From System F to Typed Assembly Language

GREG MORRISETT and DAVID WALKER
Cornell University
KARL CRARY
Carnegie Mellon University
and
NEAL GLEW
Cornell University

We motivate the design of a *typed assembly language* (TAL) and present a type-preserving translation from System F to TAL. The typed assembly language 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 user-defined abstract data types. The type system ensures that well-typed programs cannot violate these abstractions. In addition, the typing constructs admit many low-level compiler optimizations. 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 *certified code*, suitable for use in systems where untrusted and potentially malicious code must be checked for safety before execution.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, Syntax*; D.3.2 [**Programming Languages**]: Language Classifications—*Macro and Assembly Languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational Semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Closure conversion, certified code, secure extensible systems, typed assembly language, type-directed compilation, typed intermediate languages

This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-1-0317, and NSF grants CCR-9317320 and CCR-9708915. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not reflect the views of these agencies.

Authors's addresses: G. Morrisett, D. Walker, and N. Glew: 4130 Upson Hall, Ithaca, NY 14853-7501, USA; K. Crary: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA.

1. INTRODUCTION

Compilers that manipulate statically typed intermediate languages have compelling advantages over traditional compilers that manipulate untyped program representations. 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, closure conversion, unboxing, subsumption elimination, or region inference. Many of these optimizations require type information in order to succeed, and even those that do not, often do benefit from the additional structure supplied by a typing discipline. Moreover, the essence of many of these program transformations can be specified by the corresponding type translation. Types provide concise and yet precise documentation of the compilation process that can be automatically verified by a type checker. In practice, this technique has been invaluable for debugging new transformations and optimizations [Tarditi et al. 1996; Morrisett et al. 1996].

Today a small number of compilers work with typed intermediate languages in order to realize some or all of these benefits [Leroy 1992; Peyton Jones et al. 1993; Birkedal et al. 1993; Tarditi et al. 1996; Lindholm and Yellin 1996; Shao 1997; Dimock et al. 1997]. However, in all of these compilers, there is a 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 recode the untyped portions of a compiler to maintain type information through all phases of compilation and, in so doing, extend the paradigm of compiling with typed intermediate languages to compiling with typed *target* languages.

The target language in this article is a strongly typed assembly language (TAL) based on a generic RISC instruction set. The type system for TAL is surprisingly standard; supporting tuples, polymorphism, existential packages, and a restricted form of function pointer, yet it is sufficiently powerful that we can automatically generate well-typed code from high-level ML-like languages.

The TAL framework admits most conventional low-level optimizations such as global register allocation, copy propagation, constant folding, and dead-code elimination. Except for a small number of atomic code patterns, TAL also supports code motion optimizations such as instruction scheduling, common-subexpression elimination, and loop-invariant removal. Some more advanced implementation techniques are not supported by the simple typed assembly language we present here, including run-time code generation, intensional polymorphism, and array bounds check elimination. In Section 8 we discuss how to extend the type system presented here to support such techniques.

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 safely and efficiently. For example, as suggested by the SPIN project [Bershad et al. 1995], operating systems could allow users to download TAL extensions into the kernel. The kernel would type check the TAL extension, thereby ensuring that it never accesses hidden resources within the kernel, always calls kernel routines with the right number and types of arguments, and so forth. After the type checker has verified the extension, the kernel can safely assemble it and dynamically link it in. Such a TAL-based system has a number of advantages. Currently, SPIN requires that extensions be

written in a single high-level language (Modula-3) and use a single trusted compiler (along with cryptographic signatures) in order to ensure their safety. 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, since 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 better performance. TAL could also be used to support extensible Web browsers, extensible servers, active networks, or any other extensible system where security, performance, and language independence are desired. Of course, while type safety implies many important security properties such as memory safety, neither SPIN nor TAL can enforce other important security properties, such as termination, that do not follow from type safety.

Another framework for verifying safety properties in low-level programs, proposed by Necula and Lee, is called *proof-carrying code* (PCC) [Necula and Lee 1996; Necula 1997; 1998]. Necula and Lee encode the relevant operational content of simple type systems using extensions to first-order predicate logic, and automatically verify proofs of security properties such as memory safety [Necula 1997]. Because Necula and Lee use a general-purpose logic, they can encode more expressive security properties and permit some optimizations that are impossible in TAL. TAL, on the other hand, provides compiler writers with a higher-level set of abstractions than PCC. These abstractions make compiling languages with features such as higher-order functions and data types simpler. In order to do the same, a PCC programmer must build such abstractions from the low-level logical primitives, and it is not always obvious how to obtain a compact logical encoding of these language constructs that preserves the necessary security properties. Another benefit of the TAL abstractions is that they make it possible to automatically reconstruct the proof of type safety; TAL binaries can be more compact than PCC binaries because they do not need to contain a complete proof. Clearly, however, the ideal system contains both the compiler support and compact annotations of TAL and the flexibility of PCC. We leave this long-term goal to future research; here we focus on the theoretical framework for automatic compilation of high-level languages to type-safe assembly language.

2. OVERVIEW

The goals of this work are twofold: first, to define a type system for a conventional assembly language and to prove its soundness, and, second, to demonstrate the expressiveness of the resulting language by showing how to automatically compile a high-level language to type-correct assembly code. In this section, we give a brief overview of our typed assembly language and the structure of our compiler.

2.1 TAL

The primary goal of the TAL type system is to provide a fully automatic way to verify that programs will not violate the primitive abstractions of the language. In a conventional untyped assembly language, all values are represented as word-sized integers, and the primitive operations of the language apply to any such values. That is, in an untyped assembly language, there is only one abstraction: the machine word. In contrast, TAL provides a set of built-in abstractions, such

as (word-sized) integers, pointers to tuples, and code labels, for each of which only some operations are applicable. For example, arithmetic is only permitted on integer values; dereferencing is only permitted for pointer values; and control transfer is only permitted for code labels. We say that a program becomes *stuck* if it attempts to perform an unpermissible operation. Hence, the primary goal of the TAL type system is to ensure that well-typed programs do not become stuck.

Because TAL treats integers as a separate abstraction from pointers or code labels, and because arithmetic is only permitted on integers, it is possible to show, that, in addition to never becoming stuck, a well-typed TAL program satisfies a number of important safety policies relevant to security. For instance, it is possible to conclude that programs cannot manufacture pointers to arbitrary objects, or that programs cannot jump to code that has not been verified.

In addition to providing a set of built-in abstractions, TAL provides a set of type constructors that may be used by programmers or compilers to build new abstractions. For example, in the functional language compiler that we sketch, closures (a high-level language abstraction) are encoded as TAL-level abstractions using the existential type constructor. In the high-level language, it is impossible for a program to apply any primitive operation to a closure except for function application. For instance, it is impossible for a program to inspect the environment of the closure. At the TAL level, closures are represented as a pair of a label (to some code) and an environment data structure (intended to hold the free variables of the code). We use an existential type to hide the type of the environment data structure and to connect it to the code. The resulting object prevents malicious or faulty code from reading the environment, or passing anything but the closure's environment to the closure's code.

In other work, we have extended the type system to support many more abstractions, such as modules [Glew and Morrisett 1999] and the run-time stack [Morrisett et al. 1998]. Here, we have attempted to keep the type system simple enough that the formalism may be understood and proven sound, yet powerful enough that we can demonstrate how a high-level ML-like language may be compiled to type-correct TAL code automatically.

The typed assembly language we present here is based on a conventional RISC-style assembly language. In particular, all but two of the instructions are standard assembly operations. In an effort to simplify the formalism, we have omitted many typical instructions, such as a jump-and-link, that may be synthesized using our primitives. Figure 1 gives an example TAL program that, when control is transferred to the label `l_main`, computes 6 factorial and then halts with the result in register `r1`. The code looks and behaves much like standard assembly code, except that each label is annotated with a `code` precondition that associates types with registers. The precondition specifies, that, before control can be transferred to the corresponding label, the registers must contain values of the specified types. Hence, before allowing a jump to `l_fact` as in `l_main`, the type checker ensures that an integer value resides in register `r1`.

2.2 A Type-Preserving Compiler

In order to motivate the typing constructs in TAL and to justify our claims about its expressiveness, we spend a large part of this article sketching a compiler from a

```
l_main:
  code[]{}.                          % entry point
    mov r1,6
    jmp l_fact
l_fact:
  code[]{r1:int}.                    % compute factorial of r1
    mov r2,r1                        % set up for loop
    mov r1,1
    jmp l_loop
l_loop:
  code[]{r1:int,r2:int}.             % r1: the product so far,
                                     % r2: the next number to be multiplied
    bnz r2,l_nonzero                 % branch if not zero
    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
```

Fig. 1. A TAL program that computes 6 factorial.

variant of the polymorphic lambda-calculus to TAL. Our compiler is structured as five translations between six typed calculi:

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

Each calculus is a *first-class* programming language in the sense that each translation operates correctly on any well-typed program of its input calculus. The translations do *not* assume that their input is the output from the preceding translation. This fact frees a compiler to optimize code aggressively between any of the translation steps. The inspiration for the phases and their ordering is derived from SML/NJ [Appel and MacQueen 1991; Appel 1992] (which is in turn based on the Rabbit [Steele 1978] and Orbit [Kranz et al. 1986] compilers) except that our compiler uses types throughout compilation.

The rest of this article proceeds by describing each of the languages and translations in our compiler in detail. We give the syntax and static semantics of each language as well as type-preserving translations between them. The middle calculi (λ^{K}, λ^{C}, λ^{H}, and λ^{A}) have many features in common. Therefore, we only describe λ^{K} in full, and each successive calculus is defined in terms of its differences from the preceding one.

We begin by presenting the compiler's source language, λ^{F}, in Section 3. Section 4 describes the first intermediate language, λ^{K}, and gives a typed CPS translation to it based on Harper and Lillibridge [1993]. The CPS translation fixes the order of evaluation and names intermediate computations. Section 5 presents λ^{C} and gives a typed closure translation to it based on, but considerably simpler than, that of Minamide et al. [1996]. The closure translation makes the construction of functions' environments and closures explicit, thereby rendering all data structures explicit. This is followed by a simple hoisting translation that lifts the (now closed) code to the top level.

Section 6 presents λ^{A}, in which the allocation and initialization of data structures is made explicit, and gives a translation from λ^{H} to λ^{A}. At this point in compilation, the intermediate code is essentially in a lambda-calculus syntax for assembly language (following the ideas of Wand [1992]). Section 7 presents the formal details of our typed assembly language. We show type safety for TAL, and also define a translation from λ^{A} to TAL. Finally, in Section 8 we discuss extensions to TAL to support additional language constructs and optimizations. We also describe our current implementation of TAL and discuss some directions for future investigation.

3. SYSTEM F

The source language for our compiler, λ^{F}, is a variant of System F [Girard 1971; 1972; Reynolds 1974] (the polymorphic λ-calculus) augmented with integers, products, and recursion on terms. The syntax for λ^{F} appears below:

types	τ, σ	$::=$	$\alpha \mid \mathit{int} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \langle\vec{\tau}\rangle$
annotated terms	e	$::=$	u^{τ}
terms	u	$::=$	$x \mid i \mid \mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid$
			$\langle\vec{e}\rangle \mid \pi_i(e) \mid e_1\, p\, e_2 \mid \mathsf{if0}(e_1, e_2, e_3)$
primitives	p	$::=$	$+ \mid - \mid \times$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::=$	$x_1{:}\tau_1, \ldots, x_n{:}\tau_n$

Integers, the only base type, are introduced by integer literals (i), operated on by by arithmetic primitives p, and eliminated by a conditional expression $\mathsf{if0}$. The term $\mathsf{if0}(e_1, e_2, e_3)$ evaluates to e_2 when e_1 evaluates to zero, and otherwise evaluates to e_3. We use the notation $\vec{E}$ to refer to a vector of syntactic objects drawn from the syntactic class E. For example, $\langle\vec{e}\rangle$ is shorthand for a tuple $\langle e_1, \ldots, e_n\rangle$. The elimination form for tuples, $\pi_i(e)$, evaluates to the ith field of the tuple e. Recursive functions are written $\mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e$, where x is the name of the recursive function and may appear free in the body; x_1 is its argument (with type τ_1); and e is its body (with type τ_2). Polymorphic functions are written $\Lambda\alpha.e$, where α is the abstracted type, and e is the body of the polymorphic function. For example, the polymorphic identity function may be written as $\Lambda\alpha.\,\mathsf{fix}\,\mathtt{id}(\mathtt{x}{:}\alpha){:}\alpha.\mathtt{x}$. Instantiation of a polymorphic expression e is written $e\,[\tau]$. As usual, we consider types and expressions that differ only in the names of bound variables to be identical. We write the capture-avoiding substitution of E for X in E' as $E'[E/X]$.

We interpret λ^{F} with a conventional call-by-value operational semantics (which is not presented here). The static semantics (given in Figure 2) 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 second judgment $\Delta \vdash_{\mathrm{F}} \tau$ asserts that type τ is well-formed under type context Δ. In later judgments, we will use $\emptyset$ to denote an empty type or value context.

To simplify the presentation of our translations, we use type-annotated terms (e), which are unannotated terms (u) marked with their types. This decision allows us to present our translations in a simple, syntax-directed fashion, rather than making them dependent on the structure of typing derivations. The typing rules ensure that

$$\frac{}{\Delta \vdash_{\mathrm{F}} \tau}\ (FTV(\tau) \subseteq \Delta) \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{F}} u : \tau}{\Delta;\Gamma \vdash_{\mathrm{F}} u^{\tau} : \tau}$$

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

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

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

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

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

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

Fig. 2. Static Semantics of λ^{F}.

all annotations in a well-formed term are correct. In the interest of clarity, however, we will omit the type annotations in informal discussions and examples.

