From System F to Typed Assembly Language *
(Extended Version)

Greg Morrisett David Walker Karl Crary Neal Glew

Cornell University

November 21, 1997

Abstract

We motivate the design of a statically *typed assembly language* (TAL) and present a type-preserving translation from System F to TAL. The TAL we present is based on a conventional RISC assembly language, but its static type system provides support for enforcing high-level language abstractions, such as closures, tuples, and objects, as well as user-defined abstract data types. The type system ensures that well-typed programs cannot violate these abstractions. In addition, the typing constructs place almost no restrictions on low-level optimizations such as register allocation, instruction selection, or instruction scheduling.

Our translation to TAL is specified as a sequence of type-preserving transformations, including CPS and closure conversion phases; type-correct source programs are mapped to type-correct assembly language. A key contribution is an approach to polymorphic closure conversion that is considerably simpler than previous work. The compiler and typed assembly language provide a fully automatic way to produce *proof carrying code*, suitable for use in systems where untrusted and potentially malicious code must be checked for safety before execution.

1 Introduction

Compiling a source language to a statically typed intermediate language has compelling advantages over a conventional untyped compiler. An optimizing compiler for a high-level language such as ML may make as many as 20 passes over a single program, performing sophisticated analyses and transformations such as CPS conversion [15, 35, 3, 13, 19], closure conversion [21, 40, 20, 4, 27], unboxing [23, 31, 39], subsumption elimination [10, 12], or region inference [8]. Many of these optimizations require type information in order to succeed, and even those that do not often benefit from the additional structure supplied by a typing discipline [23, 19, 31, 37]. Furthermore, the ability to typecheck intermediate code provides an invaluable tool for debugging new transformations and optimizations [41, 29].

Today a small number of compilers work with typed intermediate languages in order to realize some or all of these benefits [23, 34, 7, 41, 25, 38, 14]. However, in all of these compilers, there is a

*This material is based on work supported in part by AFOSR grant F49620-97-1-0013, ARPA/RADC grant FC30602-96-1-0317, NSF grant CCR-9244739, and ONR grant N00014-92-J-1764. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of these agencies.

conceptual line where types are lost. For instance, the TIL/ML compiler preserves type information through approximately 80% of compilation, but the remaining 20% is untyped.

We show how to eliminate the untyped portions of a compiler and by so doing, extend the approach of compiling with typed intermediate languages to typed *target* languages. The target language in this paper is a strongly typed assembly language (TAL) based on a generic RISC instruction set. The type system for the language is surprisingly standard, supporting tuples, polymorphism, existentials, and a very restricted form of function pointer, yet it is sufficiently powerful that we can automatically generate well-typed and efficient code from high-level ML-like languages. Furthermore, we claim that the type system does not seriously hinder low-level optimizations such as register allocation, instruction selection, instruction scheduling, and copy propagation.

TAL not only allows us to reap the benefits of types throughout a compiler, but it also enables a practical system for executing untrusted code both safely and efficiently. For example, as suggested by the SPIN project [6], operating systems could allow users to download TAL extensions into the kernel. The kernel could typecheck the TAL code to ensure that the code never accesses hidden resources within the kernel, always calls kernel routines with the right number and types of arguments, *etc.*, and then assemble and dynamically link the code into the kernel.[1] However, SPIN currently requires the user to write the extension in a single high-level language (Modula-3) and use a single trusted compiler (along with cryptographic signatures) in order to ensure the safety of the extension. In contrast, a kernel based on a typed assembly language could support extensions written in a variety of high-level languages using a variety of untrusted compilers, as the safety of the resulting assembly code can be checked independently of the source code or the compiler. Furthermore, critical inner-loops could be hand-written in assembly language in order to achieve optimal performance. TAL could also be used to support extensible web-browsers, extensible servers, active networks, or any other "kernel" where security, performance, and language independence are desired.

Software Fault Isolation (SFI) [47] also provides memory safety and language independence. However, SFI requires the insertion of extra "sandboxing" code, corresponding to dynamic type tests, to ensure that the extension is safe. In contrast, TAL does not have the overhead of the additional sandboxing code, as typechecking is performed offline.

With regard to these security properties, TAL is an instance of Necula and Lee's *proof carrying code* (PCC) [33, 32]. Necula suggests that the relevant operational content of simple type systems may be encoded using extensions to first-order predicate logic, and proofs of relevant security properties such as memory safety may be automatically verified [32]. In addition, Necula's approach places no restrictions on code sequences or instruction scheduling, whereas TAL has a small number of such restrictions (see Section 6.2). However, in general there is no complete algorithm for constructing the proof that the code satisfies the desired security properties. In contrast, we provide a fully automatic procedure for generating typed assembly language from a well-formed source term.

[1] Of course, while type safety implies many important security properties such as memory safety, there are a variety of other important security properties, such as termination, that do not follow from type safety.

1.1 Overview

In order to motivate the typing constructs in TAL and to justify our claims about its expressiveness, we spend much of this paper sketching a compiler from a variant of the polymorphic λ-calculus to TAL. The eager reader may wish to glance at Figure 26 for a sample TAL program.

Our compiler is structured as five translations between six typed calculi:

$$\lambda^{\mathrm{F}} \xrightarrow{\text{CPS-conversion}} \lambda^{\mathrm{K}} \xrightarrow{\text{Closure conversion}} \lambda^{\mathrm{C}} \xrightarrow{\text{Hoisting}} \lambda^{\mathrm{H}} \xrightarrow{\text{Allocation}} \lambda^{\mathrm{A}} \xrightarrow{\text{Code generation}} \mathrm{TAL}$$

Each of these calculi is used as a first-class programming calculus in the sense that each translation accepts any well-typed program of its input calculus; it does not assume that the input is the output from the preceding translation. This allows the compiler to aggressively optimize code between any of the translation steps. The inspiration for the phases and their ordering is derived from SML/NJ [5, 3] (which is in turn based on the Rabbit [40] and Orbit compilers [20]) except that types are used throughout compilation.

The rest of this paper proceeds by describing each of the languages and translations in our compiler in full detail. We give the syntax and static semantics of each language as well as type-directed and type-preserving translations between them. Section 2 presents λ^{F}, the compiler's source language. Section 3 presents the first intermediate language, λ^{K}, and gives a typed CPS translation to it based on Danvy and Filinski [13] and Harper and Lillibridge [19]. Section 4 presents the next intermediate language, λ^{C}, and gives a typed closure translation based on, but considerably simpler than, the presentation of Minamide, Morrisett, and Harper [27]. Section 5 presents the λ^{A} intermediate language and a translation that makes allocation and initialization of data structures explicit. At this point in compilation, the intermediate code is essentially in a λ-calculus syntax for assembly language, following the ideas of Wand [48]. Finally, Section 6 presents our typed assembly language and defines a translation from λ^{A} to TAL. In Section 7 we show the type correctness of the compiler and in Section 8 we discuss extensions to TAL to support language constructs not considered here.

2 System F

The source language for our compiler, λ^{F}, is a call-by-value variant of System F [16, 17, 36] (the polymorphic λ-calculus) augmented with products and recursion on terms. The syntax for λ^{F} appears below:

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \alpha \mid \mathit{int} \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau \mid \langle \tau_1, \ldots, \tau_n \rangle \\
\textit{terms} & e & ::= & x \mid i \mid \mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \\
 & & & \quad \langle e_1, \ldots, e_n \rangle \mid \pi_i(e) \mid e_1\, p\, e_2 \mid \mathit{if0}(e_1, e_2, e_3) \\
\textit{primitives} & p & ::= & + \mid - \mid \times \\
\textit{type contexts} & \Delta & ::= & \alpha_1, \ldots, \alpha_n \\
\textit{value contexts} & \Gamma & ::= & x_1{:}\tau_1, \ldots, x_n{:}\tau_n
\end{array}$$

In order to simplify the presentation, λ^{F} has only integers as a base type (ranged over by the metavariable i). We use $\vec{X}$ to denote a vector of syntactic objects drawn from X. For instance,

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash_{\mathrm{F}} \tau}$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{F}} x : \tau} \, (\Gamma(x) = \tau) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{F}} i : \mathit{int}}$$

$$\frac{\Delta \vdash_{\mathrm{F}} \tau_1 \quad \Delta \vdash_{\mathrm{F}} \tau_2 \quad \Delta; \Gamma\{x{:}\tau_1 \to \tau_2, x_1{:}\tau_1\} \vdash_{\mathrm{F}} e : \tau_2}{\Delta; \Gamma \vdash_{\mathrm{F}} \mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2} \, (x, x_1 \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash_{\mathrm{F}} e_2 : \tau_1}{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 e_2 : \tau_2}$$

$$\frac{\Delta\{\alpha\}; \Gamma \vdash_{\mathrm{F}} e : \tau}{\Delta; \Gamma \vdash_{\mathrm{F}} \Lambda\alpha.e : \forall\alpha.\tau} \, (\alpha \notin \Delta) \qquad \frac{\Delta \vdash_{\mathrm{F}} \tau \quad \Delta; \Gamma \vdash_{\mathrm{F}} e : \forall\alpha.\tau'}{\Delta; \Gamma \vdash_{\mathrm{F}} e[\tau] : \tau'[\tau/\alpha]}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{F}} e_i : \tau_i}{\Delta; \Gamma \vdash_{\mathrm{F}} \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{F}} e : \langle \tau_1, \ldots, \tau_n \rangle}{\Delta; \Gamma \vdash_{\mathrm{F}} \pi_i(e) : \tau_i} \, (1 \leq i \leq n)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{F}} e_2 : \mathit{int}}{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 \, p \, e_2 : \mathit{int}} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{F}} e_2 : \tau \quad \Delta; \Gamma \vdash_{\mathrm{F}} e_3 : \tau}{\Delta; \Gamma \vdash_{\mathrm{F}} \mathit{if0}(e_1, e_2, e_3) : \tau}$$

Figure 1: Static Semantics of λ^{F}

$\langle \vec{\tau} \rangle$ is shorthand for a product type $\langle \tau_1, \ldots, \tau_n \rangle$. The term $\mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e$ represents a recursively-defined function x with argument x_1 of type τ_1 and body e. Hence, both x and x_1 are bound within e. Similarly, α is bound in e for $\Lambda\alpha.e$ and bound in τ for $\forall\alpha.\tau$. As usual, we consider syntactic objects to be equivalent up to alpha-conversion of bound variables.

We interpret λ^{F} with a conventional call-by-value operational semantics (not presented here). The static semantics (given in Figure 1) is specified as a set of inference rules for concluding judgments of the form $\Delta; \Gamma \vdash_{\mathrm{F}} e : \tau$ where Δ is a context containing the free type variables of Γ, e, and τ; Γ is a context that assigns types to the free variables of e; and τ is the type of e. A judgment $\Delta \vdash_{\mathrm{F}} \tau$ asserts that type τ is well-formed under type context Δ.

As a running example, we will consider the compilation of the following computation of 6 factorial:

$$(\mathit{fix}\, f(n{:}\mathit{int}){:}\mathit{int}.\mathit{if0}(n, 1, n \times f(n-1)))\, 6.$$

3 CPS Conversion

The first compilation stage is conversion to continuation-passing style (CPS). This stage names all intermediate computations and eliminates the need for a control stack. All unconditional control transfers, including function invocation and return, are achieved via function call. The target calculus for this phase is λ^{K}:

$$
\begin{array}{llcl}
\textit{types} & \tau & ::= & \alpha \mid \mathit{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \rightarrow \mathit{void} \mid \langle\vec{\tau}\rangle \\
\textit{terms} & e & ::= & v[\vec{\tau}](\vec{v}) \mid \mathit{if0}(v, e_1, e_2) \mid \mathit{halt}[\tau]v \mid \mathit{let}\ x = v\ \mathit{in}\ e \mid \\
 & & & \quad \mathit{let}\ x = \pi_i(v)\ \mathit{in}\ e \mid \mathit{let}\ x = v_1\, p\, v_2\ \mathit{in}\ e \\
\textit{values} & v & ::= & x \mid i \mid \langle\vec{v}\rangle \mid \mathit{fix}\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\
\textit{primitives} & p & ::= & + \mid - \mid \times \\
\textit{type contexts} & \Delta & ::= & \alpha_1, \ldots, \alpha_n \\
\textit{value contexts} & \Gamma & ::= & x_1{:}\tau_1, \ldots, x_n{:}\tau_n
\end{array}
$$

Code in λ^{K} is nearly linear: it consists of a series of let bindings followed by a function call. The exception to this is the *if0* construct, which is still a tree containing two expressions.

In λ^{K} there is only one abstraction mechanism (*fix*), which abstracts both type and value variables, thereby simplifying the rest of the compiler. The corresponding $\forall$ and $\rightarrow$ types are also combined. However, we abbreviate $\forall[\,].(\vec{\tau}) \rightarrow \mathit{void}$ as $(\vec{\tau}) \rightarrow \mathit{void}$.

Unlike in λ^{F}, functions in λ^{K} do not return values; instead, they invoke continuations. The function notation "$\rightarrow \mathit{void}$" is intended to suggest this fact. Execution is completed by the construct $\mathit{halt}[\tau]v$, which accepts a result value v of type τ and terminates the computation. Typically, this construct is used by the top-level continuation. Since expressions never return values, typing judgments for expressions do not state types. Instead, the judgment $\Delta;\Gamma \vdash_{\mathrm{K}} e$ indicates that the expression e is well-formed under type and value contexts Δ and Γ. Aside from these issues, the static semantics for λ^{K} is completely standard and appears in Figure 2.

3.1 Translating λ^{F} to λ^{K}

The implementation of CPS-conversion follows Danvy and Filinski [13] and Harper and Lillibridge [19]. Danvy and Filinski give a CPS translation with a two-level type system that distinguishes between static (or "administrative") β-redices and dynamic ones. They use this in a one-pass translation that produces an efficient CPS value and prove the resulting value $\beta\eta$-equivalent to a standard CPS translation (as given by Fischer and Plotkin [15, 35]) They also show how to modify their translation so that it is "properly tail-recursive" (*i.e.,* so that the unnecessary η-expansions of tail-recursive functions are eliminated). Our translation uses both the two-level system and the tail-recursion optimizations. Harper and Lillibridge discuss the typing properties of CPS conversion of F_ω augmented with callcc and abort primitives. We used their call-by-value translation to guide our translation. Note that a real implementation would uncurry a type abstraction followed by a value abstraction during the translation, creating a single polymorphic function.

The type translation $\mathcal{K}[\![\,\cdot\,]\!]$ mapping λ^{F} types to λ^{K} types is given in Figure 3. For any λ^{F} type τ, $\mathcal{N}[\![\tau]\!]$ represents the λ^{K} type for a τ-continuation, which is $(\mathcal{K}[\![\tau]\!]) \rightarrow \mathit{void}$. The translation of terms (Figures 4 and 5) is given by three judgments. Full programs are translated by the judgment $\vdash_{\mathrm{F}} e_{\mathrm{F}} : \tau \overset{\mathrm{cps}}{\leadsto} e_{\mathrm{cps}}$, which asserts that e_{cps} is a correct CPS conversion of the program e_{F}. Most of the work is done by two other judgments: one for terms in tailcall positions and another for all other positions.

The non-tailcall translation is given by the judgment $\Delta;\Gamma \vdash_{\mathrm{F}} e_{\mathrm{F}} : \tau \overset{\mathrm{cps}^*}{\leadsto} e_{\mathrm{cps}}$. The result, e_{cps}, is a static function that takes a static continuation representing the rest of the program and invokes it

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash_{\mathrm{K}} \tau}$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{K}} x : \tau} \ (\Gamma(x) = \tau) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{K}} i : \mathit{int}}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v_i : \tau_i}{\Delta; \Gamma \vdash_{\mathrm{K}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\Delta\{\vec{\alpha}\} \vdash_{\mathrm{K}} \tau_i \quad \Delta\{\vec{\alpha}\}; \Gamma\{x{:}\forall[\vec{\alpha}](\vec{\tau}) \to \mathit{void}, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \vdash_{\mathrm{K}} e}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{fix}\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \forall[\vec{\alpha}](\vec{\tau}) \to \mathit{void}} \ (x, x_i \notin \Gamma, \alpha_i \notin \Delta)$$

$$\frac{\Delta \vdash_{\mathrm{K}} \sigma_i \quad \Delta; \Gamma \vdash_{\mathrm{K}} v : \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathit{void} \quad \Delta; \Gamma \vdash_{\mathrm{K}} v_j : \tau_j[\vec{\sigma}/\vec{\alpha}]}{\Delta; \Gamma \vdash_{\mathrm{K}} v[\sigma_1, \ldots, \sigma_n](v_1, \ldots, v_m)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{K}} e_1 \quad \Delta; \Gamma \vdash_{\mathrm{K}} e_2}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{if0}(v, e_1, e_2)} \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \tau}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{halt}[\tau]v}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \tau \quad \Delta; \Gamma\{x{:}\tau\} \vdash_{\mathrm{K}} e}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = v\ \mathit{in}\ e} \ (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \langle \tau_1, \ldots, \tau_n \rangle \quad \Delta; \Gamma\{x{:}\tau_i\} \vdash_{\mathrm{K}} e}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = \pi_i(v)\ \mathit{in}\ e} \ (x \notin \Gamma, 1 \leq i \leq n)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v_1 : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{K}} v_2 : \mathit{int} \quad \Delta; \Gamma\{x{:}\mathit{int}\} \vdash_{\mathrm{K}} e}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = v_1\ p\ v_2\ \mathit{in}\ e} \ (x \notin \Gamma)$$

