FROM SYSTEM F TO TYPED ASSEMBLY LANGUAGE

LECTURE & NOTES BY JAMES T. PERCONTI
BASED ON THE TOPLAS 1999 PAPER OF THE SAME NAME BY MORRISETT ET AL.

1. Typed Assembly Language

A traditional assembly language provides only one type of value: the word-sized bit pattern. All operations apply alike to all strings of bits, whether they are arithmetic operations, dereferences, control transfers, or anything else.

What if we could use a low-level language that worked in terms of richer abstractions, more in line with those provided by our higher-level languages?

Typed Assembly Language is an assembly language that discriminates things like

- integers,
- pointers to tuples, and
- code labels.

It keeps these different kinds of values distinct and only allows the appropriate operations on each type. Its type system can guarantee that programs will not break these abstractions (type safety).

Why do we want types in assembly language? Here are some of the benefits of typechecking assembly programs:

- It's impossible to manufacture pointers to arbitrary space or jump to unverified code.
- Kernel routines are only called with correct number and types of arguments.
- With type-preserving compilation, we maintain many invariants of our original high-level language program, such as preventing code from looking inside the environment of a closure.

We'll show how to get from System F to TAL using a five-pass compiler. As we proceed, notice how little the types change.

Here is the overall structure of the compiler:

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

We'll talk about each stage of compilation in detail, but in general, each pass eliminates some high-level feature from our language, getting us closer to assembly-style programs.

2. System F

To begin, here is a formulation of System F with integers, tuples, and recursion. There's only a little that's new here; have a look at the syntax:

$$
\begin{array}{rcll}
\tau, \sigma & ::= & \alpha \mid \mathrm{int} \mid \tau_1 \to \tau_2 \mid \forall \alpha.\tau \mid \langle \tau_1, \dots, \tau_n \rangle & \textit{types} \\
e & ::= & u^{\tau} & \textit{annotated terms} \\
u & ::= & x \mid i \mid \mathrm{fix}\, x(x_1 \colon \tau_1) \colon \tau_2.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ [\tau] \mid \langle e_1, \dots, e_n \rangle & \textit{terms} \\
 & & \mid \pi_i(e) \mid e_1\ p\ e_2 \mid \mathrm{if0}(e_1, e_2, e_3) & \\
p & ::= & + \mid - \mid \times & \textit{primitives} \\
\Delta & ::= & \alpha_1 \dots, \alpha_n & \textit{type contexts} \\
\Gamma & ::= & x_1 \colon \tau_1, \dots, x_n \colon \tau_n & \textit{value contexts}
\end{array}
$$

Note that we don't have a separate λ-expression from the recursive fix expression, but we don't need it, because we aren't required to actually make recursive calls in the body of our functions.

The biggest thing to note is that we annotate each subexpression with its type. This gives us all the type information we need to ensure that a well-typed System F program compiles to a well-typed TAL program. Working with annotations everywhere might seem tedious, but notice that the programmer doesn't actually have to write them: they can be inserted by the typechecker. Everything we are doing happens inside the compiler, after typechecking has finished, so we can simply carry the annotations around once they are inserted. In this presentation, we will often leave the annotations implicit just to minimize clutter. The appendix has complete definitions with all the necessary annotations.

The type system enforces that annotations are used in the natural way: we have a judgment for annotated terms that just checks that the annotation matches the type of the underlying term:

$$\boxed{\Delta; \Gamma \vdash_F e \colon \tau} \qquad \frac{\Delta; \Gamma \vdash_F u \colon \tau}{\Delta; \Gamma \vdash_F u^{\tau} \colon \tau}$$

There are two other type judgments in this system: a judgment to check that types are well formed, and the main typing judgment for unannotated terms.

$$\boxed{\Delta \vdash_F \tau} \qquad\qquad \boxed{\Delta; \Gamma \vdash_F u \colon \tau}$$

The latter is where all the familiar typing rules live. We don't need to go over all of them because we've seen them many times by now, but let's remind ourselves what the rule for fix looks like:

$$\frac{\Delta; \Gamma, x \colon \tau_1 \to \tau_2, x_1 \colon \tau_1 \vdash_F e \colon \tau_2}{\Delta; \Gamma \vdash_F (\mathrm{fix}\, x(x_1 \colon \tau_1) \colon \tau_2.e) \colon \tau_1 \to \tau_2}$$

To typecheck a recursive function, we add the function's name and argument to our environment at the appropriate types, and then typecheck the body.