As a running example, we will consider compiling a term that computes 6 factorial:

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

4. 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 called λ^{K}:

types	τ, σ	$::=$	$\alpha \mid \mathit{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \to \mathit{void} \mid \langle \tau_1, \ldots, \tau_n\rangle$
annotated values	v	$::=$	u^{τ}
values	u	$::=$	$x \mid i \mid \mathsf{fix}\, x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid \langle \vec{v}\rangle$
primitives	p	$::=$	$+ \mid - \mid \times$
declarations	d	$::=$	$x = v \mid x = \pi_i\ v \mid x = v_1\, p\, v_2$
terms	e	$::=$	$\mathsf{let}\ d\ \mathsf{in}\ e$
		$\mid$	$v[\vec{\tau}](\vec{v})$
		$\mid$	$\mathsf{if0}(v, e_1, e_2)$
		$\mid$	$\mathsf{halt}[\tau]v$
type contexts	Δ	$::=$	$\alpha_1, \ldots, \alpha_n$
value contexts	Γ	$::=$	$x_1{:}\tau_1, \ldots, x_n{:}\tau_n$

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 forms a tree containing two subexpressions.

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

$$\frac{}{\Delta \vdash_{\mathrm{K}} \tau}\ (FTV(\tau) \subseteq \Delta) \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{K}} u : \tau}{\Delta;\Gamma \vdash_{\mathrm{K}} u^{\tau} : \tau}$$

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

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

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

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

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

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

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

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

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

Fig. 3. Static Semantics of λ^{K}.

$\forall$ and $\to$ types are also combined. We abbreviate $\forall[\,].(\vec{\tau}) \to \mathit{void}$ as $(\vec{\tau}) \to \mathit{void}$; we abbreviate $\mathsf{fix}\, f[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ as $\lambda[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$, when f does not appear free in e; and we omit empty type argument brackets in both the fix and λ forms and in applications.

In λ^{K}, unlike λ^{F}, functions do not return values, so function calls are just jumps. The function notation "$\to \mathit{void}$" is intended to suggest this fact. If control is to be returned to the caller, the caller must pass the callee a continuation function for it to invoke. Execution is completed by the construct $\mathsf{halt}[\tau]v$, 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, only typing judgments for values state types. The new judgment $\Delta;\Gamma \vdash_{\mathrm{K}} e$ indicates that the term e is well formed under type and value contexts Δ and Γ. Aside from these issues, the static semantics for λ^{K} is standard and appears in Figure 3.

4.1 Translation

The CPS translation that takes λ^{F} to λ^{K} is based on that of Harper and Lillibridge [1993] and appears in Figure 4. The type translation is written $\mathcal{K}[\![\cdot]\!]$. The principal translation for terms, $\mathcal{K}_{\mathrm{exp}}[\![e]\!]$, takes a continuation k, computes the value of e and hands that value to k. A second term translation for full programs, $\mathcal{K}_{\mathrm{prog}}[\![e]\!]$, calls the principal translation with a special top-level continuation that

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

$$
\mathcal{K}_{\text{prog}}[\![u^\tau]\!] \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].\mathsf{halt}[\mathcal{K}[\![\tau]\!]]x^{\mathcal{K}[\![\tau]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau]\!]}
$$

$$
\begin{array}{lcl}
\mathcal{K}_{\text{exp}}[\![y^\tau]\!]k & \stackrel{\text{def}}{=} & k(y^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![i^\tau]\!]k & \stackrel{\text{def}}{=} & k(i^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![(\mathsf{fix}\, x(x_1{:}\tau_1){:}\tau_2.e)^\tau]\!]k & \stackrel{\text{def}}{=} & k((\mathsf{fix}\, x(x_1{:}\mathcal{K}[\![\tau_1]\!], c{:}\mathcal{K}_{\text{cont}}[\![\tau_2]\!]).\mathcal{K}_{\text{exp}}[\![e]\!]c^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}[\![\tau]\!]}) \\
\mathcal{K}_{\text{exp}}[\![(u_1^{\tau_1}\, u_2^{\tau_2})^\tau]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![u_1^{\tau_1}]\!](\lambda x_1{:}\mathcal{K}[\![\tau_1]\!]. \\
 & & \quad \mathcal{K}_{\text{exp}}[\![u_2^{\tau_2}]\!](\lambda x_2{:}\mathcal{K}[\![\tau_2]\!]. \\
 & & \quad\quad x_1^{\mathcal{K}[\![\tau_1]\!]}(x_2^{\mathcal{K}[\![\tau_2]\!]}, k))^{\mathcal{K}_{\text{cont}}[\![\tau_2]\!]})^{\mathcal{K}_{\text{cont}}[\![\tau_1]\!]} \\
\mathcal{K}_{\text{exp}}[\![(\Lambda\alpha.u^\tau)^{\tau'}]\!]k & \stackrel{\text{def}}{=} & k((\lambda[\alpha](c{:}\mathcal{K}_{\text{cont}}[\![\tau]\!]).\mathcal{K}_{\text{exp}}[\![u^\tau]\!]c^{\mathcal{K}_{\text{cont}}[\![\tau]\!]})^{\mathcal{K}[\![\tau']\!]}) \\
\mathcal{K}_{\text{exp}}[\![(u^\tau[\sigma])^{\tau'}]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].x^{\mathcal{K}[\![\tau]\!]}[\mathcal{K}[\![\sigma]\!]](k))^{\mathcal{K}_{\text{cont}}[\![\tau]\!]} \\
\mathcal{K}_{\text{exp}}[\![\langle u_1^{\tau_1},\ldots,u_n^{\tau_n}\rangle^\tau]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![u_1^{\tau_1}]\!](\lambda x_1{:}\mathcal{K}[\![\tau_1]\!].\cdots \\
 & & \quad \mathcal{K}_{\text{exp}}[\![u_n^{\tau_n}]\!](\lambda x_n{:}\mathcal{K}[\![\tau_n]\!]. \\
 & & \quad\quad k(\langle x_1^{\mathcal{K}[\![\tau_1]\!]},\ldots,x_n^{\mathcal{K}[\![\tau_n]\!]}\rangle^{\mathcal{K}[\![\tau]\!]}))^{\mathcal{K}_{\text{cont}}[\![\tau_n]\!]}\ldots)^{\mathcal{K}_{\text{cont}}[\![\tau_1]\!]} \\
\mathcal{K}_{\text{exp}}[\![\pi_i(u^\tau)^{\tau'}]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\lambda x{:}\mathcal{K}[\![\tau]\!].\,\mathsf{let}\ y = \pi_i(x)\ \mathsf{in}\ k(y^{\mathcal{K}[\![\tau']\!]}))^{\mathcal{K}_{\text{cont}}[\![\tau]\!]} \\
\mathcal{K}_{\text{exp}}[\![e_1\ p\ e_2{}^\tau]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![e_1]\!](\lambda x_1{:}int. \\
 & & \quad \mathcal{K}_{\text{exp}}[\![e_2]\!](\lambda x_2{:}int. \\
 & & \quad\quad \mathsf{let}\ y = x_1\ p\ x_2\ \mathsf{in}\ k(y^{int}))^{\mathcal{K}_{\text{cont}}[\![int]\!]})^{\mathcal{K}_{\text{cont}}[\![int]\!]} \\
\mathcal{K}_{\text{exp}}[\![\mathsf{if0}(e_1,e_2,e_3)^\tau]\!]k & \stackrel{\text{def}}{=} & \mathcal{K}_{\text{exp}}[\![e_1]\!](\lambda x{:}int. \\
 & & \quad \mathsf{if0}(x^{int}, \mathcal{K}_{\text{exp}}[\![e_2]\!]k, \mathcal{K}_{\text{exp}}[\![e_3]\!]k))^{\mathcal{K}_{\text{cont}}[\![int]\!]}
\end{array}
$$

Fig. 4. Translation from λ^{F} to λ^{K}.

accepts a final answer and halts. In the translation, the variables c and x are assumed to be fresh in order to avoid variable capture.

An important property of the translation is that it translates well-formed λ^{F} expressions to well-formed λ^{K} expressions:

Lemma (CPS Conversion Type Correctness). *If* $\emptyset;\emptyset \vdash_{\text{F}} e : \tau$ *then* $\emptyset;\emptyset \vdash_{\text{K}} \mathcal{K}_{\text{prog}}[\![e]\!]$.

In this translation, and in those that follow, no particular effort is made to optimize the resulting code. A realistic compiler based on these type systems, such as the one we discuss in Section 8, would integrate optimizations into these translations. For instance, a realistic CPS-converter would eliminate "administrative" redices and optimize tail recursion [Danvy and Filinski 1992].

The factorial example coded in λ^{K} is given below. This code would be obtained

by $\mathcal{K}_{\text{prog}}[\![\cdot]\!]$ in conjunction with two optimizations mentioned above.

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

5. SIMPLIFIED POLYMORPHIC CLOSURE CONVERSION

The second compilation stage is closure conversion, which makes closures explicit, thereby separating program code from data. This is done in two steps. The first and main step, closure conversion proper, rewrites all functions so that they contain no free variables. Any variables that appear free in a function must be taken as additional arguments to that function. Those additional arguments are collected in an *environment* that is paired with the (now closed) code to create a *closure*. Function calls are performed by extracting the code and the environment from the closure, and then calling that code with the environment as an additional argument.

In the second step, hoisting, the code blocks are lifted to the top of the program, achieving the desired separation between code and data. Since those code blocks are closed, hoisting can be done without difficulty. We begin with closure conversion proper; the hoisting step is considered in Section 5.2.

Although the operational explanation of closure conversion is quite simple, there are a number of subtle issues involved in type-checking the resulting code. In the absence of polymorphic functions, our approach to typing closure conversion is based on Minamide et al. [1996], who observe that if two functions with the same type but different free variables (and therefore different environment types) were naively typed after closure conversion, the types of their closures would not be the same. To prevent this, they use *existential types* [Mitchell and Plotkin 1988] to abstract the types of environments, thereby hiding the fact that the closures' environments have different types.

In the presence of polymorphism, functions may have free type variables as well as free value variables, and, just as for free value variables, closure conversion must rewrite functions to take free type variables as additional arguments. Our approach for dealing with this issue diverges from that of Minamide et al., who desire a *type-passing* interpretation of polymorphism in which types are constructed and passed as data at run time. In such a type-passing interpretation, those additional type arguments must be collected in a *type environment*, which is the type-level equivalent of the value environment discussed earlier. Type environments necessitate two complex mechanisms: abstract kinds, to hide the differences between type environments, and translucent types, to ensure that code blocks are called with the correct type environments.

We propose a considerably simpler approach to polymorphic closure conversion. To avoid the complexities of type environments, we adopt a *type-erasure* interpretation of polymorphism as in *The Definition of Standard ML* [Milner et al. 1997]. In a type-erasure interpretation, we need not save the contents of free type variables in a type environment; instead, we substitute them directly into code blocks. Semantically, this amounts to making copies of code blocks in which the relevant

Additional syntactic constructs:

$$\begin{array}{lll} \textit{types} & \tau,\sigma & ::= \ \ldots \mid \exists\alpha.\tau \\ \\ \textit{values} & u & ::= \ \ldots \mid v[\tau] \mid \mathsf{pack}\,[\tau_1, v]\ \mathsf{as}\ \tau_2 \\ \textit{declarations} & d & ::= \ \ldots \mid [\alpha, x] = \mathsf{unpack}\ v \\ \textit{terms} & e & ::= \ \ldots \mid \text{replace } v[\vec{\tau}](\vec{v}) \text{ by } v(\vec{v}) \end{array}$$

The typing rule for $\mathsf{fix}\, x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ is replaced by:

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

The typing rule for $v[\vec{\tau}](\vec{v})$ is replaced by:

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

Additional typing rules:

$$\frac{\Delta \vdash_{\mathrm{C}} \sigma \quad \Delta; \Gamma \vdash_{\mathrm{C}} v : \forall[\alpha, \vec{\beta}].(\vec{\tau}) \to \mathit{void}}{\Delta; \Gamma \vdash_{\mathrm{C}} v[\sigma] : (\forall[\vec{\beta}].(\vec{\tau}[\sigma/\alpha]) \to \mathit{void})} \qquad \frac{\Delta \vdash_{\mathrm{C}} \tau_1 \quad \Delta; \Gamma \vdash_{\mathrm{C}} v : \tau_2[\tau_1/\alpha]}{\Delta; \Gamma \vdash_{\mathrm{C}} \mathsf{pack}\,[\tau_1, v]\ \mathsf{as}\ \exists\alpha.\tau_2 : \exists\alpha.\tau_2}$$

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

Shorthand:

$$\begin{array}{rcl} v[\,] & \stackrel{\text{def}}{=} & v \\ u^{\forall[\alpha,\vec{\beta}].(\vec{\sigma})\to\mathit{void}}[\tau, \vec{\tau}] & \stackrel{\text{def}}{=} & (u^{\forall[\alpha,\vec{\beta}].(\vec{\sigma})\to\mathit{void}}[\tau])^{\forall[\vec{\beta}].(\vec{\sigma}[\tau/\alpha])\to\mathit{void}}[\vec{\tau}] \end{array}$$

Fig. 5. Changes from λ^{K} to λ^{C}.

substitutions have been performed. However, as types will ultimately be erased, these "copies" are represented by the same term at run time, resulting in no run-time cost.

