,r_{n+1}{:}\tau_{n+1}\} \leqslant \{r_1{:}\tau_1,\ldots,r_n{:}\tau_n\}$$

There are of course alternatives. For instance, we could add a more complete notion of subtyping, and the rule to include both depth and width subtyping. However, this would add a substantial number of rules to the type system and therefore complicate the proof of soundness. Alternatively, we could eliminate subtyping all together and instead use polymorphism to abstract the types of registers we would otherwise forget. However, in practice, we have found that the width sub-typing approach yields smaller and thus more readable typing annotations.

Obviously, there are additional judgments that are needed for the other instructions, operands, heap values, etc., but these are fairly straightforward and are presented in detail in section 5. As mentioned in the previous section, the principal goal of the type system is to ensure that well-formed machine states do not become stuck. Indeed, this fact can be proven by establishing subject reduction and progress lemmas as we do in the Appendix.

2.3 Compiling to TAL

Although TAL is a fairly simple programming language and has a fairly simple type system, we can still compile high-level polymorphic functional languages, such as core ML, to type-correct TAL code. In our previous work, we described a prototypical compiler that was composed of four stages: The first stage converted code to *continuation passing style* (CPS) in order to make the control context explicit using higher-order functions. The second stage *closure converted* the resulting CPS code by representing functions as pairs of a closed piece of code abstracting the free variables of the function, and an environment which provided values for the free variables. The third stage lifted closed functions to the top-level and made allocation of tuples explicit (the latter function is made unnecessary by the simplified version of `malloc` in this paper) and resulted in code that was very C-like in nature. The final stage, code generation, simply translated the resulting code into TAL in a straightforward fashion.

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 (Minamide *et al.*, 1996),[1] 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 $int \to \langle\rangle$ has type $(int, (\langle\rangle \to void)) \to void$.[2] Then, after closures are introduced, the code has type:

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

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

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

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 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.

As an example, consider the following ML code which computes 6 factorial:

[1] The issue is that, two source level functions with the same source type, might have different environment types at the target level (e.g. because they have different free variables.) Existentially quantifying the type of the environment ensures that the target-level types remain the same.

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

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

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

Fig. 1. Typed assembly code for factorial (unoptimized).

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

Figure 1 gives equivalent TAL code that would result from our simple compiler.

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

Fig. 2. Additions to TAL for simple stacks.

3 Adding a stack to TAL

In this section, we describe how to extend TAL to obtain a Stack-Based Typed Assembly Language (STAL), focusing on the key issues. Here, we informally discuss the dynamic and static semantics for the modified language, leaving formal treatment to section 5. We also discuss how these features may be used in a type-directed compiler.

3.1 Basic developments

Figure 2 defines the new syntactic constructs for adding stacks to the TAL abstract machine. Operationally we model stacks (S) as lists of word-sized values. We augment the machine state by adding a new distinguished register sp to the register file component to hold the current value of the stack. Thus, machine states are of the form $(H, R[\texttt{sp} \mapsto S], I)$ and consist of a heap, register file (including the stack), and instruction sequence.

There are four new instructions that manipulate the stack: The salloc n instruction enlarges the stack by n words. On a conventional machine, assuming stacks grow toward lower addresses, an salloc operation would correspond to subtracting n from the stack pointer (or, more realistically, $4n$). The new stack slots are uninitialized, which we formalize by filling them with ‘nonsense’ words denoted by ns. Nonsense values are assigned the type $\top$, suggesting that there are no useful operations on values of this type. In the presence of a primitive notion of sub-typing, we could also treat $\top$ as the greatest type (top).

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 ith word (from zero) of the stack into register r, whereas the sst $\texttt{sp}(i), r$ stores register r into the ith word.

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 typed by a register assignment (Γ) mapping registers to types. However, Γ also maps the distinguished register `sp` to a stack type σ. Finally, code blocks and code types support polymorphic abstraction over both types and stack types. In the interest of clarity, from time to time we will give registers symbolic names (such as `ra` for return address).

In addition to the possibilities for stuck states arising from TAL, our new abstract machine can become stuck if we attempt to execute:

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

As usual, a type safety theorem (Theorem 5.1) dictates that no well-formed program can become stuck.

3.2 *Using the stack for temporaries*

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 the end of each step of the computation.

```
                                % $\sigma$
salloc      $n$                 % $\top::\top::\cdots::\top::\sigma$
sst         $sp(0), r_1$        % $\tau_1::\top::\cdots::\top::\sigma$
⋮
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`. A push can be seen as simply `salloc` 1 followed by a store to $\mathtt{sp}(0)$, whereas a pop is a load from $\mathtt{sp}(0)$ followed by `sfree` 1. Also, a 'jump-and-link' or 'call' instruction that 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. There are other practical issues, including allocation for sizes different than a word, or alignment constraints that we also do not treat here. However, we expect that it is not too difficult to extend the formalism to deal with such details.

3.3 Stack polymorphism and recursive functions

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:

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

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:

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

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 receives. 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. (A formal proof of this fact appears in Crary (1999).)

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

```
(H, {sp ↦ nil}, I)  where

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

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

and τ_ρ = ∀[].{r1 : int, sp : ρ}
```

Fig. 3. STAL factorial example.

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 3 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.

3.4 Calling conventions

It is interesting to note that the stack-based code is quite similar to the heap-based code of Figure 1. In a sense, the stack-based code remains in 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 fully appreciate the correspondence, consider the type of the TAL version of `l_fact` from Figure 1:

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

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 stack-based execution and continuation-passing 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):

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

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. As a simple example, Figure 4 gives STAL code for the following tail-recursive factorial code:

```
fun tail_fact n =
    let fun loop(a,n) =
        if n = 0 then a else loop(a*n,n-1)
    in
        loop(1,n)
    end
```

```
l_loop:
    code[ρ]{r1 : int,r2 : int,sp : ρ,ra : {r1 : int,sp : ρ}}.
      bneq r2,l_nonzero[ρ]  % if n = 0 continue
      jmp ra                % return
l_nonzero:
    code[ρ]{r1 : int,r2 : int,sp : ρ,ra : {r1 : int,sp : ρ}}.
      mul r1,r1,r2          % a = a * n
      sub r2,r2,1           % n = n − 1
      jmp l_loop[ρ]         % loop(a,n)
l_tail_fact:
    code[ρ]{r1 : ⟨⟩,r2 : int,sp : ρ,ra : {r1 : int,sp : ρ}}.
      mov r1,1
      jmp l_loop[ρ]         % loop(1,n)
```

Fig. 4. Tail-recursive factorial example.

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:

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

This is the type that corresponds to the callee-saves protocol of Appel and Shao (Appel & Shao, 1992). Again the close correspondence holds between the stack- and heap-oriented types. Indeed, either one can be obtained mechanically from the other.

4 Compound stacks

The simple stack mechanisms described in the previous section support encodings for simple forms of procedures. However, as we will argue, the mechanisms are not sufficient for compiling more sophisticated control mechanisms, such as exceptions, or procedures with static links. The problem is that the typing discipline treats the stack in a *linear* fashion in that it only allows access to the stack contents through the stack pointer register. That is, there is no general facility for obtaining pointers into the middle of the stack. This restriction allows us to easily re-use space on the stack for values of different types and to grow or shrink the stack, but prevents a number of useful encodings.

Unfortunately, we see no simple way to extend the type system to support arbitrary pointers into the stack soundly. However, in this section, we consider an extension to the typing discipline that supports a *limited* form of pointer into the stack without unduly complicating the type system. We motivate the mechanism by showing how it may be used to encode (one treatment of) exceptions and static links.

4.1 Exception calling conventions

We now consider one way to implement exceptions in STAL. In languages such as ML or Java, the exception mechanisms consist of a control aspect (raising and handling the exception) and a data aspect (the exception value or object, and matching or testing for a particular exception constructor). We only consider the control aspects here to avoid the need to introduce extensible sums or objects.

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 and treating rex as the register to hold the exception continuation):

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

As before, the caller could unbox the continuations:

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

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

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

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, though this fact is not captured by the types. This observation leads to the following calling convention for exceptions with stacks:

$$\begin{aligned}\forall[\rho_1, \rho_2].\{&\mathtt{sp}{:}\rho_1 \mathbin{@} \rho_2, \mathtt{r1}{:}\mathit{int}, \mathtt{ra}{:}\{\mathtt{sp}{:}\rho_1 \mathbin{@} \rho_2, \mathtt{r1}{:}\langle\rangle\},\\ &\mathtt{rex}{:}\{\mathtt{sp}{:}\rho_2, \mathtt{r1}{:}\mathit{exn}\}, \mathtt{res}{:}\mathit{ptr}(\rho_2)\}\end{aligned}$$

This type uses the notion of a *compound stack:* When σ_1 and σ_2 are stack types, the compound stack type $\sigma_1 \mathbin{@} \sigma_2$ is the result of appending the two types. Thus, in the above type, the function is presented with a stack with type $\rho_1 \mathbin{@} \rho_2$, all of which is expected by the regular continuation, but only a tail of which (ρ_2) is expected by the exception continuation. Since ρ_1 and ρ_2 are quantified, the function may still be used for any stack so long as the exception continuation accepts some tail of that stack.

To raise an exception, the exception is placed in r1 and 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 $\mathit{ptr}(\sigma)$ 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.

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

Fig. 5. Additions to TAL for compound stacks.

4.2 Compound stacks

The additional syntax to support compound stacks is summarized in Figure 5. The type constructs $\sigma_1 @ \sigma_2$ and $\mathit{ptr}(\sigma)$ were discussed above. The word value $\mathit{ptr}(i)$ 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 $\mapsto w::S$ $(\tau \neq \top)$
mov r1,sp     % r1 : $\mathit{ptr}(\tau::\sigma)$
sfree 1       % sp : $\sigma$, sp $\mapsto S$
salloc 1      % sp : $\top::\sigma$, sp $\mapsto \mathit{ns}::S$
sld r2,r1(0)  % r2 : $\tau$ but r2 $\mapsto \mathit{ns}$
```

When execution reaches the final line, r1 still has type $\mathit{ptr}(\tau::\sigma)$, but this type is no longer consistent with the state of the stack; the pointer in r1 points to ns, which does not have type τ.

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 $\mathit{ptr}(\sigma_2)$, then r is valid if σ_2 is a tail of σ_1 (formally, if there exists some σ' such that $\sigma_1 = \sigma' @ \sigma_2$). 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, $\top::\sigma$.

It may not be obvious that this simple approach of 'validating' a pointer into the middle of the stack is actually sound. Therefore, in section 5, we formalize the type system and prove a soundness result in the Appendix.

4.3 Using compound stacks

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

$$\forall[\rho_1, \rho_2].\{\mathtt{sp}{:}\rho_1 \mathbin{@} \rho_2, \mathtt{r1}{:}\mathit{int}, \mathtt{ra}{:}\{\mathtt{sp}{:}\rho_1 \mathbin{@} \rho_2, \mathtt{r1}{:}\langle\rangle\},\\ \mathtt{rex}{:}\{\mathtt{sp}{:}\rho_2, \mathtt{r1}{:}\mathit{exn}\}, \mathtt{res}{:}\mathit{ptr}(\rho_2)\}$$

An exception may be raised within the body of such a function by restoring the handler's stack from `res` and jumping to the handler. A new exception handler may be installed by copying the stack pointer to `res` and making subsequent function calls with the stack type variables instantiated to *nil* and $\rho_1 \mathbin{@} \rho_2$. 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 $\rho_1 \mathbin{@} (\tau_{\text{handler}}{::}\rho_2)$ and exception pointers with the type $\mathit{ptr}(\tau_{\text{handler}}{::}\rho_2)$ where $\tau_{\text{handler}} = \forall[\,].\{\mathtt{sp}{:}\rho_2, \mathtt{r1}{:}\mathit{exn}\}$.

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:

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

One application of this tool would be for implementing languages such as Pascal that require static links to access variables defined in outer enclosing lexical scopes. For example, the code for a procedure at lexical depth $n \geqslant 0$ would have a stack type σ_n where:

$$\begin{aligned} \sigma_0 &= \mathit{nil} \\ \sigma_i &= \tau_{i,1}{::}\tau_{i,2}{::}\cdots{::}\tau_{i,m_i}{::}\mathit{ptr}(\sigma_{i-1}){::}\rho_i \mathbin{@} \sigma_{i-1} \end{aligned}$$

For a given segment σ_i, we would have the local variables $(\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m_i})$, followed by a static link to the next segment $(\mathit{ptr}(\sigma_{i-1}))$, followed by a stack type variable ρ_i which abstracts the dynamic portion of the stack between the frames σ_i and σ_{i-1}. By loading the pointer to σ_{i-1} into a register, we can access the local variables for the statically enclosing scope. We can also access the pointer to σ_{i-2} to access the next scope, etc. If desired, the chaining overhead can be avoided by caching the link pointers in a heap-allocated tuple.

The primary limitation of this approach to placing data on the stack is that it forces us to expose the relative order of data allocated on the stack, though we can abstract the distance. Furthermore, the type system forces us to distinguish between stack-allocated values and heap-allocated values. Consequently, it does not support compilation of languages such as C that allow stack and heap pointers to be freely mixed, and that allow pointers to stack-allocated values to be passed as arguments in any order.

Finally, we note that our treatment of exceptions here is somewhat simplistic. In particular, we have ignored the data aspects and assumed a single exception value model. However, it is straightforward to generalize to multiple exception values.