Figure 2: Static Semantics of λ^{K}

$$\begin{aligned}
\mathcal{K}[\![\alpha]\!] &\stackrel{\text{def}}{=} \alpha \\
\mathcal{K}[\![\mathit{int}]\!] &\stackrel{\text{def}}{=} \mathit{int} \\
\mathcal{K}[\![\tau_1 \to \tau_2]\!] &\stackrel{\text{def}}{=} (\mathcal{K}[\![\tau_1]\!], \mathcal{N}[\![\tau_2]\!]) \to \mathit{void} \\
\mathcal{K}[\![\forall \alpha.\tau]\!] &\stackrel{\text{def}}{=} \forall[\alpha].(\mathcal{N}[\![\tau]\!]) \to \mathit{void} \\
\mathcal{K}[\![\langle \tau_1, \ldots, \tau_n \rangle]\!] &\stackrel{\text{def}}{=} \langle \mathcal{K}[\![\tau_1]\!], \ldots, \mathcal{K}[\![\tau_n]\!] \rangle \\
\\
\mathcal{N}[\![\tau]\!] &\stackrel{\text{def}}{=} (\mathcal{K}[\![\tau]\!]) \to \mathit{void}
\end{aligned}$$

Figure 3: Type Translation from λ^{F} to λ^{K}

$$\frac{\emptyset;\emptyset \vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}^*}{\leadsto} e'}{\vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}}{\leadsto} e'@(\lambda y.\mathit{halt}[\mathcal{K}[\![\tau]\!]]y)}$$

$$\frac{}{\Delta;\Gamma \vdash_{\mathrm{F}} x : \tau \overset{\mathrm{cps}^*}{\leadsto} \lambda k.k@x}\ (\Gamma(x) = \tau) \qquad \frac{}{\Delta;\Gamma \vdash_{\mathrm{F}} i : \mathit{int} \overset{\mathrm{cps}^*}{\leadsto} \lambda k.k@i}$$

$$\frac{\Delta \vdash_{\mathrm{F}} \tau_1 \quad \Delta \vdash_{\mathrm{F}} \tau_2 \quad \Delta;\Gamma\{x{:}\tau_1 \to \tau_2, x_1{:}\tau_1\} \vdash_{\mathrm{F}} e : \tau_2 \overset{\mathrm{cps}^{\mathrm{t}}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2 \overset{\mathrm{cps}^*}{\leadsto} \lambda k.k@(\mathit{fix}\, x[\,](x_1{:}\mathcal{K}[\![\tau_1]\!], c{:}\mathcal{N}[\![\tau_2]\!]).e'@c)}\ (x, x_1 \notin \Gamma)$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \tau_1 \to \tau_2 \overset{\mathrm{cps}^*}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \tau_1 \overset{\mathrm{cps}^*}{\leadsto} e_2'}{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 e_2 : \tau_2 \overset{\mathrm{cps}^*}{\leadsto} \lambda k.e_1'@(\lambda y_1.e_2'@(\lambda y_2.y_1[\,](y_2, \mathit{fix}\,\_[\,](z{:}\mathcal{K}[\![\tau_2]\!]).k@z)))}$$

$$\frac{\Delta\{\alpha\};\Gamma \vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}^{\mathrm{t}}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \Lambda\alpha.e : \forall\alpha.\tau \overset{\mathrm{cps}^*}{\leadsto} \lambda k.k@(\mathit{fix}\,\_[\alpha](c{:}\mathcal{N}[\![\tau]\!]).e'@c)}\ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash_{\mathrm{F}} \tau \quad \Delta;\Gamma \vdash_{\mathrm{F}} e : \forall\alpha.\tau' \overset{\mathrm{cps}^*}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} e[\tau] : \tau'[\tau/\alpha] \overset{\mathrm{cps}^*}{\leadsto} \lambda k.e'@(\lambda y.y[\mathcal{K}[\![\tau]\!]](\mathit{fix}\,\_[\,](z{:}\mathcal{K}[\![\tau'[\tau/\alpha]]\!]).k@z))}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_i : \tau_i \overset{\mathrm{cps}^*}{\leadsto} e_i'}{\begin{array}{l}\Delta;\Gamma \vdash_{\mathrm{F}} \langle e_1,\ldots,e_n\rangle : \langle\tau_1,\ldots,\tau_n\rangle \overset{\mathrm{cps}^*}{\leadsto} \\ \qquad \lambda k.e_1'@(\lambda y_1.e_2'@\cdots e_n'@(\lambda y_n.k@\langle y_1,\ldots,y_n\rangle)\cdots)\end{array}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e : \langle\tau_1,\ldots,\tau_n\rangle\rangle \overset{\mathrm{cps}^*}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \pi_i(e) : \tau_i \overset{\mathrm{cps}^*}{\leadsto} \lambda k.e'@(\lambda y.\mathit{let}\ z = \pi_i(y)\ \mathit{in}\ k@z)}\ (1 \le i \le n)$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \overset{\mathrm{cps}^*}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \mathit{int} \overset{\mathrm{cps}^*}{\leadsto} e_2'}{\Delta;\Gamma \vdash_{\mathrm{F}} e_1\, p\, e_2 : \mathit{int} \overset{\mathrm{cps}^*}{\leadsto} \lambda k.e_1'@(\lambda y_1.e_2'@(\lambda y_2.\mathit{let}\ z = y_1\, p\, y_2\ \mathit{in}\ k@z))}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \overset{\mathrm{cps}^*}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \tau \overset{\mathrm{cps}^{\mathrm{t}}}{\leadsto} e_2' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_3 : \tau \overset{\mathrm{cps}^{\mathrm{t}}}{\leadsto} e_3'}{\begin{array}{l}\Delta;\Gamma \vdash_{\mathrm{F}} \mathit{if0}(e_1, e_2, e_3) : \tau \overset{\mathrm{cps}^*}{\leadsto} \\ \qquad \lambda k.e_1'@(\lambda y.\mathit{let}\ c = \mathit{fix}\,\_[\,](z : \mathcal{K}[\![\tau]\!]).k@z\ \mathit{in} \\ \qquad\qquad \mathit{if0}(y, e_2'@c, e_3'@c))\end{array}}$$

Figure 4: Term Translation from λ^{F} to λ^{K} (except tailcalls)

$$\frac{}{\Delta;\Gamma \vdash_{\mathrm{F}} x : \tau \overset{\mathrm{cps^t}}{\leadsto} \lambda c.c[\,](x)}\;(\Gamma(x)=\tau) \qquad \frac{}{\Delta;\Gamma \vdash_{\mathrm{F}} i : \mathit{int} \overset{\mathrm{cps^t}}{\leadsto} \lambda c.c[\,](i)}$$

$$\frac{\Delta \vdash_{\mathrm{F}} \tau_1 \quad \Delta \vdash_{\mathrm{F}} \tau_2 \quad \Delta;\Gamma\{x{:}\tau_1 \to \tau_2, x_1{:}\tau_1\} \vdash_{\mathrm{F}} e : \tau_2 \overset{\mathrm{cps^t}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2 \overset{\mathrm{cps^t}}{\leadsto} \lambda c.c[\,](\mathit{fix}\, x[\,](x_1{:}\mathcal{K}[\![\tau_1]\!], c'{:}\mathcal{N}[\![\tau_2]\!]).e'@c')}\;(x, x_1 \notin \Gamma)$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \tau_1 \to \tau_2 \overset{\mathrm{cps^*}}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \tau_1 \overset{\mathrm{cps^*}}{\leadsto} e_2'}{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 e_2 : \tau_2 \overset{\mathrm{cps^t}}{\leadsto} \lambda c.e_1'@(\lambda y_1.e_2'@(\lambda y_2.y_1[\,](y_2, c)))}$$

$$\frac{\Delta\{\alpha\};\Gamma \vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps^t}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \Lambda\alpha.e : \forall\alpha.\tau \overset{\mathrm{cps^t}}{\leadsto} \lambda c.c[\,](\mathit{fix}\,\_[\alpha](c{:}\mathcal{N}[\![\tau]\!]).e'@c)}\;(\alpha \notin \Delta)$$

$$\frac{\Delta \vdash_{\mathrm{F}} \tau \quad \Delta;\Gamma \vdash_{\mathrm{F}} e : \forall\alpha.\tau' \overset{\mathrm{cps^*}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} e[\tau] : \tau'[\tau/\alpha] \overset{\mathrm{cps^t}}{\leadsto} \lambda c.e'@(\lambda y.y[\mathcal{K}[\![\tau]\!]](c))}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_i : \tau_i \overset{\mathrm{cps^*}}{\leadsto} e_i'}{\begin{array}{l}\Delta;\Gamma \vdash_{\mathrm{F}} \langle e_1,\ldots,e_n\rangle : \langle \tau_1,\ldots,\tau_n\rangle \overset{\mathrm{cps^t}}{\leadsto} \\ \qquad \lambda c.e_1'@(\lambda y_1.e_2'@\cdots e_n'@(\lambda y_n.c[\,](\langle y_1,\ldots,y_n\rangle))\cdots)\end{array}}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e : \langle \tau_1,\ldots,\tau_n\rangle\rangle \overset{\mathrm{cps^*}}{\leadsto} e'}{\Delta;\Gamma \vdash_{\mathrm{F}} \pi_i(e) : \tau_i \overset{\mathrm{cps^t}}{\leadsto} \lambda c.e'@(\lambda y.\mathit{let}\ z = \pi_i(y)\ \mathit{in}\ c[\,](z))}\;(1 \le i \le n)$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \overset{\mathrm{cps^*}}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \mathit{int} \overset{\mathrm{cps^*}}{\leadsto} e_2'}{\Delta;\Gamma \vdash_{\mathrm{F}} e_1\, p\, e_2 : \mathit{int} \overset{\mathrm{cps^t}}{\leadsto} \lambda c.e_1'@(\lambda y_1.e_2'@(\lambda y_2.\mathit{let}\ z = y_1\, p\, y_2\ \mathit{in}\ c[\,](z)))}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{F}} e_1 : \mathit{int} \overset{\mathrm{cps^*}}{\leadsto} e_1' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_2 : \tau \overset{\mathrm{cps^t}}{\leadsto} e_2' \quad \Delta;\Gamma \vdash_{\mathrm{F}} e_3 : \tau \overset{\mathrm{cps^t}}{\leadsto} e_3'}{\Delta;\Gamma \vdash_{\mathrm{F}} \mathit{if0}(e_1, e_2, e_3) : \tau \overset{\mathrm{cps^t}}{\leadsto} \lambda c.e_1'@(\lambda y.\mathit{if0}(y, e_2'@c, e_3'@c))}$$

Figure 5: Tailcall Term Translation from λ^{F} to λ^{K}

with the result of computing e_{F}. The tailcall translation is given by the judgment $\Delta; \Gamma \vdash_{\mathrm{F}} e_{\mathrm{F}} : \tau \overset{\mathrm{cps}^{\mathrm{t}}}{\rightsquigarrow} e_{\mathrm{cps}}$. Its result, e_{cps}, is similar except that it is a static function taking a *dynamic* continuation (that is, a continuation coded in λ^{K}).

Throughout the translation, static functions and applications are denoted by $\lambda x.e$ and $e_1 @ e_2$, and dynamic functions and applications are written in λ^{K} syntax. The static variables k, y and y_i and the dynamic variables c, c' and z, which are used internally by the translation, are always assumed fresh. An underscore is used in place of unreferenced variables.

Lemma 3.1 *$\emptyset; \emptyset \vdash_{\mathrm{F}} e : \tau$ if and only if there exists e' such that $\vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}}{\rightsquigarrow} e'$.*

Lemma 3.2 (CPS Conversion Type Correctness) *If $\vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}}{\rightsquigarrow} e'$ then $\emptyset; \emptyset \vdash_{\mathrm{K}} e'$.*

These lemmas and the others like it are proved by induction on the source derivations.

When applied to the factorial example, this translation yields the following λ^{K} term:

$$
\begin{array}{l}
(\mathit{fix}\ f\ [\,]\ (n{:}\mathit{int}, k{:}(\mathit{int}) \to \mathit{void}). \\
\quad \mathit{if0}(n, k[\,](1), \\
\quad\quad \mathit{let}\ x = n - 1\ \mathit{in} \\
\quad\quad f[\,](x, \mathit{fix}\ \_\ [\,]\ (y{:}\mathit{int}). \\
\quad\quad\quad\quad \mathit{let}\ z = n \times y \\
\quad\quad\quad\quad \mathit{in}\ k[\,](z)))) \\
[\,]\ (6, \mathit{fix}\ \_\ [\,]\ (n{:}\mathit{int}).\ \mathit{halt}[\mathit{int}]n)
\end{array}
$$

4 Simplified Polymorphic Closure Conversion

The second compilation stage is closure conversion, which separates program code from data. This is done in two steps. Most of the work is done in the first step, closure conversion proper, which rewrites all functions so that they are closed. In order to do this, any variables from the context that are used in the function must be taken as additional arguments. These additional arguments are collected in an environment, which is paired with the (now closed) code to make a closure. In the second step, hoisting, closed function code is lifted to the top of the program, achieving the desired separation between code and data. We begin with closure conversion proper; the hoisting step is considered in Section 4.1.

Our approach to typed closure conversion is based on that of Minamide *et al.* [27]: If two functions with the same type but different free variables (and therefore different environment types) were naively closure converted, the types of their closures would not be the same. To prevent this, closures are given *existential* types [28] where the type of the environment is held abstract.

However, we propose an approach to polymorphic closure conversion that is considerably simpler than that of Minamide *et al.* which requires both abstract kinds and translucent types. Both of these mechanisms arise because Minamide *et al.* desire a *type-passing* interpretation of polymorphism where types are constructed and passed to polymorphic functions at run-time. Under a

types	τ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].(\vec{\tau}) \rightarrow void \mid \langle\vec{\tau}\rangle \mid \exists\alpha.\tau$
terms	e	$::=$	$v(\vec{v}) \mid if0(v, e_1, e_2) \mid halt[\tau]v \mid let\ x = v\ in\ e \mid let\ x = \pi_i(v)\ in\ e \mid$ $let\ x = v_1\ p\ v_2\ in\ e \mid let\ [\alpha, x] = unpack\ v\ in\ e$
values	v	$::=$	$x \mid i \mid \langle\vec{v}\rangle \mid v[\tau] \mid pack\ [\tau, v]\ as\ \exists\alpha.\tau' \mid fixcode\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
primitives	p	$::=$	$+ \mid - \mid \times$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::=$	$x_1{:}\tau_1, \ldots, x_n{:}\tau_n$

Figure 6: Syntax of λ^{C}

type-passing interpretation, polymorphic instantiation cannot be treated via substitution, as this requires making a copy of the code at run-time. Instead, a closure is constructed that consists of closed code, a value environment mapping variables to values, and a *type environment* mapping type variables to types.

In our approach, we assume a *type-erasure* interpretation of polymorphism as in *The Definition of Standard ML* [26], and polymorphic instantiation is semantically handled via substitution (*i.e.,* making a copy of the code with the types substituted for the type variables). As types will ultimately be erased from terms for execution, the "copies" can (and will) be represented by the same term. This avoids the need for abstract kinds (since there are no type environments), as well as translucent types. A type-erasure interpretation is not without its costs: It precludes some advanced implementation techniques [31, 43, 1, 30] and has subtle interactions with side-effects. We address the latter concern by forcing polymorphic abstractions to be values [42, 49] (*i.e.,* they must be syntactically attached to value abstractions).

To support this interpretation, we consider the partial application of functions to type arguments to be values. For example, suppose v has the type $\forall[\vec{\alpha}, \vec{\beta}].(\vec{\tau}) \rightarrow void$ where the type variables $\vec{\alpha}$ are intended for the type environment and the type variables $\vec{\beta}$ are intended for the function's type arguments. If $\vec{\sigma}$ is a vector of types to be used for the type environment, then the partial instantiation $v[\vec{\sigma}]$ is still treated as a value and has type $\forall[\vec{\beta}].(\vec{\tau}[\vec{\sigma}/\vec{\alpha}]) \rightarrow void$. The syntax of λ^{C} is otherwise similar to λ^{K} and appears in Figure 6. The static semantics of λ^{C} appears in Figure 7. Notice in particular that the body e of a function expression $fixcode\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ must typecheck given only the type and value contexts $\vec{\alpha}$ and $\{x{:}(\tau_1, \ldots, \tau_n) \rightarrow void, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$, which result from the *fixcode* abstraction. In other words, code must be closed.

The closure conversion algorithm is formalized as a type-directed translation in Figures 8 and 9. The translation of λ^{K} types is denoted by $\mathcal{C}[\![\cdot]\!]$. Terms are translated by the judgment $\Delta; \Gamma \vdash_{\mathrm{K}} e_{\mathrm{cps}} \overset{\mathrm{clos}}{\leadsto} e_{\mathrm{clos}}$, which asserts that e_{clos} is a correct closure conversion of the term e_{cps}, and values are translated by the judgment $\Delta; \Gamma \vdash_{\mathrm{K}} v_{\mathrm{cps}} : \tau \overset{\mathrm{clos}}{\leadsto} v_{\mathrm{clos}}$, which asserts that v_{clos} is a correct closure conversion of the value v_{cps}. The variables z, x_{code} and x_{env}, which are used internally by the translation, are always assumed fresh.