The complete type system for this language and for all the other languages we'll talk about are given in the appendix. We won't give an operational semantics until we get all the way to TAL, but it should be clear throughout what the implied semantics are for each language.

OK, enough review of System F; let's start compiling!

3. CPS conversion (λ^K)

Our first pass converts the program to continuation-passing style. This makes the order of evaluation explicit and gives us a linear program (except when we have branching paths from if0 expressions). Also, like in assembly language, all intermediate results are explicitly saved in temporary variables.

We've talked some about CPS in class already, but I'll review the important parts. First, let's look at the syntax of λ^K, the CPS language we're heading for:

$$\begin{array}{rcll}
\tau, \sigma & ::= & \alpha \mid \mathrm{int} \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void} & \textit{types} \\
v, k & ::= & u^\tau & \textit{annotated values} \\
u & ::= & x \mid i \mid \langle v_1, \ldots, v_n \rangle \mid \mathrm{fix}\, x[\alpha_1, \ldots, \alpha_n](x_1 : \tau_1, \ldots, x_m : \tau_m).e & \textit{values} \\
p & ::= & + \mid - \mid \times & \textit{primitives} \\
d & ::= & x = v \mid x = \pi_i\ v \mid x = v_1\ p\ v_2 & \textit{declarations} \\
e & ::= & \texttt{let}\ d\ \texttt{in}\ e \mid v[\tau_1, \ldots, \tau_n](v_1, \ldots, v_m) \mid \mathrm{if0}(v, e_1, e_2) \mid \mathbf{halt}[\tau]v & \textit{terms} \\
\Delta & ::= & \alpha_1 \ldots, \alpha_n & \textit{type contexts} \\
\Gamma & ::= & x_1 : \tau_1, \ldots, x_n : \tau_n & \textit{value contexts}
\end{array}$$

There are several differences between λ^K and the formulation of System F we came from. Let's have a look at them one at a time:

(1) *Function abstractions and type abstractions have been merged into one syntactic form.* The fix construct now takes both type arguments and value arguments, and it can take a bunch of arguments at once. This is not really relevant to CPS itself, but it's convenient to have this minor feature when we define the translation.
(2) *Values and terms have been split up.* We no longer allow any kind of expression to appear anywhere. Instead, we distinguish between the forms that do computation (applications, projections, primitive arithmetic operations, and conditionals) and the forms that are already values. This is how we get the linearity we're looking for.
(3) *What's up with that big long type we have now instead of function types and universal types?* I'm glad you asked!

First of all, it's big and long because of the changes we talked about in point (1). But the interesting thing here is the '$\to$ void' that has replaced the return type of a function. Why is this here?

In a continuation-passing program, like an imperative assembly program, we don't really think of functions as things that return values. Functions just do some computation, and then hand off control to the *continuation* they've been carrying around. A continuation is a function that can be thought of as "the rest of the program." When we look at the translation from System F to λ^K in a minute, we'll see that every function will be given an extra argument for the continuation it should hand off to when it's done.

Of course, we don't really want our program to *never* return a value, so we also add a special form $\mathbf{halt}[\tau]v$ that stops the program and gives the final result. But to match the idea that functions don't return, we write the type of a function as taking its arguments and returning void.

Before we move on, let's look a little more closely at how this language forces our programs to be linear. Suppose we wanted to compute a simple arithmetic expression such as $(2+3)-(7\times 5)$. We can't directly write this down in λ^K syntax!