In our implementation of STAL for the Intel x86, we use the hierarchically tagged object approach of Glew (Glew, 1999) which supports both ML-style extensible sums and Java-style objects as exception constructors. When an exception is raised, the exception value is passed to the nearest (dynamically) enclosing exception handler. A primitive form of matching is used to determine if the handler can actually handle the given exception. If not, then the exception is re-raised by throwing the value to the next exception handler. In this fashion, the stack is dynamically unwound to the nearest context that can handle the exception.

Another technique, first described in the context of Clu (Liskov & Snyder, 1979), is used in most Java compilers today: When an exception is raised, a runtime routine unwinds the stack to the nearest enclosing handler that can handle the particular exception. Doing so requires that the compiler emit tables indexed by return addresses to describe the layout of stack frames, callee-save register information, and where to find handler code. The advantage of this approach is that it requires no exception register and has no overhead upon entering a try-block. However, to ensure that the code is type safe, we would have to modify the type system to ensure that the tables are properly constructed, and that the runtime unwinding routine is correct.

5 Formal STAL semantics

This section contains a complete technical description of the STAL abstract machine, which is very similar to the TAL abstract machine (described in detail in Morrisett *et al.*, 1999a). We also state a type soundness theorem for the language and give a proof of that fact in Appendix A.

5.1 Syntax

The complete abstract syntax for STAL appears in Figure 6. As discussed earlier, the type structure has five distinct syntactic classes: Types (τ) are used to classify word-sized values, operands, and heap values. Stack types (σ) are used to classify stacks. Label assignments are used to classify heaps and are partial functions from labels to types. Type assignments track the type variables and stack type variables that are in scope. Finally, register assignments map registers to types. The distinguished register sp is always mapped to a stack type, whereas all other registers are mapped to conventional types.

In a code type $\forall[\Delta].\Gamma$, we consider the type variables in Δ to be bound within Γ. In an existential type $\exists\alpha.\tau$, we consider α to be bound in τ. As usual, we consider types to be equivalent up to alpha-conversion of bound type variables. In addition, we consider register files, heaps, register type assignments and label type assignments to be equivalent up to re-ordering of their components.

Machine states consist of three components: a heap H mapping labels to heap values; a register file R mapping registers to word values and the distinguished register sp to a stack; and a current instruction sequence I. Heap values consist of typed instruction sequences or tuples of word-size values. In a typed instruction sequence $\texttt{code}[\Delta]\Gamma.I$, we require the type variables in Δ to be distinct and consider

types	τ	$::=$	$\alpha \mid \mathit{int} \mid \top \mid \langle\tau_1,\ldots,\tau_n\rangle \mid \forall[\Delta].\Gamma \mid \exists\alpha.\tau \mid \mathit{ptr}(\sigma)$
stack types	σ	$::=$	$\rho \mid \mathit{nil} \mid \tau{::}\sigma \mid \sigma_1 \,@\, \sigma_2$
label assignments	Ψ	$::=$	$\{\ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n\}$
type assignments	Δ	$::=$	$\cdot \mid \alpha,\Delta \mid \rho,\Delta$
register assignments	Γ	$::=$	$\{\mathtt{sp}{:}\sigma, r_1{:}\tau_1,\ldots,r_n{:}\tau_n\}$
registers	r	$::=$	$\mathtt{r1} \mid \mathtt{r2} \mid \cdots$
word values	w	$::=$	$\ell \mid i \mid \mathit{ns} \mid w[\tau] \mid w[\sigma] \mid \mathit{pack}\ [\tau,w]\ \mathit{as}\ \tau' \mid \mathit{ptr}(i)$
operands	v	$::=$	$r \mid w \mid v[\tau] \mid v[\sigma] \mid \mathit{pack}\ [\tau,v]\ \mathit{as}\ \tau'$
heap values	h	$::=$	$\langle w_1,\ldots,w_n\rangle \mid \mathtt{code}[\Delta]\Gamma.I$
heaps	H	$::=$	$\{\ell_1 \mapsto h_1,\ldots,\ell_n \mapsto h_n\}$
register files	R	$::=$	$\{\mathtt{sp} \mapsto S, r_1 \mapsto w_1,\ldots,r_n \mapsto w_n\}$
stacks	S	$::=$	$\mathit{nil} \mid w{::}S$
instructions	ι	$::=$	$\mathit{aop}\ r_d,r_s,v \mid \mathit{bop}\ r,v \mid \mathtt{ld}\ r_d,r_s(i) \mid$ $\mathtt{malloc}\ r_d,\langle v_1,\ldots,v_n\rangle \mid \mathtt{mov}\ r_d,v \mid \mathtt{mov}\ \mathtt{sp},r_s \mid$ $\mathtt{mov}\ r_d,\mathtt{sp} \mid \mathtt{salloc}\ n \mid \mathtt{sfree}\ n \mid$ $\mathtt{sld}\ r_d,\mathtt{sp}(i) \mid \mathtt{sld}\ r_d,r_s(i) \mid \mathtt{sst}\ \mathtt{sp}(i),r_s \mid$ $\mathtt{sst}\ r_d(i),r_s \mid \mathtt{st}\ r_d(i),r_s \mid \mathtt{unpack}\ [\alpha,r_d],v$
arithmetic ops	*aop*	$::=$	$\mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul}$
branch ops	*bop*	$::=$	$\mathtt{beq} \mid \mathtt{bneq} \mid \mathtt{bgt} \mid \mathtt{blt} \mid \mathtt{bgte} \mid \mathtt{blte}$
instruction sequences	I	$::=$	$\iota;I \mid \mathtt{jmp}\ v \mid \mathtt{halt}[\tau]$
machine states	M	$::=$	(H,R,I)

Fig. 6. Syntax of STAL.

them bound in both Γ and I, and consider such heap values as equivalent up to alpha-conversion of the bound type variables. In an instruction sequence of the form $\mathtt{unpack}\ [\alpha,r_d],v;I$, we consider α bound in the remaining sequence I.

Word values form a proper syntactic subclass of operands, as they exclude registers and operands built from them. Otherwise, the two classes are the same. They include labels, integers, a nonsense value (*ns*) used when space is allocated on the stack, polymorphic instantiations (for both regular types and stack types), and packed values. Packed values abstract a type and are used to introduce existential types.

5.2 Dynamic semantics

The formal operational semantics for STAL is given as a deterministic rewriting system in Figures 7 and 8. A *terminal configuration* is a program of the form $(H,R\{\mathtt{r1} \mapsto w\},\mathtt{halt}[\tau])$. A program is said to be *stuck* if it is irreducible and not a terminal configuration.

For each arithmetic operation *aop*, we write $||\mathit{aop}||$ for the obvious arithmetic function on integer values. For example, $||\mathtt{add}||(i_1,i_2) = i_1 + i_2$. For each branch operation *bop*, we associate the obvious unary predicate $||\mathit{bop}||$, on integers. For example, $||\mathtt{blte}||(x)$ evaluates to the same truth value as the predicate $x \leqslant 0$. As described in section 2.1, we write $\hat{R}$ to lift the register file R to map operands to word values in the obvious manner, replacing registers with their values.

The notation $a[b/c]$ denotes capture avoiding substitution of b for c in a. The

$(H,R,I) \longmapsto M$ where	
if $I =$	then $M =$
$\mathit{aop}\ r_d, r_s, v; I'$	$(H, R\{r_d \mapsto \lVert \mathit{aop} \rVert(R(r_s), \hat{R}(v))\}, I')$
$\mathit{bop}\ r, v; I'$ when not $\lVert \mathit{bop} \rVert(R(r))$	(H, R, I')
$\mathit{bop}\ r, v; I'$ when $\lVert \mathit{bop} \rVert(R(r))$	$(H, R, I''[\psi/\Delta])$ where $\hat{R}(v) = \ell[\psi]$ and $H(\ell) = \texttt{code}[\Delta]\Gamma.I''$
`jmp` v	$(H, R, I'[\psi/\Delta])$ where $\hat{R}(v) = \ell[\psi]$ and $H(\ell) = \texttt{code}[\Delta]\Gamma.I'$
`ld` $r_d, r_s(i); I'$	$(H, R\{r_d \mapsto w_i\}, I')$ where $R(r_s) = \ell$, $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$, and $0 \leqslant i < n$
`malloc` $r_d, \langle v_1, \ldots, v_n \rangle$	$(H\{\ell \mapsto \langle \hat{R}(v_1), \ldots, \hat{R}(v_n) \rangle\}, R\{r_d \mapsto \ell\}, I')$ where $\ell \notin \mathit{Dom}(H)$
`mov` $r_d, v; I'$	$(H, R\{r_d \mapsto \hat{R}(v)\}, I')$
`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 $R(r_d) = \ell$, $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$, and $0 \leqslant i < n$
`unpack` $[\alpha, r_d], v; I'$	$(H, R\{r_d \mapsto w\}, I'[\tau/\alpha])$ where $\hat{R}(v) = \mathit{pack}\ [\tau, w]\ \mathit{as}\ \tau'$

Fig. 7. Operational semantics of STAL, Part I.

notation $a\{b \mapsto c\}$, where a is a mapping, represents map update. The notation $\mathrm{fv}(a)$ denotes the free variables of a.

In general, the branching rules must instantiate bound type variables as described earlier. To make the presentation simpler, 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:

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

$(H,R,I) \longmapsto M$ where	
if $I =$	then $M =$
$\texttt{mov}\ r_d, \texttt{sp}; I'$	$(H, R\{r_d \mapsto ptr(n)\}, I')$ where $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: nil$
$\texttt{mov}\ \texttt{sp}, r_s; I'$	$(H, R\{\texttt{sp} \mapsto w_{n-j} :: \cdots :: w_{n-1} :: nil\}, I')$ where $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: nil$, $R(r_s) = ptr(j)$, and $0 \leqslant j \leqslant n$
$\texttt{salloc}\ n; I'$	$(H, R\{\texttt{sp} \mapsto \underbrace{ns :: \cdots :: ns}_{n} :: R(\texttt{sp})\}, I')$
$\texttt{sfree}\ n; I'$	$(H, R\{\texttt{sp} \mapsto S\}, I')$ where $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: S$
$\texttt{sld}\ r_d, \texttt{sp}(i); I'$	$(H, R\{r_d \mapsto w_i\}, I')$ where $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: nil$ and $0 \leqslant i < n$
$\texttt{sld}\ r_d, r_s(i); I'$	$(H, R\{r_d \mapsto w_{n-j+i}\}, I')$ where $R(r_s) = ptr(j)$, $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: nil$, and $0 \leqslant i < j \leqslant n$
$\texttt{sst}\ \texttt{sp}(i), r_s; I'$	$(H, R\{\texttt{sp} \mapsto w_0 :: \cdots :: w_{i-1} :: R(r_s) :: S\}, I')$ where $R(\texttt{sp}) = w_0 :: \cdots :: w_i :: S$ and $0 \leqslant i$
$\texttt{sst}\ r_d(i), r_s; I'$	$(H, R\{\texttt{sp} \mapsto w_0 :: \cdots :: w_{n-j+i-1} :: R(r_s) :: w_{n-j+i+1} :: \cdots :: w_{n-1} :: nil\}, I')$ where $R(r_d) = ptr(j)$, $R(\texttt{sp}) = w_0 :: \cdots :: w_{n-1} :: nil$, and $0 \leqslant i < j \leqslant n$

Fig. 8. Operational semantics of STAL, Part II.

5.3 Static semantics

The static semantics of STAL is given by a suite of judgments summarized in Figure 9. The definitions of the relations defined by the judgments are given in Figures 10–12.

The first set of judgments are used to provide simple well-formedness constraints on static objects. The judgments $\Delta \vdash \tau$ and $\Delta \vdash \sigma$ are used to ensure that types and stack types are well-formed in a given context and simply require that the free type and stack type variables be drawn from Δ. The judgment $\vdash \Psi$ asserts that a label type assignment is well formed. The types occurring in Ψ cannot mention free type variables, reflecting the fact that during evaluation, type variables are replaced with closed types. The judgment $\Delta \vdash \Gamma$ asserts that the register assignment Γ is well-formed in that the free type variables occurring within it are drawn from Δ.

The judgment $\Delta \vdash \sigma_1 = \sigma_2$ gives a standard notion of definitional equivalence on stack types. In particular, $(\tau :: \sigma_1)\ @\ \sigma_2$ is equivalent to $\tau :: (\sigma_1\ @\ \sigma_2)$. Furthermore, @ is associative, with *nil* treated as both a left and right unit. To determine when the

Judgment	Meaning	Figure
$\Delta \vdash \tau$	τ is a valid type	10
$\Delta \vdash \sigma$	σ is a valid stack type	
$\vdash \Psi$	Ψ is a valid heap type	
$\Delta \vdash \Gamma$	Γ is a valid register file type	
$\Delta \vdash \sigma_1 = \sigma_2$	σ_1 and σ_2 are equivalent stack types	
$\Delta \vdash \Gamma_1 \leqslant \Gamma_2$	Γ_1 is a register file sub-type of Γ_2	
$\vdash H : \Psi$	the heap H has type Ψ assuming Ψ	11
$\Psi \vdash S : \sigma$	the stack S has type σ	
$\Psi \vdash R : \Gamma$	the register file R has type Γ	
$\Psi \vdash h : \tau$ hval	the heap value h has type τ	
$\Psi;\Delta;\Gamma \vdash v : \tau$	the operand v has type τ	
$\Psi;\Delta;\Gamma \vdash \iota \Rightarrow \Delta';\Gamma'$	instruction ι requires a context of type $\Psi;\Delta;\Gamma$ and produces a context of type $\Psi;\Delta';\Gamma'$	12
$\Psi;\Delta;\Gamma \vdash I$	I is a valid sequence of instructions	11
$\vdash M$	M is a well-typed machine state	

Fig. 9. Static semantics of STAL (judgments).

relation holds, it is possible to calculate a normal form for stack types by orienting the various β-rules from left-to-right to generate a reduction relation, apply this (confluent) reduction in any order until the types are irreducible, and then compare the resulting normal forms up to alpha-equivalence.