The key to the translation is the treatment of function types:

$$\mathcal{C}[\![\forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \rightarrow void]\!] \overset{\mathrm{def}}{=} \exists\beta.\langle\forall[\vec{\alpha}].(\beta, \mathcal{C}[\![\tau_1]\!], \ldots, \mathcal{C}[\![\tau_n]\!]) \rightarrow void, \beta\rangle$$

The existentially-quantified variable β is the type of the value environment for the closure. The

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash_{\mathrm{C}} \tau}$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{C}} x : \tau} \; (\Gamma(x) = \tau) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{C}} i : \mathit{int}}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v_i : \tau_i}{\Delta; \Gamma \vdash_{\mathrm{C}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\Delta \vdash_{\mathrm{C}} \sigma \quad \Delta; \Gamma \vdash_{\mathrm{C}} v : \forall[\alpha, \vec{\beta}].(\vec{\tau}) \to \mathit{void}}{\Delta; \Gamma \vdash_{\mathrm{C}} v[\sigma] : \forall[\vec{\beta}].(\vec{\tau}[\sigma/\alpha]) \to \mathit{void}}$$

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

$$\frac{\vec{\alpha} \vdash_{\mathrm{C}} \tau_i \quad \vec{\alpha}; \{x{:}\forall[\vec{\alpha}].(\vec{\tau}) \to \mathit{void}, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \vdash_{\mathrm{C}} e}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{fixcode}\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \forall[\vec{\alpha}].(\vec{\tau}) \to \mathit{void}}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \forall[\,].(\tau_1, \ldots, \tau_n) \to \mathit{void} \quad \Delta; \Gamma \vdash_{\mathrm{C}} v_i : \tau_i}{\Delta; \Gamma \vdash_{\mathrm{C}} v(v_1, \ldots, v_n)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{C}} e_1 \quad \Delta; \Gamma \vdash_{\mathrm{C}} e_2}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{if0}\,(v, e_1, e_2)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \tau}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{halt}[\tau]v}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \tau \quad \Delta; \Gamma\{x{:}\tau\} \vdash_{\mathrm{C}} e}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{let}\ x = v\ \mathit{in}\ e} \; (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \langle \tau_1, \ldots, \tau_n \rangle \quad \Delta; \Gamma\{x{:}\tau_i\} \vdash_{\mathrm{C}} e}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{let}\ x = \pi_i(v)\ \mathit{in}\ e} \; (x \notin \Gamma, 1 \leq i \leq n)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v_1 : \mathit{int} \quad \Delta; \Gamma \vdash_{\mathrm{C}} v_2 : \mathit{int} \quad \Delta; \Gamma\{x{:}\mathit{int}\} \vdash_{\mathrm{C}} e}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathit{let}\ x = v_1\ p\ v_2\ \mathit{in}\ e} \; (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{C}} v : \exists\alpha.\tau \quad \Delta\alpha; \Gamma\{x{:}\tau\} \vdash_{\mathrm{C}} e}{\mathit{let}\ [\alpha, x] = \mathit{unpack}\ v\ \mathit{in}\ e} \; (x \notin \Gamma, \alpha \notin \Delta)$$

Figure 7: Static Semantics of λ^{C}

$$
\begin{array}{rcl}
\mathcal{C}[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
\mathcal{C}[\![\mathit{int}]\!] & \stackrel{\text{def}}{=} & \mathit{int} \\
\mathcal{C}[\![\forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n) \to \mathit{void}]\!] & \stackrel{\text{def}}{=} & \exists\beta.\langle\forall[\vec{\alpha}].(\beta,\mathcal{C}[\![\tau_1]\!],\ldots,\mathcal{C}[\![\tau_n]\!]) \to \mathit{void},\beta\rangle \\
\mathcal{C}[\![\langle\tau_1,\ldots,\tau_n\rangle]\!] & \stackrel{\text{def}}{=} & \langle\mathcal{C}[\![\tau_1]\!],\ldots,\mathcal{C}[\![\tau_n]\!]\rangle
\end{array}
$$

Figure 8: Type Translation from λ^{K} to λ^{C}

closure itself is a pair consisting of a piece of code that is instantiated with the type environment, and the value environment. The instantiated code takes as arguments the type arguments and value arguments of the original abstraction, as well as the value environment of the closure.

Lemma 4.1 *$\emptyset;\emptyset \vdash_{\mathrm{K}} e$ if and only if there exists e' such that $\emptyset;\emptyset \vdash_{\mathrm{K}} e \stackrel{\text{clos}}{\leadsto} e'$*

Lemma 4.2 (Closure Conversion Type Correctness) *If $\emptyset;\emptyset \vdash_{\mathrm{K}} e \stackrel{\text{clos}}{\leadsto} e'$ then $\emptyset;\emptyset \vdash_{\mathrm{C}} e'$.*

4.1 Hoisting

After closure conversion, all functions are closed and may be hoisted out to the top-level without difficulty. In a real compiler, these two phases would be combined but we have separated them here for simplicity. After hoisting, programs belong to a calculus, λ^{H}, that is similar to λ^{C} except that *fixcode* is no longer a value. Instead, code blocks are defined at the top-level by a *letrec* prefix, which is called a *heap* in anticipation of λ^{A} and TAL. A new value form, *labels* (ℓ), is used to refer to those code blocks. The syntax of λ^{H} appears in Figure 10.

These changes require a small change to the typing system. In addition to typing values, we also must assign heap types (ranged over by Ψ) to heaps. These heap types are added to value and term typing judgments as an additional context and are passed unmodified through each typing rule. There are also two new judgments: the judgment $\vdash_{\mathrm{H}} P$ indicates that the program P is well-formed, and the judgment $\Psi \vdash_{\mathrm{H}} b : \ell \mapsto \tau$ indicates that the block b gives the label ℓ type τ assuming the heap has type Ψ. The static semantics of λ^{H} appear in Figure 11; rules that appear identically (except for the addition of heap types) in the static semantics of λ^{C} are omitted for brevity.

The translation of full programs from λ^{C} to λ^{H} is given by the judgment $\vdash_{\mathrm{C}} e_{\mathbf{clos}} \stackrel{\text{hst}}{\leadsto} e_{\mathbf{hst}}$, which states that $e_{\mathbf{hst}}$ is the hoisted version of the program $e_{\mathbf{clos}}$. No translation is required for types. The translation rules are exactly as expected and are omitted in the interest of brevity.

Lemma 4.3 *$\emptyset;\emptyset \vdash_{\mathrm{C}} e$ if and only if there exists P such that $\vdash_{\mathrm{C}} e \stackrel{\text{hst}}{\leadsto} P$.*

Lemma 4.4 (Hoisting Type Correctness) *If $\vdash_{\mathrm{C}} e \stackrel{\text{hst}}{\leadsto} P$ then $\vdash_{\mathrm{H}} P$.*

In Figure 12 appears the factorial example after the closure conversion and hoisting translations are applied and some simple optimizations are performed (beta reduction and copy propagation).

$$\frac{\Delta \vdash_{\mathrm{K}} \tau_i \qquad \Gamma = \{y_1{:}\tau'_1, \ldots, y_m{:}\tau'_m\} \qquad \Delta = \vec{\beta} \qquad \Delta\vec{\alpha}; \Gamma[x{:}\tau_{\text{code}}, x_1{:}\tau_1, \ldots, x_n{:}\tau_n] \vdash_{\mathrm{K}} e \overset{\text{clos}}{\rightsquigarrow} e'}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{fix}\ x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \tau_{\text{code}} \overset{\text{clos}}{\rightsquigarrow} \mathit{pack}\ [\tau_{\text{env}}, \langle v_{\text{code}}[\vec{\beta}], v_{\text{env}}\rangle]\ \mathit{as}\ \mathcal{C}[\![\tau_{\text{code}}]\!]}\ (x, x_i \notin \Gamma, \alpha_i \notin \Delta)$$

$$\begin{array}{llcl}
\text{where} & \tau_{\text{code}} & = & \forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \to \mathit{void} \\
& \tau_{\text{env}} & = & \langle \mathcal{C}[\![\tau'_1]\!], \cdots, \mathcal{C}[\![\tau'_m]\!] \rangle \\
& v_{\text{env}} & = & \langle y_1, \ldots, y_m \rangle \\
& v_{\text{code}} & = & \mathit{fixcode}\ x_{\text{code}}[\vec{\beta}, \vec{\alpha}](x_{\text{env}}{:}\tau_{\text{env}}, x_1{:}\mathcal{C}[\![\tau_1]\!], \ldots, x_n{:}\mathcal{C}[\![\tau_n]\!]). \\
& & & \quad \mathit{let}\ x = \mathit{pack}\ [\tau_{\text{env}}, \langle x_{\text{code}}[\vec{\beta}], x_{\text{env}}\rangle]\ \mathit{as}\ \mathcal{C}[\![\tau_{\text{code}}]\!]\ \mathit{in} \\
& & & \quad \mathit{let}\ y_1 = \pi_1(x_{\text{env}})\ \mathit{in} \\
& & & \quad \vdots \\
& & & \quad \mathit{let}\ y_m = \pi_m(x_{\text{env}})\ \mathit{in}\ e'
\end{array}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \forall[\alpha_1, \ldots, \alpha_m].(\tau_1, \ldots, \tau_n) \to \mathit{void} \overset{\text{clos}}{\rightsquigarrow} v' \quad \Delta \vdash_{\mathrm{K}} \sigma_i \quad \Delta; \Gamma \vdash_{\mathrm{K}} v_i : \tau_i[\vec{\sigma}/\vec{\alpha}] \overset{\text{clos}}{\rightsquigarrow} v'_i}{\Delta; \Gamma \vdash_{\mathrm{K}} v[\sigma_1, \ldots, \sigma_m](v_1, \ldots, v_n) \overset{\text{clos}}{\rightsquigarrow} e}$$

$$\begin{array}{ll}
\text{where } e = & \mathit{let}\ [\alpha_{\text{env}}, z] = \mathit{unpack}\ v'\ \mathit{in} \\
& \mathit{let}\ x_{\text{code}} = \pi_1(z)\ \mathit{in} \\
& \mathit{let}\ x_{\text{env}} = \pi_2(z)\ \mathit{in} \\
& x_{\text{code}}[\mathcal{C}[\![\sigma_1]\!]] \cdots [\mathcal{C}[\![\sigma_m]\!]](x_{\text{env}}, v'_1, \ldots, v'_n)
\end{array}$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{K}} x : \tau \overset{\text{clos}}{\rightsquigarrow} x}\ (\Gamma(x) = \tau) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{K}} i : \mathit{int} \overset{\text{clos}}{\rightsquigarrow} i}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v_i : \tau_i \overset{\text{clos}}{\rightsquigarrow} v'_i}{\Delta; \Gamma \vdash_{\mathrm{K}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle \overset{\text{clos}}{\rightsquigarrow} \langle v'_1, \ldots, v'_n \rangle}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \mathit{int} \overset{\text{clos}}{\rightsquigarrow} v' \quad \Delta; \Gamma \vdash_{\mathrm{K}} e_1 \overset{\text{clos}}{\rightsquigarrow} e'_1 \quad \Delta; \Gamma \vdash_{\mathrm{K}} e_2 \overset{\text{clos}}{\rightsquigarrow} e'_2}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{if0}(v, e_1, e_2) \overset{\text{clos}}{\rightsquigarrow} \mathit{if0}(v', e'_1, e'_2)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \tau \overset{\text{clos}}{\rightsquigarrow} v'}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{halt}[\tau]v \overset{\text{clos}}{\rightsquigarrow} \mathit{halt}[\mathcal{C}[\![\tau]\!]]v'}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \tau \overset{\text{clos}}{\rightsquigarrow} v' \quad \Delta; \Gamma\{x{:}\tau\} \vdash_{\mathrm{K}} e \overset{\text{clos}}{\rightsquigarrow} e'}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = v\ \mathit{in}\ e \overset{\text{clos}}{\rightsquigarrow} \mathit{let}\ x = v'\ \mathit{in}\ e'}\ (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v : \langle \tau_1, \ldots, \tau_n \rangle \overset{\text{clos}}{\rightsquigarrow} v' \quad \Delta; \Gamma\{x{:}\tau_i\} \vdash_{\mathrm{K}} e \overset{\text{clos}}{\rightsquigarrow} e'}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = \pi_i(v)\ \mathit{in}\ e \overset{\text{clos}}{\rightsquigarrow} \mathit{let}\ x = \pi_i(v')\ \mathit{in}\ e'}\ (x \notin \Gamma, 1 \le i \le n)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{K}} v_1 : \mathit{int} \overset{\text{clos}}{\rightsquigarrow} v'_1 \quad \Delta; \Gamma \vdash_{\mathrm{K}} v_2 : \mathit{int} \overset{\text{clos}}{\rightsquigarrow} v'_2 \quad \Delta; \Gamma\{x{:}\mathit{int}\} \vdash_{\mathrm{K}} e \overset{\text{clos}}{\rightsquigarrow} e'}{\Delta; \Gamma \vdash_{\mathrm{K}} \mathit{let}\ x = v_1\ p\ v_2\ \mathit{in}\ e \overset{\text{clos}}{\rightsquigarrow} \mathit{let}\ x = v'_1\ p\ v'_2\ \mathit{in}\ e'}\ (x \notin \Gamma)$$

Figure 9: Term Translation from λ^{K} to λ^{C}

types	τ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].(\vec{\tau}) \to void \mid \langle\vec{\tau}\rangle \mid \exists\alpha.\tau$
terms	e	$::=$	$v(\vec{v}) \mid if0(v, e_1, e_2) \mid halt[\tau]v \mid let\ x = v\ in\ e \mid let\ x = \pi_i(v)\ in\ e \mid$ $let\ x = v_1\ p\ v_2\ in\ e \mid let\ [\alpha, x] = unpack\ v\ in\ e$
values	v	$::=$	$x \mid \ell \mid i \mid \langle\vec{v}\rangle \mid v[\tau] \mid pack\ [\tau, v]\ as\ \exists\alpha.\tau'$
blocks	b	$::=$	$\ell \mapsto \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
programs	P	$::=$	$letrec\ \vec{b}\ in\ e$
primitives	p	$::=$	$+ \mid - \mid \times$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::=$	$x_1{:}\tau_1, \ldots, x_n{:}\tau_n$
heap types	Ψ	$::=$	$\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n$

Figure 10: Syntax of λ^{H}

$$\frac{}{\Psi; \Delta; \Gamma \vdash_{\mathrm{H}} \ell : \tau}\ (\Psi(\ell) = \tau)$$

$$\frac{\vec{\alpha} \vdash_{\mathrm{H}} \tau_i \quad \Psi; \vec{\alpha}; \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \vdash_{\mathrm{H}} e}{\Psi \vdash_{\mathrm{H}} \ell \mapsto \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \ell \mapsto \forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \to void}$$

$$\frac{\emptyset \vdash_{\mathrm{H}} \tau_i \quad \Psi \vdash_{\mathrm{H}} b_i : \ell_i \mapsto \tau_i \quad \Psi; \emptyset; \emptyset \vdash_{\mathrm{H}} e}{\vdash_{\mathrm{H}} letrec\ b_1, \ldots, b_n\ in\ e} \begin{pmatrix} \Psi = \ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n \\ \ell_j \neq \ell_k \text{ when } j \neq k \end{pmatrix}$$

Figure 11: New Rules in the Static Semantics of λ^{H}

$$
\begin{array}{l}
\mathit{letrec}\ \ \ell_{fact}\ \mapsto\ \texttt{(*}\ \text{main factorial code block}\ \texttt{*)}\\
\qquad\qquad\qquad \mathtt{code}[\,](\mathit{env}{:}\langle\rangle, n{:}\mathit{int}, k{:}\tau_k).\\
\qquad\qquad\qquad\quad \mathit{if0}(n,\ \texttt{(*}\ \text{true branch: continue with 1}\ \texttt{*)}\\
\qquad\qquad\qquad\qquad\quad \mathit{let}\ [\beta, k_{unpack}] = \mathit{unpack}\ k\ \mathit{in}\\
\qquad\qquad\qquad\qquad\quad \mathit{let}\ k_{code} = \pi_0(k_{unpack})\ \mathit{in}\\
\qquad\qquad\qquad\qquad\quad \mathit{let}\ k_{env} = \pi_1(k_{unpack})\\
\qquad\qquad\qquad\qquad\quad \mathit{in}\\
\qquad\qquad\qquad\qquad\qquad k_{code}(k_{env}, 1),\\
\qquad\qquad\qquad\qquad\quad \texttt{(*}\ \text{false branch: recurse with}\ n-1\ \texttt{*)}\\
\qquad\qquad\qquad\qquad\quad \mathit{let}\ x = n - 1\ \mathit{in}\\
\qquad\qquad\qquad\qquad\quad \texttt{(*}\ \text{compute factorial of}\ n-1\ \text{and continue to}\ k'\ \texttt{*)}\\
\qquad\qquad\qquad\qquad\qquad \ell_{fact}(\mathit{env}, x, \mathit{pack}\ [\langle \mathit{int}, \tau_k\rangle,\ \langle \ell_{cont}, \langle n, k\rangle\rangle]\ \mathit{as}\ \tau_k))\\
\qquad\quad \ell_{cont}\ \mapsto\ \texttt{(*}\ \text{code block for continuation after factorial computation}\ \texttt{*)}\\
\qquad\qquad\qquad \mathtt{code}[\,](\mathit{env}{:}\langle \mathit{int}, \tau_k\rangle, y{:}\mathit{int}).\\
\qquad\qquad\qquad\quad \texttt{(*}\ \text{open the environment}\ \texttt{*)}\\
\qquad\qquad\qquad\quad \mathit{let}\ n = \pi_0(\mathit{env})\ \mathit{in}\\
\qquad\qquad\qquad\quad \mathit{let}\ k = \pi_1(\mathit{env})\ \mathit{in}\\
\qquad\qquad\qquad\quad \texttt{(*}\ \text{continue with}\ n \times y\ \texttt{*)}\\
\qquad\qquad\qquad\quad \mathit{let}\ z = n \times y\ \mathit{in}\\
\qquad\qquad\qquad\quad \mathit{let}\ [\beta, k_{unpack}] = \mathit{unpack}\ k\ \mathit{in}\\
\qquad\qquad\qquad\quad \mathit{let}\ k_{code} = \pi_0(k_{unpack})\ \mathit{in}\\
\qquad\qquad\qquad\quad \mathit{let}\ k_{env} = \pi_1(k_{unpack})\\
\qquad\qquad\qquad\quad \mathit{in}\\
\qquad\qquad\qquad\qquad k_{code}(k_{env}, z)\\
\qquad\quad \ell_{halt}\ \mapsto\ \texttt{(*}\ \text{code block for top-level continuation}\ \texttt{*)}\\
\qquad\qquad\qquad \mathtt{code}[\,](\mathit{env}{:}\langle\rangle, n{:}\mathit{int}).\ \mathit{halt}[\mathit{int}](n)\\
\mathit{in}\\
\qquad\quad \ell_{fact}(\langle\rangle, 6, \mathit{pack}\ [\langle\rangle, \langle \ell_{halt}, \langle\rangle\rangle]\ \mathit{as}\ \tau_k)
\end{array}
$$

where τ_k is $\exists\alpha.\langle(\alpha, \mathit{int}) \rightarrow \mathit{void}, \alpha\rangle$