Instead, we have to write it something like this:

$$\texttt{let}\ x = 2 + 3\ \texttt{in}\ \texttt{let}\ y = 7 \times 5\ \texttt{in}\ \texttt{let}\ z = x - y\ \texttt{in}\ \textbf{halt}[\text{int}]z.$$

Typing λ^K. Let's talk quickly about the type system for λ^K, and then get to the translation itself. We have four judgments in this type system:

$$\boxed{\Delta \vdash_K \tau} \qquad \boxed{\Delta; \Gamma \vdash_K v \colon \tau} \qquad \boxed{\Delta; \Gamma \vdash_K u \colon \tau} \qquad \boxed{\Delta; \Gamma \vdash_K e}$$

These are mostly similar to the three judgments in System F. What's different is that just as we split values from computations in the syntax, we've split them into separate judgments in the type system as well. Since we have this idea that computation expressions don't return, we don't give them types. Instead, that judgment can be said to check that expressions are well-formed.

Let's look at one of the rules from this last judgment:

$$\frac{\Delta; \Gamma \vdash_K v \colon \langle \tau_1, \ldots, \tau_n \rangle \qquad \Delta; \Gamma, x \colon \tau_i \vdash_K e \qquad 1 \le i \le n}{\Delta; \Gamma \vdash_K \texttt{let}\ x = \pi_i\ v\ \texttt{in}\ e}$$

To typecheck a projection expression $\texttt{let}\ x = \pi_i\ v\ \texttt{in}\ e$, we just need to make sure that the value being projected from is a tuple, and check that the body is well-formed in the environment where x has the type of the appropriate component of the tuple.

The other typing rules are all straightforward.

Translating System F to λ^K. Let's start by looking at how we translate types. We'll write the CPS translation of a System F type τ as $\mathcal{K}_{\text{typ}}[\![\tau]\!]$.

We'll also use a helper function $\mathcal{K}_{\text{cont}}[\![\tau]\!]$, which gives us the type of a continuation that expects to receive an argument of type $\mathcal{K}_{\text{typ}}[\![\tau]\!]$. What type is that? Simply a function type that takes no type arguments, and takes one term argument, as we just described:

$$\mathcal{K}_{\text{cont}}[\![\tau]\!] \stackrel{\text{def}}{=} \forall[].(\mathcal{K}_{\text{typ}}[\![\tau]\!]) \to \text{void}$$

With this helper, the type translation is fairly straightforward. We need to add an argument to each function (this includes type abstractions) for the continuation it's going to call off to when it's done, but other than that, we just recur structurally:

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

Our translation function for terms, $\mathcal{K}_{\text{exp}}[\![e]\!]k$, takes a continuation as an argument. To translate a whole program, we need to do some work at the top level to set up the final continuation, $\textbf{halt}[\tau]v$.

$$\mathcal{K}_{\text{prog}}[\![u^\tau]\!] \stackrel{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\text{fix}\, x_f[](x_v \colon \mathcal{K}_{\text{typ}}[\![\tau]\!]).\textbf{halt}[\mathcal{K}_{\text{typ}}[\![\tau]\!]]x_v)^{\mathcal{K}_{\text{cont}}[\![\tau]\!]}$$

But this is fairly straightforward: we just call the expression translation with an appropriate k.

We don't need to look at every rule of the term translation, but the core idea is that value forms must call their continuation,

$$\mathcal{K}_{\text{exp}}[\![i^\tau]\!]k \overset{\text{def}}{=} k[](i)$$

while computation forms must divide up their work, doing the first piece directly, and building a continuation to do the rest.

$$\mathcal{K}_{\text{exp}}[\![\pi_i(u^\tau)]\!]k \overset{\text{def}}{=} \mathcal{K}_{\text{exp}}[\![u^\tau]\!](\text{fix}\, x_f[](x_v\colon \mathcal{K}_{\text{typ}}[\![\tau]\!]).\texttt{let}\ y = \pi_i\ x_v\ \texttt{in}\ k[](y))$$