The judgment $\Delta \vdash \Gamma_1 \leqslant \Gamma_2$ provides a notion of sub-typing on register file types that is used to type check control transfers as described in Section 2.2.

The rest of the judgments are used to check well-formedness of the various term constructs. The judgment $\vdash H : \Psi$ is used to give a label assignment Ψ to a heap H. The relation holds when the heap values in H have types given by Ψ under the assumptions of Ψ, thereby allowing heap values to refer to one another. Note that we require that the heap of a machine state be closed (with respect to type variables), and thus no context is necessary for checking the heap.

The judgment $\Psi \vdash S : \sigma$ asserts that the stack S is described by the stack type σ. The only interesting rule is (stkeq), which allows us to assign a stack any of its equivalent types. The judgments $\Psi \vdash R : \Gamma$ and $\Psi \vdash h : \tau$ hval are used to type register files and heap values respectively. As with heaps, stacks, and register files, heap values must be closed with respect to type variables and thus the judgments for these terms are not parameterized by a type assignment Δ. The most interesting rule among those that define these judgments is the one for code:

$$\text{code}\,\frac{\Delta \vdash \Gamma \quad \Psi;\Delta;\Gamma \vdash I}{\Psi \vdash \texttt{code}[\Delta]\Gamma.I : \forall[\Delta].\Gamma\ \text{hval}}$$

Here, we require that the typing pre-condition (Γ) be well-formed, and that the instruction sequence I be well-formed under the assumptions of Ψ, Δ, and Γ. Thus, the instructions are type checked assuming that the type variables in Δ are abstract.

The most involved judgment for values is the one for operands. Notice that, since word values are a sub-class of operands, the rules supply typing for both syntactic classes. Most of the rules are straightforward but a few deserve special mention. In particular, the (ptr) rule:

$$\text{ptr}\frac{\Delta \vdash \sigma}{\Psi;\Delta;\cdot \vdash \mathit{ptr}(i) : \mathit{ptr}(\sigma)}(|\sigma| = i)$$

allows a pointer into the stack $\mathit{ptr}(i)$ to be given the type $\mathit{ptr}(\sigma)$ for *any* stack type σ as long as σ is well-formed and has a length of i, where we define the length of a stack type as follows:

$$\begin{array}{rcl} |\mathit{nil}| & = & 0 \\ |\tau{::}\sigma| & = & 1 + |\sigma| \\ |\sigma_1 \mathbin{@} \sigma_2| & = & |\sigma_1| + |\sigma_2| \\ |\rho| & & \text{undefined} \end{array}$$

Notice that the length is undefined when the stack type σ involves a stack type variable. At first, this may seem to make pointers into the stack useless, as our compilation strategy involves writing code that abstracts the tail of the current stack. However, as we evaluate, these stack type variables will be replaced with ground types. It also seems unusual that we can assign a stack pointer any stack type. However, recall that we validate such a pointer by checking that its type is a tail of the current 'true' type of the stack before allowing the pointer to be used.

The judgment $\Psi;\Delta;\Gamma \vdash I$ asserts that a sequence of instructions is well-formed under the assumptions that the heap is described by Ψ, that the type variables in scope are in Δ, and that the registers have types described by Γ. A sequence of the form $\iota;I$ is verified by checking that ι is well-formed under these assumptions and has a post-condition $\Delta';\Gamma'$ which we use as the pre-condition on the rest of the sequence I.

For the degenerate sequence `jmp` v, we must first check that v has a code type of the form $\forall[].\Gamma_2$. Typically, v will be of the form $v'[\psi]$ where v' has a polymorphic type $\forall[\Delta].\Gamma_2$ and ψ is an appropriate instantiation for the bound variables in Δ. We then must verify that the typing pre-condition Γ_2 for the target of the jump is a super-type of the current register type Γ_1. From a Hoare logic perspective, we are ensuring that the typing predicate describing the current state of the machine implies the pre-condition for the destination code.

The degenerate sequence `halt`$[\tau]$ is used to terminate the machine. The intention is that the 'result' of the computation will have type τ and will be placed in a particular register. In this case, the typing rule requires that register `r1` contain the result.

The definition for the remaining judgment, $\Psi;\Delta_1;\Gamma_1 \vdash \iota \Rightarrow \Delta_2;\Gamma_2$ is given in Figure 12. The judgment asserts that the instruction ι is well-formed and has a post-condition $\Delta_2;\Gamma_2$. Most of the rules are straightforward so we restrict attention to the unusual rules involving the stack.

An `salloc` n instruction results in a state where n nonsense values are pushed on the stack, so the post-condition for this instruction simply adds n copies of the type $\top$ to the current type of the stack pointer. Dually, the `sfree` n instruction removes n words from the stack. Thus, we require that the stack type have at least n words described by types $\tau_0, \tau_1, \ldots, \tau_{n-1}$ as the pre-condition, and remove those types in the post-condition. Notice that it is impossible to 'pop' the stack when it is described by *nil* or a stack type variable.

An instruction `mov` r_d, `sp` moves a stack pointer value into register r_d, and thus its post-condition gives r_d the type $ptr(\sigma)$ where σ is the current type of the stack. Dually, the instruction `mov sp`, r_d is used to restore a previously saved stack pointer. However, here we must validate that r_d is still valid, and thus we check that r_d has type $ptr(\sigma_2)$ where σ_2 is a tail of the current stack type.

Loading values from or storing values to the stack is straightforward to verify when the stack pointer is used: we simply need to check that if the instruction uses a stack offset i, then the stack has at least i word values on it. Like `sfree`, these instructions cannot 'read past' a stack described by *nil* or a stack type variable. In the case that we store a value, we must update the type of the stack pointer appropriately.

Loading or storing is not quite as straightforward when we use another register r which has a pointer back into the stack. First, we must validate r by checking that its current type is a tail of the current stack pointer's type. Second, in the case of a store, we must modify both the type of the true stack pointer as well as the type of r. Modifying the type of the stack pointer is necessary to ensure that subsequent stack operations are sound. Modifying the type of r is not strictly speaking necessary, but not doing so could result in invalidating the pointer for subsequent operations.

The principal theorem regarding the semantics is type safety:

Theorem 5.1 (*Type Safety*)
If $\vdash M$ then M cannot become stuck during evaluation.

That is, either M steps to a well-formed terminal configuration or else M diverges, but at no point during evaluation will we reach a configuration in which we are stuck due to a type error. The theorem is proved using the usual Subject Reduction and Progress lemmas, each of which are proved by induction on typing derivations.

Lemma 5.2 (*Subject Reduction*)
If $\vdash P$ and $P \longmapsto P'$ then $\vdash P'$.

Lemma 5.3 (*Progress*)
If $\vdash P$ then either P is a terminal configuration or there exists P' such that $P \longmapsto P'$.

Proofs for both lemmas appear in Appendix A.

6 Related and future work

Our work is partially inspired by Reynolds (1995), who 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 & Abadi (1999) formalize the Java bytecode verifier's treatment of subroutines by giving a type system for a subset of the JVML (Lindholm & Yellin, 1996). In particular, their type system ensures that within a procedure activation, given any program control point, the stack is of the same size each time that control point is reached during execution. Consequently, procedure call must be a primitive construct (which it is in JVML). In contrast, our treatment supports polymorphic stack recursion, and hence procedure calls can be encoded using existing assembly language primitives.

More recently, O'Callahan (1999) has used the mechanisms in this paper to devise an alternative, simpler type system for JVML bytecodes that differs from the official specification (Lindholm & Yellin, 1996). 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.

Necula and Lee introduced the idea of *proof-carrying code* (PCC) as a general framework for certifying compilers (Necula & Lee, 1996; Necula, 1997). In the PCC approach, the compiler produces an explicit proof that the target code respects a given security policy, instead of using typing annotations and an implicit proof. The general framework is quite attractive because in theory, it supports enforcement of any security policy, not just type safety, and because there are no *a priori* restrictions placed upon the code that might hamper optimizations. They demonstrated these ideas by constructing a certifying compiler called Touchstone that mapped a safe subset of C to an instance of PCC and showed that the resulting code could be as fast as the best C compilers (Necula & Lee, 1998). However, the verification condition generator and safety conditions used in the Touchstone PCC system pre-supposed a fairly restrictive calling convention and stack model. In particular, there was no provision for pointers back into the stack, and thus no support for stack allocation of exception contexts, displays, or other data. Fortunately, it seems relatively easy to adapt the ideas behind STAL to the PCC setting to achieve the advantages of each.

Tofte and others (Birkedal *et al.*, 1996; Tofte & Talpin, 1994) have developed an allocation strategy using a region-based model and effects-based type system. 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. Rather, we leverage a combination of linearity and validation to ensure that stack references are sound.

Bailey & Davidson (1995) 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 (see section 4.3); 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. Secondly, threads and advanced mechanisms for implementing first-class continuations such as the work by Hieb *et al.* (1990) cannot be modeled in this system without adding new primitives. Thirdly, and perhaps most importantly, the type system does not protect against stack overflow. One way to support this is to read- and write-protect the last page of the stack, and enforce the constraints that (a) the stack is never grown by more than a page amount, and (b) when the stack is grown, the last word is immediately written. A type-theoretic solution should also be feasible with only moderate additional complexity.

Nevertheless, we claim that the framework presented here is a practical approach to compilation. To substantiate this claim, we have constructed a certifying compiler called Popcorn that maps a type safe subset of C to a variant of STAL, suitably adapted for the 32-bit Intel architecture (Morrisett *et al.*, 1999b). 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, datatypes, 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 assembly language that supports both a heap and a stack. We ensure soundness for stack [de]allocation and stack slot re-use by treating the stack in a quasi-linear fashion and by conservatively validating pointers into the middle of the stack. This discipline forces us to maintain a distinction between stack and heap pointers and to keep track of the relative ordering of stack pointers. Though this prevents us from generally allocating values on the stack, our language is flexible enough to support many common uses of a control stack in that it allows: CPS using either heap or stack allocation, a variety of procedure calling conventions, static links and 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 a single calculus.

A Proof of type soundess for STAL

Lemma A.1 (*Derived Judgements*)

1. If $\vdash H : \Psi$ then $\vdash \Psi$.
2. If $\vdash \Psi$ and $\Psi \vdash R : \Gamma$ then $\cdot \vdash \Gamma$.
3. If $\Delta \vdash \Gamma$ then $\mathrm{fv}(\Gamma) \subseteq \Delta$.

Proof

By straightforward induction on the derivation. □

Lemma A.2 (*Context Strengthening*)

If $\Delta_1 \vdash \sigma_1 = \sigma_2$ and $\Delta_1 \subseteq \Delta_2$ then $\Delta_2 \vdash \sigma_1 = \sigma_2$.

Proof

By straightforward induction on the derivation. □

A closed stack type is always equivalent to a list of ordinary types. Thus, the idea of the *i*-th element of a stack type is useful in proving certain lemmas. We denote this $\sigma[i]$ and define it as follows:

$$\begin{array}{rcl} \rho[i] & & \text{undefined} \\ \mathit{nil}\,[i] & & \text{undefined} \\ (\tau{::}\sigma)[i] & = & \left\{ \begin{array}{ll} \tau & i = 1 \\ \sigma[i-1] & i > 1 \end{array} \right. \\ (\sigma_1 \mathbin{@} \sigma_2)[i] & = & \left\{ \begin{array}{ll} \sigma_1[i] & i \leqslant |\sigma_1| \\ \sigma_2[i - |\sigma_1|] & i > |\sigma_1| \end{array} \right. \end{array}$$

Lemma A.3

If $\cdot \vdash \sigma$ then $\cdot \vdash \sigma = \sigma[1] :: \cdots :: \sigma[|\sigma|] :: \mathit{nil}$.

Proof

By rule (stype), $\mathrm{fv}(\sigma) = \emptyset$ so σ is composed of *nil*, ::, @, and types. The proof is by induction on the structure of σ.

$\sigma = \mathit{nil}$**:** Immediate.

$\sigma = \tau{::}\sigma'$**:** By the induction hypothesis, $\cdot \vdash \sigma' = \sigma'[1] :: \cdots :: \sigma'[|\sigma'|] :: \mathit{nil}$. By definition, the latter equals $\sigma[2] :: \cdots :: \sigma[|\sigma|] :: \mathit{nil}$. The result follows by rule (seq-cons).