Figure 12: Factorial in λ^{H}

types	τ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].(\vec{\tau}) \to void \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
terms	e	$::=$	$let\ \vec{d}\ in\ v(\vec{v}) \mid let\ \vec{d}\ in\ if0(v, e_1, e_2) \mid let\ \vec{d}\ in\ halt[\tau]v$
declarations	d	$::=$	$x = v \mid x = \pi_i(v) \mid x = v_1\ p\ v_2 \mid [\alpha, x] = unpack\ v \mid$ $x = malloc[\vec{\tau}] \mid x = v[i] \leftarrow v'$
values	v	$::=$	$x \mid \ell \mid i \mid v[\tau] \mid pack\ [\tau, v]\ as\ \exists \alpha.\tau'$
blocks	b	$::=$	$\ell \mapsto \mathtt{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
programs	P	$::=$	$letrec\ \vec{b}\ in\ e$
primitives	p	$::=$	$+ \mid - \mid \times$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::=$	$x_1{:}\tau_1, \ldots, x_n{:}\tau_n$
heap types	Ψ	$::=$	$\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n$

Figure 13: Syntax of λ^{A}

5 Explicit Allocation

The λ^{H} intermediate language still has an atomic constructor for forming tuples, but machines must allocate space for a tuple and fill it out field by field; the allocation stage makes this process explicit. The syntax of λ^{A}, the target calculus of this stage, is similar to that of λ^{H}, and appears in Figure 13. Note that there is no longer a value form for tuples. The creation of an n-element tuple becomes a computation that is separated into an allocation step and n initialization steps. For example, if v_0 and v_1 are integers, the pair $\langle v_0, v_1 \rangle$ is created as follows (where types have been added for clarity):

$$\begin{array}{ll} let & x_0{:}\langle int^0, int^0 \rangle = malloc[int, int] \\ & x_1{:}\langle int^1, int^0 \rangle = x_0[0] \leftarrow v_0 \\ & x\ :\langle int^1, int^1 \rangle = x_1[1] \leftarrow v_1 \\ & \vdots \end{array}$$

The "$x_0 = malloc[int, int]$" step allocates an uninitialized tuple and binds the address (*i.e.*, label) of the tuple to x_0. The "0" superscripts on the types of the fields indicate that the fields are uninitialized, and hence no projection may be performed on those fields. The "$x_1 = x_0[0] \leftarrow v_0$" step updates the first field of the tuple with the value v_0 and binds the address of the tuple to x_1. Note that x_1 is assigned a type where the first field has a "1" superscript, indicating that this field is initialized. Finally, the "$x = x_1[1] \leftarrow v_1$" step initializes the second field of the tuple with v_1 and binds the address of the tuple to x, which is assigned the fully initialized type $\langle int^1, int^1 \rangle$. Hence, both π_0 and π_1 are allowed on x.

Like all the intermediate languages of the compiler, this code sequence need not be atomic; it may be rearranged or optimized in any well-typed manner. The initialization flags on the types ensure that a field cannot be projected unless it has been initialized. Furthermore, the syntactic value restriction ensures there is no unsoundness in the presence of polymorphism. However, it is important to note that $x[i] \leftarrow v$ is interpreted as an imperative operation, and thus at the end of the sequence, x_0, x_1, and x are all aliases for the same location, even though they have different (but compatible) types. Consequently, the initialization flags do not prevent a field from being initialized twice. It is possible to use monads [44, 22] or linear types [18, 45, 46] to ensure that a tuple is initialized exactly once, but we have avoided these approaches in the interest of a simpler type system.

The static semantics for λ^{A} are given in Figure 14. Figure 15 presents the type translation from λ^{H} to λ^{A}. All that happens is that initialization flags are added to each field of tuple types:

$$\mathcal{A}[\![\langle \tau_1, \ldots, \tau_n \rangle]\!] \stackrel{\text{def}}{=} \langle \mathcal{A}[\![\tau_1]\!]^1, \ldots, \mathcal{A}[\![\tau_n]\!]^1 \rangle$$

The term translation appears in Figure 16. In this translation, values are translated to both a value and a series of declarations that are used to construct that value. For non-tuple values, that series of declarations will be null (denoted by ϵ). The variables y_i, which are used internally by the translation, are always assumed fresh. When e is *let* $\vec{d}$ *in* E, we write *let* $\vec{d'}$ *in* e to mean *let* $\vec{d}, \vec{d'}$ *in* E.

Lemma 5.1 $\vdash_{\mathrm{H}} P$ *if and only if there exists* P' *such that* $\vdash_{\mathrm{H}} P \stackrel{\text{alloc}}{\leadsto} P'$.

Lemma 5.2 (Allocation Type Correctness) *If* $\vdash_{\mathrm{H}} P \stackrel{\text{alloc}}{\leadsto} P'$ *then* $\vdash_{\mathrm{A}} P'$.

The factorial example after application of the explicit allocation translation appears in Figure 17.

6 Typed Assembly Language

The final compilation stage, code generation, converts λ^{A} to TAL. All of the major typing constructs in TAL are present in λ^{A} and, indeed, code generation is largely syntactic. To summarize the type structure at this point, there is a combined abstraction mechanism that may simultaneously abstract a type environment, a set of type arguments, and a set of value arguments. Values of these types may be partially applied to type environments and remain values. There are existential types to support closures and other data abstractions. Finally, there are n-tuples with flags on the fields indicating whether the field has been initialized.

A key technical distinction between λ^{A} and TAL is that λ^{A} uses alpha-varying variables, whereas TAL uses register names, which like labels on records, do *not* alpha-vary.[2] Following standard practice, we assume an infinite supply of registers. Mapping to a language with a finite number of registers may be performed by spilling registers into a tuple, and reloading values from this tuple when necessary.

One of the consequences of this aspect of TAL is that a register calling convention must be used in code generation, and that calling convention must be made explicit in the types. Hence TAL includes the type $\forall[\vec{\alpha}]\{\mathtt{r1}{:}\tau_1, \ldots, \mathtt{r}n{:}\tau_n\}$, which is used to describe entry points of code blocks (*i.e.,* code labels) and is the TAL analog of the λ^{A} function type, $\forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \rightarrow \mathit{void}$. The key difference is that we assign fixed registers to the arguments of the code. Intuitively, to jump to a block of code of this type, the type variables $\vec{\alpha}$ must be suitably instantiated, and registers `r1` through $\mathtt{r}n$ must contain values of type τ_1 through τ_n, respectively.

[2] Indeed, the register file may be viewed as a record, and register names as field labels for this record.

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash_{\mathrm{A}} \tau}$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} x : \tau}\ (\Gamma(x) = \tau) \qquad \frac{}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \ell : \tau}\ (\Psi(\ell) = \tau) \qquad \frac{}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} i : \mathit{int}}$$

$$\frac{\Delta \vdash_{\mathrm{A}} \sigma \quad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \forall[\alpha, \vec{\beta}].(\vec{\tau}) \to \mathit{void}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v[\sigma] : \forall[\vec{\beta}].(\vec{\tau}[\sigma/\alpha]) \to \mathit{void}}$$

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

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \forall[\,].(\tau_1, \ldots, \tau_n) \to \mathit{void} \quad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v_i : \tau_i}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ \ \mathit{in}\ v(v_1, \ldots, v_n)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \mathit{int} \quad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} e_1 \quad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} e_2}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ \ \mathit{in}\ \mathit{if0}(v, e_1, e_2)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \tau}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ \ \mathit{in}\ \mathit{halt}[\tau]v}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \tau \quad \Psi; \Delta; \Gamma\{x{:}\tau\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ x = v, \vec{d}\ \mathit{in}\ e}\ (x \notin \Gamma)$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \quad \Psi; \Delta; \Gamma\{x{:}\tau_i\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ x = \pi_i(v), \vec{d}\ \mathit{in}\ e}\ (\varphi_i = 1, x \notin \Gamma, 1 \leq i \leq n)$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v_1 : \mathit{int} \quad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v_2 : \mathit{int} \quad \Psi; \Delta; \Gamma\{x{:}\mathit{int}\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ x = v_1\ p\ v_2, \vec{d}\ \mathit{in}\ e}\ (x \notin \Gamma)$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \exists\alpha.\tau \quad \Psi; \Delta\alpha; \Gamma\{x{:}\tau\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ [\alpha, x] = \mathit{unpack}\ v, \vec{d}\ \mathit{in}\ e}\ (x \notin \Gamma, \alpha \notin \Delta)$$

$$\frac{\Delta \vdash_{\mathrm{A}} \tau_i \quad \Psi; \Delta; \Gamma\{x{:}\langle \tau_1^0, \ldots, \tau_n^0 \rangle\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ x = \mathit{malloc}[\tau_1, \ldots, \tau_n], \vec{d}\ \mathit{in}\ e}\ (x \notin \Gamma)$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{A}} v' : \tau_i \\ \Psi; \Delta; \Gamma\{x{:}\langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle\} \vdash_{\mathrm{A}} \mathit{let}\ \vec{d}\ \mathit{in}\ e\end{array}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{A}} \mathit{let}\ x = v[i] \leftarrow v', \vec{d}\ \mathit{in}\ e}\ (x \notin \Gamma, 1 \leq i \leq n)$$

$$\frac{\vec{\alpha} \vdash_{\mathrm{A}} \tau_i \quad \Psi; \vec{\alpha}; \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\} \vdash_{\mathrm{A}} e}{\Psi \vdash_{\mathrm{A}} \ell \mapsto \mathtt{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \ell \mapsto \forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \to \mathit{void}}$$

$$\frac{\emptyset \vdash_{\mathrm{A}} \tau_i \quad \Psi \vdash_{\mathrm{A}} b_i : \ell_i \mapsto \tau_i \quad \Psi; \emptyset; \emptyset \vdash_{\mathrm{A}} e}{\vdash_{\mathrm{A}} \mathit{letrec}\ b_1, \ldots, b_n\ \mathit{in}\ e} \left(\begin{array}{l} \Psi = \ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n \\ \ell_j \neq \ell_k \text{ when } j \neq k \end{array} \right)$$

Figure 14: Static Semantics of λ^{A}

$$
\begin{array}{rcl}
\mathcal{A}[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
\mathcal{A}[\![\mathit{int}]\!] & \stackrel{\text{def}}{=} & \mathit{int} \\
\mathcal{A}[\![\forall[\vec{\alpha}].(\vec{\tau}) \rightarrow \mathit{void}]\!] & \stackrel{\text{def}}{=} & \forall[\vec{\alpha}].(\mathcal{A}[\![\vec{\tau}]\!]) \rightarrow \mathit{void} \\
\mathcal{A}[\![\langle \tau_1, \ldots, \tau_n \rangle]\!] & \stackrel{\text{def}}{=} & \langle \mathcal{A}[\![\tau_1]\!]^1, \ldots, \mathcal{A}[\![\tau_n]\!]^1 \rangle \\
\mathcal{A}[\![\exists \alpha.\tau]\!] & \stackrel{\text{def}}{=} & \exists \alpha.\mathcal{A}[\![\tau]\!]
\end{array}
$$

Figure 15: Type Translation from λ^{H} to λ^{A}

Another technical point is that registers may contain only *word* values, which are integers, pointers into the heap (*i.e.,* labels), and instantiated or packed word values. Tuples and code blocks are *large* values and must be heap allocated. In this manner, TAL makes the layout of data in memory explicit.

In the remainder of this section, we present the syntax of TAL (Section 6.1), its dynamic semantics (Section 6.2), and its full static semantics (Section 6.3). Finally, we sketch the translation from λ^{A} to TAL (Section 6.4).

6.1 TAL Syntax

We present the full syntax of TAL in Figure 18. A TAL abstract machine or *program* consists of a heap, a register file and a sequence of instructions. The heap is a mapping of labels to heap values, which are tuples and code. The register file is a mapping of registers (ranged over by the metavariable r) to word values. Heaps, register files, and their respective types are not considered syntactically correct if they repeat labels or registers. When r appears in R, the notation $R\{r \mapsto w\}$ represents the register file R with the r binding replaced with w, and a similar notation is used for register file types; if r does not appear in R, the indicated binding is merely added, as usual.

Although heap values are not word values, the labels that point to them are. The other word values are integers, instantiations of word values, existential packages, and junk values ($?\tau$), which are used by the operational semantics to represent uninitialized data. A small value is either a word value, a register, or an instantiated or packed small value. The distinction between word and small values is drawn because a register must contain a word, not another register. Code blocks are linear sequences of instructions that abstract a set of type variables, and state their register assumptions. The sequence of instructions is always terminated by a `jmp` or `halt` instruction. Expressions that differ only by alpha-variation (of type variables) are considered identical, as are programs that differ only in the order of fields in a heap or register file.

6.2 TAL Operational Semantics

The operational semantics of TAL is presented in Figure 19 as a deterministic rewriting system $P \longmapsto P'$ that maps programs to programs. Although, as discussed above, we ultimately intend a type-erasure interpretation, we do not erase the types from the operational semantics presented here, so that we may more easily state and prove a subject reduction theorem (Lemma 6.1). The

$$\frac{}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} x:\tau \stackrel{\mathrm{alloc}}{\rightsquigarrow} x,\epsilon}\ (\Gamma(x)=\tau) \qquad \frac{}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \ell:\tau \stackrel{\mathrm{alloc}}{\rightsquigarrow} \ell,\epsilon}\ (\Psi(\ell)=\tau)$$

$$\frac{}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} i:\mathit{int} \stackrel{\mathrm{alloc}}{\rightsquigarrow} i,\epsilon}$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v_i:\tau_i \stackrel{\mathrm{alloc}}{\rightsquigarrow} v_i',\vec{d}_i}{\begin{array}{l}\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \langle v_1,\ldots,v_n\rangle:\langle\tau_1,\ldots,\tau_n\rangle \stackrel{\mathrm{alloc}}{\rightsquigarrow} y_n,(\vec{d}_1,\ldots,\vec{d}_n,\\ \qquad y_0=\mathit{malloc}[\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]],\\ \qquad y_1=y_0[1]\leftarrow v_1',\\ \qquad \vdots\\ \qquad y_n=y_{n-1}[n]\leftarrow v_n')\end{array}}$$

$$\frac{\Delta\vdash_{\mathrm{H}}\sigma \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\forall[\alpha,\vec{\beta}](\vec{\tau})\rightarrow\mathit{void} \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d}}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v[\sigma]:\forall[\vec{\beta}](\vec{\tau}[\sigma/\alpha])\rightarrow\mathit{void}\rightsquigarrow v'[\mathcal{A}[\![\sigma]\!]],\vec{d}}$$

$$\frac{\Delta\vdash_{\mathrm{H}}\tau \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\tau'[\tau/\alpha] \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d}}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{pack}\,[\tau,v]\ \mathit{as}\ \exists\alpha.\tau':\exists\alpha.\tau' \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{pack}\,[\mathcal{A}[\![\tau]\!],v']\ \mathit{as}\ \mathcal{A}[\![\exists\alpha.\tau']\!],\vec{d}}$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\forall[\,].(\tau_1,\ldots,\tau_n)\rightarrow\mathit{void} \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d} \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v_i:\tau_i \stackrel{\mathrm{alloc}}{\rightsquigarrow} v_i',\vec{d}_i}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v(v_1,\ldots,v_n) \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d},\vec{d}_1,\ldots,\vec{d}_n\ \mathit{in}\ v'(v_1',\ldots,v_n')}$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\mathit{int} \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d} \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} e_1 \stackrel{\mathrm{alloc}}{\rightsquigarrow} e_1' \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} e_2 \stackrel{\mathrm{alloc}}{\rightsquigarrow} e_2'}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{if0}(v,e_1,e_2) \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d}\ \mathit{in}\ \mathit{if0}(v',e_1',e_2')}$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\tau \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d}}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{halt}[\tau]v\rightsquigarrow \mathit{let}\ \vec{d}\ \mathit{in}\ \mathit{halt}[\mathcal{A}[\![\tau]\!]]v'}$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\tau \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d} \quad \Psi;\Delta;\Gamma\{x{:}\tau\}\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{let}\ x=v\ \mathit{in}\ e \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d},x=v'\ \mathit{in}\ e'}\ (x\notin\Gamma)$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\langle\tau_1,\ldots,\tau_n\rangle \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d} \quad \Psi;\Delta;\Gamma\{x{:}\tau_i\}\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{let}\ x=\pi_i(v)\ \mathit{in}\ e \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d},x=\pi_i(v')\ \mathit{in}\ e'}\ (x\notin\Gamma,1\leq i\leq n)$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v_1:\mathit{int} \stackrel{\mathrm{alloc}}{\rightsquigarrow} v_1',\vec{d}_1 \quad \Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v_2:\mathit{int} \stackrel{\mathrm{alloc}}{\rightsquigarrow} v_2',\vec{d}_2 \quad \Psi;\Delta;\Gamma\{x{:}\mathit{int}\}\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} \mathit{let}\ x=v_1\,p\,v_2\ \mathit{in}\ e \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d}_1,\vec{d}_2,x=v_1'\,p\,v_2'\ \mathit{in}\ e'}\ (x\notin\Gamma)$$

$$\frac{\Psi;\Delta;\Gamma\vdash_{\mathrm{H}} v:\exists\alpha.\tau \stackrel{\mathrm{alloc}}{\rightsquigarrow} v',\vec{d} \quad \Psi;\Delta\alpha;\Gamma\{x{:}\tau\}\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\Psi;\mathit{let}\ [\alpha,x]=\mathit{unpack}\ v\ \mathit{in}\ e \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{let}\ \vec{d},[\alpha,x]=\mathit{unpack}\ v'\ \mathit{in}\ e'}\ (x\notin\Gamma,\alpha\notin\Delta)$$

$$\frac{\vec{\alpha}\vdash_{\mathrm{H}}\tau_i \quad \Psi;\vec{\alpha};\{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\}\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\begin{array}{l}\Psi\vdash_{\mathrm{H}} \ell\mapsto \mathtt{code}[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e:\ell\mapsto\forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n)\rightarrow\mathit{void} \stackrel{\mathrm{alloc}}{\rightsquigarrow}\\ \qquad\qquad \ell\mapsto\mathtt{code}[\vec{\alpha}](x_1{:}\mathcal{A}[\![\tau_1]\!],\ldots,x_n{:}\mathcal{A}[\![\tau_n]\!]).e'\end{array}}$$

$$\frac{\emptyset\vdash_{\mathrm{H}}\tau_i \quad \Psi\vdash_{\mathrm{H}} b_i:\ell_i\mapsto\tau_i \stackrel{\mathrm{alloc}}{\rightsquigarrow} b_i' \quad \Psi;\emptyset;\emptyset\vdash_{\mathrm{H}} e \stackrel{\mathrm{alloc}}{\rightsquigarrow} e'}{\vdash_{\mathrm{H}} \mathit{letrec}\ b_1,\ldots,b_n\ \mathit{in}\ e \stackrel{\mathrm{alloc}}{\rightsquigarrow} \mathit{letrec}\ b_1',\ldots,b_n'\ \mathit{in}\ e'}\ \left(\begin{array}{l}\Psi=\ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n\\ \ell_j\neq\ell_k\ \text{when}\ j\neq k\end{array}\right)$$