Functions and type abstractions behave just like other value forms, except that we have to add the extra continuation argument.

$$\mathcal{K}_{\text{exp}}[\![\Lambda\alpha.u^\tau]\!]k \overset{\text{def}}{=} k[](\text{fix}\, x_f[\alpha](x_k\colon \mathcal{K}_{\text{cont}}[\![\tau]\!]).\mathcal{K}_{\text{exp}}[\![e]\!]x_k)$$

4. Interlude: Type Preservation; Optimization

None of the passes in our compiler perform any optimization. But of course we want to enable optimizing compilation. This is not a problem, though, because we can just add more passes that perform optimization in between translation passes. For instance, we could write something that takes λ^K programs to optimized λ^K programs and just add it to the pipeline right after CPS conversion.

The thing to worry about when we add passes is that our goal is to preserve well-typedness as we compile System F programs to TAL. Fortunately, we get this property by composing a series of lemmas for each translation step: we can prove that $\mathcal{K}_{\text{prog}}[\![e]\!]$ maps well-typed System F programs to well-typed λ^K programs, $\mathcal{C}_{\text{prog}}[\![e]\!]$ maps well-typed λ^K programs to well-typed λ^C programs, and so on.

Further, the type system for each intermediate language is defined for that whole language, not just for programs that come directly out of the translation from the previous language. So all we need to do is ensure that additional compilation passes within our intermediate languages also preserve well-typedness, and then we maintain our result for the full compilation process.

5. Closure Conversion (λ^C)

The most obvious high-level feature in our language are *closures*. Functions in λ^K have an implicit environment that gives values to their free variables. We can't write down free variables in an assembly program, so they have to go!

We'll add arguments to each function corresponding to each free variable, and package the function with an *explicit* environment object. Let's try to write down the type translation for functions:

$$\mathcal{C}_{\text{typ}}[\![\forall[\alpha_1,\ldots,\alpha_n](\tau_1,\ldots,\tau_m)\to\text{void}]\!] \overset{\text{def}}{=} \qquad \langle\forall[\alpha_1,\ldots,\alpha_n](\tau_1,\ldots,\tau_m,?)\to\text{void},?\rangle$$

At the type level, we don't know what the environment should look like, because we can't actually check the free variables that appear in the function. But we know it will have *some* type, so let's just say exactly that:

$$\mathcal{C}_{\text{typ}}[\![\forall[\alpha_1,\ldots,\alpha_n](\tau_1,\ldots,\tau_m)\to\text{void}]\!] \overset{\text{def}}{=} \exists\beta.\langle\forall[\alpha_1,\ldots,\alpha_n](\tau_1,\ldots,\tau_m,\beta)\to\text{void},\beta\rangle$$

In fact, using an existential type is quite appropriate here, because we want to protect the invariants that the closure abstraction gave us. The type we wrote down enforces that each function is called only with a compatible environment, and that nothing else can look at the pieces of another function's environment.

Let's add existential types and the corresponding terms to our language:

$$\begin{array}{rll} \tau, \sigma & ::= \cdots \mid \exists \alpha.\tau & \textit{types} \\ u & ::= \cdots \mid \mathrm{pack}\,[\tau_1, v]\,\mathrm{as}\,\tau_2 & \textit{values} \\ d & ::= \cdots \mid [\alpha, x] = \mathrm{unpack}\,v & \textit{declarations} \end{array}$$

We now have a solid plan for dealing with term variables (we'll look at the term translation in a moment), but what about type variables? We also need to eliminate free type variables from our functions.

We could try to use the same strategy as with term variables, but this requires a bunch of machinery (existential kinds! and more) that we don't want to deal with, so instead let's think about the computational content of types.