Formally this means, that, in a type-erasure interpretation, we consider the partial application of a function to a type argument to be a value. For example, suppose v has the type $\forall[\vec{\alpha}, \vec{\beta}].(\vec{\tau}) \to \mathit{void}$ where the type variables $\vec{\alpha}$ stand for the function's free type variables, and the type variables $\vec{\beta}$ are the function's ordinary type arguments. If $\vec{\sigma}$ are the contents of those free type variables, then the partial instantiation $v[\vec{\sigma}]$ is considered a value and has type $\forall[\vec{\beta}].(\vec{\tau}[\vec{\sigma}/\vec{\alpha}]) \to \mathit{void}$. This instantiation takes the place of the construction of a type environment.

The work of Minamide et al. arose from the TIL compiler [Morrisett et al. 1996], which uses run-time type information to optimize data layout [Tarditi et al. 1996]. At first, it seems that a type-erasure semantics precludes these optimizations. However, recent work of Crary et al. [1998; 1999] shows how to encode run-time type information in a type-erasure language. Rather than manipulating types directly, programs manipulate values that represent types. Using this device, the type environment can become part of the value environment, and closure conversion may be performed in a similar fashion as described here. These mechanisms can be added to TAL, and the optimizations above can be used in a compiler targeting it.

Figure 5 presents the differences between λ^{C} and λ^{K}. The principal difference is that the body of a function must type check in a context containing only its formal arguments. In other words, code blocks must be closed, as desired. As discussed

above, we also make type instantiation a value form. Finally, we add existential types [Mitchell and Plotkin 1988] to support the typing of closures. Note that in a type-erasure interpretation, the type portion of an existential package (all but v of $\mathsf{pack}\,[\tau, v]\ \mathsf{as}\ \exists\alpha.\tau'$) is erased at run time, and hence the creation of such a package has no run-time cost.

5.1 Translation

The closure conversion algorithm is formalized in Figure 6. The translation for types is denoted by $\mathcal{C}[\![\cdot]\!]$, the only interesting rule of which is the one for function types:

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

The existentially quantified variable β represents the type of the value environment for the closure. The closure itself is a pair consisting of a piece of code instantiated with types for its free type variables, and a value environment. The instantiated code takes as arguments its original type and value arguments, as well as the value environment. Closures are invoked by extracting the code and environment from the closure and then applying the code to the environment and the function's arguments.

The term translation has three parts: one for terms, $\mathcal{C}_{\mathrm{exp}}[\![\cdot]\!]$, one for declarations, $\mathcal{C}_{\mathrm{dec}}[\![\cdot]\!]$, and one for values, $\mathcal{C}_{\mathrm{val}}[\![\cdot]\!]$. For uniformity with other translations, we also provide a whole program translation ($\mathcal{C}_{\mathrm{prog}}[\![\cdot]\!]$), which in this case simply invokes the term translation. To avoid variable capture, the variables z and γ are assumed to be fresh.

Again, we may show that the translation preserves well-formedness of programs:

Lemma (Closure Conversion Type Correctness). *If* $\emptyset;\emptyset\vdash_{\mathrm{K}} e$ *then* $\emptyset;\emptyset\vdash_{\mathrm{C}} \mathcal{C}_{\mathrm{prog}}[\![e]\!]$.

5.2 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. The target of the hoisting translation is the calculus λ^{H}, in which fix is no longer a value form. Instead, code blocks are defined by a *letrec* prefix, which we call a *heap* in anticipation of the heaps of λ^{A} and TAL. This change is made precise in Figure 7.

Programs are translated from λ^{C} to λ^{H} by replacing all fix expressions with fresh variables and binding those variables to the corresponding code expressions in the heap. This translation, denoted by $\mathcal{H}_{\mathrm{prog}}[\![\cdot]\!]$, is straightforward to formalize, so we omit the formalization in the interest of brevity.

Lemma (Hoisting Type Correctness). *If* $\emptyset;\emptyset\vdash_{\mathrm{C}} e$ *then* $\vdash_{\mathrm{H}} \mathcal{H}_{\mathrm{prog}}[\![e]\!]$.

Some examples of closure conversion and hoisting appear in Figures 8 and 9. Figure 8 gives the factorial example after closure conversion, hoisting, and few simplifying optimizations (beta reduction and copy propagation). To illustrate polymorphic closure conversion we consider another example in Figure 9, the polymorphic, higher-order function *twice* that takes a function and composes it with itself. The *twice* function contains two nested functions, *twicef* and *oncef*, each

$$
\begin{array}{lcl}
\mathcal{C}[\![\alpha]\!] & \stackrel{\text{def}}{=} & \alpha \\
\mathcal{C}[\![int]\!] & \stackrel{\text{def}}{=} & int \\
\mathcal{C}[\![\forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n)\to void]\!] & \stackrel{\text{def}}{=} & \exists\beta.\langle\forall[\vec{\alpha}].(\beta,\mathcal{C}[\![\tau_1]\!],\ldots,\mathcal{C}[\![\tau_n]\!])\to 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 \\
 & & \\
\mathcal{C}_{\text{prog}}[\![e]\!] & \stackrel{\text{def}}{=} & \mathcal{C}_{\text{exp}}[\![e]\!] \\
 & & \\
\mathcal{C}_{\text{exp}}[\![\mathsf{let}\ d\ \mathsf{in}\ e]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathcal{C}_{\text{dec}}[\![d]\!]\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!] \\
\mathcal{C}_{\text{exp}}[\![u^\tau[\sigma_1,\ldots,\sigma_m](v_1,\ldots,v_n)]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ [\gamma,z]=\mathsf{unpack}\ \mathcal{C}_{\text{val}}[\![u^\tau]\!]\ \mathsf{in} \\
 & & \quad \mathsf{let}\ z_{code}=\pi_1(z^{\langle\tau_{code},\gamma\rangle})\ \mathsf{in} \\
 & & \quad \mathsf{let}\ z_{env}=\pi_2(z^{\langle\tau_{code},\gamma\rangle})\ \mathsf{in} \\
 & & \quad ({z_{code}}^{\tau_{code}}[\mathcal{C}[\![\sigma_1]\!],\ldots,\mathcal{C}[\![\sigma_m]\!]]) \\
 & & \qquad ({z_{env}}^{\gamma},\mathcal{C}_{\text{val}}[\![v_1]\!],\ldots,\mathcal{C}_{\text{val}}[\![v_n]\!]) \\
 & & \quad \text{where} \\
 & & \quad \mathcal{C}[\![\tau]\!]=\exists\gamma.\langle\tau_{code},\gamma\rangle \\
\mathcal{C}_{\text{exp}}[\![\mathsf{if0}(v,e_1,e_2)]\!] & \stackrel{\text{def}}{=} & \mathsf{if0}(\mathcal{C}_{\text{val}}[\![v]\!],\mathcal{C}_{\text{exp}}[\![e_1]\!],\mathcal{C}_{\text{exp}}[\![e_2]\!]) \\
\mathcal{C}_{\text{exp}}[\![\mathsf{halt}[\tau]v]\!] & \stackrel{\text{def}}{=} & \mathsf{halt}[\mathcal{C}[\![\tau]\!]]\mathcal{C}_{\text{val}}[\![v]\!] \\
 & & \\
\mathcal{C}_{\text{dec}}[\![x=v]\!] & \stackrel{\text{def}}{=} & x=\mathcal{C}_{\text{val}}[\![v]\!] \\
\mathcal{C}_{\text{dec}}[\![x=\pi_i(v)]\!] & \stackrel{\text{def}}{=} & x=\pi_i(\mathcal{C}_{\text{val}}[\![v]\!]) \\
\mathcal{C}_{\text{dec}}[\![x=v_1\ p\ v_2]\!] & \stackrel{\text{def}}{=} & x=\mathcal{C}_{\text{val}}[\![v_1]\!]\ p\ \mathcal{C}_{\text{val}}[\![v_2]\!] \\
 & & \\
\mathcal{C}_{\text{val}}[\![x^\tau]\!] & \stackrel{\text{def}}{=} & x^{\mathcal{C}[\![\tau]\!]} \\
\mathcal{C}_{\text{val}}[\![i^\tau]\!] & \stackrel{\text{def}}{=} & i^{\mathcal{C}[\![\tau]\!]} \\
\mathcal{C}_{\text{val}}[\![\langle v_1,\ldots,v_n\rangle^\tau]\!] & \stackrel{\text{def}}{=} & \langle\mathcal{C}_{\text{val}}[\![v_1]\!],\ldots,\mathcal{C}_{\text{val}}[\![v_n]\!]\rangle^{\mathcal{C}[\![\tau]\!]} \\
\mathcal{C}_{\text{val}}[\![(\mathsf{fix}\,x[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e)^\tau]\!] & \stackrel{\text{def}}{=} & (\mathsf{pack}\ [\tau_{env},\langle v_{code}[\vec{\beta}],v_{env}\rangle^{\langle\tau_{code},\tau_{env}\rangle}]\ \mathsf{as}\ \mathcal{C}[\![\tau]\!])^{\mathcal{C}[\![\tau]\!]} \\
\quad \text{where}\ y_1^{\sigma_1},\ldots,y_m^{\sigma_m} & = & FV(\mathsf{fix}\,x[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e) \\
\quad \vec{\beta} & = & FTV(\mathsf{fix}\,x[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e) \\
\quad \tau_{env} & = & \mathcal{C}[\![\langle\sigma_1,\ldots,\sigma_m\rangle]\!] \\
\quad \tau_{rawcode} & = & \forall[\vec{\beta},\vec{\alpha}].(\tau_{env},\mathcal{C}[\![\tau_1]\!],\ldots,\mathcal{C}[\![\tau_n]\!])\to void \\
\quad \tau_{code} & = & \forall[\vec{\alpha}].(\tau_{env},\mathcal{C}[\![\tau_1]\!],\ldots,\mathcal{C}[\![\tau_n]\!])\to void \\
\quad v_{code} & = & (\mathsf{fix}\,z_{code}[\vec{\beta},\vec{\alpha}](z_{env}{:}\tau_{env},x_1{:}\mathcal{C}[\![\tau_1]\!],\ldots,x_n{:}\mathcal{C}[\![\tau_n]\!]). \\
 & & \qquad \mathsf{let}\ x=\mathsf{pack}\ [\tau_{env},\langle {z_{code}}^{\tau_{rawcode}}[\vec{\beta}],{z_{env}}^{\tau_{env}}\rangle^{\langle\tau_{code},\tau_{env}\rangle}] \\
 & & \qquad\quad \mathsf{as}\ \mathcal{C}[\![\tau]\!]\ \mathsf{in} \\
 & & \qquad \mathsf{let}\ y_1=\pi_1({z_{env}}^{\tau_{env}})\ \mathsf{in} \\
 & & \qquad \vdots \\
 & & \qquad \mathsf{let}\ y_m=\pi_m({z_{env}}^{\tau_{env}})\ \mathsf{in}\ \mathcal{C}_{\text{exp}}[\![e]\!])^{\tau_{rawcode}} \\
\quad v_{env} & = & \langle {y_1}^{\mathcal{C}[\![\sigma_1]\!]},\ldots,{y_m}^{\mathcal{C}[\![\sigma_m]\!]}\rangle^{\tau_{env}}
\end{array}
$$

Fig. 6. Translation from λ^{K} to λ^{C}.

of which contains the free type variable α, and therefore, after closure conversion, α becomes part of the type environment for these functions. Consequently, the type argument to $twice_{code}$ is an ordinary type argument, but the type arguments to the code blocks $twicef_{code}$ and $oncef_{code}$ stand for free type variables and are instantiated appropriately whenever closures are formed from those code blocks.

Syntax changes:

$$\begin{array}{lrcl} \textit{values} & u & ::= & \text{delete } \mathsf{fix}\, x(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ \textit{heap values} & h & ::= & \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \\ \textit{programs} & P & ::= & \mathsf{letrec}\, x_1 \mapsto h_1, \ldots, x_n \mapsto h_n \;\mathsf{in}\; e \end{array}$$

The typing rule for fix is replaced by a heap value rule for code:

$$\frac{\vec{\alpha} \vdash_{\mathrm{H}} \tau_i \quad \vec{\alpha}; (\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \vdash_{\mathrm{H}} e}{\Gamma \vdash_{\mathrm{H}} \mathsf{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \forall[\vec{\alpha}](\tau_1, \ldots, \tau_n) \to \mathit{void}\ \text{hval}} \ (x_1, \ldots, x_n \notin \Gamma)$$

New typing rule:

$$\frac{\emptyset \vdash_{\mathrm{H}} \tau_i \quad x_1{:}\tau_1, \ldots, x_n : \tau_n \vdash_{\mathrm{H}} h_i : \tau_i\ \text{hval} \quad \emptyset; x_1{:}\tau_1, \ldots, x_n : \tau_n \vdash_{\mathrm{H}} e}{\vdash_{\mathrm{H}} \mathsf{letrec}\, x_1 \mapsto h_1, \ldots, x_n \mapsto h_n \;\mathsf{in}\; e} \ (x_i \neq x_j \text{ for } i \neq j)$$

Fig. 7. Changes from λ^{C} to λ^{H}.

```
letrec f_code    ↦ (* main factorial code block *)
                   code[](env:⟨⟩, n:int, k:τ_k).
                     if0(n, (* true branch: continue with 1 *)
                            let [β, k_unpack] = unpack k in
                            let k_code = π1(k_unpack) in
                            let k_env = π2(k_unpack) in
                            k_code(k_env, 1),
                            (* false branch: recurse with n − 1 *)
                            let x = n − 1 in
                            f_code(env, x, pack [⟨int, τ_k⟩, ⟨cont_code, ⟨n, k⟩⟩] as τ_k))
     cont_code   ↦ (* code block for continuation after factorial computation *)
                   code[](env:⟨int, τ_k⟩, y:int).
                     (* open the environment *)
                     let n = π1(env) in
                     let k = π2(env) in
                     (* compute n! into z *)
                     let z = n × y in
                     (* continue with z *)
                     let [β, k_unpack] = unpack k in
                     let k_code = π1(k_unpack) in
                     let k_env = π2(k_unpack) in
                     k_code(k_env, z)
     halt_code   ↦ (* code block for top-level continuation *)
                   code[](env:⟨⟩, n:int). halt[int]n
in
     f_code(⟨⟩, 6, pack [⟨⟩, ⟨halt_code, ⟨⟩⟩] as τ_k)
```

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

Fig. 8. Factorial in λ^{H}.