$\sigma = \sigma_1 \mathbin{@} \sigma_2$**:** By the induction hypothesis, $\cdot \vdash \sigma_1 = \sigma_1[1] :: \cdots :: \sigma_1[|\sigma_1|] :: \mathit{nil}$ and $\cdot \vdash \sigma_2 = \sigma_2[1] :: \cdots :: \sigma_2[|\sigma_2|] :: \mathit{nil}$. Thus, $\cdot \vdash \sigma = (\sigma[1] :: \cdots :: \sigma[|\sigma_1|] :: \mathit{nil}) \mathbin{@} (\sigma[|\sigma_1| + 1] :: \cdots :: \sigma[|\sigma|] :: \mathit{nil})$. Then by an inner induction on the length of σ_1, using the (stkβ3), (stkβ1), and (seq-trans) rules, the result follows.

□

Lemma A.4 (*Stack Equality*)

1. $\cdot \vdash \sigma_1 = \sigma_2$ if and only if $|\sigma_1| = |\sigma_2|$ and $\forall 1 \leqslant i \leqslant |\sigma_1| : \sigma_1[i] = \sigma_2[i]$.
2. If $\cdot \vdash \sigma_1 = \sigma_2$ then $|\sigma_1| = |\sigma_2|$.
3. If $\cdot \vdash \tau_1 :: \cdots :: \tau_n :: \sigma = \tau_1' :: \cdots :: \tau_n' :: \sigma'$ then $\tau_i = \tau_i'$ for $1 \leqslant i \leqslant n$ and $\cdot \vdash \sigma = \sigma'$.
4. If $\cdot \vdash \tau_n :: \cdots :: \tau_1 :: \mathit{nil} = \sigma_1 \mathbin{@} \sigma_2$ then $\cdot \vdash \tau_n :: \cdots :: \tau_{n-|\sigma_1|+1} :: \mathit{nil} = \sigma_1$ and $\cdot \vdash \tau_{|\sigma_2|} :: \cdots :: \tau_1 :: \mathit{nil} = \sigma_2$.

Proof
Items 2–4 are corollaries of part 1. The left to right part of item 1 is by a straightforward induction on the derivation. The right to left part of item 1 follows by rules (seq-sym), (seq-trans), (seq-cons), and (seq-refl) from Lemma A.3. □

Lemma A.5 (*Type Substitution 1*)
If $\Delta_1 \vdash \tau_i$ then:

1. If $\Delta_2, \vec{\alpha}, \Delta_1 \vdash \Gamma_1 \leqslant \Gamma_2$ then $\Delta_2, \Delta_1 \vdash \Gamma_1[\vec{\tau}/\vec{\alpha}] \leqslant \Gamma_2[\vec{\tau}/\vec{\alpha}]$
2. If $\Delta_2, \vec{\alpha}, \Delta_1 \vdash \sigma_1 = \sigma_2$ then $\Delta_2, \Delta_1 \vdash \sigma_1[\vec{\tau}/\vec{\alpha}] = \sigma_2[\vec{\tau}/\vec{\alpha}]$
3. If $\Delta_2, \vec{\alpha}, \Delta_1 \vdash \tau$ then $\Delta_2, \Delta_1 \vdash \tau[\vec{\tau}/\vec{\alpha}]$
4. If $\Delta_2, \vec{\alpha}, \Delta_1 \vdash \sigma$ then $\Delta_2, \Delta_1 \vdash \sigma[\vec{\tau}/\vec{\alpha}]$
5. If $\Delta_2, \vec{\alpha}, \Delta_1 \vdash \Gamma$ then $\Delta_2, \Delta_1 \vdash \Gamma[\vec{\tau}/\vec{\alpha}]$

Proof
By induction on the derivation using Context Strengthening. □

Lemma A.6 (*Heap Extension*)
If $\vdash H : \Psi$, $\cdot \vdash \tau$, $\Psi\{\ell{:}\tau\} \vdash h : \tau$ hval, and $\ell \notin H$ then:

1. $\vdash \Psi\{\ell{:}\tau\}$
2. $\vdash H\{\ell \mapsto h\} : \Psi\{\ell{:}\tau\}$
3. If $\Psi \vdash R : \Gamma$ then $\Psi\{\ell{:}\tau\} \vdash R : \Gamma$
4. If $\Psi \vdash S : \sigma$ then $\Psi\{\ell{:}\tau\} \vdash S : \sigma$
5. If $\Psi; \Delta; \Gamma \vdash I$ then $\Psi\{\ell{:}\tau\}; \Delta; \Gamma \vdash I$
6. If $\Psi; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma'$ then $\Psi\{\ell{:}\tau\}; \Delta; \Gamma \vdash \iota \Rightarrow \Delta'; \Gamma'$
7. If $\Psi \vdash h : \tau'$ hval then $\Psi\{\ell{:}\tau\} \vdash h : \tau'$ hval
8. If $\Psi; \Delta; \Gamma \vdash v : \tau'$ then $\Psi\{\ell{:}\tau\}; \Delta; \Gamma \vdash v : \tau'$

Proof
Part 1 is immediate. Part 2 follows from parts 1 and 7. Parts 3–8 are by straightforward induction on derivations. □

Lemma A.7
If $\Psi \vdash w_1 :: w_2 :: \cdots :: w_n :: \mathit{nil} : \sigma$, then for some $\tau_1, \ldots, \tau_n$, $\cdot \vdash \sigma = \tau_1 :: \cdots :: \tau_n :: \mathit{nil}$ and $\Psi; \cdot; \cdot \vdash w_i : \tau_i$ for $1 \leqslant i \leqslant n$.

Proof
We proceed by induction on the derivation of $\Psi \vdash w_1 :: w_2 :: \cdots :: w_n :: \mathit{nil} : \sigma$.

(nil)**:** Trivial.

(cons)**:** We know σ is $\tau_1 :: \sigma'$ for some τ_1 and σ' and $\Psi; \cdot; \cdot \vdash w_1 : \tau_1$ and $\Psi \vdash w_2 :: \cdots :: w_n :: \mathit{nil} : \sigma'$. Then by the induction hypothesis, $\cdot \vdash \sigma' = \tau_2 :: \cdots :: \tau_n :: \mathit{nil}$ and $\Psi; \cdot; \cdot \vdash w_i : \tau_i$ for $2 \leqslant i \leqslant n$.

(stkeq)**:** We have $\cdot \vdash \sigma = \sigma'$ and $\Psi \vdash w_1{::}w_2{::}\cdots{::}w_n{::}nil : \sigma'$. By the induction hypothesis, $\cdot \vdash \sigma' = \tau_1{::}\tau_2{::}\cdots{::}\tau_n{::}nil$ and $\Psi;\cdot;\cdot \vdash w_i : \tau_i$ for $1 \leqslant i \leqslant n$. The result then follows from (seq-trans).

□

Lemma A.8 (*Canonical Stack Forms*)

If $\Psi \vdash R : \Gamma$ then $R(\mathtt{sp}) = w_1{::}\cdots{::}w_n{::}nil$ for some $w_1,\ldots,w_n$, $\cdot \vdash \Gamma(\mathtt{sp}) = \tau_1{::}\cdots{::}\tau_n{::}nil$ for some $\tau_1,\ldots,\tau_n$, and $\Psi;\cdot;\cdot \vdash w_i : \tau_i$ for $1 \leqslant i \leqslant n$. Note also that $|R(\mathtt{sp})| = n = |\Gamma(\mathtt{sp})|$.

Proof

By the definition of the abstract syntax, $R(\mathtt{sp}) = w_1{::}\cdots w_n{::}nil$ for some $w_1,\ldots,w_n$ where $n \geqslant 0$. By (regfile) it must be that $\Psi \vdash w_1{::}\cdots w_n{::}nil : \Gamma(\mathtt{sp})$. By the previous Lemma, there exists $\tau_1,\cdots,\tau_n$ such that $\cdot \vdash \Gamma(\mathtt{sp}) = \tau_1{::}\cdots{::}\tau_n{::}nil$ and $\Psi;\cdot;\cdot \vdash w_i : \tau_i$. □

Lemma A.9 (*$\hat{R}$ Typing*)

If $\Psi \vdash R : \Gamma$ and $\Psi;\cdot;\Gamma \vdash v : \tau$ then $\Psi;\cdot;\cdot \vdash \hat{R}(v) : \tau$.

Proof

The proof is by induction on the derivation of $\Psi;\cdot;\Gamma \vdash v : \tau$. Consider the following cases for the last rule used in the derivation:

(label)**,** (int)**,** (ns)**, and** (ptr)**:** Immediate.

(reg)**:** This rule requires $\tau = \Gamma(r)$. The only rule that can type R is (regfile), and this rule requires $\Psi;\cdot;\cdot \vdash R(r) : \Gamma(r)$. The conclusion follows since $\hat{R}(r) = R(r)$.

(tapp)**:** This rule requires that $\tau = \forall[\Delta'].\Gamma'[\tau'/\alpha]$, $v = v'[\tau']$, and $\Psi;\cdot;\Gamma \vdash v' : \forall[\alpha,\Delta'].\Gamma'$. By the induction hypothesis, we deduce $\Psi;\cdot;\cdot \vdash \hat{R}(v') : \forall[\alpha,\Delta'].\Gamma'$, and the rule (tapp) proves $\Psi;\cdot;\cdot \vdash \hat{R}(v')[\tau'] : \tau$. The result follows since $\hat{R}(v) = \hat{R}(v')[\tau']$.

(stapp)**:** This case follows by the same argument as for (tapp).

(pack)**:** This rule requires $\tau = \exists\alpha.\tau'$, $v = pack\ [\tau'',v']\ as\ \exists\alpha.\tau'$, and $\Psi;\cdot;\Gamma \vdash v' : \tau'[\tau''/\alpha]$. By the induction hypothesis, we deduce $\Psi;\cdot;\cdot \vdash \hat{R}(v') : \tau'[\tau''/\alpha]$, and the rule (pack) proves $\Psi;\cdot;\cdot \vdash pack\ [\tau',\hat{R}(v')]\ as\ \tau : \tau$. The result follows since $\hat{R}(v) = pack\ [\tau',\hat{R}(v')]\ as\ \tau$.

□

Lemma A.10 (*Register File Weakening*)

If $\cdot \vdash \Gamma_1 \leqslant \Gamma_2$ and $\Psi \vdash R : \Gamma_1$ then $\Psi \vdash R : \Gamma_2$.

Proof

The judgments $\cdot \vdash \Gamma_1 \leqslant \Gamma_2$ and $\Psi \vdash R : \Gamma_1$ can only be derived by the rules (rf-leq) and (regfile) respectively. It follows that $R = \{\mathtt{sp} \mapsto S, r_1 \mapsto w_1,\ldots,r_m \mapsto w_m\}$, $\Gamma_1 = \{\mathtt{sp}{:}\sigma, r_1{:}\tau_1,\ldots,r_n{:}\tau_n\}$, $\Gamma_2 = \{\mathtt{sp}{:}\sigma', r_1{:}\tau_1,\ldots,r_p{:}\tau_p\}$, $m \geqslant n \geqslant p$, $\Psi \vdash S : \sigma$, $\Psi;\cdot;\cdot \vdash w_i : \tau_i$ for $1 \leqslant i \leqslant n$, and $\cdot \vdash \sigma = \sigma'$. By rule (stkeq) $\Psi \vdash S : \sigma'$. The result follows by rule (regfile). □

Lemma A.11 (*Register File Update*)

1. If $\Psi \vdash R : \Gamma$ and $\Psi;\cdot;\cdot \vdash w : \tau$ then $\Psi \vdash R\{r \mapsto w\} : \Gamma\{r{:}\tau\}$.
2. If $\Psi \vdash R : \Gamma$ and $\Psi \vdash S : \sigma$ then $\Psi \vdash R\{\mathtt{sp} \mapsto S\} : \Gamma\{\mathtt{sp}{:}\sigma\}$.

Proof

For part 1, suppose R is $\{\mathtt{sp} \mapsto S, r_1 \mapsto w_1, \ldots, r_m \mapsto w_m\}$ where r may or may not be in $\{r_1, \ldots, r_m\}$ and Γ is $\{\mathtt{sp}{:}\sigma, r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$. Since $\Psi \vdash R : \Gamma$, by the rule (regfile) it must be the case that $m \geqslant n$ and $\Psi;\cdot;\cdot \vdash w_i : \tau_i$ (for all $1 \leqslant i \leqslant m$ and some $\tau_{n+1}, \ldots, \tau_m$). So certainly for i such that $r_i \neq r$, $\Psi;\cdot;\cdot \vdash w_i : \tau_i$, and by hypothesis $\Psi;\cdot;\cdot \vdash w : \tau$ so by rule (regfile) $\Psi \vdash R\{r \mapsto w\} : \Gamma\{r{:}\tau\}$. Part 2 follows by a similar argument. □

Lemma A.12 (*Canonical Heap Forms*)
If $\Psi \vdash h : \tau$ hval then:

1. If $\tau = \forall[\Delta].\Gamma$ then:
   (a) $h = \mathtt{code}[\Delta]\Gamma.I$
   (b) $\Psi;\Delta;\Gamma \vdash I$
2. If $\tau = \langle \tau_0, \ldots, \tau_n \rangle$ then:
   (a) $h = \langle w_0, \ldots, w_n \rangle$
   (b) $\Psi;\cdot;\cdot \vdash w_i : \tau_i$
3. There are no other possible forms for τ.