Once we finish using types for any static checking and for compilation, they don't really have any effect on the program. We can *erase* types at the end of our compilation process and run the program without them. When we take this view, type instantiation doesn't perform computation, so let's replace our current form for function application with two new forms that move types out of the world of computation:

$$\begin{array}{rll} u & ::= \cdots \mid v[\tau] & \textit{values} \\ e & ::= \cdots \mid v(v_1, \ldots, v_n) & \textit{terms} \end{array}$$

With type application now a value form, we are finished constructing λ^C, the target language of our closure conversion translation. Aside from those two sets of changes, everything is the same as λ^K. For the most part, the type system is also similar, with the same four judgments and all the rules you would expect. It is worth noting, however, that the rule for functions now requires that the function body typecheck in an environment containing only the variables and type variables bound by that function:

$$\frac{\begin{array}{c}\forall i.\ \alpha_1, \ldots, \alpha_n \vdash_C \tau_i \\ \alpha_1, \ldots, \alpha_n; x \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void}, x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash_C e\end{array}}{\Delta; \Gamma \vdash_C (\mathrm{fix}\,x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e) \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void}}$$

This matches what our closure-converted language is supposed to do: free variables have gone away.

We still have to finish defining how to translate λ^K into λ^C. The rest of the type translation just recurs structurally, so there is nothing interesting to do. For the term translation, the only place where something interesting happens is, of course, when we translate functions. The full rule is quite complicated, but let's take at least a partial look:

$$\begin{aligned} &\mathcal{C}_{\mathrm{val}}[\![(\mathrm{fix}\,x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_n \colon \tau_n).e)^\tau]\!] \\ &\qquad\qquad\qquad \overset{\mathrm{def}}{=} \mathrm{pack}\,[\tau_{\mathrm{env}}, \langle v_{\mathrm{code}}[\beta_1] \cdots [\beta_n], v_{\mathrm{env}} \rangle]\,\mathrm{as}\,\mathcal{C}_{\mathrm{typ}}[\![\tau]\!] \end{aligned}$$

The formal definitions of all these pieces are in the appendix, but informally:

- v_{env} is a tuple containing all the free term variables of the function, and τ_{env} is its corresponding type (τ_{env} is easy to calculate because the variables still have type annotations on them).
- The β_i are the free type variables of the function.

- v_{code} transforms the function by replacing each occurrence of a former free variable with a lookup into the corresponding component of the environment tuple.

With this translation, we've eliminated free variables from our language! What's next?

6. Hoisting (λ^H)

Another feature we need to remove from our language is nested functions. Since functions no longer have free variables, this is easy! We're going to pull functions up to a special top-level form. Let's begin by adding the new syntax we'll need.

$h ::= \text{code}[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e$ *heap values*

$P ::= \texttt{letrec}\ x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\ \texttt{in}\ e$ *programs*

We have a new class of values h for the things that live on the heap in assembly programs. Code blocks are one such thing, and currently the only heap value form we have. Our "heap" for the moment is the letrec form we've added.

To perform the hoisting translation, we just have to find all the fix expressions in our program, move them to the heap under a fresh variable, and replace their original position with that variable.

With that done, we can remove the old function form from the syntax, prohibiting λ^H programs from putting functions anywhere but in the initial heap. And that's it!

A quick word about λ^H's type system before we move on: while not much has changed from λ^C, we have added new syntactic categories, so we need new judgments to typecheck them:

$$\boxed{\vdash_H P} \qquad \boxed{\Gamma \vdash_H h \colon \tau \text{ hval}}$$

The judgment for programs just calls off to the other judgments to check that all the heap values and the body are well-typed in an environment containing all the top-level names. The judgment for heap values contains only a rule for code blocks, which works like the previous languages' rule for functions, except that we can once again use variables bound in the external environment. This doesn't undo the restriction we added for λ^C because now the environment can only ever contain top-level code labels. (The heap value judgment includes the literal symbol hval just to make it easier to distinguish which judgment it is.)

