From System F to Typed Assembly Language[*]

Greg Morrisett David Walker Karl Crary Neal Glew

Cornell University

Abstract

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

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

To appear at the 1998 Symposium on Principles of Programming Languages

1 Introduction and Motivation

Compiling a source language to a statically typed intermediate language has compelling advantages over a conventional untyped compiler. An optimizing compiler for a high-level language such as ML may make as many as 20 passes over a single program, performing sophisticated analyses and transformations such as CPS conversion [14, 35, 2, 12, 18], closure conversion [20, 40, 19, 3, 26], unboxing [22, 28, 38], subsumption elimination [9, 11], or region inference [7]. Many of these optimizations require type information in order to succeed, and even those that do not often benefit from the additional structure supplied by a typing discipline [22, 18, 28, 37]. Furthermore, the ability to type-check intermediate code provides an invaluable tool for debugging new transformations and optimizations [41, 30].

Today a small number of compilers work with typed intermediate languages in order to realize some or all of these benefits [22, 34, 6, 41, 24, 39, 13]. 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 eliminate the untyped portions of a compiler and by so doing, extend the approach of compiling with typed intermediate languages to typed *target* languages. The target language in this paper is a strongly typed assembly language (TAL) based on a generic RISC instruction set. The type system for the language is surprisingly standard, supporting tuples, polymorphism, existentials, and a very restricted form of function pointer, yet it is sufficiently powerful that we can automatically generate well-typed and efficient code from high-level ML-like languages. Furthermore, we claim that the type system does not seriously hinder low-level optimizations such as register allocation, instruction selection, instruction scheduling, and copy propagation.

TAL not only allows us to reap the benefits of types throughout a compiler, but it also enables a practical system for executing untrusted code both safely and

---

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

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

Figure 1: Compilation of System F to Typed Assembly Language

efficiently. For example, as suggested by the SPIN project [5], operating systems could allow users to download TAL extensions into the kernel. The kernel could type-check the TAL code to ensure that the code never accesses hidden resources within the kernel, always calls kernel routines with the right number and types of arguments, *etc.*, and then assemble and dynamically link the code into the kernel.[1] However, SPIN currently requires the user to write the extension in a single high-level language (Modula-3) and use a single trusted compiler (along with cryptographic signatures) in order to ensure the safety of the extension. In contrast, a kernel based on a typed assembly language could support extensions written in a variety of high-level languages using a variety of untrusted compilers, as the safety of the resulting assembly code can be checked independently of the source code or the compiler. Furthermore, critical inner-loops could be hand-written in assembly language in order to achieve optimal performance. TAL could also be used to support extensible web-browsers, extensible servers, active networks, or any other "kernel" where security, performance, and language independence are desired.

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

With regard to these security properties, TAL is an instance of Necula and Lee's *proof carrying code* (PCC) [33, 32]. Necula suggests that the relevant operational content of simple type systems may be encoded using extensions to first-order predicate logic, and proofs of relevant security properties such as memory safety may be automatically verified [32]. In addition, Necula's approach places no restrictions on code sequences, or instruction scheduling, whereas our TAL has a small number of such restrictions (see Section 5.2). However, in general there is no complete algorithm for constructing the proof that the code satisfies the desired security properties. In contrast, we provide a fully automatic procedure for generating typed assembly language from a well-formed source term.

1.1 Overview

In order to motivate the typing constructs in TAL and to justify our claims about its expressiveness, we spend much of this paper sketching a compiler from a variant of the polymorphic λ-calculus to TAL. The anxious reader may wish to glance at Figure 11 for a sample TAL program.

The compiler in this paper is structured as four translations between the five typed calculi given in Figure 1. Each of these calculi is used as a first-class programming calculus in the sense that each translation accepts any well-typed program of its input calculus; it does not assume that the input is the output from the preceding translation. This allows the compiler to aggressively optimize code between any of the translation steps. The inspiration for the phases and their ordering is derived from SML/NJ [4, 2] (which is in turn based on the Rabbit [40] and Orbit compilers [19]) except that types are used throughout compilation.