Figure 16: Term Translation from λ^{H} to λ^{A}

$$
\begin{array}{l}
\mathit{letrec}\ \ \ell_{fact}\ \mapsto \mathtt{code}[\,](env{:}\langle\rangle, n{:}int, k{:}\tau_k).\\
\qquad\qquad\qquad\quad \mathit{let}\ \ \mathit{in}\ \mathit{if0}(n,\\
\qquad\qquad\qquad\qquad\qquad \mathit{let}\ [\beta, k_{unpack}] = \mathit{unpack}\ k\\
\qquad\qquad\qquad\qquad\qquad\quad k_{code} = \pi_0(k_{unpack})\\
\qquad\qquad\qquad\qquad\qquad\quad k_{env} = \pi_1(k_{unpack})\\
\qquad\qquad\qquad\qquad\qquad \mathit{in}\ k_{code}(k_{env}, 1),\\
\qquad\qquad\qquad\qquad\qquad \mathit{let}\ x = n - 1\\
\qquad\qquad\qquad\qquad\qquad\quad y_5 = \mathit{malloc}[int, \tau_k]\\
\qquad\qquad\qquad\qquad\qquad\quad y_6 = y_5[1] \leftarrow n\\
\qquad\qquad\qquad\qquad\qquad\quad y_7 = y_6[2] \leftarrow k\\
\qquad\qquad\qquad\qquad\qquad\quad y_8 = \mathit{malloc}[(\langle int^1, \tau_k^1\rangle, int) \rightarrow void, \langle int^1, \tau_k^1\rangle,]\\
\qquad\qquad\qquad\qquad\qquad\quad y_9 = y_8[1] \leftarrow \ell_{cont}\\
\qquad\qquad\qquad\qquad\qquad\quad y_{10} = y_9[2] \leftarrow y_7\\
\qquad\qquad\qquad\qquad\qquad \mathit{in}\ \ell_{fact}(env, x, \mathit{pack}\ [\langle int^1, \tau_k^1\rangle, y_{10}]\ \mathit{as}\ \tau_k))\\
\qquad\ \ell_{cont}\ \mapsto \mathtt{code}[\,](env{:}\langle int^1, \tau_k^1\rangle, y{:}int).\\
\qquad\qquad\qquad\quad \mathit{let}\ n = \pi_0(env)\\
\qquad\qquad\qquad\qquad k = \pi_1(env)\\
\qquad\qquad\qquad\qquad z = n \times y\\
\qquad\qquad\qquad\qquad [\beta, k_{unpack}] = \mathit{unpack}\ k\\
\qquad\qquad\qquad\qquad k_{code} = \pi_0(k_{unpack})\\
\qquad\qquad\qquad\qquad k_{env} = \pi_1(k_{unpack})\\
\qquad\qquad\qquad\quad \mathit{in}\ k_{code}(k_{env}, z)\\
\qquad\ \ell_{halt}\ \mapsto \mathtt{code}[\,](env{:}\langle\rangle, n{:}int).\ \mathit{let}\ \ \mathit{in}\ \mathit{halt}[int](n)\\
\mathit{in}\\
\qquad \mathit{let}\ y_0 = \mathit{malloc}[\,]\\
\qquad\quad y_1 = \mathit{malloc}[\,]\\
\qquad\quad y_2 = \mathit{malloc}[(\langle\rangle, int) \rightarrow void, \langle\rangle]\\
\qquad\quad y_3 = y_2[1] \leftarrow \ell_{halt}\\
\qquad\quad y_4 = y_3[2] \leftarrow y_1\\
\qquad \mathit{in}\ \ell_{fact}(y_0, 6, \mathit{pack}\ [\langle\rangle, y_4]\ \mathit{as}\ \tau_k)
\end{array}
$$

where τ_k is $\exists\alpha.\langle(\alpha, int) \rightarrow void^1, \alpha^1\rangle$

Figure 17: Factorial in λ^{A}

types	τ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
heap types	Ψ	$::=$	$\{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}$
register file types	Γ	$::=$	$\{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}$
type contexts	Δ	$::=$	$\vec{\alpha}$
registers	r	$::=$	$\texttt{r1} \mid \texttt{r2} \mid \texttt{r3} \mid \cdots$
word values	w	$::=$	$\ell \mid i \mid ?\tau \mid w[\tau] \mid \mathit{pack}\,[\tau, w]\;\mathit{as}\;\tau'$
small values	v	$::=$	$r \mid w \mid v[\tau] \mid \mathit{pack}\,[\tau, v]\;\mathit{as}\;\tau'$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle \mid \texttt{code}[\vec{\alpha}]\Gamma.S$
heaps	H	$::=$	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	ι	$::=$	$\texttt{add}\, r_d, r_s, v \mid \texttt{bnz}\, r, v \mid \texttt{ld}\, r_d, r_s[i] \mid \texttt{malloc}\, r_d[\vec{\tau}] \mid \texttt{mov}\, r_d, v \mid$ $\texttt{mul}\, r_d, r_s, v \mid \texttt{st}\, r_d[i], r_s \mid \texttt{sub}\, r_d, r_s, v \mid \texttt{unpack}[\alpha, r_d], v$
instruction sequences	S	$::=$	$\iota; S \mid \texttt{jmp}\, v \mid \texttt{halt}[\tau]$
programs	P	$::=$	(H, R, S)

Figure 18: Syntax of TAL

well-formed terminal configurations of the rewriting system have the form $(H, R\{\texttt{r1} \mapsto w\}, \texttt{halt}[\tau])$. This corresponds to a machine state where the register `r1` contains the value computed by the computation. All other terminal configurations are considered to be "stuck" programs.

If we erase the types from the instructions, then their meaning is intuitively clear and there is a one-to-one correspondence with conventional assembly language instructions. The two exceptions to this are the `unpack` and `malloc` instructions, which are discussed below.

Intuitively, the $\texttt{ld}\, r_d, r_s[i]$ instruction loads the i^{th} component (counting from 0) of the tuple bound to the label in r_s, and places this word value in r_d. Conversely, $\texttt{st}\, r_d[i], r_s$ places the word value in r_s at the i^{th} position of the tuple bound to the label in r_d. The instruction $\texttt{jmp}\ v$, where v is a value of the form $\ell[\vec{\tau}]$, transfers control to the code bound to the label ℓ, instantiating the abstracted type variables of ℓ with $\vec{\tau}$. The $\texttt{bnz}\, r, v$ instruction tests the value in r to see if it is zero. If so, then control continues with the next instruction. Otherwise control is transferred to v as with the `jmp` instruction.

The instruction $\texttt{unpack}[\alpha, r_d], v$, where v is a value of the form $\mathit{pack}\,[\tau', v']\;\mathit{as}\;\tau$, is evaluated by substituting τ' for α in the remainder of the sequence of instructions currently being executed, and by binding the register r_d to the value v'. If types are erased, the `unpack` instruction can be implemented with a `mov` instruction.

As at the λ^{A} level, $\texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n]$ allocates a fresh, uninitialized tuple in the heap and binds the address of this tuple to r_d. Of course, real machines do not provide a primitive `malloc` instruction. Our intention is that, as types are erased, `malloc` is expanded into a fixed instruction sequence that allocates a tuple of the appropriate size. Because this instruction sequence is abstract, it prevents optimization from re-ordering and interleaving these underlying instructions with the surrounding TAL code. However, this is the only instruction sequence that is abstract in TAL.

$(H,R,S) \longmapsto P$ where	
if $S =$	then $P =$
`add` $r_d, r_s, v; S'$	$(H, R\{r_d \mapsto R(r_s) + \hat{R}(v)\}, S')$ and similarly for `mul` and `sub`
`bnz` $r, v; S'$ when $R(r) = 0$	(H, R, S')
`bnz` $r, v; S'$ when $R(r) = i$ and $i \neq 0$	$(H, R, S''[\vec{\tau}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\tau}]$ and $H(\ell) = \mathtt{code}[\vec{\alpha}]\Gamma.S''$
`jmp` v	$(H, R, S'[\vec{\tau}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\tau}]$ and $H(\ell) = \mathtt{code}[\vec{\alpha}]\Gamma.S'$
`ld` $r_d, r_s[i]; S'$	$(H, R\{r_d \mapsto w_i\}, S')$ where $R(r_s) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle$ with $0 \leq i < n$
`malloc` $r_d[\tau_1, \ldots, \tau_n]; S'$	$(H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n\rangle\}, R\{r_d \mapsto \ell\}, S')$ where $\ell \notin H$
`mov` $r_d, v; S'$	$(H, R\{r_d \mapsto \hat{R}(v)\}, S')$
`st` $r_d[i], r_s; S'$	$(H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1}\rangle\}, R, S')$ where $R(r_d) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle$ with $0 \leq i < n$
`unpack`$[\alpha, r_d], v; S'$	$(H, R\{r_d \mapsto w\}, S'[\tau/\alpha])$ where $\hat{R}(v) = \mathit{pack}\ [\tau, w]\ \mathit{as}\ \tau'$

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

Figure 19: Operational Semantics of TAL

Real machines also have a finite amount of heap space. It is straightforward to link our TAL to a conservative garbage collector [9] in order to reclaim unused heap values. Support for an accurate collector would require introducing tags so that we may distinguish pointers from integers, or else require a type-passing interpretation [43, 30]. The tagging approach is readily accomplished in our framework.

6.3 TAL Static Semantics

The static semantics for TAL appears in Figures 21 and 22 and consists of thirteen judgments, summarized in Figure 20 and elaborated briefly below. The static semantics is inspired by and follows the conventions of Morrisett and Harper's $\lambda_{\mathrm{gc}}^{\to\forall}$ [30].

Judgment	Meaning
$\Delta \vdash_{\mathrm{TAL}} \tau$ type	τ is a well-formed type
$\vdash_{\mathrm{TAL}} \Psi$ htype	Ψ is a well-formed heap type
$\Delta \vdash_{\mathrm{TAL}} \Gamma$ rftype	Γ is a well-formed register file type
$\Delta \vdash_{\mathrm{TAL}} \tau_1 \leq \tau_2$ type	τ_1 is a subtype of τ_2
$\Delta \vdash_{\mathrm{TAL}} \Gamma_1 \leq \Gamma_2$ rftype	Γ_1 is register file subtype of Γ_2
$\vdash_{\mathrm{TAL}} H : \Psi$ heap	H is a well-formed heap of heap type Ψ
$\Psi \vdash_{\mathrm{TAL}} R : \Gamma$ regfile	R is a well-formed register file of register file type Γ
$\Psi \vdash_{\mathrm{TAL}} h : \tau$ hval	h is a well-formed heap value of type τ
$\Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau$ wval	w is a well-formed word value of type τ
$\Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau^{\varphi}$ fwval	w is a well-formed word value of flagged type τ^{φ} (*i.e.*, w has type τ or w is $?\tau$ and φ is 0)
$\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \tau$	v is a well-formed small value of type τ
$\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} S$	S is a well-formed instruction sequence
$\vdash_{\mathrm{TAL}} P$	P is a well-formed program

Figure 20: TAL Static Semantic Judgments

The first five judgments state well-formedness and subtyping of various sorts of types. Each of these judgments are relative to a type context except the heap type well-formedness judgment; heaps and heap types are required to be closed, so no type context is used. Next are two judgments for assigning types to heaps and to register files; neither heaps nor register files may contain free type variables, but register files may contain references to the heap.

The next four judgments are for assigning types to values. In addition to one rule for each sort of value, there is a rule for assigning *flagged types* to word values: the junk value $?\tau$ may not be assigned any regular type, but it may be assigned the flagged type τ^0. Each value sort may contain references to the heap, all but heap values may contain free type variables, but (as discussed above) only small values may contain registers. The final two judgments assert well-formedness of instruction sequences and programs.

A few additional words are merited on the two subtyping judgments. Neither of these are intended to support subtyping in the usual sense, although they could be expanded to do so. Instead, they are used to allow the forgetting of information in particular places where necessary. The subtyping judgment makes it possible to forget that a field of a tuple has been initialized. This is used in the subject reduction argument (Lemma 6.1) where it is sometimes necessary that references to an initialized tuple be given the old uninitialized type. The register file subtyping judgment makes it possible to forget the types of some registers. This makes it possible to jump to a code block when "too many" registers are defined.

The following two lemmas and their corollary establish type safety for TAL. Their proofs appear in Appendix A.

Lemma 6.1 (Subject Reduction) *If* $\vdash_{\mathrm{TAL}} P$ *and* $P \longmapsto P'$*, then* $\vdash_{\mathrm{TAL}} P'$*.*

$$\boxed{\Delta \vdash_{\text{TAL}} \tau \text{ type} \quad \vdash_{\text{TAL}} \Psi \text{ htype} \quad \Delta \vdash_{\text{TAL}} \Gamma \text{ rftype}}$$

$$\text{(type)}\ \frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash_{\text{TAL}} \tau \text{ type}} \qquad \text{(heap-type)}\ \frac{\emptyset \vdash_{\text{TAL}} \tau_i \text{ type}}{\vdash_{\text{TAL}} \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\} \text{ htype}}$$

$$\text{(reg-type)}\ \frac{\Delta \vdash_{\text{TAL}} \tau_i \text{ type}}{\Delta \vdash_{\text{TAL}} \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \text{ rftype}}$$

$$\boxed{\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2 \text{ type} \quad \Delta \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2 \ \text{ rftype}}$$

$$\text{(reflex)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type}}{\Delta \vdash_{\text{TAL}} \tau \leq \tau \text{ type}} \qquad \text{(trans)}\ \frac{\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2 \text{ type} \quad \Delta \vdash_{\text{TAL}} \tau_2 \leq \tau_3 \text{ type}}{\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_3 \text{ type}}$$

$$\text{(0-1)}\ \frac{\Delta \vdash_{\text{TAL}} \tau_i \text{ type}}{\Delta \vdash_{\text{TAL}} \langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle \leq \langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^0, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle \text{ type}}$$

$$\text{(weaken)}\ \frac{\Delta \vdash_{\text{TAL}} \tau_i \text{ type} \quad (\text{for } 1 \leq i \leq m)}{\Delta \vdash_{\text{TAL}} \{r_1 : \tau_1, \ldots, r_m : \tau_m\} \leq \{r_1 : \tau_1, \ldots, r_n : \tau_n\} \text{ rftype}}\ (m \geq n)$$

$$\boxed{\vdash_{\text{TAL}} P \quad \vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile}}$$

$$\text{(prog)}\ \frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} S}{\vdash_{\text{TAL}} (H, R, S)}$$

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

$$\text{(reg)}\ \frac{\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i \text{ wval} \quad (\text{for } 1 \leq i \leq m)}{\Psi \vdash_{\text{TAL}} \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m\} : \{r_1 \mapsto \tau_1, \ldots, r_n \mapsto \tau_n\} \text{ regfile}}\ (m \geq n)$$

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

$$\text{(tuple)}\ \frac{\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i} \text{ fwval}}{\Psi \vdash_{\text{TAL}} \langle w_1, \ldots, w_n \rangle : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \text{ hval}} \qquad \text{(code)}\ \frac{\vec{\alpha} \vdash_{\text{TAL}} \Gamma \text{ rftype} \quad \Psi; \vec{\alpha}; \Gamma \vdash_{\text{TAL}} S}{\Psi \vdash_{\text{TAL}} \texttt{code}[\vec{\alpha}]\Gamma.S : \forall[\vec{\alpha}].\Gamma \text{ hval}}$$

$$\text{(label)}\ \frac{\Delta \vdash_{\text{TAL}} \tau' \leq \tau \text{ type}}{\Psi; \Delta \vdash_{\text{TAL}} \ell : \tau \text{ wval}}\ (\Psi(\ell) = \tau') \qquad \text{(int)}\ \frac{}{\Psi; \Delta \vdash_{\text{TAL}} i : \mathit{int} \text{ wval}}$$

$$\text{(tapp-word)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type} \quad \Psi; \Delta \vdash_{\text{TAL}} w : \forall[\alpha, \vec{\beta}].\Gamma \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} w[\tau] : \forall[\vec{\beta}].\Gamma[\tau/\alpha] \text{ wval}}$$

$$\text{(pack-word)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type} \quad \Psi; \Delta \vdash_{\text{TAL}} w : \tau'[\tau/\alpha] \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} \mathit{pack}\ [\tau, w]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau' \text{ wval}}$$

$$\text{(init)}\ \frac{\Psi; \Delta \vdash_{\text{TAL}} w : \tau \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} w : \tau^{\varphi} \text{ fwval}} \qquad \text{(uninit)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type}}{\Psi; \Delta \vdash_{\text{TAL}} ?\tau : \tau^0 \text{ fwval}}$$

$$\text{(reg-val)}\ \frac{}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} r : \tau}\ (\Gamma(r) = \tau) \qquad \text{(word-val)}\ \frac{\Psi; \Delta \vdash_{\text{TAL}} w : \tau \text{ wval}}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} w : \tau}$$

$$\text{(tapp-val)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type} \quad \Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \forall[\alpha, \vec{\beta}].\Gamma'}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v[\tau] : \forall[\vec{\beta}].\Gamma'[\tau/\alpha]}$$

$$\text{(pack-val)}\ \frac{\Delta \vdash_{\text{TAL}} \tau \text{ type} \quad \Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \tau'[\tau/\alpha]}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \mathit{pack}\ [\tau, v]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}$$