Proof

The only applicable rules are (tuple) and (code). The result follows by inspection of those rules. □

Lemma A.13 (*Canonical Forms*)
If $\vdash H : \Psi$, $\Psi \vdash R : \Gamma$, and $\Psi;\cdot;\Gamma \vdash v : \tau$ then:

1. If $\tau = \mathit{int}$ then $\hat{R}(v) = i$.
2. If $\tau = \forall[\Delta_2].\Gamma'$ then:
   (a) $\hat{R}(v) = \ell[\psi]$
   (b) for each $\tau \in \psi$, $\cdot \vdash \tau$
   (c) for each $\sigma \in \psi$, $\cdot \vdash \sigma$
   (d) $H(\ell) = \mathtt{code}[\Delta_1, \Delta_2]\Gamma''.I$
   (e) $\Gamma' = \Gamma''[\psi/\Delta_1]$
   (f) $\Psi;(\Delta_1, \Delta_2);\Gamma'' \vdash I$
3. If $\tau = \langle \tau_0, \ldots, \tau_n \rangle$ then:
   (a) $\hat{R}(v) = \ell$
   (b) $H(\ell) = \langle w_0, \ldots, w_n \rangle$
   (c) $\Psi;\cdot;\cdot \vdash w_i : \tau_i$
4. If $\tau = \exists\alpha.\tau'$ then $\hat{R}(v) = \mathit{pack}\ [\tau'', w']\ \mathit{as}\ \tau'''$, $\cdot \vdash \tau''$, and $\Psi;\cdot;\cdot \vdash w' : \tau'[\tau''/\alpha]$.
5. If $\tau = \mathit{ptr}(\sigma)$ then $\hat{R}(v) = \mathit{ptr}(|\sigma|)$.

Proof
Let $w = \hat{R}(v)$. By $\hat{R}$ Typing, $\Psi;\cdot;\cdot \vdash w : \tau$. The proof proceeds by induction on this judgment's derivation. Consider the last rule used in the derivation:

(label)**:** This rule requires $w = \ell$ and $\Psi(\ell) = \tau$. Since $\vdash H : \Psi$, it follows that $\Psi \vdash H(\ell) : \tau$ hval. Then the result follows by Canonical Heap Forms.
(int)**,** (ns)**, and** (ptr)**:** Immediate.
(reg)**:** This case is not possible since the register assignment is empty.
(tapp)**:** This rule requires that $\tau = \forall[\Delta].\Gamma'[\tau'/\alpha]$, $w = w'[\tau']$, and $\Psi;\cdot;\cdot \vdash w' : \forall[\alpha,\Delta].\Gamma'$. By the induction hypothesis, $w' = \ell[\psi]$, $H(\ell) = \texttt{code}[\Delta',\alpha,\Delta]\Gamma''.I$, $\Gamma' = \Gamma''[\psi/\Delta']$, and $\Psi;(\Delta',\alpha,\Delta);\Gamma'' \vdash I$. Clearly $\Gamma'[\tau'/\alpha] = \Gamma''[\psi/\Delta'][\tau'/\alpha] = \Gamma'[\psi,\tau'/\Delta',\alpha]$. The conclusion follows since $w = \ell[\psi,\tau']$.
(stapp)**:** This case follows by the same argument as for (tapp).
(pack)**:** Immediate.

□

Lemma A.14 (*Type Substitution 2*)
If $\cdot \vdash \tau_i$ then:

1. If $\Psi;(\Delta,\vec{\alpha});\Gamma \vdash I$ then $\Psi;\Delta;\Gamma[\vec{\tau}/\vec{\alpha}] \vdash I[\vec{\tau}/\vec{\alpha}]$.
2. If $\Psi;(\Delta,\vec{\alpha});\Gamma \vdash \iota \Rightarrow (\Delta',\Delta,\vec{\alpha});\Gamma'$ then
   $\Psi;\Delta;\Gamma[\vec{\tau}/\vec{\alpha}] \vdash \iota[\vec{\tau}/\vec{\alpha}] \Rightarrow (\Delta',\Delta);\Gamma'[\vec{\tau}/\vec{\alpha}]$
3. If $\Psi;(\Delta,\vec{\alpha});\Gamma \vdash v : \tau$ then $\Psi;\Delta;\Gamma[\vec{\tau}/\vec{\alpha}] \vdash v[\vec{\tau}/\vec{\alpha}] : \tau[\vec{\tau}/\vec{\alpha}]$

Proof
The proof is by induction on derivations, Context Strengthening, and Type Substitution 1. □

Theorem A.15 (*Subject Reduction*)
If $\vdash P$ and $P \longmapsto P'$ then $\vdash P'$.

Proof
P has the form $(H,R,\iota;I)$ or $(H,R,\texttt{jmp}\ v)$. Let TD be the derivation of $\vdash P$. Consider the following cases for `jmp` or ι:

case *aop***:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_s : \mathit{int} \quad \Psi;\cdot;\Gamma \vdash v : \mathit{int}}{\Psi;\cdot;\Gamma \vdash \mathit{aop}\ r_d,r_s,v \Rightarrow \cdot;\Gamma\{r_d{:}\mathit{int}\}} \quad \Psi;\cdot;\Gamma\{r_d{:}\mathit{int}\} \vdash I}{\Psi;\cdot;\Gamma \vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H,R',I)$ where $R' = R\{r_d \mapsto i\}$ and $i = ||\mathit{aop}||(R(r_s),\hat{R}(v))$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Forms it follows that $R(r_s)$ and $\hat{R}(v)$ are integer literals, and therefore $\Psi;\cdot;\cdot \vdash i : \mathit{int}$. Hence $\Psi \vdash R' : \Gamma\{r_d{:}\mathit{int}\}$ by Register File Update.
3. $\Psi;\cdot;\Gamma\{r_d{:}\mathit{int}\} \vdash I$ is in TD.

case *bop***:** TD has the form:

$$\dfrac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma\vdash r:\mathit{int} \quad \Psi;\cdot;\Gamma\vdash v:\forall[].\Gamma' \quad \cdot\vdash\Gamma\leqslant\Gamma'}{\Psi;\cdot;\Gamma\vdash \mathit{bop}\ r,v\Rightarrow\cdot;\Gamma} \qquad \Psi;\cdot;\Gamma\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

If not $||bop||(R(r))$ then $P'=(H,R,I)$ and $\vdash P'$ follows since $\Psi;\cdot;\Gamma\vdash I$ is in TD. Otherwise the reasoning is exactly as in the case for `jmp` below.

case `jmp`**:** TD has the form:

$$\dfrac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \quad \dfrac{\Psi;\cdot;\Gamma\vdash v:\forall[].\Gamma' \quad \cdot\vdash\Gamma\leqslant\Gamma'}{\Psi;\cdot;\Gamma\vdash \mathtt{jmp}\ v}}{\vdash P}$$

By the operational semantics, $P'=(H,R,I[\psi/\Delta])$ where $\hat{R}(v)=\ell[\psi]$ and $H(\ell)=\mathtt{code}[\Delta]\Gamma''.I$. Then:

1. $\vdash H:\Psi$ is in TD.
2. From $\cdot\vdash\Gamma\leqslant\Gamma'$ and $\Psi\vdash R:\Gamma$ it follows by Register File Weakening that $\Psi\vdash R:\Gamma'$.
3. By Canonical Forms it follows from $\Psi;\cdot;\Gamma\vdash v:\forall[].\Gamma'$ that $\cdot\vdash\psi$, $\Gamma'=\Gamma''[\psi/\Delta]$, and $\Psi;\Delta;\Gamma''\vdash I$. By Type Substitution 2 $\Psi;\cdot;\Gamma''[\psi/\Delta]\vdash I[\psi/\Delta]$, which is the same as $\Psi;\cdot;\Gamma'\vdash I[\psi/\Delta]$.

case `ld`**:** TD has the form:

$$\dfrac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma\vdash r_s:\langle\tau_0,\ldots,\tau_n\rangle \quad 0\leqslant i\leqslant n}{\Psi;\cdot;\Gamma\vdash \mathtt{ld}\ r_d,r_s(i)\Rightarrow\cdot;\Gamma\{r_d{:}\tau_i\}} \qquad \Psi;\cdot;\Gamma\{r_d{:}\tau_i\}\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

By the operational semantics $P'=(H,R',I)$ where $R'=R\{r_d\mapsto w_i\}$, $R(r_s)=\ell$, $H(\ell)=\langle w_0,\ldots,w_m\rangle$, and $0\leqslant i\leqslant m$. Then:

1. $\vdash H:\Psi$ is in TD.
2. By Canonical Forms it follows from $\Psi;\cdot;\Gamma\vdash r_s:\langle\tau_0,\ldots,\tau_n\rangle$ that $m=n$ and $\Psi;\cdot;\cdot\vdash w_j:\tau_j$ for $0\leqslant j\leqslant n$. By Register File update we conclude $\Psi\vdash R':\Gamma\{r_d{:}\tau_i\}$.
3. $\Psi;\cdot;\Gamma\{r_d{:}\tau_i\}\vdash I$ is in TD.

case `malloc`**:** TD has the form:

$$\dfrac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma\vdash v_i:\tau_i\ (\text{for } 1\leqslant i\leqslant n)}{\Psi;\cdot;\Gamma\vdash \mathtt{malloc}\ r,\langle v_1,\ldots,v_n\rangle\Rightarrow\cdot;\Gamma'} \qquad \Psi;\cdot;\Gamma'\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

where $\Gamma'=\Gamma\{r{:}\tau\}$ and $\tau=\langle\tau_1,\ldots,\tau_n\rangle$. By the operational semantics $P'=(H',R',I)$ where $H'=H\{\ell\mapsto\langle\hat{R}(v_1),\ldots,\hat{R}(v_n)\rangle\}$, $R'=R\{r\mapsto\ell\}$, and $\ell\notin H$. Let $\Psi'=\Psi\{\ell{:}\tau\}$, then:

1. By $\hat{R}$ Typing $\Psi;\cdot;\cdot\vdash\hat{R}(v_i):\tau_i$ for $1\leqslant i\leqslant n$. By rule (tuple) $\Psi\vdash\langle\hat{R}(v_1),\ldots,\hat{R}(v_n)\rangle:\tau$ hval. By Heap Extension it follows that $\vdash H':\Psi'$.

2. By rule (label) $\Psi';\cdot;\cdot \vdash \ell : \tau$. By Heap Extension $\Psi' \vdash R : \Gamma$ and it follows by Register File update that $\Psi' \vdash R' : \Gamma'$.
3. By Heap Extension $\Psi';\cdot;\Gamma' \vdash I$.

case `mov` r,v**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash v : \tau}{\Psi;\cdot;\Gamma \vdash \texttt{mov}\ r,v \Rightarrow \cdot;\Gamma\{r{:}\tau\}} \quad \Psi;\cdot;\Gamma\{r{:}\tau\} \vdash I}{\Psi;\cdot;\Gamma \vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H, R', I)$ where $R' = R\{r_d \mapsto \hat{R}(v)\}$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By $\hat{R}$ Typing it follows from $\Psi;\cdot;\Gamma \vdash v : \tau$ that $\Psi;\cdot;\cdot \vdash \hat{R}(v) : \tau$. Using Register File Update we conclude that $\Psi \vdash R' : \Gamma\{r{:}\tau\}$.
3. $\Psi;\cdot;\Gamma\{r{:}\tau\} \vdash I$ is in TD.

case `mov` $r_d,$`sp`**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\begin{array}{c}\dfrac{}{\Psi;\cdot;\Gamma \vdash \texttt{mov}\ r_d,\texttt{sp} \Rightarrow \cdot;\Gamma\{r_d{:}ptr(\sigma)\}}\ (\Gamma(\texttt{sp}) = \sigma) \\ \Psi;\cdot;\Gamma\{r_d{:}ptr(\sigma)\} \vdash I\end{array}}{\Psi;\cdot;\Gamma \vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H, R', I)$ where $R' = R\{r_d \mapsto ptr(|R(\texttt{sp})|)\}$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms it follows from $\Gamma(\texttt{sp}) = \sigma$ that $|\sigma| = |R(\texttt{sp})|$. By Derived Judgments and inversion of rule (rftype) $\cdot \vdash \sigma$, so by (ptr) $\Psi;\cdot;\cdot \vdash ptr(|R(\texttt{sp})|) : ptr(\sigma)$. By Register File Update, $\Psi \vdash R' : \Gamma\{r_d{:}ptr(\sigma)\}$.
3. $\Psi;\cdot;\Gamma\{r_d{:}ptr(\sigma)\} \vdash I$ is in TD.

case `mov sp`$,r_s$**:** TD has the form:

$$\dfrac{\begin{array}{c}\vdash H : \Psi \\ \Psi \vdash R : \Gamma\end{array} \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_s : ptr(\sigma_2) \quad \cdot \vdash \Gamma(\texttt{sp}) = \sigma_1 \mathbin{@} \sigma_2}{\Psi;\cdot;\Gamma \vdash \texttt{mov sp},r_s \Rightarrow \cdot;\Gamma\{\texttt{sp}{:}\sigma_2\}} \quad \Psi;\cdot;\Gamma\{\texttt{sp}{:}\sigma_2\} \vdash I}{\Psi;\cdot;\Gamma \vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H, R', I)$ where $R' = R\{\texttt{sp} \mapsto S\}$, $S = w_i{::}\cdots{::}w_1{::}nil$, $R(\texttt{sp}) = w_n{::}\cdots w_i{::}\cdots{::}w_1{::}nil$, $R(r_s) = ptr(i)$, and $0 \leqslant i \leqslant n$. Let $\sigma_3 = \tau_i{::}\cdots{::}\tau_1{::}nil$, then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms, $\cdot \vdash \Gamma(\texttt{sp}) = \tau_n{::}\cdots{::}\tau_1{::}nil$ $(*)$ and $\Psi;\cdot;\cdot \vdash w_j : \tau_j$ for $1 \leqslant j \leqslant n$. By repeated use of rules (nil) and (cons), $\Psi \vdash S : \sigma_3$. By $\hat{R}$ Typing, $\Psi;\cdot;\cdot \vdash ptr(i) : ptr(\sigma_2)$, and by inversion on (ptr), we have $|\sigma_2| = i$. By Stack Equality, it follows from $\cdot \vdash \Gamma(\texttt{sp}) = \sigma_1 \mathbin{@} \sigma_2$, $(*)$, (seq-sym), and (seq-trans) that $\cdot \vdash \sigma_3 = \sigma_2$. By rule (stkeq) $\Psi \vdash S : \sigma_2$. By Register File Update it follows that $\Psi \vdash R' : \Gamma\{\texttt{sp}{:}\sigma_2\}$.