7. Explicit Allocation (λ^A)

In assembly language, we can't create or manipulate large pieces of data directly; instead, we keep them on the heap and load in a word-sized piece at a time to work with. We need to change the way our language handles tuples to match this model. Syntactically, we replace the previous version of tuples and tuple types with the following constructs:

$\tau, \sigma ::= \cdots \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle$ *types*

$\varphi ::= 1 \mid 0$ *initialization flags*

$v ::= \cdots \mid ?\tau$ *annotated values*

$h ::= \cdots \mid \langle v_1, \ldots, v_n \rangle$ *heap values*

$d ::= \cdots \mid x = \texttt{malloc}[\tau_1, \ldots, \tau_n] \mid x = v_1[i] \leftarrow v_2$ *declarations*

Tuple types are now labeled with a flag that indicates whether each element of the tuple is initialized (1), or not (0). The tuple value form has been moved to a heap value form, and we've added a few constructs to let us build them:

- The $?\tau$ form represents an uninitialized element of a tuple.
- The `malloc` declaration makes a new tuple of appropriately-typed uninitialized values.
- The other new declaration form lets us set the value of a tuple element.

Translating λ^H to λ^A is simple: when a tuple appears in our λ^H program, we just add the sequence of declarations necessary to build it and replace the occurrence of the tuple itself with the appropriate variable. At the type level, since tuples that appeared in our λ^H program were always fully-initialized, we translate tuple types by adding the flag for an initialized element to each type in the tuple:

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

Finally, let's note the changes to the type system. Most obviously, the rule for tuples moves to the judgment for heap values:

$$\frac{\forall i.\ \cdot; \Gamma \vdash_A v_i \colon \tau_i^{\varphi_i}}{\Gamma \vdash_A \langle v_1, \ldots, v_n \rangle \colon \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \text{ hval}}$$

The judgment used in the premise of this rule is also new: it deals with initialization flags.

$$\boxed{\Delta; \Gamma \vdash_A v \colon \tau^{\varphi}} \qquad \frac{\Delta; \Gamma \vdash_A v \colon \tau}{\Delta; \Gamma \vdash_A v \colon \tau^{\varphi}} \qquad \frac{}{\Delta; \Gamma \vdash_A ?\tau \colon \tau^0}$$

Any of our old value forms can typecheck with either initialization flag, since we don't prevent re-initialization of the same tuple element. The $?\tau$ form, of course, may only have the flag for an uninitialized element.

8. Code Generation (TAL)

We're now very close to having programs that work at an assembly level! Let's have a look at the language we want to end up with. Here is the full syntax of TAL:

τ, σ	$::=$	$\alpha \mid \text{int} \mid \forall[\alpha_1, \ldots, \alpha_n].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau$	*types*
φ	$::=$	$1 \mid 0$	*initialization flags*
Ψ	$::=$	$\{\ell_1 \colon \tau_1, \ldots, \ell_n \colon \tau_n\}$	*heap types*
Γ	$::=$	$\{r_1 \colon \tau_1, \ldots, r_n \colon \tau_n\}$	*register file types*
Δ	$::=$	$\alpha_1 \ldots, \alpha_n$	*type contexts*
r	$::=$	$\texttt{r1} \mid \texttt{r2} \mid \texttt{r3} \mid \cdots$	*registers*
w	$::=$	$\ell \mid i \mid ?\tau \mid w[\tau] \mid \text{pack}\,[\tau_1, w]\,\text{as}\,\tau_2$	*word values*
v	$::=$	$r \mid w \mid v[\tau] \mid \text{pack}\,[\tau_1, v]\,\text{as}\,\tau_2$	*small values*
h	$::=$	$\langle v_1, \ldots, v_n \rangle \mid \text{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I$	*heap values*
H	$::=$	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$	*heaps*
R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$	*register files*
ι	$::=$	$\texttt{add}\, r_d, r_s, v \mid \texttt{mul}\, r_d, r_s, v \mid \texttt{sub}\, r_d, r_s, v \mid \texttt{bnz}\, r, v$	*instructions*
		$\mid \texttt{ld}\, r_d, r_s[i] \mid \texttt{st}\, r_d[i].r_s \mid \texttt{mov}\, r_d, v$	
		$\mid \texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n] \mid \texttt{unpack}[\alpha, r_d], v$	
I	$::=$	$\iota; I \mid \texttt{jmp}\, v \mid \texttt{halt}[\tau]$	*instruction sequences*