The rest of this paper proceeds as follows: Section 2 presents λ^{F}, the compiler's source language, and sketches a typed CPS translation based on Harper and Lillibridge [18] and Danvy and Filinski [12], to our first intermediate language λ^{K}. Section 3 presents the next intermediate language, λ^{C}, and gives a typed closure translation based on, but considerably simpler than, the presentation of Minamide, Morrisett, and Harper [26]. Section 4 presents the λ^{A} intermediate language and a translation that makes allocation and initialization of data structures explicit. At this point in compilation, the intermediate code is essentially in a λ-calculus syntax for assembly language, following the ideas of Wand [48]. Finally, Section 5 presents our typed assembly language and defines a translation from λ^{A} to TAL. Section 6 discusses extensions to TAL to support language constructs not considered here.

Space considerations prevent us from giving all of the details of the term translations. We encourage those interested to read the companion technical report [31], which gives formal static semantics for each of the intermediate languages. Also, the report gives a full proof that the type system for our assembly language is sound.

2 System F and CPS Conversion

The source language for our compiler, λ^{F}, is a call-by-value variant of System F [15, 16, 36] (the polymorphic λ-calculus) augmented with products and recursion on

---

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

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

Figure 2: CPS type translation

terms. The syntax for λ^{F} appears below:

$$\begin{array}{llll}
\mathit{types} & \tau & ::= & \alpha \mid \mathit{int} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \langle\tau_1, \cdots, \tau_n\rangle \\
\mathit{terms} & e & ::= & x \mid i \mid \mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid \\
 & & & \quad e[\tau] \mid \langle e_1, \cdots, e_n\rangle \mid \pi_i(e) \mid \\
 & & & \quad e_1\, p\, e_2 \mid \mathit{if0}(e_1, e_2, e_3) \\
\mathit{prims} & p & ::= & + \mid - \mid \times
\end{array}$$

In order to simplify the presentation, we assume λ^{F} has only integers as a base type (ranged over by the metavariable i). We use $\vec{X}$ to denote a vector of syntactic objects drawn from X. For instance, $\langle\vec{\tau}\rangle$ is shorthand for a product type $\langle\tau_1, \ldots, \tau_n\rangle$. The term $\mathit{fix}\, x(x_1{:}\tau_1){:}\tau_2.e$ represents a recursively-defined function x with argument x_1 of type τ_1 and body e. Hence, both x and x_1 are bound within e. Similarly, α is bound in e for $\Lambda\alpha.e$ and bound in τ for $\forall\alpha.\tau$. As usual, we consider syntactic objects to be equivalent up to alpha-conversion of bound variables.

We interpret λ^{F} with a conventional call-by-value operational semantics (not presented here). The static semantics is specified as a set of inference rules that allow us to conclude judgments of the form $\Delta;\Gamma \vdash_{\mathrm{F}} e : \tau$ where Δ is a set containing the free type variables of Γ, e, and τ; Γ assigns types to the free variables of e; and τ is the type of e.

As a running example, we will be considering compilation and evaluation of 6 factorial:

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

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

$$\begin{array}{llll}
\mathit{types} & \tau & ::= & \alpha \mid \mathit{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \to \text{void} \mid \langle\vec{\tau}\rangle \\
\mathit{terms} & e & ::= & v[\vec{\tau}](\vec{v}) \mid \mathit{if0}(v, e_1, e_2) \mid \mathit{halt}[\tau]v \mid \\
 & & & \quad \mathit{let}\, x = v\ \mathit{in}\ e \mid \\
 & & & \quad \mathit{let}\, x = \pi_i(v)\ \mathit{in}\ e \mid \\
 & & & \quad \mathit{let}\, x = v_1\, p\, v_2\ \mathit{in}\ e \\
\mathit{values} & v & ::= & x \mid i \mid \langle\vec{v}\rangle \mid \mathit{fix}\, x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_k{:}\tau_k).e
\end{array}$$

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

There is only one abstraction mechanism (*fix*), which abstracts both type and value variables, simplifying the rest of the compiler. The corresponding $\forall$ and $\to$ types are also combined. However, we abbreviate $\forall[\,].(\vec{\tau}) \to \text{void}$ as $(\vec{\tau}) \to \text{void}$.

Unlike λ^{F}, functions do not return a value; instead, they invoke continuations. The function notation "$\to$ void" is intended to suggest this. Execution is completed by the construct $\mathit{halt}[\tau]v$, which accepts a result value v of type τ and terminates the computation. Typically, this construct is used by the top-level continuation. Aside from these differences, the static and dynamic semantics for λ^{K} is completely standard.