3. $\Psi;\cdot;\Gamma\{\mathtt{sp}{:}\sigma_2\} \vdash$ is in TD.

case `salloc`**:** TD has the form:

$$\frac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \qquad \dfrac{\dfrac{\Gamma(\mathtt{sp})=\sigma}{\Psi;\cdot;\Gamma\vdash \mathtt{salloc}\ n \Rightarrow \cdot;\Gamma'} \quad \Psi;\cdot;\Gamma'\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

where $\Gamma' = \Gamma\{\mathtt{sp}{:}\sigma\}$ and $\sigma = \underbrace{\top::\cdots::\top}_{n}::\Gamma(\mathtt{sp})$. By the operational semantics $P' = (H,R',I)$ where $R' = R\{\mathtt{sp}\mapsto S\}$ and $S = \underbrace{ns::\cdots::ns}_{n}::R(\mathtt{sp})$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By the (regfile) rule it must be that $\Psi;\cdot;\cdot\vdash R(\mathtt{sp}) : \Gamma(\mathtt{sp})$. By repeated use of the (ns) and (cons) rules we can conclude that $\Psi;\cdot\vdash S : \sigma$. Using Register File Update we conclude that $\Psi\vdash R' : \Gamma'$.
3. $\Psi;\cdot;\Gamma'\vdash I$ is in TD.

case `sfree`**:** TD has the form:

$$\frac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \qquad \dfrac{\dfrac{\cdot\vdash\Gamma(\mathtt{sp}) = \tau_1::\cdots::\tau_n::\sigma_2}{\Psi;\cdot;\Gamma\vdash \mathtt{sfree}\ n \Rightarrow \cdot;\Gamma\{\mathtt{sp}{:}\sigma_2\}} \quad \Psi;\cdot;\Gamma\{\mathtt{sp}{:}\sigma_2\}\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H,R',I)$ where $R' = R\{\mathtt{sp}\mapsto S\}$ and $R(\mathtt{sp}) = w_1::\cdots::w_n::S$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms, for some $m \geqslant n$, $S = w_{n+1}::\cdots::w_m::nil$, $\cdot\vdash\Gamma(\mathtt{sp}) = \tau'_1::\cdots::\tau'_m::nil$ $(*)$, and $\Psi;\cdot;\cdot\vdash w_i : \tau'_i$ for $1\leqslant i\leqslant m$. By repeated use of the (nil) and (cons) rules, $\Psi\vdash S : \tau'_{n+1}::\cdots::\tau'_m::nil$. By Stack Equality it follows from $\cdot\vdash\Gamma(\mathtt{sp}) = \tau_1::\cdots::\tau_n::\sigma_2$, $(*)$, (seq-sym), and (seq-trans) that $\cdot\vdash\tau'_{n+1}::\cdots::\tau'_m::nil = \sigma_2$. By rule (stkeq) $\Psi\vdash S : \sigma_2$. By Register File Update $\Psi\vdash R' : \Gamma\{\mathtt{sp}{:}\sigma_2\}$.
3. $\Psi;\cdot;\Gamma\{\mathtt{sp}{:}\sigma_2\}\vdash I$ is in TD.

case `sld` $r_d, \mathtt{sp}(i)$**:** TD has the form:

$$\frac{\vdash H:\Psi \quad \Psi\vdash R:\Gamma \qquad \dfrac{\dfrac{\cdot\vdash\Gamma(\mathtt{sp}) = \tau_0::\cdots::\tau_i::\sigma_2 \quad 0\leqslant i}{\Psi;\cdot;\Gamma\vdash \mathtt{sld}\ r_d,\mathtt{sp}(i) \Rightarrow \cdot;\Gamma\{r_d{:}\tau_i\}} \quad \Psi;\cdot;\Gamma\{r_d{:}\tau_i\}\vdash I}{\Psi;\cdot;\Gamma\vdash \iota;I}}{\vdash P}$$

By the operational semantics $P' = (H,R',I)$ where $R' = R\{r_d\mapsto w_i\}$, $R(\mathtt{sp}) = w_0::\cdots::w_n::nil$, and $0\leqslant i\leqslant n$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms, $\vdash\Gamma(\mathtt{sp}) = \tau'_0::\cdots::\tau'_n::nil$ $(*)$ and $\Psi;\cdot;\cdot\vdash w_i : \tau'_i$. By Stack Equality, it follows from $\cdot\vdash\Gamma(\mathtt{sp}) = \tau_0::\cdots::\tau_i::\sigma_2$, $(*)$, (seq-sym), and (seq-trans) that $\tau'_i = \tau_i$. By Register File Update, we may conclude $\Psi\vdash R' : \Gamma\{r_d{:}\tau_i\}$.

3. $\Psi;\cdot;\Gamma\{r_d{:}\tau_i\} \vdash I$ is in TD.

case $\mathtt{sld}\ r_d, r_s(i)$**:** TD has the form:

$$\dfrac{\begin{array}{c}\vdash H : \Psi \\ \Psi \vdash R : \Gamma\end{array} \qquad \dfrac{\dfrac{\begin{array}{cc} 0 \leqslant i & \cdot \vdash \Gamma(\mathtt{sp}) = \sigma_1 \mathbin{@} \sigma_2 \\ \Psi;\cdot;\Gamma \vdash r_s : ptr(\sigma_2) & \cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3 \end{array}}{\Psi;\cdot;\Gamma \vdash \mathtt{sld}\ r_d, r_s(i) \Rightarrow \cdot;\Gamma\{r_d{:}\tau_i\}} \qquad \Psi;\cdot;\Gamma\{r_d{:}\tau_i\} \vdash I}{\Psi;\cdot;\Gamma \vdash \mathtt{sld}\ r_d, r_s(i); I}}{\vdash P}$$

By the operational semantics $P' = (H, R', I)$ where $R' = R\{r_d \mapsto w_{j-i}\}$, $R(\mathtt{sp}) = w_n :: \cdots :: w_1 :: nil$, $R(r_s) = ptr(j)$, and $0 \leqslant i < j \leqslant n$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms, $\cdot \vdash \Gamma(\mathtt{sp}) = \tau'_n :: \cdots :: \tau'_1 :: nil$ (1) and $\Psi;\cdot;\cdot \vdash w_k : \tau'_k$. By Canonical Forms, $j = |\sigma_2|$ (2). By Stack Equality, it follows from $\cdot \vdash \Gamma(\mathtt{sp}) = \sigma_1 \mathbin{@} \sigma_2$, (1), (2), (seq-sym), and (seq-trans) that $\cdot \vdash \tau'_j :: \cdots :: \tau'_1 :: nil = \sigma_2$. From the latter, $\cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3$, and rule (seq-trans) it follows that $\cdot \vdash \tau'_j :: \cdots :: \tau'_1 :: nil = \tau_0 :: \cdots :: \tau_i :: \sigma_3$. Then by Stack Equality, $\tau'_{j-i} = \tau_i$. By Register File Update, $\Psi \vdash R' : \Gamma\{r_d{:}\tau_i\}$.
3. $\Psi;\cdot;\Gamma\{r_d{:}\tau_i\} \vdash I$ is in TD.

case $\mathtt{sst}\ \mathtt{sp}(i), r_s$**:** TD has the form:

$$\dfrac{\begin{array}{c}\vdash H : \Psi \\ \Psi \vdash R : \Gamma\end{array} \qquad \dfrac{\dfrac{\cdot \vdash \Gamma(\mathtt{sp}) = \tau_0 :: \cdots :: \tau_i :: \sigma_2 \quad \Psi;\cdot;\Gamma \vdash r_s : \tau \quad 0 \leqslant i}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ \mathtt{sp}(i), r_s \Rightarrow \cdot;\Gamma'} \qquad \Psi;\cdot;\Gamma' \vdash I}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ \mathtt{sp}(i), r_s; I}}{\vdash P}$$

where $\Gamma' = \Gamma\{\mathtt{sp}{:}\sigma\}$ and $\sigma = \tau_0 :: \cdots :: \tau_{i-1} :: \tau :: \sigma_2$. By the operational semantics $P' = (H, R', I)$ where $R' = R\{\mathtt{sp} \mapsto w_0 :: \cdots :: w_{i-1} :: R(r_s) :: S\}$ and $R(\mathtt{sp}) = w_0 :: \cdots :: w_i :: S$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Stack Forms, $\cdot \vdash \Gamma(\mathtt{sp}) = \tau'_0 :: \cdots :: \tau'_n :: nil$ (1) and for some $n \geqslant i$ and w_{i+1} through w_n, $S = w_{i+1} :: \cdots :: w_n :: nil$ and $\Psi;\cdot;\cdot \vdash w_j : \tau'_j$. Let $\sigma' = \tau'_0 :: \cdots :: \tau'_{i-1} :: \tau :: \tau'_{i+1} :: \cdots :: \tau'_n :: nil$. By $\hat{R}$ Typing, $\Psi;\cdot;\cdot \vdash R(r_s) : \tau$. By repeated use of the (cons) and (nil) rules, $\Psi \vdash w_0 :: \cdots :: w_{i-1} :: R(r_s) :: S : \sigma'$. By Stack Equality it follows from $\cdot \vdash \Gamma(\mathtt{sp}) = \tau_0 :: \cdots :: \tau_i :: \sigma_2$, (1), (seq-sym), and (seq-trans) that $\tau'_j = \tau_j$ and $\cdot \vdash \tau'_{i+1} :: \cdots :: \tau'_n :: nil = \sigma_2$. By repeated use of the (seq-cons) rule, $\cdot \vdash \sigma' = \sigma$. By rule (stkeq), $\Psi \vdash w_0 :: \cdots :: w_{i-1} :: R(r_s) :: S : \sigma$. By Register File Update it follows that $\Psi \vdash R' : \Gamma'$.
3. $\Psi;\cdot;\Gamma' \vdash I$ is in TD.

case `sst` $r_d(i), r_s$**:** TD has the form:

$$\frac{\begin{array}{c}\vdash H : \Psi\\ \Psi \vdash R : \Gamma\end{array} \qquad \dfrac{\dfrac{\begin{array}{cc}0 \leqslant i & \cdot \vdash \Gamma(\mathtt{sp}) = \sigma_1 \mathbin{@} \sigma_2\\ \Psi;\cdot;\Gamma \vdash r_d : ptr(\sigma_2) & \cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3\\ \Psi;\cdot;\Gamma \vdash r_s : \tau & \cdot \vdash \sigma_4 = \tau_0 :: \cdots :: \tau_{i-1} :: \tau :: \sigma_3\end{array}}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ r_d(i), r_s \Rightarrow \cdot;\Gamma'} \qquad \Psi;\cdot;\Gamma' \vdash I}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ r_d(i), r_s; I}}{\vdash P}$$