6. 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. To do so, we eliminate the value form for tuples, and introduce new declaration forms for allocating and initializing tuples, as shown in Figure 10. 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_1 and

λ^{F} source:

$twice = \Lambda\alpha.\, \lambda f{:}\alpha \to \alpha.\, \lambda x{:}\alpha.\, f(fx)$

λ^{K} source:

$$
\begin{array}{l}
twice = \\
\quad \lambda[\alpha](f{:}\tau_f, c{:}(\tau_f) \to void). \\
\qquad \mathsf{let}\ twicef = \\
\qquad\quad \lambda(x{:}\alpha, c'{:}(\alpha) \to void). \\
\qquad\qquad \mathsf{let}\ oncef = \lambda(z{:}\alpha).\, f(z, c')\ \mathsf{in} \\
\qquad\qquad f(x, oncef) \\
\qquad \mathsf{in} \\
\qquad\quad c[\,](twicef) \\
\\
\quad \text{where } \tau_f = (\alpha, (\alpha) \to void) \to void
\end{array}
$$

λ^{H} translation:

$$
\begin{array}{l}
\mathsf{letrec}\ \ twice_{code}[\alpha](env{:}\langle\rangle, f{:}\tau_f, c{:}\exists\rho_3.\langle(\rho_3, \tau_f) \to void, \rho_3\rangle). \\
\qquad\qquad \mathsf{let}\ twicef = \mathsf{pack}\,[\langle\tau_f\rangle, \langle twicef_{code}[\alpha], \langle f\rangle\rangle]\ \mathsf{as}\ \tau_f\ \mathsf{in} \qquad \texttt{(* create closure *)} \\
\qquad\qquad \mathsf{let}\,[\rho_3, c_{unpack}] = \mathsf{unpack}\ c\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ c_{code} = \pi_1(c_{unpack})\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ c_{env} = \pi_2(c_{unpack})\ \mathsf{in} \\
\qquad\qquad\quad c_{code}(c_{env}, twicef) \\
\qquad twicef_{code}[\alpha](env{:}\langle\tau_f\rangle, x{:}\alpha, c'{:}\tau_{\alpha c}). \\
\qquad\qquad \mathsf{let}\ f = \pi_1(env)\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ oncef = \mathsf{pack}\,[\langle\tau_f, \tau_{\alpha c}\rangle, \langle oncef_{code}[\alpha], \langle f, c'\rangle\rangle]\ \mathsf{as}\ \tau_{\alpha c}\ \mathsf{in} \qquad \texttt{(* create closure *)} \\
\qquad\qquad \mathsf{let}\,[\rho_1, f_{unpack}] = \mathsf{unpack}\ f\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ f_{code} = \pi_1(f_{unpack})\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ f_{env} = \pi_2(f_{unpack})\ \mathsf{in} \\
\qquad\qquad\quad f_{code}(f_{env}, x, oncef) \\
\qquad oncef_{code}[\alpha](env : \langle\tau_f, \tau_{\alpha c}\rangle, z : \alpha). \\
\qquad\qquad \mathsf{let}\ f = \pi_1(env)\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ c' = \pi_2(env)\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\,[\rho_1, f_{unpack}] = \mathsf{unpack}\ f\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ f_{code} = \pi_1(f_{unpack})\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ f_{env} = \pi_2(f_{unpack})\ \mathsf{in} \\
\qquad\qquad\quad f_{code}(f_{env}, z, c') \\
\mathsf{in}\ \cdots \\
\\
\text{where}\ \ \tau_f \ = \ \exists\rho_1.\langle(\rho_1, \alpha, \tau_{\alpha c}) \to void, \rho_1\rangle \\
\qquad\quad \tau_{\alpha c} \ = \ \exists\rho_2.\langle(\rho_2, \alpha) \to void, \rho_2\rangle
\end{array}
$$

Fig. 9. Polymorphic example.

v_2 are integers, the pair $\langle v_1, v_2\rangle$ is created as follows (where types have been added for clarity):

$$
\begin{array}{ll}
\mathsf{let} & x_1{:}\langle int^0, int^0\rangle = \mathsf{malloc}[int, int] \\
& x_2{:}\langle int^1, int^0\rangle = x_1[1] \leftarrow v_1 \\
& x\ {:}\langle int^1, int^1\rangle = x_2[2] \leftarrow v_2 \\
& \vdots
\end{array}
$$

The "$x_1 = \mathsf{malloc}[int, int]$" step allocates an uninitialized tuple and binds x_1 to the address of the tuple. The "0" superscripts on the types of the fields indicate that the fields are uninitialized, and hence no projection may be performed on those

Syntax changes:

$$\begin{array}{lrcl}
\textit{types} & \tau, \sigma & ::= & \ldots \mid \text{replace } \langle \vec{\tau} \rangle \text{ by } \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \\
\textit{initialization flags} & \varphi & ::= & 1 \mid 0 \\
\textit{values} & u & ::= & \text{delete } \langle \vec{v} \rangle \\
\textit{declarations} & d & ::= & \ldots \mid x = \mathsf{malloc}[\vec{\tau}] \mid x = v_1[i] \leftarrow v_2 \\
\textit{heap values} & h & ::= & \ldots \mid \langle \vec{v} \rangle
\end{array}$$

The typing rule for projection is replaced by:

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

The typing rule for tuples is replaced by a heap value rule:

$$\frac{\emptyset; \Gamma \vdash v_i : \tau_i}{\Gamma \vdash_{\mathrm{A}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1^1, \ldots, \tau_n^1 \rangle \ \text{hval}}$$

New typing rules:

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

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\mathrm{A}} v_1 : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \qquad \Delta; \Gamma \vdash_{\mathrm{A}} v_2 : \tau_i \\ \Delta; \Gamma, x{:}\langle \tau_1^{\varphi_1}, \ldots, \tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \tau_{i+1}^{\varphi_{i+1}}, \ldots, \tau_n^{\varphi_n} \rangle \vdash_{\mathrm{A}} e\end{array}}{\Delta; \Gamma \vdash_{\mathrm{A}} \mathsf{let}\ x = v_1[i] \leftarrow v_2\ \mathsf{in}\ e} \ (x \notin \Gamma \wedge 1 \leq i \leq n)$$

Shorthand:

$$\begin{array}{rcl}
\mathsf{let}\ \epsilon\ \mathsf{in}\ e & \stackrel{\text{def}}{=} & e \\
\mathsf{let}\ d, \vec{d}\ \mathsf{in}\ e & \stackrel{\text{def}}{=} & \mathsf{let}\ d\ \mathsf{in}\ \mathsf{let}\ \vec{d}\ \mathsf{in}\ e
\end{array}$$

Fig. 10. Changes from λ^{H} to λ^{A}.

fields. The "$x_2 = x_1[1] \leftarrow v_1$" step updates the first field of the tuple with the value v_1 and binds x_2 to the address of the tuple. Note that x_2 is assigned a type where the first field has a "1" superscript, indicating that this field is initialized. Finally, the "$x = x_2[2] \leftarrow v_2$" step initializes the second field of the tuple with v_2 and binds x to the address of the tuple, which is assigned the fully initialized type $\langle \mathit{int}^1, \mathit{int}^1 \rangle$. Hence, both π_1 and π_2 are allowed on x.

The code sequence above need not be atomic; it may be rearranged or interleaved with projections in any well-typed manner. The initialization flags on the types ensure that a field cannot be projected unless it has been initialized. Moreover, a syntactic value restriction ensures there is no unsoundness in the presence of polymorphism. Operationally, the declaration $x[i] \leftarrow v$ is interpreted as an imperative operation, and thus at the end of the sequence, x_1, x_2, and x are all aliases for the same location, even though they have different types. Consequently, the initialization flags do not prevent a field from being initialized twice. It is possible to use monads [Wadler 1990a; Launchbury and Peyton Jones 1995] or linear types [Girard 1987; Wadler 1990b; 1993] to ensure that a tuple is initialized exactly once, but we have avoided these approaches in the interest of a simpler type system. The presence of uninitialized values also raises some garbage collection issues; in Section 8 we discuss how our implementation deals with these issues.

6.1 Translation

The translation of types from λ^{H} to λ^{A} is simple; it amounts to adding initialization flags 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 is formalized in Figure 11 as five translations: full programs ($\mathcal{A}_{\text{prog}}[\![\cdot]\!]$), heap values ($\mathcal{A}_{\text{hval}}[\![\cdot]\!]$), expressions ($\mathcal{A}_{\text{exp}}[\![\cdot]\!]$), declarations ($\mathcal{A}_{\text{dec}}[\![\cdot]\!]$), and values ($\mathcal{A}_{\text{val}}[\![\cdot]\!]$). The focus of the translation is on the last rule, which generalizes the informal translation of tuples given in the previous section. This rule returns the sequence of declarations to allocate and initialize a tuple. Although the other (non-tuple) values are more or less unchanged by the translation, they too must return sequences of declarations needed to construct those values. Such sequences will be empty unless the value in question contains a tuple. Similarly, the declaration translation produces a sequence of declarations. To avoid variable capture, the variable y is assumed to be fresh.

Lemma (Allocation Type Correctness). *If* $\vdash_{\text{H}} P$ *then* $\vdash_{\text{A}} \mathcal{A}_{\text{prog}}[\![P]\!]$.

The factorial example after application of the explicit allocation translation appears in Figure 12.

7. 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.[1] 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 calling convention must be used in code generation, and calling conventions must be made explicit in the types. Hence TAL includes the type $\forall[\vec{\alpha}].\{\texttt{r1}{:}\tau_1, \ldots, \texttt{r}n{:}\tau_n\}$, which is used to describe entry points of code blocks and is the TAL analog of the λ^{A} function type, $\forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \to \mathit{void}$. 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 `r`n must contain values of type τ_1 through τ_n, respectively.

Another distinction between λ^{A} and TAL is, that, while λ^{A} has one mechanism (variables) for identifying values, TAL follows real machines and distinguishes between labels (which may be thought of as pointers) and registers. Registers may

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

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

$$
\begin{array}{l}
\mathcal{A}_{\text{prog}}[\![\mathsf{letrec}\, x_1\mapsto h_1,\ldots,x_n\mapsto h_n\ \mathsf{in}\ e]\!] \\
\qquad\stackrel{\text{def}}{=}\ \mathsf{letrec}\, x_1\mapsto\mathcal{A}_{\text{hval}}[\![h_1]\!],\ldots,x_n\mapsto\mathcal{A}_{\text{hval}}[\![h_n]\!]\ \mathsf{in}\ \mathcal{A}_{\text{exp}}[\![e]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}_{\text{hval}}[\![\mathsf{code}[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e]\!] \\
\qquad\stackrel{\text{def}}{=}\ \mathsf{code}[\vec{\alpha}](x_1{:}\mathcal{A}[\![\tau_1]\!],\ldots,x_n{:}\mathcal{A}[\![\tau_n]\!]).\mathcal{A}_{\text{exp}}[\![e]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{A}_{\text{exp}}[\![\mathsf{let}\ d\ \mathsf{in}\ e]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathcal{A}_{\text{dec}}[\![d]\!]\ \mathsf{in}\ \mathcal{A}_{\text{exp}}[\![e]\!] \\
\mathcal{A}_{\text{exp}}[\![v(v_1,\ldots,v_n)]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ \vec{d},\vec{d}_1,\ldots,\vec{d}_n\ \mathsf{in}\ v'(v'_1,\ldots,v'_n) \\
 & & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \text{ and } \langle\vec{d}_i,v'_i\rangle=\mathcal{A}_{\text{val}}[\![v_i]\!] \\
\mathcal{A}_{\text{exp}}[\![\mathsf{if0}(v,e_1,e_2)]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ \vec{d}\ \mathsf{in}\ \mathsf{if0}(v',\mathcal{A}_{\text{exp}}[\![e_1]\!],\mathcal{A}_{\text{exp}}[\![e_2]\!]) \\
 & & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \\
\mathcal{A}_{\text{exp}}[\![\mathsf{halt}[\tau]v]\!] & \stackrel{\text{def}}{=} & \mathsf{let}\ \vec{d}\ \mathsf{in}\ \mathsf{halt}[\mathcal{A}[\![\tau]\!]]v' \\
 & & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!]
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{A}_{\text{dec}}[\![x=v]\!] & \stackrel{\text{def}}{=} & \vec{d},\ x=v' & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \\
\mathcal{A}_{\text{dec}}[\![x=\pi_i(v)]\!] & \stackrel{\text{def}}{=} & \vec{d},\ x=\pi_i(v') & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \\
\mathcal{A}_{\text{dec}}[\![x=v_1\ p\ v_2]\!] & \stackrel{\text{def}}{=} & \vec{d}_1,\vec{d}_2,\ x=v'_1\ p\ v'_2 & \text{where } \langle\vec{d}_i,v'_i\rangle=\mathcal{A}_{\text{val}}[\![v_i]\!] \\
\mathcal{A}_{\text{dec}}[\![[\alpha,x]=\mathsf{unpack}\ v]\!] & \stackrel{\text{def}}{=} & \vec{d},\ [\alpha,x]=\mathsf{unpack}\ v' & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!]
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{A}_{\text{val}}[\![x^\tau]\!] & \stackrel{\text{def}}{=} & \langle\epsilon,x^{\mathcal{A}[\![\tau]\!]}\rangle \\
\mathcal{A}_{\text{val}}[\![i^\tau]\!] & \stackrel{\text{def}}{=} & \langle\epsilon,i^{\mathcal{A}[\![\tau]\!]}\rangle \\
\mathcal{A}_{\text{val}}[\![(v[\sigma])^\tau]\!] & \stackrel{\text{def}}{=} & \langle\vec{d},(v'[\mathcal{A}[\![\sigma]\!]])^{\mathcal{A}[\![\tau]\!]}\rangle \qquad \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \\
\mathcal{A}_{\text{val}}[\![(\mathsf{pack}\,[\tau,v]\ \mathsf{as}\ \tau')^{\tau''}]\!] & \stackrel{\text{def}}{=} & \langle\vec{d},(\mathsf{pack}\,[\mathcal{A}[\![\tau]\!],v']\ \mathsf{as}\ \mathcal{A}[\![\tau']\!])^{\mathcal{A}[\![\tau'']\!]}\rangle \\
 & & \text{where } \langle\vec{d},v'\rangle=\mathcal{A}_{\text{val}}[\![v]\!] \\
\mathcal{A}_{\text{val}}[\![\langle u_1{}^{\tau_1},\ldots,u_n{}^{\tau_n}\rangle^\tau]\!] & \stackrel{\text{def}}{=} & \langle(\vec{d}_1,\ldots,\vec{d}_n,y_0=\mathsf{malloc}[\mathcal{A}[\![\tau_1]\!],\ldots,\mathcal{A}[\![\tau_n]\!]], \\
 & & \qquad y_1=y_0^{(\tau^{(0)})}[1]\leftarrow v'_1, \\
 & & \qquad\vdots \\
 & & \qquad y_n=y_{n-1}^{(\tau^{(n-1)})}[n]\leftarrow v'_n), \\
 & & y_n^{\mathcal{A}[\![\tau]\!]}\rangle \\
 & & \text{where}\quad \langle\vec{d}_i,v'_i\rangle=\mathcal{A}_{\text{val}}[\![u_i{}^{\tau_i}]\!] \\
 & & \text{and}\quad \tau^{(i)}=\langle\mathcal{A}[\![\tau_1]\!]^1,\ldots,\mathcal{A}[\![\tau_i]\!]^1,\mathcal{A}[\![\tau_{i+1}]\!]^0,\ldots,\mathcal{A}[\![\tau_n]\!]^0\rangle
\end{array}
$$