The implementation of typed CPS-conversion is based upon that of Harper and Lillibridge [18]. The type translation $\mathcal{K}[\![\cdot]\!]$ mapping λ^{F} types to λ^{K} types is given in Figure 2. The translation on terms is given by a judgment $\Delta;\Gamma \vdash_{\mathrm{F}} e_{\mathrm{F}} : \tau \leadsto v_{\mathrm{cps}}$ where $\Delta;\Gamma \vdash_{\mathrm{F}} e_{\mathrm{F}} : \tau$ is a derivable λ^{F} typing judgment, and v_{cps} is a λ^{K} value with type $((\mathcal{K}[\![\tau]\!]) \to \text{void}) \to \text{void}$.

Following Danvy and Filinski [12], our term translation simultaneously CPS converts the term, performs tail-call optimization, and eliminates administrative redices (see the technical report [31] for details). When applied to the factorial example, this translation yields the following λ^{K} term:

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

3 Simplified Polymorphic Closure Conversion

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

Our approach to typed closure conversion is based on that of Minamide *et al.* [26]: If two functions with the same type but different free variables (and therefore different environment types) were naively closure converted, the types of their closures would not be the same. To prevent this, closures are given *existential* types [27] where the type of the environment is held abstract.

However, we propose an approach to polymorphic closure conversion that is considerably simpler than that of Minamide *et al.*, which requires both abstract kinds and translucent types. Both of these mechanisms arise because Minamide *et al.* desire a *type-passing* interpretation of polymorphism where types are constructed and passed to polymorphic functions at run-time. Under a type-passing interpretation, polymorphic instantiation cannot easily be treated via substitution, as this requires making a copy of the code at run-time. Instead, a closure is constructed that consists of closed code, a value environment mapping variables to values, and a *type environment* mapping type variables to types.

In our approach, we assume a *type-erasure* interpretation of polymorphism as in *The Definition of Standard ML* [25], and polymorphic instantiation is semantically handled via substitution (*i.e.,* making a copy of the code with the types substituted for the type variables). As we will ultimately erase the types on terms before execution, the "copies" can (and will) be represented by the same term. This avoids the need for abstract kinds (since there are no type environments), as well as translucent types. A type-erasure interpretation is not without its costs: It precludes some advanced implementation techniques [28, 43, 1, 29] and has subtle interactions with side-effects. We address the latter concern by forcing polymorphic abstractions to be values [42, 49] (*i.e.,* they must be syntactically attached to value abstractions).

To support this interpretation, we consider the partial application of functions to type arguments to be values. For example, suppose v has the type $\forall[\vec{\alpha},\vec{\beta}].(\vec{\tau}) \rightarrow \mathrm{void}$ where the type variables $\vec{\alpha}$ are intended for the type environment and the type variables $\vec{\beta}$ are intended for the function's type arguments. If $\vec{\sigma}$ is a vector of types to be used for the type environment, then the partial instantiation $v[\vec{\sigma}]$ is still treated as a value and has type $\forall[\vec{\beta}].(\vec{\tau}[\vec{\sigma}/\vec{\alpha}]) \rightarrow \mathrm{void}$. The syntax of λ^{C} is otherwise similar to λ^{K}:

$$\begin{array}{lcll}
\mathit{types} & \tau & ::= & \alpha \mid \mathit{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \rightarrow \mathrm{void} \mid \langle\vec{\tau}\rangle \mid \exists\alpha.\tau \\
\mathit{terms} & e & ::= & v(\vec{v}) \mid \mathit{if0}(v, e_1, e_2) \mid \mathit{halt}[\tau]v \mid \\
 & & & \mathit{let}\, x = v \;\mathit{in}\; e \mid \\
 & & & \mathit{let}\, x = \pi_i(v) \;\mathit{in}\; e \mid \\
 & & & \mathit{let}\, x = v_1 \, p \, v_2 \;\mathit{in}\; e \mid \\
 & & & \mathit{let}\, [\alpha, x] = \mathit{unpack}\, v \;\mathit{in}\; e \\
\mathit{values} & v & ::= & x \mid i \mid \langle\vec{v}\rangle \mid v[\tau] \mid \mathit{pack}\, [\tau, v] \;\mathit{as}\; \exists\alpha.\tau' \mid \\
 & & & \mathit{fixcode}\, x[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_k{:}\tau_k).e
\end{array}$$