where $\Gamma' = \Gamma\{\mathtt{sp}{:}\sigma_1 \mathbin{@} \sigma_4, r_d{:}ptr(\sigma_4)\}$. By the operational semantics $P'{=}(H, R', I)$ where $R' = R\{\mathtt{sp} \mapsto S'\}$, $S' = w_n :: \cdots :: w_{j-i+1} :: R(r_s) :: w_{j-i-1} :: \cdots :: w_1 :: nil$, $R(r_d) = ptr(j)$, $R(\mathtt{sp}) = w_n :: \cdots :: w_1 :: nil$, and $0 \leqslant i < j \leqslant n$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Forms $j = |\sigma_2|$, and by Stack Equality $|\sigma_2| = 1 + i + |\sigma_3|$ and $|\sigma_4| = 1 + i + |\sigma_3|$. Therefore $j = |\sigma_4|$, and by (ptr) $\Psi;\cdot;\cdot \vdash ptr(j) : \sigma_4$. By Register File Update, $\Psi \vdash R : \Gamma\{r_d{:}ptr(\sigma_4)\}$. By Canonical Stack Forms, $\cdot \vdash \Gamma(\mathtt{sp}) = \tau'_n :: \cdots :: \tau'_1 :: nil$ (1) and $\Psi;\cdot;\cdot \vdash w_k : \tau'_k$ for $1 \leqslant k \leqslant n$. Let $\sigma = \tau'_n :: \cdots :: \tau'_{j-i+1} :: \tau :: \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil$. By $\hat{R}$ Typing, $\Psi;\cdot;\cdot \vdash R(r_s) : \tau$. By repeated use of the (cons) and (nil) rules, $\Psi \vdash S' : \sigma$. By Stack Equality it follows from $\cdot \vdash \Gamma(sp) = \sigma_1 \mathbin{@} \sigma_2$, (1), (seq-sym), and (seq-trans) that $\cdot \vdash \tau'_n :: \cdots :: \tau'_{j+1} :: nil = \sigma_1$ and $\cdot \vdash \tau'_j :: \cdots :: \tau'_1 :: nil = \sigma_2$. By the latter, $\cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3$, (seq-sym), and (seq-trans), by Stack Equality it follows that $\tau_k = \tau'_{j-k}$ (for $0 \leqslant k \leqslant i$) and $\cdot \vdash \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil = \sigma_3$. By repeated use of rule (seq-cons) $\cdot \vdash \tau'_j :: \cdots :: \tau'_{j-i+1} :: \tau :: \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil = \tau_0 :: \cdots :: \tau_{i-1} :: \tau :: \sigma_3$. By $\cdot \vdash \sigma_4 = \tau_0 :: \cdots :: \tau_{i-1} :: \tau :: \sigma_3$, (seq-sym), and (seq-trans), it follows that $\cdot \vdash \tau'_j :: \cdots :: \tau'_{j-i+1} :: \tau :: \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil = \sigma_4$. Using rule (seq-append), it follows that $\cdot \vdash (\tau'_n :: \cdots :: \tau'_{j+1} :: nil) \mathbin{@} (\tau'_j :: \cdots :: \tau'_{j-i+1} :: \tau :: \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil) = \sigma_1 \mathbin{@} \sigma_4$. By repeated use of rules (cons), (stkβ3), and (stkβ1) it follows that $\cdot \vdash (\tau'_n :: \cdots :: \tau'_{j+1} :: nil) \mathbin{@} (\tau'_j :: \cdots :: \tau'_{j-i+1} :: \tau :: \tau'_{j-i-1} :: \cdots :: \tau'_1 :: nil) = \sigma$. Then using rules (seq-sym) and (seq-trans), we may conclude $\cdot \vdash \sigma = \sigma_1 \mathbin{@} \sigma_4$. By (stkeq) $\Psi \vdash S' : \sigma_1 \mathbin{@} \sigma_4$. By Register File Update it follows that $\Psi \vdash R' : \Gamma'$.
3. $\Psi;\cdot;\Gamma' \vdash I$ is in TD.

case `st`**:** TD has the form:

$$\frac{\begin{array}{c}\vdash H : \Psi\\ \Psi \vdash R : \Gamma\end{array} \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_d : \langle \tau_0, \ldots, \tau_n \rangle \quad \Psi;\cdot;\Gamma \vdash r_s : \tau_i \quad 0 \leqslant i \leqslant n}{\Psi;\cdot;\Gamma \vdash \mathtt{st}\ r_d(i), r_s \Rightarrow \cdot;\Gamma} \qquad \Psi;\cdot;\Gamma \vdash I}{\Psi;\cdot;\Gamma \vdash \iota; I}}{\vdash P}$$

By the operational semantics, $P' = (H', R, I)$ where $H' = H\{\ell \mapsto h'\}$ and $h' = \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_n \rangle$, $R(r_d) = \ell$, $H(\ell) = \langle w_0, \ldots, w_n \rangle$, and $0 \leqslant i \leqslant n$. Then:

1. Inspection of the rule (heap) reveals that $\vdash H' : \Psi$ if $\Psi \vdash h' : \Psi(\ell)$ hval. By $\hat{R}$ Typing, $\Psi;\cdot;\cdot \vdash \ell : \langle \tau_0, \ldots, \tau_n \rangle$ and, by Canonical Forms, $\Psi;\cdot;\cdot \vdash w_j : \tau_j$ for $0 \leqslant j \leqslant n$ (*). Since the former can only be concluded by the rule (label), it

must be that $\Psi(\ell) = \langle \tau_0, \ldots, \tau_n \rangle$. By $\hat{R}$ Typing $\Psi;\cdot;\cdot \vdash R(r_s) : \tau_i$. By $(*)$ and rule (tuple) $\Psi \vdash h' : \Psi(\ell)$ hval as required.

2. $\Psi \vdash R : \Gamma$ is in TD.
3. $\Psi;\cdot;\Gamma \vdash I$ is in TD.

case `unpack`**:** TD has the form:

$$\dfrac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash v : \exists\alpha.\tau}{\Psi;\cdot;\Gamma \vdash \texttt{unpack}\ [\alpha, r], v \Rightarrow \alpha;\Gamma\{r{:}\tau\}} \qquad \Psi;\alpha;\Gamma\{r{:}\tau\} \vdash I}{\Psi;\cdot;\Gamma \vdash \iota;I}}{\vdash P}$$

By the operational semantics, $P' = (H, R\{r \mapsto w\}, I[\tau'/\alpha])$ where $\hat{R}(v) = \mathit{pack}\ [\tau', w]$ $\mathit{as}\ \tau''$. Let $\tau''' = \tau[\tau'/\alpha]$. Then:

1. $\vdash H : \Psi$ is in TD.
2. By Canonical Forms it follows from $\Psi;\cdot;\Gamma \vdash v : \exists\alpha.\tau$ that $\cdot \vdash \tau'$ and $\Psi;\cdot;\cdot \vdash w : \tau'''$. By Register File Update it follows that $\Psi \vdash R\{r \mapsto w\} : \Gamma\{r{:}\tau'''\}$.
3. By Type Substitution 2 and $\Psi;\alpha;\Gamma\{r{:}\tau\} \vdash I$ it follows that $\Psi';\cdot;\Gamma\{r{:}\tau'''\} \vdash I[\tau'/\alpha]$. (Note that $\Gamma[\tau'/\alpha] = \Gamma$ follows from Derived Judgments.)

□

Theorem A.16 (*Progress*)
If $\vdash P$ then either P has the form $(H, R\{\texttt{r1} \mapsto w\}, \texttt{halt}[\tau])$ (and, moreover, $\Psi;\cdot;\cdot \vdash w : \tau$ for some Ψ such that $\vdash H : \Psi$) or there exists P' such that $P \longmapsto P'$.

Proof
Suppose $P = (H, R, I_{\text{full}})$. Let TD be the derivation of $\vdash P$. The proof is by cases on the first instruction of I_{full}.

case *aop***:** TD has the form:

$$\dfrac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_s : \mathit{int} \qquad \Psi;\cdot;\Gamma \vdash v : \mathit{int}}{\Psi;\cdot;\Gamma \vdash \mathit{aop}\ \ r_d, r_s, v \Rightarrow \cdot;\Gamma\{r_d{:}\mathit{int}\}} \qquad \Psi;\cdot;\Gamma\{r_d{:}\mathit{int}\} \vdash I}{\Psi;\cdot;\Gamma \vdash \mathit{aop}\ \ r_d, r_s, v;I}}{\vdash P}$$

By Canonical Forms, $R(r_s)$ and $\hat{R}(v)$ each represent integer literals. Hence $P \longmapsto (H, R\{r_d \mapsto ||\mathit{aop}||(R(r_s), \hat{R}(v)), I)$.

case *bop***:** TD has the form:

$$\dfrac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r : \mathit{int} \qquad \Psi;\cdot;\Gamma \vdash v : \forall[\,].\Gamma' \qquad \cdot \vdash \Gamma \leqslant \Gamma'}{\Psi;\cdot;\Gamma \vdash \mathit{bop}\ \ r, v \Rightarrow \cdot;\Gamma} \qquad \Psi;\cdot;\Gamma \vdash I}{\Psi;\cdot;\Gamma \vdash \mathit{bop}\ \ r, v;I}}{\vdash P}$$

By Canonical Forms $R(r)$ is an integer literal and $\hat{R}(v) = \ell[\psi]$ where $H(\ell) = \texttt{code}[\Delta]\Gamma''.I'$ and $|\psi| = |\Delta|$. If not $||\mathit{bop}||R(r)$ then $P \longmapsto (H, R, I)$. If $||\mathit{bop}||R(r)$ then $P \longmapsto (H, R, I'[\psi/\Delta])$.

case `halt`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\Psi;\cdot;\Gamma \vdash \texttt{r1} : \tau}{\Psi;\cdot;\Gamma \vdash \texttt{halt}[\tau]}}{\vdash (H, R, \texttt{halt}[\tau])}$$

By $\hat{R}$ Typing we may deduce that $\hat{R}(\texttt{r1})$ is defined and $\Psi;\cdot;\cdot \vdash \hat{R}(\texttt{r1}) : \tau$. In other words, $R = R'\{\texttt{r1} \mapsto w\}$ and $\Psi;\cdot;\cdot \vdash w : \tau$.

case `jmp`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\Psi;\cdot;\Gamma \vdash v : \forall[\,].\Gamma' \quad \cdot \vdash \Gamma \leqslant \Gamma'}{\Psi;\cdot;\Gamma \vdash \texttt{jmp}\ v}}{\vdash P}$$

By Canonical Forms, $\hat{R}(v) = \ell[\psi]$ where $H(\ell) = \texttt{code}[\Delta]\Gamma''.I'$ and $|\psi| = |\Delta|$. Hence $P \longmapsto (H, R, I'[\psi/\Delta])$.

case `ld`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_s : \langle \tau_0, \ldots, \tau_n \rangle \quad 0 \leqslant i \leqslant n}{\Psi;\cdot;\Gamma \vdash \texttt{ld}\ r_d, r_s(i) \Rightarrow \cdot;\Gamma\{r_d{:}\tau_i\}} \quad \Psi;\cdot;\Gamma\{r_d{:}\tau_i\} \vdash I}{\Psi;\cdot;\Gamma \vdash \texttt{ld}\ r_d, r_s(i); I}}{\vdash P}$$

By Canonical Forms, $R(r_s) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_n \rangle$. Therefore, by the operational semantics $P \longmapsto (H, R\{r_d \mapsto w_i\}, I)$.

case `malloc`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash v_i : \tau_i \text{ (for } 1 \leqslant i \leqslant n)}{\Psi;\cdot;\Gamma \vdash \texttt{malloc}\ r, \langle v_1, \ldots, v_n \rangle \Rightarrow \cdot;\Gamma'} \quad \Psi;\cdot;\Gamma' \vdash I}{\Psi;\cdot;\Gamma \vdash \iota; I}}{\vdash P}$$

By $\hat{R}$ Typing, $\hat{R}(v_i)$ is well-defined (for $1 \leqslant i \leqslant n$). Then $P \longmapsto (H\{\ell \mapsto \langle \hat{R}(v_1), \ldots, \hat{R}(v_n) \rangle\}, R\{r \mapsto \ell\}, I)$ for some $\ell \notin H$.

case `mov` r, v**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash v : \tau}{\Psi;\cdot;\Gamma \vdash \texttt{mov}\ r, v \Rightarrow \cdot;\Gamma\{r{:}\tau\}} \quad \Psi;\cdot;\Gamma\{r{:}\tau\} \vdash I}{\Psi;\cdot;\Gamma \vdash \texttt{mov}\ r, v; I}}{\vdash P}$$

By $\hat{R}$ Typing $\hat{R}(v)$ is well-defined. Hence $P \longmapsto (H, R\{r \mapsto \hat{R}(v)\}, I)$.

case `mov` r_d, `sp`**:** Suppose I_{full} has the form `mov` $r_d, \texttt{sp}; I$ then $P \longmapsto (H, R\{r_d \mapsto ptr(|R(\texttt{sp})|)\}, I)$.

case `mov sp`, r_s**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\cdot \vdash \Gamma(\texttt{sp}) = \sigma_1 \mathbin{@} \sigma_2 \quad \Psi;\cdot;\Gamma \vdash r_s : ptr(\sigma_2)}{\Psi;\cdot;\Gamma \vdash \texttt{mov sp}, r_s \Rightarrow \cdot;\Gamma\{\texttt{sp}{:}\sigma_2\}}}{\Psi;\cdot;\Gamma \vdash \texttt{mov sp}, r_s; I}}{\vdash P}$$

By Canonical Forms, $R(r_s) = ptr(|\sigma_2|)$. By Canonical Stack Forms $R(\texttt{sp}) =$

$w_n :: \cdots :: w_1 :: \mathit{nil}$ where $n = |\Gamma(\texttt{sp})|$. By Stack Equality $|\Gamma(\texttt{sp})| = |\sigma_1 \mathbin{@} \sigma_2|$, and by definition the latter equals $|\sigma_1| + |\sigma_2|$. So $0 \leqslant |\sigma_2| \leqslant n$, hence $P \longmapsto (H, R\{\texttt{sp} \mapsto w_{|\sigma_2|} :: \cdots :: w_1 :: \mathit{nil}\}, I)$.

case $\texttt{salloc}$**:** Suppose I_{full} has the form $\texttt{salloc}\ n; I$ then:

$$P \longmapsto (H, R\{\texttt{sp} \mapsto \underbrace{\mathit{ns} :: \cdots :: \mathit{ns}}_{n} :: R(\texttt{sp})\}, I)$$

case $\texttt{sfree}$**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\cdot \vdash \Gamma(\texttt{sp}) = \tau_1 :: \cdots :: \tau_n :: \sigma_2}{\Psi; \cdot; \Gamma \vdash \texttt{sfree}\ n \Rightarrow \cdot; \Gamma\{\texttt{sp}:\sigma_2\}} \quad \Psi; \cdot; \Gamma\{\texttt{sp}:\sigma_2\} \vdash I}{\Psi; \cdot; \Gamma \vdash \texttt{sfree}\ n; I}}{\vdash P}$$