Fig. 11. Translation from λ^{H} to λ^{A}.

contain only *word* values, which are integers or pointers. As in λ^{A}, tuples and code blocks are *large* values and must be heap allocated. Heap objects are identified by labels, which may reside in registers. 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 7.1), its dynamic semantics (Section 7.2), and its full static semantics (Section 7.3). Finally,

```
letrec f_code  ↦ (* main factorial code block *)
                 code[](env:⟨⟩, n:int, k:τ_k).
                     if0(n, (* true branch: continue with 1 *)
                             let [β, k_unpack] = unpack k in
                             let k_code = π1(k_unpack) in
                             let k_env = π2(k_unpack) in
                             k_code(k_env, 1),
                             (* false branch: recurse with n − 1 *)
                             let x = n − 1 in
                             let y1 = malloc[int, τ_k] in
                             let y2 = y1[1] ← n in
                             let y3 = y2[2] ← k in        (* ⟨n, k⟩ *)
                             let y4 = malloc[(⟨int, τ_k⟩, int) → void, ⟨int, τ_k⟩] in
                             let y5 = y4[1] ← cont_code in
                             let y6 = y5[2] ← y3 in       (* ⟨cont_code, ⟨n, k⟩⟩ *)
                             f_code(env, x, pack [⟨int, τ_k⟩, y6] as τ_k))
      cont_code ↦ (* code block for continuation after factorial computation *)
                 code[](env:⟨int, τ_k⟩, y:int).
                     (* open the environment *)
                     let n = π1(env) in
                     let k = π2(env) in
                     (* continue with n × y *)
                     let z = n × y in
                     let [β, k_unpack] = unpack k in
                     let k_code = π1(k_unpack) in
                     let k_env = π2(k_unpack) in
                     k_code(k_env, z)
      halt_code  ↦ (* code block for top-level continuation *)
                 code[](env:⟨⟩, n:int). halt[int]n
in
      let y7 = malloc[ ] in        (* ⟨⟩ *)
      let y8 = malloc[ ] in        (* ⟨⟩ *)
      let y9 = malloc[(⟨⟩, int) → void, ⟨⟩] in
      let y10 = y9[1] ← halt_code in
      let y11 = y10[2] ← y8 in     (* ⟨halt_code, ⟨⟩⟩ *)
      f_code(y7, 6, pack [⟨⟩, y11] as τ_k)

where τ_k is ∃α.⟨(α, int) → void, α⟩
```

Fig. 12. Factorial in λ^{A}.

we present the translation from λ^{A} to TAL (Section 7.4).

7.1 TAL Syntax

We present the syntax of TAL in Figure 13. A TAL abstract machine state, or *program,* is a triple consisting of a heap (H), a register file (R), and a sequence of instructions (I). The heap is a mapping of labels (ℓ) to heap values (tuples and code blocks). The register file is a mapping of registers (r) to word values. Heaps, register files, and their respective types are not 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; if r does not appear in R, the indicated binding is simply added to R. Similar notation is used for heaps, register file types, and heap types.

$$
\begin{array}{lrcl}
\textit{types} & \tau,\sigma & ::= & \alpha \mid \mathit{int} \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau \\
\textit{initialization flags} & \varphi & ::= & 0 \mid 1 \\
\textit{heap types} & \Psi & ::= & \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\} \\
\textit{register file types} & \Gamma & ::= & \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\} \\
\textit{type contexts} & \Delta & ::= & \alpha_1, \ldots, \alpha_n \\
 & & & \\
\textit{registers} & r & ::= & \mathtt{r1} \mid \mathtt{r2} \mid \mathtt{r3} \mid \cdots \\
\textit{word values} & w & ::= & \ell \mid i \mid ?\tau \mid w[\tau] \mid \mathsf{pack}\,[\tau, w]\ \mathsf{as}\ \tau' \\
\textit{small values} & v & ::= & r \mid w \mid v[\tau] \mid \mathsf{pack}\,[\tau, v]\ \mathsf{as}\ \tau' \\
\textit{heap values} & h & ::= & \langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\vec{\alpha}]\Gamma.I \\
\textit{heaps} & H & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \\
\textit{register files} & R & ::= & \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} \\
 & & & \\
\textit{instructions} & \iota & ::= & \mathtt{add}\,r_d, r_s, v \mid \mathtt{bnz}\,r, v \mid \mathtt{ld}\,r_d, r_s[i] \mid \mathtt{malloc}\,r_d[\vec{\tau}] \mid \mathtt{mov}\,r_d, v \mid \\
 & & & \mathtt{mul}\,r_d, r_s, v \mid \mathtt{st}\,r_d[i], r_s \mid \mathtt{sub}\,r_d, r_s, v \mid \mathtt{unpack}[\alpha, r_d], v \\
\textit{instruction sequences} & I & ::= & \iota; I \mid \mathtt{jmp}\,v \mid \mathtt{halt}[\tau] \\
\textit{programs} & P & ::= & (H, R, I)
\end{array}
$$

Fig. 13. Syntax of TAL.

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. We draw a distinction between word values and small values 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 bound type variables are considered identical, as are expressions that differ only in the order of the fields in a heap, a register file, or a heap or register file type.

7.2 TAL Operational Semantics

The operational semantics of TAL is presented in Figure 14 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 1). 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 $\mathtt{ld}\,r_d, r_s[i]$ instruction loads the ith component (counting from 0) of the tuple bound to the label in r_s, and places this word value in r_d. Conversely, $\mathtt{st}\,r_d[i], r_s$ places the word value in r_s into the ith position of the tuple bound to the label in r_d. The instruction $\mathtt{jmp}\,v$, 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 that code with $\vec{\tau}$. The $\mathtt{bnz}\,r, v$ instruction tests the value in r to see if it is zero. If so, control continues with the next instruction; otherwise control is transferred

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

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

Fig. 14. Operational semantics of TAL.

to v as with the `jmp` instruction.

The instruction $\texttt{unpack}[\alpha, r_d], v$, where v is a value of the form $\mathsf{pack}\, [\tau', v']\, \mathsf{as}\, \tau$, 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 reduces to a simple `mov` instruction.

As in λ^{A}, $\texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n]$ allocates a fresh, uninitialized tuple in the heap and binds r_d to the address of this tuple. Of course, real machines do not provide a primitive `malloc` instruction. Our intention is, that, when 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 reordering and interleaving these underlying instructions with the surrounding TAL code. However, this is the only instruction sequence that is abstract in TAL.

7.3 TAL Static Semantics

The purpose of the static semantics is to specify when programs are well formed and to ensure that well-formed programs do not get stuck. As programs are closed and self-contained, this is expressed by the judgment $\vdash_{\text{TAL}} P$. The well-formedness of a program is defined by the well-formedness of its three components: the heap, the register file, and the instruction stream. Consequently, formation judgments are required for heaps, register files, and instruction sequences, which in turn require judgments for the various sorts of values and types. The static semantics for TAL appears in Figures 16–18 and consists of 13 judgments, summarized in Figure 15

Judgment	Meaning
$\Delta \vdash_{\text{TAL}} \tau$	τ is a well-formed type
$\vdash_{\text{TAL}} \Psi$	Ψ is a well-formed heap type
$\Delta \vdash_{\text{TAL}} \Gamma$	Γ is a well-formed register file type
$\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$	Γ_1 is register file subtype of Γ_2
$\vdash_{\text{TAL}} H : \Psi$	H is a well-formed heap of heap type Ψ
$\Psi \vdash_{\text{TAL}} R : \Gamma$	R is a well-formed register file of register file type Γ
$\Psi \vdash_{\text{TAL}} h : \tau$ hval	h is a well-formed heap value of type τ
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau$ wval	w is a well-formed word value of type τ
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau^{\varphi}$	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)
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \tau$	v is a well-formed small value of type τ
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} I$	I is a well-formed instruction sequence
$\vdash_{\text{TAL}} P$	P is a well-formed program

Fig. 15. TAL static semantic judgments.

and elaborated on below. The large number of judgments is a reflection more of the large number of syntactic classes, than of any inherent semantic complexity. The static semantics is inspired by and follows the conventions of Morrisett and Harper's [1997] $\lambda_{\text{gc}}^{\rightarrow\forall}$.

The first five judgments in Figure 15 specify the well-formedness conditions for types and define subtyping relationships. Four of the five judgments include a type context that indicates which type variables are in scope. Heaps and heap types must be closed, and as a result, their judgments do not include type contexts.

The subtyping judgments are not intended to support subtyping in the usual generality, although they could be expanded to do so. Instead, they are used to allow the forgetting of information. The judgment $\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2$, for instance, makes it possible to forget that a field of a tuple has been initialized. This is used in the subject reduction argument (Lemma 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 about the contents of some registers. This makes it possible to jump to a code block when too many registers are defined.

The rest of the judgments check the well-formedness of the term constructs. Neither heaps nor register files may contain free type variables, so their judgments do not include a type context. Since values in the heap are mutually recursive, the heap's own type is used while typing the heap; to make this sound we separately require that heap types be well formed. The next four judgments are for assigning types to values. In addition to one judgment for each sort of value, there is a judgment 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 sort of value may contain references to the heap; all but heap values may contain free type variables, but only small values may contain registers. Consequently, heap value formation requires only a heap type; word value formation adds a type context; and small value formation adds a type context and a register file type.

The central result is the type safety of TAL programs: well-formed programs

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

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

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

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

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

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

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

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

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

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

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

Fig. 16. Static semantics of TAL (miscellaneous).

never get "stuck." The well-formed terminal configurations of the operational semantics have the form $(H, R\{\mathtt{r1} \mapsto w\}, \mathsf{halt}[\tau])$. All other terminal configurations are considered *stuck*. Type safety follows from the usual Subject Reduction and Progress theorems. Their proofs are given in the Appendix.

Theorem (Subject Reduction). *If* $\vdash_{\text{TAL}} P$ *and* $P \longmapsto P'$, *then* $\vdash_{\text{TAL}} P'$.

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

7.4 Code Generation

The translation from λ^{A} to TAL appears in Figures 19 and 20. 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:

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

The most interesting part of the term translation is the translation of declara-

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

$$\text{(tuple)}\ \frac{\Psi;\emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}}{\Psi \vdash_{\text{TAL}} \langle w_1,\ldots,w_n\rangle : \langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle \text{ hval}}$$

$$\text{(code)}\ \frac{\vec{\alpha} \vdash_{\text{TAL}} \Gamma \quad \Psi;\vec{\alpha};\Gamma \vdash_{\text{TAL}} I}{\Psi \vdash_{\text{TAL}} \mathtt{code}[\vec{\alpha}]\Gamma.I : \forall[\vec{\alpha}].\Gamma \text{ hval}}$$

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

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

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

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

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

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

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

Fig. 17. Static semantics of TAL (values).

tions. Informally, declarations 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 `arith` is the appropriate arithmetic instruction.

— $[\alpha, x] = \mathsf{unpack}\ v$ is mapped to $\mathtt{unpack}[\alpha, r_x], v$.

— $x = \mathsf{malloc}[\vec{\tau}]$ is mapped to $\mathtt{malloc}\, r_x[\vec{\tau}]$.