Our closure conversion technique is formalized as a type-directed translation in the companion technical report [31]. We summarize here by giving the type translation for function types:

$$\begin{array}{c}
\mathcal{C}\llbracket\forall[\vec{\alpha}].(\tau_1, \ldots, \tau_k) \rightarrow \mathrm{void}\rrbracket \\
\overset{\mathrm{def}}{=} \\
\exists\beta.\langle\forall[\vec{\alpha}].(\beta, \mathcal{C}\llbracket\tau_1\rrbracket, \ldots, \mathcal{C}\llbracket\tau_k\rrbracket) \rightarrow \mathrm{void}, \beta\rangle
\end{array}$$

The existentially-quantified variable β is the type of the value environment for the closure. The closure itself is a pair consisting of a piece of code that is instantiated with the type environment, and the value environment. The instantiated code takes as arguments the type arguments and value arguments of the original abstraction, as well as the value environment of the closure. The rules for closure converting λ^{K} abstractions and applications are given in Figure 3.

After closure conversion, all functions are closed and may be hoisted out to the top level without difficulty. After hoisting, programs belong to a calculus that is similar to λ^{C} except that *fixcode* is no longer a value. Rather, code must be referred to by *labels* (ℓ) and the syntax of programs includes a *letrec* prefix, which binds labels to code. A closure converted and hoisted factorial, as it might appear after some optimization, is shown in Figure 4.

4 Explicit Allocation

The λ^{C} intermediate language still has an atomic constructor for forming tuples, but machines must allocate space for a tuple and fill it out field by field; the allocation stage makes this process explicit. The syntax of λ^{A}, the target calculus of this stage, is similar to that of λ^{C}, and appears in Figure 5. Note that there is no longer a value form for tuples. The creation of an n-element tuple becomes a computation that is separated into an allocation step and n initialization steps. For example, if v_0 and v_1 are integers, the pair $\langle v_0, v_1\rangle$ is created as follows (where types have been added for clarity):

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

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

$$\frac{\Delta \vdash_{\mathrm{K}} \tau_i \quad \Gamma = \{y_1{:}\tau_1', \ldots, y_m{:}\tau_m'\} \quad \Delta = \{\vec{\beta}\} \quad \Delta[\vec{\alpha}];\Gamma[x{:}\tau_{\mathrm{code}}, x_1{:}\tau_1,\ldots,x_n{:}\tau_n] \vdash_{\mathrm{K}} e : \mathrm{void} \rightsquigarrow e'}{\Delta;\Gamma \vdash_{\mathrm{K}} \mathit{fix}\, x[\vec{\alpha}](x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e : \tau_{\mathrm{code}} \rightsquigarrow \mathit{pack}\ [\tau_{\mathrm{env}}, \langle v_{\mathrm{code}}[\vec{\beta}], v_{\mathrm{env}}\rangle]\ \mathit{as}\ \mathcal{C}[\![\tau_{\mathrm{code}}]\!]}\ (\mathrm{abs})$$

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

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\mathrm{K}} v : \forall[\vec{\alpha}].(\tau_1,\ldots,\tau_n) \to \mathrm{void} \rightsquigarrow v' \qquad \Delta \vdash_{\mathrm{K}} \vec{\sigma} \\ \Delta;\Gamma \vdash_{\mathrm{K}} v_1 : \tau_1[\vec{\sigma}/\vec{\alpha}] \rightsquigarrow v_1' \quad \cdots \quad \Delta;\Gamma \vdash_{\mathrm{K}} v_n : \tau_n[\vec{\sigma}/\vec{\alpha}] \rightsquigarrow v_n'\end{array}}{\Delta;\Gamma \vdash_{\mathrm{K}} v[\vec{\sigma}](v_1,\ldots,v_n) : \tau \rightsquigarrow e}\ (\mathrm{app})$$

$$\begin{array}{ll}
\text{where } e = & \mathit{let}\ [\alpha_{\mathrm{env}}, x] = \mathit{unpack}\ v'\ \mathit{in} \\
& \mathit{let}\ x_{\mathrm{code}} = \pi_1(x)\ \mathit{in} \\
& \mathit{let}\ x_{\mathrm{env}} = \pi_2(x)\ \mathit{in} \\
& x_{\mathrm{code}}[\mathcal{C}[\![\vec{\sigma}]\!]](x_{\mathrm{env}}, v_1',\ldots,v_n')
\end{array}$$