$P ::= (H, R, I)$ *programs*

Look first at the set of available instructions ι, and note that all but the last two are typical instructions we might see in any assembly language. Once we perform type erasure, the $\texttt{unpack}[\alpha, r_d], v$ instruction is equivalent to $\texttt{mov}\, r_d, v$, so it should be easy to accept.

The `malloc` instruction is a little more complicated, but essentially it can be implemented as an atomic code sequence that allocates space on the heap equal to the size of the desired tuple. Our formulation of TAL assumes we have an unlimited amount of heap space and, for that matter, an unlimited number of registers, which of course is not the case on a real machine. But for our purposes here, we are not going to worry about allocation issues.

Next, let's look at the kinds of values used in TAL. Word values (w) are things that can fit in a register: pointers, integers, and the "uninitialized tuple element" value $?\tau$. We also have type applications of and existential packages containing other word values, but these things don't take up any space or have any operational meaning because of type erasure.

"Small" values (v) are the values that can appear in an instruction. They include all the same things as word values, as well as register names r. When given a register name as a small value, the semantics of TAL (given on the last page of the appendix) automatically looks up the underlying word value in that register.

Heap values (h) include code blocks and tuples, just as in λ^A.

Typing TAL. Overall, our language of types has changed very little from what we started with in System F. The surrounding type system is also fairly straightforward. But there are a few things that should be addressed. First, there have been two fairly simple syntactic changes:

(1) In assembly language, our λ-calculus notion of variables has to shift to notions of pointers and registers. Thus type environments must become heap types Ψ and register file types Γ, which map heap pointers and register names respectively to their types.
(2) Instead of having a list of argument types, the function type now explicitly states which registers the arguments are expected to be given in, in the form of the register file type the function expects. (We've dropped the "$\rightarrow$ void" from the function type as well, but there's no deep reason for this. It was just a symbol anyway.)

Since there are a lot of syntactic categories in TAL, there are a lot of judgments in the type system. But don't worry! Most of them are very simple. First, we have the usual judgment for well formed types, and we have judgments that check that all the types appearing in a register file type or heap type are well-formed:

$$\boxed{\Delta \vdash_{\mathrm{TAL}} \tau} \qquad \boxed{\vdash_{\mathrm{TAL}} \Psi} \qquad \boxed{\Delta \vdash_{\mathrm{TAL}} \Gamma}$$

Note that the heap type judgment doesn't have any environment: heap values can only have closed types.

There are two more judgments that deal exclusively with constructs at the type level:

$$\boxed{\Delta \vdash_{\mathrm{TAL}} \tau_1 \leq \tau_2} \qquad \boxed{\Delta \vdash_{\mathrm{TAL}} \Gamma_1 \leq \Gamma_2}$$

These "subtyping" judgments each have one specific purpose, and contain only exactly the rules needed to carry it out. The first one lets us reset some of the initialization flags on a tuple type, essentially forgetting that we have initialized those values. It's used when we typecheck a pointer. The second one lets us forget about registers we no longer need when we jump or branch to a code block that doesn't expect those registers to have values.

At the term level, we need judgments to typecheck whole program configurations and each program component:

$$\boxed{\vdash_{\text{TAL}} P} \qquad \boxed{\vdash_{\text{TAL}} H \colon \Psi} \qquad \boxed{\Psi \vdash_{\text{TAL}} R \colon \Gamma} \qquad \boxed{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} I}$$

These in turn rely on judgments to typecheck each kind of value:

$$\boxed{\Psi \vdash_{\text{TAL}} h \colon \tau \text{ hval}} \qquad \boxed{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau \text{ wval}} \qquad \boxed{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau^{\varphi}}$$

$$\boxed{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \tau}$$

There is not much that is new or interesting in the various value judgments, and the program, heap, and register file judgments just call out to the others in the natural way. However, it is worth noting that all program components and values can reference the heap, so the type of the heap Ψ is an environment in all the corresponding judgments. Similarly, instructions and small values can refer to registers, so those judgments contain the register file type Γ. Finally, free type variables can be found in a word or small value, or in instructions, so an appropriate environment Δ is used, but heap values must be closed.