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

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

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

$$\begin{array}{l}\mathtt{mov}\, r_{\text{temp}}, v\,;\, \mathtt{mov}\, r_{\text{temp}_1}, v_1\,;\,\ldots\,;\, \mathtt{mov}\, r_{\text{temp}_n}, v_n\,;\\ \mathtt{mov}\, \mathtt{r1}, r_{\text{temp}_1}\,;\,\ldots\,;\, \mathtt{mov}\, \mathtt{r}n, r_{\text{temp}_n}\,;\, \mathtt{jmp}\, r_{\text{temp}}\end{array}.$$

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

— $\mathsf{if0}(v, e_1, e_2)$ is mapped to the sequence

$$\mathtt{mov}\, r_{\text{temp}}, v\,;\, \mathtt{bnz}\, r_{\text{temp}}, \ell[\vec{\alpha}]\,;\, I_1$$

$\boxed{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} I}$

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

$$\text{(s-bnz)}\ \frac{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} r : int \qquad \Psi;\Delta;\Gamma \vdash_{\text{TAL}} v : \forall[].\Gamma' \qquad \Delta \vdash_{\text{TAL}} \Gamma \le \Gamma' \qquad \Psi;\Delta;\Gamma \vdash_{\text{TAL}} I}{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} \texttt{bnz}\, r, v; I}$$

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

$$\text{(s-malloc)}\ \frac{\Delta \vdash_{\text{TAL}} \tau_i \quad \Psi;\Delta;\Gamma\{r_d{:}\langle \tau_1^0, \ldots, \tau_n^0 \rangle\} \vdash_{\text{TAL}} I}{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} \texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n]; I}$$

$$\text{(s-mov)}\ \frac{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} v : \tau \quad \Psi;\Delta;\Gamma\{r_d : \tau\} \vdash_{\text{TAL}} I}{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} \texttt{mov}\, r_d, v; I}$$

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

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

$$\text{(s-jmp)}\ \frac{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} v : \forall[].\Gamma' \quad \Delta \vdash_{\text{TAL}} \Gamma \le \Gamma'}{\Psi;\Delta;\Gamma \vdash_{\text{TAL}} \texttt{jmp}\, v}$$

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

Fig. 18. Static semantics of TAL (instructions).

where ℓ is bound in the heap to $\texttt{code}[\vec{\alpha}]\Gamma.I_2$; the translation of e_i is I_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.

— $\mathsf{halt}[\tau]v$ is mapped to the sequence $\texttt{mov r1}, v\,;\, \texttt{halt}[\tau]$

The formal translation uses a mapping γ that tracks what label or register is used to implement each term variable. As discussed above, if0 terms are implemented by a conditional branch to a new code block representing the else-clause. These new code blocks must be heap allocated, so translations of terms (and translations of heap values, which can contain terms) must return an addition to the heap as well as an instruction sequence. Also, the translation of terms must track the current type context Δ and register file type Γ in order to place that information into new code blocks resulting from if0 terms.

Lemma (Code Generation Type Correctness). *If* $\vdash_{\text{A}} P$ *then* $\vdash_{\text{TAL}} \mathcal{T}_{\text{prog}}[\![P]\!]$.

By composing the five translations (CPS conversion, closure conversion, hoisting, allocation, and code generation), we obtain a translation from λ^{F} to TAL. The type correctness of the composite translation follows from the preceding type correctness

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

$$
\begin{array}{lcl}
\mathcal{T}_{\text{prog}}[\![\mathsf{letrec}\, x_1\mapsto h_1,\ldots,x_n\mapsto h_n\ \mathsf{in}\ e]\!] & \stackrel{\text{def}}{=} & (H,\emptyset,I) \\
& & \quad \text{where } \gamma = \{x_1\mapsto \ell_1,\ldots,x_n\mapsto \ell_n\} \\
& & \qquad \langle H_i, h_i'\rangle = \mathcal{T}^{\gamma}_{\text{hval}}[\![h_i]\!] \\
& & \qquad \langle H_{exp}, I\rangle = \mathcal{T}^{\gamma,\emptyset,\emptyset}_{\text{exp}}[\![e]\!] \\
& & \qquad H_{root} = \{\ell_1\mapsto h_1',\ldots,\ell_n\mapsto h_n'\} \\
& & \qquad H = H_{root}H_1\cdots H_nH_{exp} \\
& & \qquad \ell_i \text{ are distinct}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}^{\gamma}_{\text{hval}}[\![\mathsf{code}[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e]\!] & \stackrel{\text{def}}{=} & \langle H, \mathtt{code}[\vec{\alpha}]\Gamma.I\rangle \\
& & \quad \text{where } \Gamma = \{\mathtt{r1}{:}\mathcal{T}[\![\tau_1]\!],\ldots,\mathtt{r}n{:}\mathcal{T}[\![\tau_n]\!]\} \\
& & \qquad \gamma' = \gamma\{x_1\mapsto \mathtt{r1},\ldots,x_n\mapsto \mathtt{r}n\} \\
& & \qquad \langle H, I\rangle = \mathcal{T}^{\gamma',\vec{\alpha},\Gamma}_{\text{exp}}[\![e]\!] \\
\mathcal{T}^{\gamma}_{\text{hval}}[\![\langle v_1,\ldots,v_n\rangle]\!] & \stackrel{\text{def}}{=} & \langle\emptyset, \langle \mathcal{T}^{\gamma}_{\text{val}}[\![v_1]\!],\ldots,\mathcal{T}^{\gamma}_{\text{val}}[\![v_n]\!]\rangle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}^{\gamma}_{\text{val}}[\![x^{\tau}]\!] & \stackrel{\text{def}}{=} & \gamma(x) \\
\mathcal{T}^{\gamma}_{\text{val}}[\![i^{\tau}]\!] & \stackrel{\text{def}}{=} & i \\
\mathcal{T}^{\gamma}_{\text{val}}[\![(v[\sigma])^{\tau}]\!] & \stackrel{\text{def}}{=} & \mathcal{T}^{\gamma}_{\text{val}}[\![v]\!][\mathcal{T}[\![\sigma]\!]] \\
\mathcal{T}^{\gamma}_{\text{val}}[\![(\mathsf{pack}\,[\tau_1,v]\ \mathsf{as}\ \tau_2)^{\tau}]\!] & \stackrel{\text{def}}{=} & \mathsf{pack}\,[\mathcal{T}[\![\tau_1]\!],\mathcal{T}^{\gamma}_{\text{val}}[\![v]\!]]\ \mathsf{as}\ \mathcal{T}[\![\tau_2]\!]
\end{array}
$$

Fig. 19. Translation from λ^{A} to TAL (except expressions).

lemmas.

Corollary (Compiler Type Correctness). *If* $\vdash_{\text{F}} e:\tau$ *then* $\vdash_{\text{TAL}} (\mathcal{T}_{\text{prog}}\circ\mathcal{A}_{\text{prog}}\circ\mathcal{H}_{\text{prog}}\circ\mathcal{C}_{\text{prog}}\circ\mathcal{K}_{\text{prog}})[\![e]\!]$.

7.5 TAL Factorial

The factorial computation translated into TAL appears in Figure 21. 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. Were the efficiency of this version unsatisfactory, a more efficient version could be obtained by improving the λ^{F} source program (e.g., by using tail recursion), by optimizing intermediate language programs (e.g., by eliminating unnecessary closure creation), or by hand-coding a highly optimized version directly in TAL, such as the one in Figure 1.

8. EXTENSIONS AND PRACTICE

The previous sections provide a theoretical basis for compiling high-level languages to typed assembly language. In this section we discuss some issues that arise when putting this technology into practice.

$$
\begin{array}{lcl}
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ x = u^{\tau}\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{mov}\, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![u^{\tau}]\!]; I)\rangle \\
& & \text{where } \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta,\Gamma\{r:\mathcal{T}[\![\tau]\!]\}}_{\mathrm{exp}}[\![e]\!] \\
& & \qquad r \text{ is fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ x = \pi_i(u^{\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle})\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{mov}\, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![u^{\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}]\!]; \\
& & \qquad \mathtt{ld}\, r, r[i-1]; I)\rangle \\
& & \text{where } \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta,\Gamma\{r:\mathcal{T}[\![\tau_i]\!]\}}_{\mathrm{exp}}[\![e]\!] \\
& & \qquad r \text{ is fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ x = v_1\ p\ v_2\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{mov}\, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v_1]\!]; \\
& & \qquad \mathtt{arith}_p\, r, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v_2]\!]; I)\rangle \\
& & \text{where } \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta,\Gamma\{r:int\}}_{\mathrm{exp}}[\![e]\!] \\
& & \qquad \mathtt{arith}_{+} \ = \ \mathtt{add} \\
& & \qquad \mathtt{arith}_{-} \ = \ \mathtt{sub} \\
& & \qquad \mathtt{arith}_{\times} \ = \ \mathtt{mul} \\
& & \qquad r \text{ is fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ [\alpha, x] = \mathsf{unpack}\ u^{\exists\alpha.\tau}\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{unpack}[\alpha, r], \mathcal{T}^{\gamma}_{\mathrm{val}}[\![u^{\exists\alpha.\tau}]\!]; I)\rangle \\
& & \text{where } \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta\{\alpha\},\Gamma\{r:\mathcal{T}[\![\tau]\!]\}}_{\mathrm{exp}}[\![e]\!] \\
& & \qquad \alpha, r \text{ are fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ x = \mathsf{malloc}[\tau_1,\ldots,\tau_n]\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{malloc}\, r[\mathcal{T}[\![\tau_1]\!],\ldots,\mathcal{T}[\![\tau_n]\!]]; I)\rangle \\
& & \text{where} \\
& & \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta,\Gamma\{r:\langle \mathcal{T}[\![\tau_1]\!]^0,\ldots,\mathcal{T}[\![\tau_n]\!]^0\rangle\}}_{\mathrm{exp}}[\![e]\!] \\
& & r \text{ is fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{let}\ x = u^{\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}[i] \leftarrow v\ \mathsf{in}\ e]\!] & \stackrel{\mathrm{def}}{=} & \langle H, (\mathtt{mov}\, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![u^{\langle \tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}]\!]; \\
& & \qquad \mathtt{mov}\, r', \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v]\!]; \\
& & \qquad \mathtt{st}\, r[i-1], r'; I)\rangle \\
& & \text{where} \\
& & \langle H, I\rangle \ = \ \mathcal{T}^{\gamma\{x\mapsto r\},\Delta,\Gamma'}_{\mathrm{exp}}[\![e]\!] \\
& & \Gamma' \ = \ \Gamma\{r:\mathcal{T}[\![\langle \tau_1^{\varphi_1},\ldots,\tau_{i-1}^{\varphi_{i-1}}, \tau_i^1, \\
& & \qquad\qquad \tau_{i+1}^{\varphi_{i+1}},\ldots,\tau_n^{\varphi_n}\rangle]\!]\} \\
& & r, r' \text{ are fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![v(v_1,\ldots,v_n)]\!] & \stackrel{\mathrm{def}}{=} & \langle \emptyset, (\mathtt{mov}\, r'_0, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v]\!]; \\
& & \quad \mathtt{mov}\, r'_1, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v_1]\!]; \ldots; \\
& & \quad \mathtt{mov}\, r'_n, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v_n]\!]; \\
& & \quad \mathtt{mov}\, \mathtt{r1}, r'_1; \ldots \\
& & \quad \mathtt{mov}\, \mathtt{r}n, r'_n; \\
& & \quad \mathtt{jmp}\, r'_0)\rangle \\
& & \text{where } r'_i \text{ are fresh and } r'_i \notin \{\mathtt{r1},\ldots,\mathtt{r}n\} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{if0}(v, e_1, e_2)]\!] & \stackrel{\mathrm{def}}{=} & \langle H_1 H_2\{\ell \mapsto h\}, (\mathtt{mov}\, r, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v]\!]; \\
& & \qquad\qquad \mathtt{bnz}\, r, \ell[\Delta]; I_1)\rangle \\
& & \text{where } \langle H_i, I_i\rangle \ = \ \mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![e_i]\!] \\
& & \qquad h \ = \ \mathtt{code}[\Delta]\Gamma.I_2 \\
& & \qquad \ell, r \text{ are fresh} \\
\mathcal{T}^{\gamma,\Delta,\Gamma}_{\mathrm{exp}}[\![\mathsf{halt}[\tau] v]\!] & \stackrel{\mathrm{def}}{=} & \langle \emptyset, (\mathtt{mov}\, \mathtt{r1}, \mathcal{T}^{\gamma}_{\mathrm{val}}[\![v]\!]; \\
& & \quad \mathtt{halt}[\mathcal{T}[\![\tau]\!]])\rangle
\end{array}
$$

Fig. 20. Translation of expressions from λ^{A} to TAL.

$(H, \{\}, I)$ 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_fact                      % call fact(env, n − 1, l_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 I =

```
        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 halt code
        st r3[1],r2                     % store halt environment ⟨⟩
        mov r2,6                        % load argument (6)
        mov r3,pack [⟨⟩,r3] as τ_k      % abstract the type of the environment
        jmp l_fact                      % call fact(⟨⟩, 6, l_halt)
```

and τ_k = $\exists\alpha.\langle\forall[].\{\texttt{r1}{:}\alpha,\texttt{r2}{:}int\}^1, \alpha^1\rangle$

Fig. 21. Typed assembly code for factorial.

8.1 Implementation

In order to investigate the applicability of our approach to realistic modern programming languages, we have implemented a version of TAL for the Intel 32-bit Architecture (IA32) [Intel 1996], and have compilers for a number of different source languages including a safe C-like language [Morrisett et al. 1999] and a higher-order, dynamically typed language (a subset of Scheme). Compilers for Standard ML and a small object-oriented language are also in development.