Figure 3: Closure Conversion for λ^{K} Abstractions and Applications

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

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

Figure 4: Closure Converted, Hoisted Factorial Code

$$
\begin{array}{llcl}
\textit{types} & \tau & ::= & \alpha \mid \textit{int} \mid \forall[\vec{\alpha}].(\vec{\tau}) \to \mathrm{void} \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau \\
\textit{initialization flags} & \varphi & ::= & 0 \mid 1 \\
\textit{terms} & e & ::= & \textit{let}\ \vec{d}\ \textit{in}\ v(\vec{v}) \mid \textit{let}\ \vec{d}\ \textit{in}\ \textit{if0}(v, e_1, e_2) \mid \textit{let}\ \vec{d}\ \textit{in}\ \textit{halt}[\tau]v \\
\textit{declarations} & d & ::= & x = v \mid x = \pi_i(v) \mid x = v_1\ p\ v_2 \mid [\alpha, x] = \textit{unpack}\ v \mid \\
 & & & \qquad x = \textit{malloc}[\vec{\tau}] \mid x = v[i] \leftarrow v' \\
\textit{values} & v & ::= & x \mid \ell \mid i \mid v[\tau] \mid \textit{pack}\ [\tau, v]\ \textit{as}\ \exists \alpha.\tau' \\
\textit{blocks} & b & ::= & \ell = \texttt{code}[\vec{\alpha}](x_1{:}\tau_1, \ldots, x_k{:}\tau_k).e \\
\textit{programs} & P & ::= & \textit{letrec}\ \vec{b}\ \textit{in}\ e
\end{array}
$$

Figure 5: Syntax of λ^{A}

Like all the intermediate languages of our compiler, this code sequence need not be atomic; it may be rearranged or optimized in any well-typed manner. The initialization flags on the types ensure that we cannot project a field unless it has been initialized. Furthermore, our syntactic value restriction ensures there is no unsoundness in the presence of polymorphism. However, it is important to note that we interpret $x[i] \leftarrow v$ as an imperative operation, and thus at the end of the sequence, x_0, x_1, and x are all aliases for the same location, even though they have different (but compatible) types. Consequently, the initialization flags do not prevent a field from being initialized twice. It is possible to use monads [44, 21] or linear types [17, 45, 46] to ensure that a tuple is initialized exactly once, but we have avoided these approaches in the interest of a simpler type system.

The type translation from λ^{C} to λ^{A} is trivial. All that happens is that initialization flags are added to each field of tuple types:

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

The term translation is also straightforward. As mentioned above, tuple values are exploded into a sequence of declarations consisting of a *malloc* and appropriate initializations.

5 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, we have 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. We have existential types to support closures and other data abstractions. Finally, we have n-tuples with flags on the fields indicating whether the field has been initialized.

In the remainder of this section, we present the syntax of TAL (Section 5.1), its dynamic semantics (Section 5.2), and its full static semantics (Section 5.3). Finally, we sketch the translation from λ^{A} to TAL (Section 5.4).

5.1 TAL Syntax

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

The types of TAL include type variables, integers, existentials, and tuple types augmented with initialization flags, as in λ^{A}. The type $\forall[\vec{\alpha}]\{\texttt{r1}{:}\tau_1, \ldots, \texttt{r}n{:}\tau_n\}$ is used to describe entry points of basic blocks (*i.e.,* code labels) and is the TAL analog of the λ^{A} function type, $\forall[\vec{\alpha}].(\tau_1, \ldots, \tau_n) \to \mathrm{void}$. 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 technical point is that registers may contain only *word* values, which are integers, pointers into the heap (*i.e.,* labels), and instantiated or packed word values. Tuples and code blocks are *large* values and must be heap allocated. In this manner, TAL makes the layout of data in memory explicit.

With these technical points in mind, we present the full syntax of TAL in Figure 6. A TAL abstract machine or *program* consists of a heap, a register file and a sequence of instructions. The heap is a mapping of labels to heap values, which are tuples and code. The register file is a mapping of registers (ranged over by the metavariable r) to word values. 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; this distinction is drawn because a register must contain a word, not another register. Code blocks are linear sequences

---

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

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 are considered identical, as are programs that differ only in the order of fields in a heap or register file.

5.2 TAL Operational Semantics

The operational semantics of TAL is presented in Figure 7 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 5.1).

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

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

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

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

Real machines also have a finite amount of heap space. It is straightforward to link our TAL to a conservative garbage collector [8] in order to reclaim unused heap values. Support for an accurate collector would require introducing tags so that we may distinguish pointers from integers, or else require a type-passing interpretation [43, 29]. The tagging approach is readily accomplished in our framework.