By Canonical Stack Forms $R(\texttt{sp}) = w_m :: \cdots :: w_1 :: \mathit{nil}$ where $m = |\Gamma(\texttt{sp})|$. By Stack Equality $|\Gamma(\texttt{sp})| = |\tau_1 :: \cdots :: \tau_n :: \sigma_2|$, and the latter equals $n + |\sigma_2|$, so $m \geqslant n$. Hence $P \longmapsto (H, R\{\texttt{sp} \mapsto w_{m-n} :: \cdots :: w_1 :: \mathit{nil}\}, I)$.

case $\texttt{sld}\ r_d, \texttt{sp}(i)$**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\cdot \vdash \Gamma(\texttt{sp}) = \tau_0 :: \cdots :: \tau_i :: \sigma_2 \quad 0 \leqslant i}{\Psi; \cdot; \Gamma \vdash \texttt{sld}\ r_d, \texttt{sp}(i) \Rightarrow \cdot; \Gamma\{r_d : \tau_i\}}}{\Psi; \cdot; \Gamma \vdash \texttt{sld}\ r_d, \texttt{sp}(i); I}}{\vdash P}$$

By Canonical Stack Forms $R(\texttt{sp}) = w_n :: \cdots :: w_1 :: \mathit{nil}$ where $n = |\Gamma(\texttt{sp})|$. By Stack Equality $|\Gamma(\texttt{sp})| = |\tau_0 :: \cdots :: \tau_i :: \sigma_2|$, and the latter equals $1 + i + |\sigma_2|$, so $0 \leqslant i < n$. Hence $P \longmapsto (H, R\{r_d \mapsto w_{n-i}\}, I)$.

case $\texttt{sld}\ r_d, r_s(i)$**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\begin{matrix} 0 \leqslant i & \cdot \vdash \Gamma(\texttt{sp}) = \sigma_1 \mathbin{@} \sigma_2 \\ \Psi; \cdot; \Gamma \vdash r_s : \mathit{ptr}(\sigma_2) & \cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3 \end{matrix}}{\Psi; \cdot; \Gamma \vdash \texttt{sld}\ r_d, r_s(i) \Rightarrow \cdot; \Gamma\{r_d : \tau_i\}}}{\Psi; \cdot; \Gamma \vdash \texttt{sld}\ r_d, r_s(i); I}}{\vdash P}$$

By Canonical Forms $R(r_s) = \mathit{ptr}(|\sigma_2|)$. By Canonical Stack Forms $R(\texttt{sp}) = w_n :: \cdots :: w_1 :: \mathit{nil}$ where $n = |\Gamma(\texttt{sp})|$. By Stack Equality $|\Gamma(\texttt{sp})| = |\sigma_1 \mathbin{@} \sigma_2|$, and the latter equals $|\sigma_1| + |\sigma_2|$, so $|\sigma_2| \leqslant n$. Again by Stack Equality $|\sigma_2| = |\tau_0 :: \cdots :: \tau_i :: \sigma_3|$, and the latter equals $1 + i + |\sigma_3|$, so $0 \leqslant i < |\sigma_2|$. Hence $P \longmapsto (H, R\{r_d \mapsto w_{|\sigma_2|-i}\}, I)$.

case $\texttt{sst}\ \texttt{sp}(i), r_s$**:** TD has the form:

$$\dfrac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \dfrac{\dfrac{\cdot \vdash \Gamma(\texttt{sp}) = \tau_0 :: \cdots :: \tau_i :: \sigma_2 \quad \Psi; \cdot; \Gamma \vdash r_s : \tau \quad 0 \leqslant i}{\Psi; \cdot; \Gamma \vdash \texttt{sst}\ \texttt{sp}(i), r_s \Rightarrow \cdot; \Gamma'}}{\Psi; \cdot; \Gamma \vdash \texttt{sst}\ \texttt{sp}(i), r_s; I}}{\vdash P}$$

By Canonical Stack Forms $R(\texttt{sp}) = w_n :: \cdots :: w_1 :: \mathit{nil}$ where $n = |\Gamma(\texttt{sp})|$. By Stack Equality $|\Gamma(\texttt{sp})| = |\tau_0 :: \cdots :: \tau_i :: \sigma_2|$, and the latter equals $1 + i + |\sigma_2|$, so $0 \leqslant i < n$. By $\hat{R}$ Typing, $R(r_s)$ is defined. Hence $P \longmapsto (H, R\{\texttt{sp} \mapsto w_n :: \cdots :: w_{n-i+1} :: R(r_s) :: w_{n-i-1} :: \cdots \mathit{nil}\}, I)$.

case `sst` $r_d(i), r_s$**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\begin{array}{cc} 0 \leqslant i & \cdot \vdash \Gamma(\mathtt{sp}) = \sigma_1 \mathbin{@} \sigma_2 \\ \Psi;\cdot;\Gamma \vdash r_d : ptr(\sigma_2) & \cdot \vdash \sigma_2 = \tau_0 :: \cdots :: \tau_i :: \sigma_3 \\ \Psi;\cdot;\Gamma \vdash r_s : \tau & \end{array}}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ r_d(i), r_s \Rightarrow \cdot;\Gamma'}}{\Psi;\cdot;\Gamma \vdash \mathtt{sst}\ r_d(i), r_s; I}}{\vdash P}$$

By Canonical Forms, $R(r_d) = ptr(|\sigma_2|)$. By Canonical Stack Forms $R(\mathtt{sp}) = w_n :: \cdots :: w_1 :: nil$ where $n = |\Gamma(\mathtt{sp})|$. By Stack Equality $|\Gamma(\mathtt{sp})| = |\sigma_1 \mathbin{@} \sigma_2|$, and the latter equals $|\sigma_1| + |\sigma_2|$, so $|\sigma_2| \leqslant n$. Again by Stack Equality $|\sigma_2| = |\tau_0 :: \cdots :: \tau_i :: \sigma_3|$, and the latter equals $1 + i + |\sigma_3|$, so $0 \leqslant i < |\sigma_2|$. By $\hat{R}$ Typing, $R(r_s)$ is defined. Hence $P \longmapsto (H, R\{\mathtt{sp} \mapsto w_n :: \cdots :: w_{|\sigma_2|-i+1} :: R(r_s) :: w_{|\sigma_2|-i-1} :: \cdots :: nil\}, I)$.

case `st`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash r_d : \langle \tau_0, \ldots, \tau_n \rangle \quad \Psi;\cdot;\Gamma \vdash r_s : \tau_i \quad 0 \leqslant i \leqslant n}{\Psi;\cdot;\Gamma \vdash \mathtt{st}\ r_d(i), r_s \Rightarrow \cdot;\Gamma'}}{\Psi;\cdot;\Gamma \vdash \mathtt{st}\ r_d(i), r_s; I}}{\vdash P}$$

By Canonical Forms, $R(r_d) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_n \rangle$. By $\hat{R}$ Typing, $R(r_s)$ is defined. Hence $P \longmapsto (H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_n.\}, R, I)$.

case `unpack`**:** TD has the form:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \qquad \dfrac{\dfrac{\Psi;\cdot;\Gamma \vdash v : \exists\alpha.\tau}{\Psi;\cdot;\Gamma \vdash \mathtt{unpack}\ [\alpha, r], v \Rightarrow \alpha;\Gamma\{r{:}\tau\}} \quad \Psi;\alpha;\Gamma\{r{:}\tau\} \vdash I}{\Psi;\cdot;\Gamma \vdash \mathtt{unpack}\ [\alpha, r], v; I}}{\vdash P}$$

By Canonical Forms, $\hat{R}(v) = pack\ [\tau', w]\ as\ \tau''$. Hence $P \longmapsto (H, R\{r \mapsto w\}, I[\tau'/\alpha])$.

□

References

Appel, A. and Shao, Z. (1992) Callee-saves registers in continuation-passing style. *Lisp and Symbolic Computation*, **5**, 189–219.

Appel, A. W. (1992) *Compiling with Continuations.* Cambridge University Press.

Appel, A. W. and MacQueen, D. B. (1991) Standard ML of New Jersey. In: Wirsing, M. (editor), *Third International Symposium on Programming Language Implementation and Logic Programming: Lecture Notes in Computer Science 528*, pp. 1–13. Springer-Verlag.

Bailey, M. and Davidson, J. (1995) A formal model of procedure calling conventions. *22nd ACM Symposium on Principles of Programming Languages*, pp. 298–310.

Birkedal, L., Tofte, M. and Vejlstrup, M. (1996) From region inference to von Neumann machines via region representation inference. *23rd ACM Symposium on Principles of Programming Languages*, pp. 171–183.

Coglio, A., Goldberg, A. and Qian, Z. (1998) Towards a provably-correct implementation of the JVM bytecode verifier. *Proceedings OOPSLA'98 Workshop on the Formal Underpinnings of Java.*

Crary, K. (1999) A simple proof technique for certain parametricity results. *ACM SIGPLAN International Conference on Functional Programming*, pp. 82–89.

Felleisen, M. (1987) *The calculi of lambda-v-cs-conversion: A syntactic theory of control and state in imperative higher-order programming languages.* PhD thesis, Indiana University.

Freund, S. and Mitchell, J. (1998) A type system for object initialization in the Java bytecode language. *Proc. Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pp. 310–328. ACM Press.

Freund, S. and Mitchell, J. (1999) *Specification and verification of Java bytecode subroutines and exceptions.* Technical report, Computer Science Department, Stanford University.

Glew, N. (1999) Type dispatch for named hierarchical types. *ACM SIGPLAN International Conference on Functional Programming*, pp. 172–182.

Goldberg, A. (1998) A specification of Java loading and bytecode verification. *Proc. 5th ACM Conf. on Computer and Communications Security.*

Gosling, J., Joy, W. and Steele, G. (1996) *The Java Language Specification.* Second ed. Addison-Wesley.

Harper, R. (1994) A simplified account of polymorphic references. *Infor. Process. Lett.* **51**(4), 201–206. (Follow-up note in *Infor. Process. Lett.*, 57(1), 1996.)

Hieb, R., Dybvig, R. K. and Bruggeman, C. (1990) Representing control in the presence of first-class continuations. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 66–77. (Published as *SIGPLAN Notices*, **25**(6).)

Landin, P. J. (1964) The mechanical evaluation of expressions. *Computer J.* **6**, 308–320.

Lindholm, T. and Yellin, F. (1996) *The Java Virtual Machine Specification.* Addison-Wesley.

Liskov, B. H. and Snyder, A. (1979) Exception handling in Clu. *IEEE Trans. Softw. Eng.* **5**(6), 546–558.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML (revised).* MIT Press.

Minamide, Y., Morrisett, G. and Harper, R. (1996) Typed closure conversion. *23rd ACM Symposium on Principles of Programming Languages*, pp. 271–283.

Mitchell, J. C. and Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Progamming Languages and Systems*, **10**(3), 470–502.

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1998a) From System F to typed assembly language. *25th ACM Symposium on Principles of Programming Languages.* (Extended version published as Cornell University technical report TR97-1651, November 1997.)

Morrisett, G., Crary, K., Glew, N. and Walker, D. (1998b) Stack-based typed assembly language. *ACM Workshop on Types in Compilation*, pp. 95–118.

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1999a) From System F to typed assembly language. *ACM Trans. Programming Languages and Systems*, **21**(3), 528–569.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S. and Zdancewic, S. (1999b) TALx86: a realistic typed assembly language. *Proc. ACM SIGPLAN Workshop on Compiler Support for Systems Software*, pp. 25–35.

Necula, G. (1997) Proof-carrying code. *24th ACM Symposium on Principles of Programming Languages*, pp. 106–119.

Necula, G. and Lee, P. (1996) Safe kernel extensions without run-time checking. *Proc. Operating System Design and Implementation*, pp. 229–243.

Necula, G. C. and Lee, P. (1998) Design and implementation of a certifying compiler. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 333–344.

O'Callahan, R. (1999) A simple, comprehensive type system for Java bytecode subroutines. *26th ACM Symposium on Principles of Programming Languages.*

Qian, Z. (1998) A formal specification of Java(tm) virtual machine instructions for objects, methods, and subroutines. In: Alves-Foss, J. (editor), *Formal Syntax and Semantics of Java(tm)*. Springer-Verlag.

Reynolds, J. (1995) Using functor categories to generate intermediate code. *22nd ACM Symposium on Principles of Programming Languages*, pp. 25–36.

Scheme (1998) Revised^5 report on the algorithmic language Scheme. *J. Higher Order & Symbolic Computation*, **11**(1), 7–105. (Also appears as *ACM SIGPLAN Notices*, **33**(9), September 1998.)

Stata, R. and Abadi, M. (1999) A type system for Java bytecode subroutines. *ACM Trans. Progamming Languages and Systems*, **21**(1), 90–137.

Steele Jr., G. L. (1978) *Rabbit: A compiler for Scheme.* MPhil thesis, MIT.

Tofte, M. and Talpin, J.-P. (1994) Implementation of the typed call-by-value λ-calculus using a stack of regions. *21st ACM Symposium on Principles of Programming Languages*, pp. 188–201.