TALx86, the target language for our compilers, is a strongly typed version of the IA32 assembly language. Our type checker verifies standard MASM assembly code in which type annotations and complex instructions such as malloc are assembly language macros. The MASM assembler processes this annotated code as it would any other assembly code, expanding the instruction macros as their definitions dictate and erasing types as it translates the assembly code into concrete machine instructions. We have also implemented our own assembler and are extending it to produced typed object files. Such typed object files include code/data segments and a type segment similar to Necula and Lee's code and proof segments in their PCC binaries [Necula 1997]. We have implemented a tool that reads TALx86 files, type checks them, and assembles them into object files or invokes MASM.

The TALx86 type system is based on the type system described in this article but enriched with a variety of standard constructs including floats, sums, arrays, references, recursive types, and higher-order type constructors. In order to deal with floating-point values correctly in the presence of polymorphism, we use a kind structure that distinguishes types of objects that are 32 bits wide (such as pointers and integers) from types of objects possibly of other sizes. If a polymorphic type variable α has the 32-bit kind, then objects of type α can be passed in general-purpose registers, and tuple offsets may be computable for fields appearing after a field of type α. If, on the other hand, α has the more general kind "Type," the type checker cannot tell how large objects of type α are, and these operations are disallowed.

To support separate compilation and type-safe linking, we have also augmented our typed assembly language with a module system [Glew and Morrisett 1999]. A TAL interface file specifies the types and terms that a TAL implementation file defines. The types may either be opaque to support information hiding and modularity, or transparent to allow information sharing and admit some cross-module optimizations. Our system performs a series of checks to ensure that implementations are well formed and that their interfaces are compatible and complete. Once interface compatibility and completeness have been verified, we assemble the code as described above and invoke a standard untyped linker.

To deal with the creation and examination of exception packets TALx86 includes a type-tagging mechanism [Glew 1999]. The basic idea is that freshly created heap pointers may be associated with a type, and that a tag for an unknown type α can be tested against a tag for a known type τ. If the test succeeds, the unknown type is refined to the known type. Using these tags, we implement an exception packet as an existentially packaged pair containing a tag of the hidden type (serving as the exception name) and a value of that type.

TALx86 also contains some support for compiling objects. The type system

has a more general notion of subtyping than this article that includes the usual contravariant rule for code, right-extension and depth subtyping for tuples, and a variance mechanism for arrays and references. Furthermore, the type-tagging mechanism can also be used to tag objects with their class. This mechanism provides us with a way to implement down-casting. However, while TALx86 contains the necessary constructs to admit some simple object encodings, we are still developing the theoretical and practical tools we need to admit efficient object encodings.

Although this article describes a CPS-based compiler, all of the compilers we have built use a stack-based compilation model. Both standard continuations and exceptions are allocated on the stack, and the stack is also used to store spilled temporary values. The details of our stack typing discipline are discussed in Morrisett et al. [1998]. 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. The combination of stack types and our register typing discipline allows us to model almost any standard calling convention. Arguments, results, and continuations (or return addresses) may be placed in registers, on the stack, or both, and the implementer may specialize conventions for known functions for better register allocation.

Real machines also have a finite amount of heap space. It is straightforward to link TALx86 to a conservative garbage collector [Boehm and Weiser 1988] and reclaim dead heap values. It is worth noting that our use of conservative collection is sound. Conservative collectors make assumptions about the way pointers can be used in programs that untyped assembly language programs can violate. However, the TAL type system guarantees that these assumptions do hold because labels are a strong abstraction; labels cannot be synthesized out of integers, and operations like pointer arithmetic are disallowed. TAL guarantees that other GC constraints hold because values that disobey the constraints cannot be constructed. For example, TAL disallows pointers into the middle of objects and ensures alignment constraints are obeyed.

Support for an accurate collector would require introducing tags so that we may distinguish pointers from integers, or else require a type-passing interpretation [Tolmach 1994; Morrisett and Harper 1997]. In the former case, we would have to ensure that all values are properly tagged and fully initialized at every potential garbage collection site. We believe that it is feasible to devise a type system to ensure these constraints, but we have not seriously investigated this option.

8.2 Future Work

There remain several directions in which TAL could be improved. One of the most important is to make array manipulation efficient. In order to ensure safe access to arrays, TALx86 uses complex instructions (which expand into three real instructions) that perform subscript and update operations after checking that the array offset is in bounds. These bounds checks cannot be eliminated in the current TAL framework. As a result, array-intensive applications will suffer the same performance penalties that they do in Java just-in-time compilers where there is no time to perform analyses to eliminate the checks. However, Xi [1999] and Xi

and Pfenning [1998; 1999] have shown how to eliminate array bounds checks effectively using dependent types. TALx86 can be extended with similar constructs. We have implemented a prototype version in which these checks can be eliminated, but we have not yet added compiler support for generating code with unchecked array subscripts.

Another important direction is to augment our compiler with data-layout optimizations such as those used in the TIL compiler [Tarditi et al. 1996]. As discussed in Section 5, such optimizations require programs to have the ability to analyze types at run-time, which is not directly compatible with the type-erasure interpretation adopted here. To make such optimizations permissible, we are augmenting the TALx86 language so that TAL programs can construct values that represent types and analyze those values when necessary, following the work of Crary et al. [1998; 1999].

Although we believe our translations are operationally correct, we are still searching for robust proofs of correctness. Similar CPS [Danvy and Filinski 1992] and closure conversion [Minamide et al. 1996] translations have already been proven correct, but these results do not easily extend to languages that include recursive types or objects. The principal problem is that these arguments are based on inductively defined, type-indexed logical relations between source and target language terms. Extending this framework so that it supports recursive types or objects is difficult because the relations can no longer be constructed in a simple inductive fashion. A syntactic proof of correctness seems possible (we have constructed such arguments for the CPS and closure conversion phases), but the proofs are overly specific to the details of the translation. Moreover, security-conscious applications might require translations that are not only operationally correct but also fully abstract. We hope further research on the proof theory of similar systems will eventually allow us to construct these arguments.

Other avenues of future research include extension of our type system to the same level of generality as PCC through the use of a dependent type theory, an investigation of the support required to compile Java classes and objects into TAL, and an exploration of type-theoretic mechanisms for performing explicit memory management.

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, although the type system may preclude some advanced optimizations, many common compiler-introduced, low-level optimizations, such as register allocation, instruction selection, or instruction scheduling, are largely unaffected. Furthermore, programmers concerned with efficiency can hand-code routines in assembly, as long as the resulting code passes the type checker. Consequently, TAL provides a foundation for high-performance computing in environments where untrusted code must be checked for safety before being executed.

APPENDIX

LEMMA (CONTEXT STRENGTHENING). *If* $\Delta \subseteq \Delta'$ *then*

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

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

LEMMA (SUBTYPING REGULARITY). *If* $\Delta \vdash_{\text{TAL}} \tau \leq \tau'$ *then* $\Delta \vdash_{\text{TAL}} \tau$ *and* $\Delta \vdash_{\text{TAL}} \tau'$.

PROOF. By induction on derivations. □

LEMMA (HEAP EXTENSION). *If* $\vdash_{\text{TAL}} H : \Psi$, $\emptyset \vdash_{\text{TAL}} \tau$, $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \tau$ hval, *and* $\ell \notin H$ *then*

(1) $\vdash_{\text{TAL}} \Psi\{\ell : \tau\}$
(2) $\vdash_{\text{TAL}} H\{\ell \mapsto h\} : \Psi\{\ell : \tau\}$
(3) If $\Psi \vdash_{\text{TAL}} R : \Gamma$ *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} R : \Gamma$
(4) If $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} I$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} I$
(5) If $\Psi \vdash_{\text{TAL}} h : \sigma$ hval *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \sigma$ hval
(6) If $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$
(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 (htype). Part 2 follows from parts 1 and 5. Parts 3–8 are by induction on derivations. □

LEMMA (HEAP UPDATE). *If* $\vdash_{\text{TAL}} H : \Psi$, $\emptyset \vdash_{\text{TAL}} \tau \leq \Psi(\ell)$, *and* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \tau$ *then*

(1) $\vdash_{\text{TAL}} \Psi\{\ell : \tau\}$
(2) $\vdash_{\text{TAL}} H\{\ell \mapsto h\} : \Psi\{\ell : \tau\}$
(3) If $\Psi \vdash_{\text{TAL}} R : \Gamma$ *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} R : \Gamma$
(4) If $\Psi; \Delta; \Gamma \vdash_{\text{TAL}} I$ *then* $\Psi\{\ell : \tau\}; \Delta; \Gamma \vdash_{\text{TAL}} I$
(5) If $\Psi \vdash_{\text{TAL}} h : \sigma$ hval *then* $\Psi\{\ell : \tau\} \vdash_{\text{TAL}} h : \sigma$ hval
(6) If $\Psi; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$ *then* $\Psi\{\ell : \tau\}; \Delta \vdash_{\text{TAL}} w : \sigma^{\varphi}$
(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 (htype) 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 with the following:

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

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

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

□

LEMMA (REGISTER FILE UPDATE). *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval *then* $\Psi \vdash_{\text{TAL}} R\{r \mapsto w\} : \Gamma\{r : \tau\}$.

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 $r_i \neq r$, we have $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i$ wval, and by hypothesis we have $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau$ wval; so by rule (reg) $\Psi \vdash_{\text{TAL}} R\{r \mapsto w\} : \Gamma\{r \mapsto \tau\}$. □

LEMMA (CANONICAL HEAP FORMS). *If* $\Psi \vdash_{\text{TAL}} h : \tau$ hval *then*

(1) *If* $\tau = \forall[\vec{\alpha}].\Gamma$ *then*
   (a) $h = \mathsf{code}[\vec{\alpha}]\Gamma.I$
   (b) $\Psi; \vec{\alpha}; \Gamma \vdash_{\text{TAL}} I$.
(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) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$.

PROOF. By inspection. □

LEMMA (CANONICAL WORD FORMS). *If* $\vdash_{\text{TAL}} H : \Psi$ *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) $H(\ell) = \mathsf{code}[\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m]\Gamma'.I$
   (c) $\Gamma = \Gamma'[\vec{\sigma}/\vec{\alpha}]$
   (d) $\Psi; \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m; \Gamma' \vdash_{\text{TAL}} I$.
(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) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$.
(4) *If* $\tau = \exists\alpha.\tau$ *then* $w = \mathsf{pack}\ [\tau', w']\ \mathsf{as}\ \exists\alpha.\tau$ *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 we have $w = \ell$, $\Psi(\ell) = \tau'$, and $\emptyset \vdash_{\text{TAL}} \tau' \leq \forall[\vec{\beta}].\Gamma$. Inspection of the subtyping rules then reveals that $\tau' = \forall[\vec{\beta}].\Gamma$. Since $\vdash_{\text{TAL}} H : \Psi$, 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 $\emptyset \vdash_{\text{TAL}} \tau' \leq \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle$. 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 $\vdash_{\text{TAL}} H : \Psi$, 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 $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi'_i}$ by Canonical Heap Forms. It remains to show that $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$ for all $0 \leq i \leq n-1$. Suppose $\varphi'_i = 1$ and $\varphi_i = 0$ (otherwise the conclusion is immediate). Then $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^1$ is shown by the (init) rule, which also permits the deduction of $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^0$.

(4) By inspection. □

LEMMA ($\hat{R}$ TYPING). *If* $\Psi \vdash_{\text{TAL}} R : \Gamma$ *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. Consider the cases for v:

Case $v = w$. Immediate.

Case $v = r$. 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 $\hat{R}(r) = R(r)$.

Case $v = v'[\sigma]$. 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 $\hat{R}(v'[\sigma]) = \hat{R}(v')[\sigma]$.

Case $v = \mathsf{pack}\,[\sigma, v']\,\mathsf{as}\,\exists\alpha.\tau'$. 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 $\Psi; \emptyset \vdash_{\text{TAL}} \mathsf{pack}\,[\sigma, \hat{R}(v')]\,\mathsf{as}\,\exists\alpha.\tau' : \exists\alpha.\tau'$ wval. The result follows, since $\hat{R}(\mathsf{pack}\,[\sigma, v']\,\mathsf{as}\,\exists\alpha.\tau') = \mathsf{pack}\,[\sigma, \hat{R}(v')]\,\mathsf{as}\,\exists\alpha.\tau'$. □

LEMMA (CANONICAL FORMS). *If* $\vdash_{\text{TAL}} H : \Psi$ hval, $\Psi \vdash_{\text{TAL}} R : \Gamma$, *and* $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau$ *then*

(1) *If* $\tau = int$ *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) $H(\ell) = \mathsf{code}[\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m]\Gamma'.I$
   (c) $\Gamma = \Gamma'[\vec{\sigma}/\vec{\alpha}]$
   (d) $\Psi; \alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m; \Gamma' \vdash_{\text{TAL}} I$.
(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) $\Psi; \emptyset \vdash_{\text{TAL}} w_i : \tau_i^{\varphi_i}$.
(4) *If* $\tau = \exists\alpha.\tau$ *then* $\hat{R}(v) = \mathsf{pack}\,[\tau', w]\,\mathsf{as}\,\exists\alpha.\tau$ *and* $\Psi; \emptyset \vdash_{\text{TAL}} w : \tau[\tau'/\alpha]$ wval.

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

LEMMA (TYPE SUBSTITUTION). *If* $\vec{\beta} \vdash_{\text{TAL}} \tau_i$ *then*

(1) *If* $\Psi; \vec{\alpha}, \vec{\beta}; \Gamma \vdash_{\text{TAL}} I$ *then* $\Psi; \vec{\beta}; \Gamma[\vec{\tau}/\vec{\alpha}] \vdash_{\text{TAL}} I[\vec{\tau}/\vec{\alpha}]$
(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* $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$ *then* $\vec{\beta} \vdash_{\text{TAL}} \Gamma_1[\vec{\tau}/\vec{\alpha}] \leq \Gamma_2[\vec{\tau}/\vec{\alpha}]$
(5) *If* $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \tau_1 \leq \tau_2$ *then* $\vec{\beta} \vdash_{\text{TAL}} \tau_1[\vec{\tau}/\vec{\alpha}] \leq \tau_2[\vec{\tau}/\vec{\alpha}]$
(6) *If* $\vec{\alpha}, \vec{\beta} \vdash_{\text{TAL}} \tau$ *then* $\vec{\beta} \vdash_{\text{TAL}} \tau[\vec{\tau}/\vec{\alpha}]$.