Figure 21: Static Semantics of TAL (except instructions)

$$\boxed{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} S}$$

$$\text{(s-arith)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_s : \mathit{int} \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \mathit{int} \qquad \Psi; \Delta; \Gamma\{r_d{:}\mathit{int}\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{arith}\ r_d, r_s, v; S}\ (\mathtt{arith} \in \{\mathtt{add}, \mathtt{mul}, \mathtt{sub}\})$$

$$\text{(s-bnz)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r : \mathit{int} \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \qquad \Delta \vdash_{\mathrm{TAL}} \Gamma \leq \Gamma' \ \text{rftype} \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{bnz}\ r, v; S}$$

$$\text{(s-ld)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \qquad \Psi; \Delta; \Gamma\{r_d{:}\tau_i\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{ld}\ r_d, r_s[i]; S}\ (\varphi_i = 1, 0 \leq i < n)$$

$$\text{(s-malloc)}\ \frac{\Delta \vdash_{\mathrm{TAL}} \tau_i \ \text{type} \qquad \Psi; \Delta; \Gamma\{r_d{:}\langle \tau_1^0, \ldots, \tau_n^0 \rangle\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{malloc}\ r_d[\tau_1, \ldots, \tau_n]; S}$$

$$\text{(s-mov)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \tau \qquad \Psi; \Delta; \Gamma\{r_d : \tau\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{mov}\ r_d, v; S}$$

$$\text{(s-sto)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_s : \tau_i \qquad \Psi; \Delta; \Gamma\{r_d{:}\langle \tau_0^{\varphi_0}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{st}\ r_d[i], r_s; S}\ (0 \leq i < n)$$

$$\text{(s-unpack)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \exists \alpha.\tau \qquad \Psi; \Delta\alpha; \Gamma\{r_d{:}\tau\} \vdash_{\mathrm{TAL}} S}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{unpack}[\alpha, r_d], v; S}\ (\alpha \notin \Delta)$$

$$\text{(s-jmp)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \qquad \Delta \vdash_{\mathrm{TAL}} \Gamma \leq \Gamma' \ \text{rftype}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{jmp}\ v}$$

$$\text{(s-halt)}\ \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{r1} : \tau}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{halt}[\tau]}$$

Figure 22: Static Semantics of TAL instructions

$$
\begin{array}{rcl}
\mathcal{T}[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
\mathcal{T}[\![int]\!] & \stackrel{\text{def}}{=} & int \\
\mathcal{T}[\![\forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n) \to void]\!] & \stackrel{\text{def}}{=} & \forall[\vec{\alpha}].\{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!],\ldots,\mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\} \\
\mathcal{T}[\![\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle]\!] & \stackrel{\text{def}}{=} & \langle\mathcal{T}[\![\tau_1]\!]^{\varphi_1},\ldots,\mathcal{T}[\![\tau_n]\!]^{\varphi_n}\rangle \\
\mathcal{T}[\![\exists\alpha.\tau]\!] & \stackrel{\text{def}}{=} & \exists\alpha.\mathcal{T}[\![\tau]\!] \\
\\
\mathcal{T}[\![x_1{:}\tau_1,\ldots,x_n{:}\tau_n]\!] & \stackrel{\text{def}}{=} & \{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!],\ldots,\mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\}
\end{array}
$$

Figure 23: Type Translation from λ^{A} to TAL

Lemma 6.2 (Progress) *If $\vdash_{\mathrm{TAL}} P$, then either there exists P' such that $P \longmapsto P'$ or P is of the form $(H, R\{\mathtt{r1} \mapsto w\}, \mathtt{halt}[\tau])$.*

Corollary 6.3 (Type Safety) *If $\vdash_{\mathrm{TAL}} P$, then there is no stuck P' such that $P \longmapsto^* P'$.*

6.4 Code Generation

The type translation, $\mathcal{T}[\![\cdot]\!]$ from λ^{A} to TAL is straightforward. The only point of interest is the translation of function types, which must assign registers to value arguments:

$$\mathcal{T}[\![\forall[\vec{\alpha}](\tau_1,\cdots,\tau_n) \to void]\!] \stackrel{\text{def}}{=} \forall[\vec{\alpha}]\{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!],\ldots,\mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\}$$

The term translation appears in Figures 24 and 25. In this translation, values are translated to small values, terms are translated to pairs of instruction sequences and heaps, blocks are translated to heaps, and programs are translated to programs. The translation for terms needs to heap allocate the false branch of *if0* expressions, which requires the use of a fresh label. In the other translations of the compiler, variables used internally by the translation have always been assumed fresh, but, since labels have global scope, choosing fresh labels is not necessarily simple. Consequently, the translation handles fresh label generation explicitly by supplying a set of used labels to the term and block translation judgments. The translation uses the notation $|\Gamma|$, representing the number of bindings in Γ, and the notations $\overline{H}$ and $\overline{\Psi}$, representing the sets of labels bound in H and Ψ.

Informally, terms are translated to instruction sequences as follows:

- $x = v$ is mapped to $\mathtt{mov}\, r_x, v$.
- $x = \pi_i(v)$ is mapped to the sequence $\mathtt{mov}\, r_x, v\,;\, \mathtt{ld}\, r_x, r_x[i-1]$
- $x = v_1\ p\ v_2$ is mapped to the sequence $\mathtt{mov}\, r_x, v_1\,;\, \mathtt{arith}\ r_x, r_x, v_2$ where $\mathtt{arith}$ is the appropriate arithmetic instruction.
- $[\alpha, x] = unpack\ v$ is mapped to $\mathtt{unpack}[\alpha, r_x], v$.
- $x = malloc[\vec{\tau}]$ is mapped to $\mathtt{malloc}\, r_x[\vec{\tau}]$.

$$\frac{}{\Psi;\Delta;\{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\} \vdash_{\mathrm{A}} x_i : \tau_i \overset{\mathrm{TAL}}{\leadsto} \mathtt{r}i}\ (1 \leq i \leq n)$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} \ell : \tau \overset{\mathrm{TAL}}{\leadsto} \ell}\ (\Psi(\ell) = \tau) \qquad \frac{}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} i : \mathit{int} \overset{\mathrm{TAL}}{\leadsto} i}$$

$$\frac{\Delta \vdash_{\mathrm{A}} \sigma \quad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \forall[\alpha,\vec{\beta}].(\vec{\tau}) \rightarrow \mathit{void} \overset{\mathrm{TAL}}{\leadsto} v'}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v[\sigma] : \forall[\vec{\beta}].(\vec{\tau}[\sigma/\alpha]) \rightarrow \mathit{void} \overset{\mathrm{TAL}}{\leadsto} v'[\mathcal{T}[\![\sigma]\!]]}$$

$$\frac{\Delta \vdash_{\mathrm{A}} \tau \quad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \tau'[\tau/\alpha] \overset{\mathrm{TAL}}{\leadsto} v'}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} \mathit{pack}\ [\tau, v]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau' \overset{\mathrm{TAL}}{\leadsto} \mathit{pack}\ [\mathcal{T}[\![\tau]\!], v']\ \mathit{as}\ \mathcal{T}[\![\exists\alpha.\tau']\!]}$$

$$\frac{\vec{\alpha} \vdash_{\mathrm{A}} \tau_i \quad \Psi;\vec{\alpha};\{x_1{:}\tau_1,\ldots,x_n{:}\tau_n\} \vdash_{\mathrm{A}} e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H}{\begin{array}{r}\Psi \vdash_{\mathrm{A}} \ell \mapsto \mathtt{code}[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e : \ell \mapsto \forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n) \rightarrow \mathit{void} \mid L \overset{\mathrm{TAL}}{\leadsto} \\ H, \ell \mapsto \mathtt{code}[\vec{\alpha}]\{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!],\ldots,\mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\}.S\end{array}}$$

$$\frac{\begin{array}{c}\Psi \vdash_{\mathrm{A}} b_i : \ell_i \mapsto \tau_i \mid L_{i-1} \overset{\mathrm{TAL}}{\leadsto} H_i \qquad \emptyset \vdash_{\mathrm{A}} \tau_i \\ \Psi;\emptyset;\emptyset \vdash_{\mathrm{A}} e \mid L_n \overset{\mathrm{TAL}}{\leadsto} S, H\end{array}}{\vdash_{\mathrm{A}} \mathit{letrec}\, b_1,\ldots,b_n\ \mathit{in}\ e \overset{\mathrm{TAL}}{\leadsto} ((H_1,\ldots,H_n,H),\emptyset,S)}\ \left(\begin{array}{l}L_i = \overline{\Psi} \cup \overline{H_1} \cup \cdots \cup \overline{H_i} \\ \Psi = \ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n \\ \ell_j \neq \ell_k \text{ when } j \neq k\end{array}\right)$$

Figure 24: Program Translation from λ^{A} to TAL (except terms)

- $x = v[i] \leftarrow v'$ is mapped to the sequence:

$$\mathtt{mov}\ r_x, v\ ;\ \mathtt{mov}\ r_{\mathrm{temp}}, v'\ ;\ \mathtt{st}\ r_x[i-1], r_{\mathrm{temp}}$$

- $v(v_1,\ldots,v_n)$ is mapped to the sequence:

$$\mathtt{mov}\ r_{\mathrm{temp}_1}, v_1\ ;\ \ldots\ ;\ \mathtt{mov}\ r_{\mathrm{temp}_n}, v_n\ ;\ \mathtt{mov}\ \mathtt{r1}, r_{\mathrm{temp}_1}\ ;\ \ldots\ ;\ \mathtt{mov}\ \mathtt{r}n, r_{\mathrm{temp}_n}\ ;\ \mathtt{jmp}\ v$$

  Note that the arguments cannot be moved immediately into the registers r1, ..., rn because those registers may be used in later arguments.

- $\mathit{if0}(v, e_1, e_2)$ is mapped to the sequence:

$$\mathtt{mov}\ r_{\mathrm{temp}}, v\ ;\ \mathtt{bnz}\ r_{\mathrm{temp}}, \ell[\vec{\alpha}]\ ;\ S_1$$

  where ℓ is bound in the heap to $\mathtt{code}[\vec{\alpha}]\Gamma.S_2$, the translation of e_i is S_i, the free type variables of e_2 are contained in $\vec{\alpha}$, and Γ is the register file type corresponding to the free variables of e_2.

- $\mathit{halt}[\tau]v$ is mapped to the sequence $\mathtt{mov}\ \mathtt{r1}, v\ ;\ \mathtt{halt}[\tau]$

Lemma 6.4 $\vdash_{\mathrm{A}} P$ *if and only if there exists* P' *such that* $\vdash_{\mathrm{A}} P \overset{\mathrm{TAL}}{\leadsto} P'$

Lemma 6.5 (TAL Conversion Type Correctness) *If* $b \vdash_{\mathrm{A}} P \overset{\mathrm{TAL}}{\leadsto} P'$ *then* $\vdash_{\mathrm{TAL}} P'$.

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \forall[\,].(\tau_1,\ldots,\tau_n) \to void \overset{\mathrm{TAL}}{\leadsto} v' \quad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v_i : \tau_i \overset{\mathrm{TAL}}{\leadsto} v_i'}{\begin{array}{l}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let \;\; in\; v(v_1,\ldots,v_n) \mid L \overset{\mathrm{TAL}}{\leadsto} \\ \quad (\mathtt{mov}\,\mathtt{r}(k+1), v_1' \,;\, \ldots \,;\, \mathtt{mov}\,\mathtt{r}(k+n), v_n' \,; \\ \quad\; \mathtt{mov}\,\mathtt{r1}, \mathtt{r}(k+1) \,;\, \ldots \,;\, \mathtt{mov}\,\mathtt{r}n, \mathtt{r}(k+n) \,;\, \mathtt{jmp}\, v'), \epsilon\end{array}} \; (|\Gamma| = k)$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : int \overset{\mathrm{TAL}}{\leadsto} v' \\ \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} e_1 \mid L \overset{\mathrm{TAL}}{\leadsto} S_1, H_1 \qquad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} e_2 \mid L \cup \overline{H_1} \overset{\mathrm{TAL}}{\leadsto} S_2, H_2\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let \;\; in\; if0(v, e_1, e_2) \mid L \overset{\mathrm{TAL}}{\leadsto} \\ (\mathtt{mov}\, r, v \,;\, \mathtt{bnz}\, r, \ell[\vec{\alpha}] \,;\, S_1), (H_1, H_2, \ell \mapsto \mathtt{code}[\vec{\alpha}]\mathcal{T}[\![\Gamma]\!].S_2)\end{array}} \; \begin{pmatrix} \ell \notin L \cup \overline{H_1} \cup \overline{H_2} \\ r = \mathtt{r}(|\Gamma|+1) \\ \Delta = \vec{\alpha} \end{pmatrix}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \tau \overset{\mathrm{TAL}}{\leadsto} v'}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let \;\; in\; halt[\tau]v \mid L \overset{\mathrm{TAL}}{\leadsto} (\mathtt{mov}\,\mathtt{r1}, v' \,;\, \mathtt{halt}[\mathcal{T}[\![\tau]\!]]), \epsilon}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \tau \overset{\mathrm{TAL}}{\leadsto} v' \quad \Psi;\Delta;\Gamma\{x{:}\tau\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; x = v, \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto}\!\leadsto (\mathtt{mov}\, r, v' \,;\, S), H} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ x \notin \Gamma \end{pmatrix}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle \overset{\mathrm{TAL}}{\leadsto} v' \quad \Psi;\Delta;\Gamma\{x{:}\tau_i\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; x = \pi_i(v), \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} (\mathtt{mov}\, r, v' \,;\, \mathtt{ld}\, r, r[i-1] \,;\, S), H} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ \varphi_i = 1 \\ x \notin \Gamma \\ 1 \le i \le n \end{pmatrix}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v_1 : int \overset{\mathrm{TAL}}{\leadsto} v_1' \qquad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v_2 : int \overset{\mathrm{TAL}}{\leadsto} v_2' \\ \Psi;\Delta;\Gamma\{x{:}int\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H\end{array}}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; x = v_1\, p\, v_2, \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} (\mathtt{mov}\, r, v_1' \,;\, \mathtt{arith}_p\, r, r, v_2' \,;\, S), H} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ \mathtt{arith}_+ = \mathtt{add} \\ \mathtt{arith}_- = \mathtt{sub} \\ \mathtt{arith}_\times = \mathtt{mul} \\ x \notin \Gamma \end{pmatrix}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v : \exists\alpha.\tau \overset{\mathrm{TAL}}{\leadsto} v' \quad \Psi;\Delta\alpha;\Gamma\{x{:}\tau\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H}{\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; [\alpha, x] = unpack\; v, \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} (\mathtt{unpack}[\alpha, r], v' \,;\, S), H} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ x \notin \Gamma \\ \alpha \notin \Delta \end{pmatrix}$$

$$\frac{\Delta \vdash_{\mathrm{A}} \tau_i \quad \Psi;\Delta;\Gamma\{x{:}\langle \tau_1^0,\ldots,\tau_n^0\rangle\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H}{\begin{array}{l}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; x = malloc[\tau_1,\ldots,\tau_n], \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} \\ \qquad (\mathtt{malloc}[\mathcal{T}[\![\tau_1]\!],\ldots,\mathcal{T}[\![\tau_n]\!]] \,;\, S), H\end{array}} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ x \notin \Gamma \end{pmatrix}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v_1 : \langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle \overset{\mathrm{TAL}}{\leadsto} v_1' \qquad \Psi;\Delta;\Gamma \vdash_{\mathrm{A}} v_2 : \tau_i \overset{\mathrm{TAL}}{\leadsto} v_2' \\ \Psi;\Delta;\Gamma\{x{:}\langle \tau_1^{\varphi_1},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^1,\tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_n^{\varphi_n}\rangle\} \vdash_{\mathrm{A}} let\; \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} S, H\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma \vdash_{\mathrm{A}} let\; x = v_1[i] \leftarrow v_2, \vec{d}\; in\; e \mid L \overset{\mathrm{TAL}}{\leadsto} \\ (\mathtt{mov}\, r, v_1' \,;\, \mathtt{mov}\, r', v_2' \,;\, \mathtt{st}\, r[i-1], r' \,;\, S), H\end{array}} \; \begin{pmatrix} r = \mathtt{r}(|\Gamma|+1) \\ r' = \mathtt{r}(|\Gamma|+2) \\ x \notin \Gamma \\ 1 \le i \le n \end{pmatrix}$$

Figure 25: Term Translation from λ^{A} to TAL

Before the resulting TAL code may be run on a real machine, a compiler must perform a final register allocation step to ensure that the TAL code runs with the number of registers supplied by the machine. This may be viewed as a type-preserving optimization within the framework of TAL. A clever register allocator could greatly improve the performance of the output code by choosing registers in order to make moves unnecessary. A practical compiler would want to perform intelligent register allocation as well as other traditional low-level optimizations such as improving instruction selection and scheduling, strength reduction, common subexpression elimination, flow-control optimizations, and dead code elimination. Each of these may be implemented as a type-preserving transformation on TAL code using well-known techniques [2].