5.3 TAL Static Semantics

The static semantics for TAL appears in Figures 9 and 10 and consists of thirteen judgments, summarized in Figure 8. The static semantics is inspired by and follows the conventions of Morrisett and Harper's $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ [29]. A weak notion of subtyping is included for technical reasons related to subject reduction, so that an initialized tuple may still be given its old uninitialized type (see the technical report [31] for details).

Lemma 5.1 (Subject Reduction) *If* $\vdash_{\mathrm{TAL}} P$ *and* $P \longmapsto P'$ *then* $\vdash_{\mathrm{TAL}} P'$.

Lemma 5.2 (Progress) *If* $\vdash_{\mathrm{TAL}} P$ *then either:*

1. *there exists* P' *such that* $P \longmapsto P'$*, or*
2. P *is of the form* $(H, R\{\texttt{r1} \mapsto w\}, \texttt{halt}[\tau])$ *where there exists* Ψ *such that* $\vdash_{\mathrm{TAL}} H : \Psi$ *and* $\Psi; \emptyset \vdash_{\mathrm{TAL}} w : \tau$.

Corollary 5.3 (Type Soundness) *If* $\vdash_{\mathrm{TAL}} P$*, then there is no stuck* P' *such that* $P \longmapsto^{*} P'$.

5.4 Code Generation

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

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

The term translation is also straightforward, except that we must keep track of the register to which a variable maps, as well as the registers used thus far so that we may allocate fresh registers. When translating a block of code, we assume that registers `r1` through $\texttt{r}n$ contain the value arguments. We informally summarize the rest of the translation as follows:

- $x = v$ is mapped to $\texttt{mov}\, r_x, v$.
- $x = \pi_i(v)$ is mapped to the sequence:

$$\texttt{mov}\, r_x, v\, ;\, \texttt{ld}\, r_x, r_x[i]$$

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

Figure 6: Syntax of TAL

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

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

Figure 7: Operational Semantics of TAL

Judgment	Meaning
$\Delta \vdash_{\text{TAL}} \tau$	τ is a valid type
$\vdash_{\text{TAL}} \Psi$	Ψ is a valid heap type (no context is used because heap types must be closed)
$\Delta \vdash_{\text{TAL}} \Gamma$	Γ is a valid register file type
$\Delta \vdash_{\text{TAL}} \tau_1 \leq \tau_2$	τ_1 is a subtype of τ_2
$\Delta \vdash_{\text{TAL}} \Gamma_1 \subseteq \Gamma_2$	the register file Γ_1 weakens Γ_2
$\vdash_{\text{TAL}} H : \Psi$	the heap H has type Ψ
$\Psi; \Delta \vdash_{\text{TAL}} R : \Gamma$	the register file R has type Γ
$\Psi \vdash_{\text{TAL}} h : \tau$	the heap value h has type τ
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau$	the word value w has type τ
$\Psi; \Delta \vdash_{\text{TAL}} w : \tau^{\varphi}$	either the word value w has type τ or w is $?\tau$ and φ is 0
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v : \tau$	the small value v has type τ
$\Psi; \Delta; \Gamma \vdash_{\text{TAL}} S$	S is a valid sequence of instructions
$\vdash_{\text{TAL}} (H, R, S)$	(H, R, S) is a valid program

Figure 8: TAL Static Semantic Judgments

- $x = v_1\ p\ v_2$ is mapped to the sequence:

  $$\texttt{mov}\ r_x, v_1\ ;\ \texttt{arith}\ r_x, r_x, v_2$$

  where `arith` is the appropriate arithmetic instruction.

- $[\alpha, x] = \mathit{unpack}\ v$ is mapped to $\texttt{unpack}\ [\alpha, r_x], v$.

- $x = \mathit{malloc}[\vec{\tau}]$ is mapped to $\texttt{malloc}\ r_x[\vec{\tau}]$.

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

  $$\texttt{mov}\ r_x, v\ ;\ \texttt{mov}\ r_{\text{temp}}, v'\ ;\ \texttt{sto}\ r_x[i], r_{\text{temp}}$$

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

  $$\begin{array}{l}\texttt{mov}\ \texttt{r}_{\text{temp}_1}, v_1\ ;\ \ldots\ ;\ \texttt{mov}\ \texttt{r}_{\text{temp}_n}, v_n\ ;\\ \quad \texttt{mov}\ \texttt{r1}, \texttt{r}_{\text{temp}_1}\ ;\ \ldots\ ;\ \texttt{mov}\ \texttt{r}n, \texttt{r}_{\text{temp}_n}\ ;\ \texttt{jmp}\ v\end{array}$$

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

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

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

- $\mathit{halt}[\tau]v$ is mapped to the sequence:

  $$\texttt{mov}\ \texttt{r1}, v\ ;\ \texttt{halt}[\tau]$$

Figure 11 gives a TAL representation of the factorial computation.