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

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

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

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

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

LEMMA (REGISTER FILE WEAKENING). *If* $\Delta \vdash_{\text{TAL}} \Gamma_1 \leq \Gamma_2$ *and* $\Psi; \Delta \vdash_{\text{TAL}} R : \Gamma_1$ *then* $\Psi; \Delta \vdash_{\text{TAL}} R : \Gamma_2$.

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

THEOREM (SUBJECT REDUCTION). *If* $\vdash_{\text{TAL}} P$ *and* $P \longmapsto P'$ *then* $\vdash_{\text{TAL}} P'$.

PROOF. P has the form $(H, R, \iota; I)$ 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

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

By the operational semantics, $P' = (H, R, I[\vec{\sigma}/\vec{\alpha}])$ where $\hat{R}(v) = \ell[\vec{\sigma}]$ and $H(\ell) = \texttt{code}[\vec{\alpha}]\Gamma''.I$. Then
(1) $\vdash_{\text{TAL}} H : \Psi$ is in TD.
(2) From $\emptyset \vdash_{\text{TAL}} \Gamma \leq \Gamma'$ and $\Psi \vdash_{\text{TAL}} R : \Gamma$ it follows by Register File Weakening that $\Psi \vdash_{\text{TAL}} R : \Gamma'$.
(3) By Canonical Forms it follows from $\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma'$ that $\Gamma' = \Gamma''[\vec{\sigma}/\vec{\alpha}]$ and $\Psi; \vec{\alpha}; \Gamma'' \vdash_{\text{TAL}} I$. By Type Substitution we conclude $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} I[\vec{\sigma}/\vec{\alpha}]$.

Case `add`, `mul`, `sub`. TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \quad \Psi \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\begin{array}{c}\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_s : \mathit{int} \qquad \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \mathit{int} \\ \Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} I\end{array}}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{arith}_p\, r_d, r_s, v; I}}{\vdash_{\text{TAL}} P}$$

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

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

Case `bnz`. TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \quad \Psi \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\begin{array}{cc}\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r : \mathit{int} & \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \\ \emptyset \vdash_{\text{TAL}} \Gamma \leq \Gamma' & \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} I\end{array}}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{bnz}\, r, v; I}}{\vdash_{\text{TAL}} P}\ .$$

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

Case `ld`. TD has the form

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

where $\Gamma' = \Gamma\{r_d : \tau_i\}$. By the operational semantics, $P' = (H, R', I)$ where $R' = R\{r_d \mapsto w_i\}$, $R(r_s) = \ell$, $H(\ell) = \langle w_0, \ldots, w_{m-1} \}$, and $0 \leq i < m$. Then
(1) $\vdash_{\text{TAL}} H : \Psi$ 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 $\Psi; \emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j}$ 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 $\Psi \vdash_{\text{TAL}} R' : \Gamma'$.
(3) $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} I$ is in TD.

Case `malloc`. TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \quad \Psi \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\emptyset \vdash_{\text{TAL}} \tau_i \quad \Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} I}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n]; I}}{\vdash_{\text{TAL}} P}$$

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, $P' = (H', R', I)$ 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 $\Psi' \vdash_{\text{TAL}} \langle ?\tau_1, \ldots, ?\tau_n \rangle : \sigma$ hval. By Heap Extension it follows that $\vdash_{\text{TAL}} H' : \Psi'$.
(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 $\Psi' \vdash_{\text{TAL}} R : \Gamma$, and it follows by Register File Update that $\Psi' \vdash_{\text{TAL}} R' : \Gamma'$.
(3) By Heap Extension, $\Psi'; \emptyset; \Gamma' \vdash_{\text{TAL}} I$.

Case `mov`. TD has the form

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

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

(1) $\vdash_{\text{TAL}} H : \Psi$ 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 $\Psi \vdash_{\text{TAL}} R' : \Gamma'$.

(3) $\Psi; \emptyset; \Gamma' \vdash_{\text{TAL}} I$ is in TD.

Case `st`. TD has the form

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

where

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

By the operational semantics, $P' = (H', R, I)$ 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 $\Psi \vdash_{\text{TAL}} R : \Gamma$ 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 $\emptyset \vdash_{\text{TAL}} \sigma_0' \le \sigma_0$ where $\Psi(\ell) = \sigma_0'$. Note that it follows from Subtyping Regularity and the definition of σ_0 that $\emptyset \vdash_{\text{TAL}} \tau_j$ 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 $\emptyset \vdash_{\text{TAL}} \sigma_1' \le \sigma_0'$ and $\emptyset \vdash_{\text{TAL}} \sigma_1' \le \sigma_1$. Since $\vdash_{\text{TAL}} H : \Psi$, we may deduce that $m = n$ and $\Psi; \emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j'}$ for $0 \le j < n$. Let $\Psi' = \Psi\{\ell : \sigma_1'\}$. By Heap Update it follows that $\Psi'; \emptyset \vdash_{\text{TAL}} w_j : \tau_j^{\varphi_j'}$.
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 that

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

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

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

(3) By Heap Update, $\Psi'; \emptyset; \Gamma' \vdash_{\text{TAL}} I$.

Case `unpack`. TD has the form

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

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

(1) $\vdash_{\text{TAL}} H : \Psi$ 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 $\Psi \vdash_{\text{TAL}} R' : \Gamma'$.

(3) By Type Substitution it follows from $\Psi;\alpha;\Gamma\{r{:}\tau'\} \vdash_{\text{TAL}} I$ that $\Psi;\emptyset;\Gamma' \vdash_{\text{TAL}} I'$.

□

THEOREM (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 $\vdash_{\text{TAL}} H : \Psi$).*

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

Case `halt` TD has the form

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

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

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \quad \Psi;\emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r_s : \mathit{int} \quad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} v : \mathit{int} \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \texttt{arith}_p\, r_d, r_s, v; I}}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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

Case `bnz` TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \quad \Psi;\emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} r : \mathit{int} \quad \Psi;\emptyset;\Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \texttt{bnz}\, r, v; I}}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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

Case `jmp` TD has the form

$$\dfrac{\vdash_{\text{TAL}} H : \Psi \quad \Psi;\emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} v : \forall[\,].\Gamma' \quad \cdots}{\Psi;\emptyset;\Gamma \vdash_{\text{TAL}} \texttt{jmp}\, v}}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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

Case `ld` TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \quad \Psi; \emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_s : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \quad \cdots}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{ld}\, r_d, r_s[i]; I} \ (1 \leq i < n)}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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

Case `malloc` Suppose that I_{full} is of the form $\texttt{malloc}\, r[\tau_1, \ldots, \tau_n]; I$. Then $P \longmapsto (H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n \rangle\}, R\{r \mapsto \ell\}, I)$ for some $\ell \notin H$.

Case `mov` TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \quad \Psi; \emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} v : \tau \quad \cdots}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{mov}\, r, v; I}}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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

Case `st` TD has the form

$$\frac{\vdash_{\text{TAL}} H : \Psi \quad \Psi; \emptyset \vdash_{\text{TAL}} R : \Gamma \qquad \dfrac{\begin{array}{c}\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_d : \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \Psi; \emptyset; \Gamma \vdash_{\text{TAL}} r_s : \tau_i \qquad \cdots\end{array}}{\Psi; \emptyset; \Gamma \vdash_{\text{TAL}} \texttt{st}\, r_d[i], r_s; I} \ (1 \leq i < n)}{\vdash_{\text{TAL}} (H, R, I_{\text{full}})} .$$

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 $P \longmapsto (H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1} \rangle\}, R, I)$.

Case `unpack` TD has the form

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

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

ACKNOWLEDGEMENTS

We are grateful to Dan Grossman, Dexter Kozen, Frederick Smith, Stephanie Weirich, Steve Zdancewic, and the anonymous referees for their many helpful comments and suggestions.

REFERENCES

Appel, A. 1992. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *3rd International Symposium on Programming Language Implementation and Logic Programming*, M. Wirsing, Ed. Springer-Verlag, New York, NY, USA, 1–13. Volume 528 of *Lecture Notes in Computer Science*.

Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C., and Eggers, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*. ACM Press, New York, NY, USA, 267–284.

Birkedal, L., Rothwell, N., Tofte, M., and Turner, D. 1993. The ML Kit (version 1). Tech. Rep. 93/14, Department of Computer Science, University of Copenhagen.

Boehm, H. and Weiser, M. 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exper. 18,* 9 (Sept.), 807–820.

Crary, K. and Weirich, S. 1999. Flexible type analysis. In *1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA. To appear.

Crary, K., Weirich, S., and Morrisett, G. 1998. Intensional polymorphism in type-erasure semantics. In *1998 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA, 301–312.

Danvy, O. and Filinski, A. 1992. Representing control: a study of the CPS transformation. *Math. Struct. Comput. Sci. 2,* 4 (Dec.), 361–391.

Dimock, A., Muller, R., Turbak, F., and Wells, J. B. 1997. Strongly typed flow-directed reprsentation transformations. In *1997 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA, 85–98.

Girard, J.-Y. 1971. 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 *Proceedings of the Second Scandinavian Logic Symposium*, J. E. Fenstad, Ed. North-Holland, 63–92.

Girard, J.-Y. 1972. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII.

Girard, J.-Y. 1987. Linear logic. *Theor. Comput. Sci. 50*, 1–102.

Glew, N. 1999. Type dispatch for named hierarchical types. In *1999 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA.

Glew, N. and Morrisett, G. 1999. Type-safe linking and modular assembly language. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 250–261.

Harper, R. and Lillibridge, M. 1993. Explicit polymorphism and CPS conversion. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 206–219.

Intel. 1996. *Intel Architecture Software Developer's Manual*. Intel Corporation, P.O. Box 7641, Mt. Prospect IL 60056-7641.

Kranz, D., Kelsey, R., Rees, J., Hudak, P. R., Philbin, J., and Adams, N. 1986. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*. ACM Press, New York, NY, USA, 219–233.

Launchbury, J. and Peyton Jones, S. 1995. State in Haskell. *J. Lisp Symb. Comput. 8,* 4 (Dec.), 293–341.

Leroy, X. 1992. Unboxed objects and polymorphic typing. In *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 177–188.

Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, CA, USA.

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

Minamide, Y., Morrisett, G., and Harper, R. 1996. Typed closure conversion. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 271–283.

Mitchell, J. and Plotkin, G. 1988. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10,* 3 (July), 470–502.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., and Zdancewic, S. 1999. TALx86: A realistic typed assembly language. In *ACM*

SIGPLAN Workshop on Compiler Support for System Software. INRIA Research Report, vol. 0228. INRIA, Centre de Diffusion, INRIA, BP 105-78153 Le Chesnay Cedex, France.

Morrisett, G., Crary, K., Glew, N., and Walker, D. 1998. Stack-based typed assembly language. In *Workshop on Types in Compilation*. Lecture Notes in Computer Science, vol. 1473. Springer-Verlag, Berlin, Germany, 28–52.

Morrisett, G. and Harper, R. 1997. Semantics of memory management for polymorphic languages. In *1st Workshop on Higher Order Operational Techniques in Semantics*, A. Gordon and A. Pitts, Eds. Publications of the Newton Institute. Cambridge University Press, Cambridge, UK.

Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*.

Necula, G. 1997. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 106–119.

Necula, G. 1998. Compiling with proofs. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA. Also available as CMU technical report CMU-CS-98-154.

Necula, G. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating System Design and Implementation*. USENIX Assocation, Berkeley, CA, USA, 229–243.

Peyton Jones, S., Hall, C., Hammond, K., Partain, W., and Wadler, P. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*. 249–257.

Reynolds, J. 1974. Towards a theory of type structure. In *Programming Symposium*. Lecture Notes in Computer Science, vol. 19. Springer-Verlag, New York, NY, USA, 408–425.

Shao, Z. 1997. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*. Boston College Computer Science Department Technical Report, vol. BCSS-97-03. Boston College, Fulton Hall, Room 460, Chestnull Hill, MA 02467-3808, USA.

Steele, Jr., G. 1978. Rabbit: A compiler for Scheme. M.S. thesis, MIT.

Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. 1996. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 181–192.

Tolmach, A. 1994. Tag-free garbage collection using explicit type parameters. In *1994 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, NY, USA, 1–11.

Wadler, P. 1990a. Comprehending monads. In *1990 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, NY, USA, 61–78.

Wadler, P. 1990b. Linear types can change the world! In *Programming Concepts and Methods*, M. Broy and C. Jones, Eds. North-Holland, Sea of Galilee, Israel. IFIP TC 2 Working Conference.

Wadler, P. 1993. A taste of linear logic. In *Mathematical Foundations of Computer Science*. Lecture Notes in Computer Science, vol. 711. Springer-Verlag, Gdansk, Poland.

Wand, M. 1992. Correctness of procedure representations in higher-order assembly language. In *Proceedings Mathematical Foundations of Programming Semantics '91*, S. Brookes, Ed. Lecture Notes in Computer Science, vol. 598. Springer-Verlag, New York, NY, USA, 294–311.

Xi, H. 1999. Dependent types in practical programming. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA.

Xi, H. and Pfenning, F. 1998. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 249–257.

Xi, H. and Pfenning, F. 1999. Dependent types in practical programming. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 214–227.

Received December 1997; revised December 1998; accepted March 1999