6.5 TAL Factorial

The factorial computation translated into TAL appears in Figure 26. To obtain the code shown, a few standard optimizations were applied; in particular, a clever (but automatable) register allocation and the removal of redundant moves. If the efficiency of this version is unsatisfactory, a more efficient, tail-recursive version of factorial could be coded in λ^{F}, or a highly optimized version could be hand-coded in TAL:

```
l_loop:
  code[]{r1:int,r2:int}.      % r1: the product so far, r2: the next number to be multiplied
    bnz r2,l_nonzero          % check if done
    halt[int]                 % halt with result in r1
l_nonzero:
  code[]{r1:int,r2:int}.
    mul r1,r1,r2              % multiply next number
    sub r2,r2,1               % decrement the counter
    jmp l_loop
l_fact:
  code[]{r1:int}.             % compute factorial of r1
    mov r2,r1                 % set up for loop
    mov r1,1
    jmp l_loop
```

with $S =$ `mov r1,6; jmp l_fact`.

7 Compiler Type Correctness

We have presented a compiler for System F which translates programs through a series of four intermediate lambda calculi before finally generating typed assembly language. The full compiler may then be defined by the judgment $\vdash_{\mathrm{F}} e : \tau \leadsto P$:

$$\frac{\vdash_{\mathrm{F}} e : \tau \overset{\mathrm{cps}}{\leadsto} e_{\mathrm{cps}} \quad \emptyset; \emptyset \vdash_{\mathrm{K}} e_{\mathrm{cps}} \overset{\mathrm{clos}}{\leadsto} e_{\mathrm{clos}} \quad \vdash_{\mathrm{C}} e_{\mathrm{clos}} \overset{\mathrm{hst}}{\leadsto} P_{\mathrm{hst}} \quad \vdash_{\mathrm{H}} P_{\mathrm{hst}} \overset{\mathrm{alloc}}{\leadsto} P_{\mathrm{alloc}} \quad \vdash_{\mathrm{A}} P_{\mathrm{alloc}} \overset{\mathrm{TAL}}{\leadsto} P}{\vdash_{\mathrm{F}} e : \tau \leadsto P}$$

For each translation we have given a type correctness lemma. A corollary of these type correctness lemmas is the type correctness of the entire compiler:

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

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

  and τ_k = ∃α.⟨∀[].{r1:α,r2:int}^1, α^1⟩
```

Figure 26: Typed Assembly Code for Factorial

Lemma 7.1 $\emptyset; \emptyset \vdash_{\mathrm{F}} e : \tau$ *if and only if there exists* P *such that* $\vdash_{\mathrm{F}} e : \tau \leadsto P$

Lemma 7.2 (Compiler Type Correctness) *If* $\vdash_{\mathrm{F}} e : \tau \leadsto P$ *then* $\vdash_{\mathrm{TAL}} P$

8 Extensions and Practice

We claim that the framework presented here is a practical approach to compilation. To substantiate this claim, we are constructing a compiler called TALC that maps the KML programming language [11] to a variant of the TAL described here, suitably adapted for the Intel x86 family of processors. We have found it straightforward to enrich the target language type system to include support for other type constructors, such as references, higher-order constructors, and recursive types. We omitted discussion of these features here in order to simplify the presentation.

Although this paper describes a CPS-based compiler, we opted to use a stack-based compilation model in the TALC compiler. Space considerations preclude a complete discussion of the details needed to support stacks, but the primary mechanisms are as follows: The size of the stack and the types of its contents are specified by *stack types,* and code blocks indicate stack types describing the state of the stack they expect. Since code is typically expected to work with stacks of varying size, functions may quantify over stack type variables, resulting in stack polymorphism.

Efficient support for disjoint sums and arrays also requires considerable additions to the type system. For sums, the critical issue is making the projection and testing of tags explicit. In a naive implementation, the connection between a sum and its tag is forgotten once the tag is loaded. For arrays the issue is that the index for a subscript or update operation must be checked to see that it is in bounds. Exposing the bounds check either requires a fixed code sequence, thereby constraining optimization, or else the type system must be strengthened so that some (decidable) fragment of arithmetic can be encoded in the types. Sums may also be implemented with either of the above techniques, or by using abstract types to tie sums to their tags. In the TALC compiler, in order to retain a simple type system and economical typechecking, we have initially opted for fixed code sequences but are exploring the implications of the more complicated type systems.

Finally, since we chose a type-erasure interpretation of polymorphism, adding floats to the language requires a boxing translation. However, recent work by Leroy [24] suggests that it is only important to unbox floats in arrays and within compilation units, which is easily done in our framework.

9 Summary

We have given a compiler from System F to a statically typed assembly language. The type system for the assembly language ensures that source level abstractions such as closures and polymorphic functions are enforced at the machine-code level. Furthermore, the type system does not preclude aggressive low-level optimization, such as register allocation, instruction selection, or instruction scheduling. In fact, programmers concerned with efficiency can hand-code routines in assembly, as long as the resulting code typechecks. Consequently, TAL provides a foundation for high-performance computing in environments where untrusted code must be checked for safety before being executed.

Acknowledgements

We are grateful to Martin Elsman, Úlfar Erlingsson, Dan Grossman, Robert Harper, Jason Hickey, Dexter Kozen, Frederick Smith, Stephanie Weirich, and Steve Zdancewic for their many helpful comments and suggestions.

References

[1] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *ACM Conference on Lisp and Functional Programming*, pages 12–23, Orlando, June 1994.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[4] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, January 1989.

[5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science.*

[6] Brain Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, December 1995.

[7] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[8] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.

[9] Hans J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[10] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[11] Karl Crary. *KML Reference Manual.* Department of Computer Science, Cornell University, 1996.

[12] Karl Crary. Foundations for the implementation of higher-order subtyping. In *ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.

[13] Olivier Danvy and Andrzej Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[14] Allyn Dimock, Robert Muller, Franklyn Turbak, and J. B. Wells. Strongly typed flow-directed reprsentation transformations. In *ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[15] M. J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, 1972.

[16] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[17] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[18] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[19] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, January 1993.

[20] David Kranz, R. Kelsey, J. Rees, P. R. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

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

[22] John Launchbury and Simon L. Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341, December 1995.

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

[24] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

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

[26] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[27] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, January 1996.

[28] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Progamming Languages and Systems*, 10(3):470–502, July 1988.

[29] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[30] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1997.

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

[32] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[33] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[34] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[35] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[36] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, 1974.

[37] Erik Ruf. Partitioning dataflow analyses using types. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 15–26, Paris, January 1997.

[38] Z. Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Dept. Technical Report BCCS-97-03.

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

[40] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978.

[41] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, May 1996.

[42] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[43] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.

[44] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, pages 61–78, Nice, June 1990.

[45] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[46] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, Gdansk, Poland, August 1993. Springer-Verlag.

[47] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[48] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In S. Brookes, editor, *Proceedings Mathematical Foundations of Programming Semantics '91*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311. Springer-Verlag, 1992.

[49] A. K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4), December 1995.

A Soundness of TAL

Lemma A.1 (Context Strengthening) *If* $\Delta \subseteq \Delta'$ *then:*

1. *If* $\Delta \vdash_{\text{TAL}} \tau$ type *then* $\Delta' \vdash_{\text{TAL}} \tau$ type
2. *If* $\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2$ type *then* $\Delta' \vdash_{\text{TAL}} \tau_1 \leq \tau_2$ type.

Proof:

Part 1 is immediate by (type). Part 2 is by induction on derivations.

□

Lemma A.2 (Subtyping Regularity) *If* $\Delta \vdash_{\text{TAL}} \tau \leq \tau'$ type *then* $\Delta \vdash_{\text{TAL}} \tau$ type *and* $\Delta \vdash_{\text{TAL}} \tau'$ type.

Proof:

By induction on derivations.

□

Lemma A.3 (Heap Extension) *If* $\vdash_{\text{TAL}} H : \Psi$ heap, $\emptyset \vdash_{\text{TAL}} \tau$ type, $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \tau$ hval, *and* $\ell \notin H$ *then:*

1. $\vdash_{\text{TAL}} \Psi\{\ell : \tau\}$ htype
2. $\vdash_{\text{TAL}} H\{\ell \mapsto h\} : \Psi\{\ell : \tau\}$ heap
3. *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} R : \Gamma$ regfile
4. *If* $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} S$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} S$
5. *If* $\Psi \vdash_{\text{TAL}} h : \sigma$ hval *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \sigma$ hval
6. *If* $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ fwval *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ fwval
7. *If* $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma$ wval *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma$ wval
8. *If* $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \sigma$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} v : \sigma$

Proof:

Part 1 is immediate by (heap-type). Part 2 follows from parts 1 and 5. Parts 3–8 are by induction on derivations.

□

Lemma A.4 (Heap Update) *If* $\vdash_{\text{TAL}} H : \Psi$ heap, $\emptyset \vdash_{\text{TAL}} \tau \leq \Psi(\ell)$ type, *and* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \tau$ *then:*

1. $\vdash_{\text{TAL}} \Psi\{\ell : \tau\}$ htype

2. $\vdash_{\text{TAL}} H\{\ell \mapsto h\} : \Psi\{\ell : \tau\}$ heap

3. *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} R : \Gamma$ regfile

4. *If* $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} S$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} S$

5. *If* $\Psi \vdash_{\text{TAL}} h : \sigma$ hval *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \sigma$ hval

6. *If* $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ fwval *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ fwval

7. *If* $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma$ wval *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma$ wval

8. *If* $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \sigma$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} v : \sigma$

Proof:

Part 1 is immediate by (heap-type) and Subtyping Regularity. Part 2 follows from parts 1 and 5. Parts 3–8 are by induction on derivations. The only interesting case is the case for the rule (label). The derivation must end:

$$\frac{\Delta \vdash_{\text{TAL}} \sigma' \leq \sigma \text{ type}}{\Psi; \Delta \vdash_{\text{TAL}} \ell' : \sigma \text{ wval}} \; (\Psi(\ell') = \sigma')$$

If $\ell \neq \ell'$ then clearly the inference also holds for $\Psi\{\ell : \tau\}$. Suppose $\ell = \ell'$. By hypothesis and Context Strengthening, we deduce $\Delta \vdash_{\text{TAL}} \tau \leq \sigma'$ type. Then the conclusion may be proven with the (trans) rule:

$$\frac{\dfrac{\Delta \vdash_{\text{TAL}} \tau \leq \sigma' \text{ type} \quad \Delta \vdash_{\text{TAL}} \sigma' \leq \sigma \text{ type}}{\Delta \vdash_{\text{TAL}} \tau \leq \sigma \text{ type}}}{\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} \ell : \sigma \text{ wval}} \; (\Psi\{\ell : \tau\}(\ell) = \tau)$$

□

Lemma A.5 (Register File Update) *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval *then* $\Psi \vdash_{\text{TAL}} R\{r \mapsto w\} : \Gamma\{r : \tau\}$ regfile.

Proof:

Suppose R is $\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$ and Γ is $\{r_1 \mapsto \tau_1, \ldots, r_m \mapsto \tau_m\}$ where r may or may not be in $\{r_1, \ldots, r_n\}$. Since $\Psi \vdash_{\text{TAL}} R : \Gamma$, by the rule (reg) it must be the case that $n \geq m$ and $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i$ wval (for all $1 \leq i \leq n$ and some $\tau_{m+1}, \ldots, \tau_n$). So certainly for i such that $r_i \neq r$, $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i$ wval, and by hypothesis $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval so by rule (reg) $\Psi \vdash_{\text{TAL}} R\{r \mapsto w\} : \Gamma\{r \mapsto \tau\}$ regfile.

□

Lemma A.6 (Canonical Heap Forms) *If* $\Psi \vdash_{\text{TAL}} h : \tau$ hval *then:*

1. *If* $\tau = \forall[\vec{\alpha}].\Gamma$ *then:*
   (a) $h = \mathtt{code}[\vec{\alpha}]\Gamma.S$
   (b) $\Psi; \vec{\alpha}; \Gamma \vdash_{\text{TAL}} S$

2. *If* $\tau = \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$ *then:*
   (a) $h = \langle w_0, \ldots, w_{n-1} \rangle$

(b) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$ fwval

Proof:

By inspection.

$\square$

Lemma A.7 (Canonical Word Forms) *If* $\vdash_{\text{TAL}} H : \Psi$ heap *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval *then:*

1. *If* $\tau = int$ *then* $w = i$.
2. *If* $\tau = \forall[\beta_1, \ldots, \beta_m].\Gamma$ *then:*
   (a) $w = \ell[\sigma_1, \ldots, \sigma_n]$
   (b) $H(\ell) = \mathtt{code}[\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m]\Gamma'.S$
   (c) $\Gamma = \Gamma'[\vec{\sigma}/\vec{\alpha}]$
   (d) $\Psi; \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m; \Gamma' \vdash_{\text{TAL}} S$
3. *If* $\tau = \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$ *then:*
   (a) $w = \ell$
   (b) $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$
   (c) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$ fwval
4. *If* $\tau = \exists\alpha.\tau$ *then* $w = pack\ [\tau', w']\ as\ \exists\alpha.\tau$ *and* $\Psi; \emptyset \vdash_{\text{TAL}} w' : \tau[\tau'/\alpha]$ wval.

Proof:

1. By inspection.
2. By induction on the derivation of $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval: The derivation must end with either the (label) or the (tapp-word) rule. Suppose the former. Then $w = \ell$, $\Psi(\ell) = \tau'$ and $\emptyset \vdash_{\text{TAL}} \tau' \leq \forall[\vec{\beta}].\Gamma$ type. Inspection of the subtyping rules then reveals that $\tau' = \forall[\vec{\beta}].\Gamma$. Since $\vdash_{\text{TAL}} H : \Psi$ heap, we may deduce that $\Psi \vdash_{\text{TAL}} H(\ell) : \forall[\vec{\beta}].\Gamma$ hval. The conclusion follows by Canonical Heap Forms.

   Alternatively, suppose the derivation ends with (tapp-word). Then $w = w'[\sigma]$ and $\Psi; \emptyset \vdash_{\text{TAL}} w' : \forall[\alpha, \vec{\beta}].\Gamma'$ wval with $\Gamma = \Gamma'[\sigma/\alpha]$. The conclusion follows by induction.
3. The derivation $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval must be shown by use of the (label) rule. Thus, $w = \ell$, $\Psi(\ell) = \tau'$ and $\emptyset \vdash_{\text{TAL}} \tau' \leq \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$ type. Let us say that $\varphi \leq \varphi$ and $1 \leq 0$. Then inspection of the subtype rules reveals that τ' must be of the form $\langle \tau_0^{\varphi'_0}, \ldots, \tau_{n-1}^{\varphi'_{n-1}} \rangle$ with $\varphi'_i \leq \varphi_i$ (for each $0 \leq i \leq n-1$). Since $\vdash_{\text{TAL}} H : \Psi$ heap, we may deduce that $\Psi \vdash_{\text{TAL}} H(\ell) : \langle \tau_0^{\varphi'_0}, \ldots, \tau_{n-1}^{\varphi'_{n-1}} \rangle$ hval. Thus $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$ and $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi'_i}$ fwval by Canonical Heap Forms. It remains to show that $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$ fwval for all $0 \leq i \leq n-1$. Suppose $\varphi'_i = 1$ and $\varphi_i = 0$ (otherwise the conclusion is immediate). Then $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^1$ fwval is shown by the (init) rule, which also permits the deduction of $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^0$ fwval.
4. By inspection.

$\square$

Lemma A.8 ($\hat{R}$ Typing) *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile *and* $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau$ *then* $\Psi; \emptyset \vdash_{\text{TAL}} \hat{R}(v) : \tau$ wval.

Proof:

The proof is by induction on the syntax of v. Cases:

$v = w$: Immediate.

$v = r$: The only rule that can type v is (reg-val) and this rule requires $\tau = \Gamma(r)$. The only rule that can type R is (reg) and this rule requires $\Psi; \emptyset \vdash_{\text{TAL}} R(r) : \tau$ wval, the conclusion follows since $\hat{R}(r) = R(r)$.

$v = v'[\sigma]$: The only rule that can type v is (tapp-val), so $\tau = \forall[\vec{\beta}].\Gamma'[\sigma/\alpha]$ and $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v' : \forall[\alpha, \vec{\beta}].\Gamma'$. By induction we deduce $\Psi; \emptyset \vdash_{\text{TAL}} \hat{R}(v') : \forall[\alpha, \vec{\beta}].\Gamma'$ wval, and then the rule (tapp-word) proves $\Psi; \emptyset \vdash_{\text{TAL}} \hat{R}(v')[\sigma] : \forall[\vec{\beta}].\Gamma'[\sigma/\alpha]$ wval. The result follows since $\hat{R}(v'[\sigma]) = \hat{R}(v')[\sigma]$.

$v = \textit{pack}\,[\sigma, v']\ \textit{as}\ \exists\alpha.\tau'$: The only rule that can type v is (pack-val), so $\tau = \exists\alpha.\tau'$ and $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v' : \tau'[\sigma/\alpha]$. By induction we deduce $\Psi; \emptyset \vdash_{\text{TAL}} \hat{R}(v') : \tau'[\sigma/\alpha]$ wval and then the rule (pack-word) proves $\Psi; \emptyset \vdash_{\text{TAL}} \textit{pack}\,[\sigma, \hat{R}(v')]\ \textit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'$ wval. The result follows since $\hat{R}(\textit{pack}\,[\sigma, v']\ \textit{as}\ \exists\alpha.\tau') = \textit{pack}\,[\sigma, \hat{R}(v')]\ \textit{as}\ \exists\alpha.\tau'$.

□

Lemma A.9 (Canonical Forms) *If* $\vdash_{\text{TAL}} H : \Psi$ hval, $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile, *and* $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau$ *then:*

1. *If* $\tau = \mathit{int}$ *then* $\hat{R}(v) = i$.
2. *If* $\tau = \forall[\beta_1, \ldots, \beta_m].\Gamma$ *then:*
   (a) $\hat{R}(v) = \ell[\sigma_1, \ldots, \sigma_n]$
   (b) $H(\ell) = \texttt{code}[\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m]\Gamma'.S$
   (c) $\Gamma = \Gamma'[\vec{\sigma}/\vec{\alpha}]$
   (d) $\Psi; \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m; \Gamma' \vdash_{\text{TAL}} S$
3. *If* $\tau = \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$ *then:*
   (a) $\hat{R}(v) = \ell$
   (b) $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$
   (c) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$ fwval
4. *If* $\tau = \exists\alpha.\tau$ *then* $\hat{R}(v) = \textit{pack}\,[\tau', w]\ \textit{as}\ \exists\alpha.\tau$ *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau[\tau'/\alpha]$ wval.

Proof:

Immediate from $\hat{R}$ Typing and Canonical Word Forms.

□

Lemma A.10 (Type Substitution) *If* $\vec{\beta} \vdash_{\text{TAL}} \tau_i$ type *then:*

1. *If* $\Psi; \vec{\alpha}, \vec{\beta}; \Gamma \vdash_{\text{TAL}} S$ *then* $\Psi; \vec{\beta}; \Gamma[\vec{\tau}/\vec{\alpha}] \vdash_{\text{TAL}} S[\vec{\tau}/\vec{\alpha}]$
2. *If* $\Psi; \vec{\alpha}, \vec{\beta}; \Gamma \vdash_{\text{TAL}} v : \tau$ *then* $\Psi; \vec{\beta}; \Gamma[\vec{\tau}/\vec{\alpha}] \vdash_{\text{TAL}} v[\vec{\tau}/\vec{\alpha}] : \tau[\vec{\tau}/\vec{\alpha}]$
3. *If* $\Psi; \vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} w : \tau$ wval *then* $\Psi; \vec{\beta} \vdash_{\text{TAL}} w[\vec{\tau}/\vec{\alpha}] : \tau[\vec{\tau}/\vec{\alpha}]$ wval

4. If $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$ rftype *then* $\vec{\beta} \vdash_{\text{TAL}} \Gamma_1[\vec{\tau}/\vec{\alpha}] \leq \Gamma_2[\vec{\tau}/\vec{\alpha}]$ rftype

5. If $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \tau_1 \leq \tau_2$ type *then* $\vec{\beta} \vdash_{\text{TAL}} \tau_1[\vec{\tau}/\vec{\alpha}] \leq \tau_2[\vec{\tau}/\vec{\alpha}]$ type

6. If $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \tau \vdash_{\text{TAL}}$ type *then* $\vec{\beta} \vdash_{\text{TAL}} \tau[\vec{\tau}/\vec{\alpha}]$ type

Proof:

By induction on derivations. The only interesting case the case for the rule (type):

$$\frac{FTV(\tau) \subseteq \{\vec{\alpha}, \vec{\beta}\}}{\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \tau \text{ type}}$$

The hypothesis must also be proven with the rule (type), so $FTV(\tau_i) \subseteq \{\vec{\beta}\}$. Consequently:

$$\begin{array}{rcl} FTV(\tau[\vec{\tau}/\vec{\alpha}]) & \subseteq & FTV(\tau) \setminus \{\vec{\alpha}\} \cup (\bigcup_i FTV(\tau_i)) \\ & \subseteq & \{\vec{\alpha}, \vec{\beta}\} \setminus \{\vec{\alpha}\} \cup \{\vec{\beta}\} \\ & = & \{\vec{\beta}\} \end{array}$$

Hence we may prove $\vec{\beta} \vdash_{\text{TAL}} \tau[\vec{\tau}/\vec{\alpha}]$ type using the (type) rule.

□

Lemma A.11 (Register File Weakening)
If $\Delta \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$ rftype *and* $\Psi; \Delta \vdash_{\text{TAL}} R : \Gamma_1$ regfile *then* $\Psi; \Delta \vdash_{\text{TAL}} R : \Gamma_2$ regfile*.*

Proof:

By inspection of the rules (weaken) and (reg).

□

Lemma A.12 (Subject Reduction) *If* $\vdash_{\text{TAL}} P$ *and* $P \longmapsto P'$ *then* $\vdash_{\text{TAL}} P'$*.*

Proof:

P has the form $(H, R, \iota; S)$ or $(H, R, \texttt{jmp}\, v)$. Let TD be the derivation of $\vdash_{\text{TAL}} P$. Consider the following cases for jmp or ι:

case jmp: TD has the form:

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \quad \emptyset \vdash_{\text{TAL}} \Gamma \leq \Gamma' \text{ rftype}}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{jmp}\, v}}{\vdash_{\text{TAL}} P}$$

By the operational semantics, $P' = (H, R, S[\vec{\sigma}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\sigma}]$ and $H(\ell) = \texttt{code}[\vec{\alpha}]\Gamma''.S$. Then:

1. $\vdash_{\text{TAL}} H : \Psi$ heap is in TD.
2. From $\emptyset \vdash_{\text{TAL}} \Gamma \leq \Gamma'$ rftype and $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile it follows by Register File Weakening that $\Psi \vdash_{\text{TAL}} R : \Gamma'$ regfile.
3. By Canonical Forms it follows from $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma'$ that $\Gamma' = \Gamma''[\vec{\sigma}/\vec{\alpha}]$ and $\Psi; \vec{\alpha}; \Gamma'' \vdash_{\text{TAL}} S$. By Type Substitution we conclude $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} S[\vec{\sigma}/\vec{\alpha}]$.

case add, mul, sub: TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r_s : \mathit{int} \qquad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} v : \mathit{int} \qquad \Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} S}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \mathtt{arith}_p\, r_d, r_s, v; S}}{\vdash_{\text{TAL}} P}$$

where $\Gamma' = \Gamma\{r_d : \mathit{int}\}$. By the operational semantics, $P' = (H, R', S)$ where $R' = R\{r_d \mapsto R(r_s)\, p\, \hat{R}(v)\}$. Then:

1. $\vdash_{\text{TAL}} H : \Psi$ heap is in TD.
2. By Canonical Forms it follows that $R(r_s)$ and $\hat{R}(v)$ are integer literals, and therefore $\Psi;\emptyset \vdash_{\text{TAL}} R(r_s)\, p\, \hat{R}(v)$ wval. We conclude $\Psi \vdash_{\text{TAL}} R' : \Gamma'$ regfile by Register File Update.
3. $\Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} S$ is in TD.

case bnz: TD has the form:

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r : \mathit{int} \qquad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \qquad \emptyset \vdash_{\text{TAL}} \Gamma \leq \Gamma' \text{ rftype} \qquad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} S}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \mathtt{bnz}\, r, v; S}}{\vdash_{\text{TAL}} P}$$

If $R(r) = 0$ then $P' = (H, R, S)$ and $\vdash_{\text{TAL}} P'$ follows since $\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} S$ is in TD. Otherwise the reasoning is exactly as in the case for jmp.

case ld: TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{0 \leq i \leq n-1 \qquad \varphi_i = 1 \qquad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \qquad \Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} S}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \mathtt{ld}\, r_d, r_s[i]; S}}{\vdash_{\text{TAL}} P}$$

where $\Gamma' = \Gamma\{r_d : \tau_i\}$. By the operational semantics, $P' = (H, R', S)$ where $R' = R\{r_d \mapsto w_i\}$, $R(r_s) = \ell$, $H(\ell) = \langle w_0, \ldots, w_{m-1}\}$ and $0 \leq i < m$. Then:

1. $\vdash_{\text{TAL}} H : \Psi$ heap is in TD.
2. By Canonical Forms it follows from $\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$ that $m = n$ and $\Psi;\emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j}$ fwval for $0 \leq j < n$. Since $\varphi_i = 1$ it must be the case (by inspection of the (init) rule) that $\Psi;\emptyset \vdash_{\text{TAL}} w_i : \tau_i$ wval. By Register File we conclude $\Psi \vdash_{\text{TAL}} R' : \Gamma'$ regfile.
3. $\Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} S$ is in TD.

case malloc: TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\emptyset \vdash_{\text{TAL}} \tau_i \text{ type} \quad \Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} S \text{ type}}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \mathtt{malloc}\, r_d[\tau_1, \ldots, \tau_n]; S}}{\vdash_{\text{TAL}} P}$$

where $\sigma = \langle \tau_1^0, \ldots, \tau_n^0 \rangle$, $\Psi' = \Psi\{\ell : \sigma\}$, and $\Gamma' = \Gamma\{r_d : \sigma\}$. By the operational semantics, $P' = (H', R', S)$ where $H' = H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}$, $R' = R\{r_d \mapsto \ell\}$, and $\ell \notin H$. Then:

1. By the (tuple) and (uninit) rules we may deduce $\Psi' \vdash_{\text{TAL}} \langle ?\tau_1, \ldots, ?\tau_n \rangle$ hval : σ. By Heap Extension it follows that $\vdash_{\text{TAL}} H' : \Psi'$ heap.
2. By the (type), (reflex), and (label) rules we may deduce that $\Psi'; \emptyset \vdash_{\text{TAL}} \ell : \sigma$ wval. By Heap Extension we deduce that $\Psi' \vdash_{\text{TAL}} R : \Gamma$ regfile and it follows by Register File Update that $\Psi' \vdash_{\text{TAL}} R' : \Gamma'$ regfile.
3. By Heap Extension, $\Psi'; \emptyset; \Gamma' \vdash_{\text{TAL}} S$.

case `mov`: TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau \quad \Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} S}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{mov}\, r, v; S}}{\vdash_{\text{TAL}} P}$$

where $\Gamma' = \Gamma\{r : \tau\}$. By the operational semantics, $P' = (H, R', S)$ where $R' = R\{r \mapsto \hat{R}(v)\}$. Then:

1. $\vdash_{\text{TAL}} H : \Psi$ heap is in TD.
2. By $\hat{R}$ Typing it follows from $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau$ that $\Psi; \emptyset \vdash_{\text{TAL}} \hat{R}(v) : \tau$ wval. Using Register File Update we conclude that $\Psi \vdash_{\text{TAL}} R' : \Gamma'$ regfile.
3. $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} S$ is in TD.

case `st`: TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\begin{array}{cc} 0 \le i \le n-1 & \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_d : \sigma_0 \\ \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_s : \tau_i & \Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} S \end{array}}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{st}\, r_d[i], r_s; S}}{\vdash_{\text{TAL}} P}$$

where:

$$\begin{array}{rcl} \sigma_0 & = & \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \sigma_1 & = & \langle \tau_0^{\varphi_0}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \Gamma' & = & \Gamma\{r_d : \sigma_1\} \end{array}$$

By the operational semantics, $P' = (H', R, S)$ where

$$H' = H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{m-1} \rangle\}$$

and $R(r_d) = \ell$, $H(\ell) = \langle w_0, \ldots, w_m \rangle$, and $0 \le i < m$. Then:

1. Since $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_d : \sigma_0$, it must be the case that $\Gamma(r_d) = \sigma_0$ and thus since $\Psi \vdash_{\text{TAL}} R : \Gamma$ regfile and $R(r_d) = \ell$ we may deduce $\Psi; \emptyset \vdash_{\text{TAL}} \ell : \sigma_0$ wval. The latter judgment must be proven with the (label) rule, hence $\emptyset \vdash_{\text{TAL}} \sigma_0' \le \sigma_0$ type where $\Psi(\ell) = \sigma_0'$. Note that it follows from Subtyping Regularity and the definition of σ_0 that $\emptyset \vdash_{\text{TAL}} \tau_j$ type for each $0 \le j < n$.
   Let us say that $\varphi \le \varphi$ and $1 \le 0$. Inspection of the subtyping rules reveals that σ_0' must be of the form $\langle \tau_0^{\varphi_0'}, \ldots, \tau_{n-1}^{\varphi_{n-1}'} \rangle$ with $\varphi_j' \le \varphi_j$. Let:

$$\sigma_1' = \langle \tau_0^{\varphi_0'}, \ldots, \tau_{i-1}^{\varphi_{i-1}'}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}'}, \ldots, \tau_{n-1}^{\varphi_{n-1}'} \rangle$$

Then $\emptyset \vdash_{\text{TAL}} \sigma_1' \leq \sigma_0'$ type and $\emptyset \vdash_{\text{TAL}} \sigma_1' \leq \sigma_1$ type. Since $\vdash_{\text{TAL}} H : \Psi$ heap, we may deduce that $m = n$ and $\Psi; \emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j'}$ fwval for $0 \leq j < n$. Let $\Psi' = \Psi\{r_d : \sigma_1'\}$. By Heap Update it follows that $\Psi'; \emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j'}$ fwval.

Using $\hat{R}$ Typing and Heap Update, we may deduce that $\Psi'; \emptyset \vdash_{\text{TAL}} R(r_s) : \tau_i$ wval and, by applying the (init) and (tuple) rules, we may conclude:

$$\Psi'; \emptyset \vdash_{\text{TAL}} \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{m-1} \rangle : \sigma_1' \text{ hval}$$

Hence $\vdash_{\text{TAL}} H' : \Psi'$ heap by Heap Update.

2. By Heap Update we may deduce $\Psi' \vdash_{\text{TAL}} R : \Gamma$. Recall that $\emptyset \vdash_{\text{TAL}} \sigma_1' \leq \sigma_1$ type. Thus, $\Psi'; \emptyset \vdash_{\text{TAL}} \ell : \sigma_1$ wval, and by Register File Update we may conclude that $\Psi' \vdash_{\text{TAL}} R : \Gamma'$ regfile (since $R = R\{r_d \mapsto \ell\}$).
3. By Heap Update, $\Psi'; \emptyset; \Gamma' \vdash_{\text{TAL}} S$.

case `unpack`: TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \exists\alpha.\tau' \quad \Psi; \alpha; \Gamma\{r{:}\tau'\} \vdash_{\text{TAL}} S}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{unpack}[\alpha, r], v; S}}{\vdash_{\text{TAL}} P}$$

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

1. $\vdash_{\text{TAL}} H : \Psi$ heap is in TD.
2. By Canonical Forms it follows from $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \exists\alpha.\tau'$ that $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau'[\tau/\alpha]$ wval. Let $\Gamma' = \Gamma\{r : \tau'[\tau/\alpha]\}$. By Register File Update if follows that $\Psi \vdash_{\text{TAL}} R' : \Gamma'$ regfile.
3. By Type Substitution it follows from $\Psi; \alpha; \Gamma\{r{:}\tau'\} \vdash_{\text{TAL}} S$ that $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} S'$.

□

Lemma A.13 (Progress) *If* $\vdash_{\text{TAL}} P$ *then either there exists* P' *such that* $P \longmapsto P'$ *or* P *is of the form* $(H, R\{\texttt{r1} \mapsto w\}, \texttt{halt}[\tau])$ *(and, moreover,* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval *for some* Ψ *such that* $\vdash_{\text{TAL}} H : \Psi$ heap*).*

Proof:

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

case `halt`: TD has the form:

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi; \emptyset \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{r1} : \tau}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \mathit{halt}[\tau]}}{\vdash_{\text{TAL}} (H, R, \mathit{halt}[\tau])}$$

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

case `add`, `mul`, `sub`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} r_s : int \quad \Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} v : int \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{arith}_p\, r_d, r_s, v; S}}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

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

case `bnz`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} r : int \quad \Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{bnz}\, r, v; S}}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

By Canonical Forms, $R(r)$ is an integer literal and $\hat{R}(v) = \ell[\sigma_1, \ldots, \sigma_n]$ with $H(\ell) = \mathtt{code}[\alpha_1, \ldots, \alpha_n].\Gamma''.S'$. If $R(r) = 0$ then $P \longmapsto (H, R, S)$. If $R(r) \neq 0$ then $P \longmapsto (H, R, S'[\vec{\sigma}/\vec{\alpha}])$.

case `jmp`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{jmp}\, v; S}}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

By Canonical Forms, $\hat{R}(v) = \ell[\sigma_1, \ldots, \sigma_n]$ with $H(\ell) = \mathtt{code}[\alpha_1, \ldots, \alpha_n].\Gamma''.S'$. Hence $P \longmapsto (H, R, S'[\vec{\sigma}/\vec{\alpha}])$.

case `ld`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{ld}\, r_d, r_s[i]; S} (1 \leq i < n)}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

By Canonical Forms, $R(r_s) = \ell$ with $H(\ell) = \langle w_0, \ldots, w_{n-1} \rangle$. Hence $P \longmapsto (H, R\{r_d \mapsto w_i\}, S)$.

case `malloc`: Suppose S_{full} is of the form $\mathtt{malloc}\, r[\tau_1, \ldots, \tau_n]; S$. Then $P \longmapsto (H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}, R\{r \mapsto \ell, S)$ for some $\ell \notin H$.

case `mov`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} v : \tau \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{mov}\, r, v; S}}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

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

case `st`: TD has the form:

$$\dfrac{\vdash_{\mathrm{TAL}} H : \Psi \text{ heap} \quad \Psi;\emptyset \vdash_{\mathrm{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\begin{array}{c}\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} r_s : \tau_i \quad \cdots\end{array}}{\Psi;\emptyset;\Gamma \vdash_{\mathrm{TAL}} \mathtt{st}\, r_d[i], r_s; S} (1 \leq i < n)}{\vdash_{\mathrm{TAL}} (H, R, S_{\mathrm{full}})}$$

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

case `unpack`: TD has the form:

$$\frac{\vdash_{\text{TAL}} H : \Psi \text{ heap} \quad \Psi; \emptyset \vdash_{\text{TAL}} R : \Gamma \text{ regfile} \quad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \exists \alpha.\tau \quad \cdots}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{unpack}[\alpha, r], v; S}}{\vdash_{\text{TAL}} (H, R, S_{\text{full}})}$$

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

□