The CPS, closure conversion, allocation, and code generation translations each take a well-typed source term, and produce a well-typed target term. Hence, the composition of these translations is a type-preserving compiler that takes well-typed λ^{F} terms, and produces well-typed TAL code. The soundness of the TAL type system (Corollary 5.3) ensures that the resulting code will either diverge or produce a TAL value of the appropriate type.

6 Extensions and Practice

We claim that the framework presented here is a practical approach to compilation. To substantiate this claim, we are constructing a compiler called TALC that maps the KML programming language [10] to a variant of the TAL described here, suitably adapted for the Intel x86 family of processors. We have found it straightforward to enrich the target language type system to include support for other type constructors, such as references, higher-order constructors, and recursive types. We omitted discussion of these features here in order to simplify the presentation.

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

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

Finally, since we chose a type-erasure interpretation of polymorphism, adding floats to the language requires a boxing translation. However, recent work by Leroy [23] suggests that it is only important to unbox floats in arrays and within compilation units, which is easily done in our framework.

7 Summary

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

References

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

[2] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Jan. 1989.

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

[5] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Dec. 1995.

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

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

[8] H. J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, Sept. 1988.

[9] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

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

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

[12] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.

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

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

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

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

[17] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[18] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, Jan. 1993.

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

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

[21] J. Launchbury and S. L. Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341, Dec. 1995.

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

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

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

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

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

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

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

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

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

[31] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language (extended version). Technical Report TR97-1651, Cornell University, Nov. 1997.

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

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

$$\frac{\emptyset \vdash_{\mathrm{TAL}} \tau_i}{\vdash_{\mathrm{TAL}} \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}} \qquad \frac{\Delta \vdash_{\mathrm{TAL}} \tau_i}{\Delta \vdash_{\mathrm{TAL}} \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}}$$

$$\frac{\Delta \vdash_{\mathrm{TAL}} \tau}{\Delta \vdash_{\mathrm{TAL}} \tau \leq \tau} \qquad \frac{\Delta \vdash_{\mathrm{TAL}} \tau_1 \leq \tau_2 \quad \Delta \vdash_{\mathrm{TAL}} \tau_2 \leq \tau_3}{\Delta \vdash_{\mathrm{TAL}} \tau_1 \leq \tau_3}$$

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

$$\frac{\Delta \vdash_{\mathrm{TAL}} \tau_i}{\Delta \vdash_{\mathrm{TAL}} \{r_1{:}\tau_1, \ldots, r_m{:}\tau_m\} \subseteq \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\}} \; (m \leq n)$$

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

$$\frac{\Psi; \Delta \vdash_{\mathrm{TAL}} w_i : \tau_i}{\Psi; \Delta \vdash_{\mathrm{TAL}} \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} : \Gamma} \; (\Gamma = \{r_1{:}\tau_1, \ldots, r_n{:}\tau_n\})$$

$$\frac{\Psi; \emptyset \vdash_{\mathrm{TAL}} w_i : \tau_i^{\varphi_i}}{\Psi \vdash_{\mathrm{TAL}} \langle w_1, \ldots, w_n \rangle : \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle} \qquad \frac{\vec{\alpha} \vdash_{\mathrm{TAL}} \Gamma \quad \Psi; \vec{\alpha}; \Gamma \vdash_{\mathrm{TAL}} S}{\Psi \vdash_{\mathrm{TAL}} \mathtt{code}[\vec{\alpha}]\Gamma.S : \forall[\vec{\alpha}].\Gamma}$$

$$\frac{}{\Psi; \Delta \vdash_{\mathrm{TAL}} i : \mathit{int}} \qquad \frac{\Psi(\ell) = \tau' \quad \Delta \vdash_{\mathrm{TAL}} \tau' \leq \tau}{\Psi; \Delta \vdash_{\mathrm{TAL}} \ell : \tau}$$

$$\frac{\Psi; \Delta \vdash_{\mathrm{TAL}} w : \forall[\alpha, \vec{\beta}].\Gamma \quad \Delta \vdash_{\mathrm{TAL}} \tau}{\Psi; \Delta \vdash_{\mathrm{TAL}} w[\tau] : \forall[\vec{\beta}].\Gamma[\tau/\alpha]} \qquad \frac{\Delta \vdash_{\mathrm{TAL}} \tau \quad \Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau'[\tau/\alpha]}{\Psi; \Delta \vdash_{\mathrm{TAL}} \mathit{pack}\ [\tau, w]\ \mathit{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}$$

$$\frac{\Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau}{\Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau^1} \qquad \frac{\Delta \vdash_{\mathrm{TAL}} \tau}{\Psi; \Delta \vdash_{\mathrm{TAL}} ?\tau : \tau^0}$$

$$\frac{\Gamma(r) = \tau}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r : \tau} \qquad \frac{\Psi; \Delta \vdash_{\mathrm{TAL}} w : \tau}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} w : \tau}$$

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

$$\frac{\vdash_{\mathrm{TAL}} H : \Psi \quad \Psi; \emptyset \vdash_{\mathrm{TAL}} R : \Gamma \quad \Psi; \emptyset; \Gamma \vdash_{\mathrm{TAL}} S}{\vdash_{\mathrm{TAL}} (H, R, S)}$$

Figure 9: Static Semantics of TAL (except instructions)

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

$$\frac{\Gamma(r) = \mathit{int} \quad \Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \quad \Delta \vdash_{\mathrm{TAL}} \Gamma' \subseteq \Gamma \quad \Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} S}{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} \mathtt{bnz}\, r, v; S}$$

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

$$\frac{\Delta \vdash_{\mathrm{TAL}} \tau_i \quad \Psi;\Delta;\Gamma\{r_d{:}\langle \tau_1^0, \ldots, \tau_n^0 \rangle\} \vdash_{\mathrm{TAL}} S}{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} \mathtt{malloc}\, r_d[\tau_1, \ldots, \tau_n]; S}$$

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} v : \tau \quad \Psi;\Delta;\Gamma\{r_d{:}\tau\} \vdash_{\mathrm{TAL}} S}{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} \mathtt{mov}\, r_d, v; S}$$

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

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

$$\frac{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} v : \forall[\,].\Gamma' \quad \Delta \vdash_{\mathrm{TAL}} \Gamma' \subseteq \Gamma}{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} \mathtt{jmp}\, v} \qquad \frac{\Gamma(\mathtt{r1}) = \tau}{\Psi;\Delta;\Gamma \vdash_{\mathrm{TAL}} \mathtt{halt}[\tau]}$$

Figure 10: Static Semantics of TAL instructions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, Dec. 1993.

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

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

```
fact = (H,{},S) where
H =
  l_fact:
      code[]{r1:⟨⟩,r2:int,r3:τ_k}.
        bnz r2,l_nonzero
        unpack [α,r3],r3                          % zero branch: call k (in r3) with 1
        ld r4,r3[0]                               % project k code
        ld r1,r3[1]                               % project k environment
        mov r2,1
        jmp r4                                    % jump with {r1 = env, r2 = 1}
  l_nonzero:
      code[]{r1:⟨⟩,r2:int,r3:τ_k}.
        sub r6,r2,1                               % n − 1
        malloc r4[int,τ_k]                        % create environment for cont in r4
        sto r4[0],r2                              % store n into environment
        sto r4[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 r5,l_cont
        sto r3[0],r5                              % store cont code
        sto r3[1],r4                              % store environment ⟨n,k⟩
        mov r2,r6                                 % arg := n − 1
        mov r3,pack [⟨int^1,τ_k^1⟩,r3] as τ_k     % abstract the type of the environment
        jmp l_fact                                % jump to k with {r1 = env, r2 = n − 1, r3 = cont}
  l_cont:
      code[]{r1:⟨int^1,τ_k^1⟩,r2:int}.            % r2 contains (n − 1)!
        ld r3,r1[0]                               % retrieve n
        ld r4,r1[1]                               % retrieve k
        mul r2,r3,r2                              % n × (n − 1)!
        unpack [α,r4],r4                          % unpack k
        ld r5,r4[0]                               % project k code
        ld r1,r4[1]                               % project k environment
        jmp r5                                    % jump to k with {r1 = env, r2 = n!}
  l_halt:
      code[]{r1:⟨⟩,r2:int}.
        mov r1,r2
        halt[int]                                 % halt with result in r1


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

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

Figure 11: Typed Assembly Code for Factorial