The judgment for an instruction sequence is a bit more interesting, so let's look at a couple of example rules. For a typical rule, let's typecheck an instruction sequence beginning with an addition instruction. We need to ensure that the inputs r_s and v are integers, and then typecheck the remainder of the instruction sequence in an environment where the register file type has been extended or updated to map the destination register r_d to the type int.

$$\frac{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} r_s \colon \text{int} \qquad \Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \text{int} \qquad \Psi; \Delta; \Gamma\{r_d \colon \text{int}\} \vdash_{\text{TAL}} I}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \texttt{add}\, r_d, r_s, v; I}$$

Let's also look at the two instructions that end a sequence: `jmp` and **halt**. When we jump to a code block, the register file type the code block expects must be a subset of the register file type we currently have. As discussed above, we are allowed to ignore registers we don't need anymore thanks to the subtyping judgment on register file types:

$$\frac{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \forall[].\Gamma' \qquad \Delta \vdash_{\text{TAL}} \Gamma \leq \Gamma'}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \texttt{jmp}\, v}$$

The final configuration of a program, signified by the **halt** instruction, expects the result to be stored in the first register:

$$\frac{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \texttt{r1} \colon \tau}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \textbf{halt}[\tau]}$$

Translating λ^A to TAL. At the type level, the only thing that happens in the translation is that function argument types are given explicit register names, as we already discussed.

Programs `letrec` $x_1 \mapsto h_1, \ldots, x_n \mapsto h_n$ `in` e are translated to an initial configuration with a heap built from the h_i, an empty register file, and an instruction sequence given by the translation of e. The translations of program components are parameterized by a map γ that gives heap locations and register names to correspond to λ^A's variables. This initially maps the x_i to a fresh set of heap locations.

The translations for λ^A's value forms are not worth detailing any more than this. The expression translation is more interesting, but involves a lot of bookkeeping, so we will just give a high-level idea of how each form is compiled. In general, each expression becomes an appropriate instruction sequence and also affects the starting heap.

- To translate `let` $x = v$ `in` e, we assign a fresh register r to correspond to x in γ, and generate a `mov` instruction, storing the value from the translation of v in r. Then we continue the instruction sequence with the translation of e.
- For the projection `let` $x = \pi_i\ v$ `in` e, we again assign a register $x \mapsto r$, and then we generate a `mov` instruction to put the pointer resulting from translating v into r and a `ld` instruction to put the appropriate tuple element into r, continuing from there by translating e.
- A primitive operation `let` $x = v_1\ p\ v_2$ `in` e translates to the corresponding arithmetic instruction, preceded by a `mov` to put one of the arguments in a register, since one input to an arithmetic instruction must come from a register.
- An unpack expression simply translates to an `unpack` instruction.
- A `malloc` expression simply translates to a `malloc` instruction.
- An update to a tuple is the counterpart to a projection. We do a `st` after preparing the arguments with a couple of `mov`s.
- To translate an application $v(v_1, \ldots, v_n)$, since our program is in continuation-passing style and functions are closed and never return, all we have to do is prepare the arguments and jump to the appropriate code block. Functions in TAL always expect their arguments to be in the first n registers, we just generate a sequence of `mov` instructions to store the translations of the v_i into the registers `r`i, and then conclude with a `jmp` instruction using the translation of v.
- An if0 expression has to build a new code block on the heap for one of its bodies, but other than that, we just generate a `mov` to put the value to test into a register, and then a `bnz`.
- Finally, λ^A's **halt**$[\tau]v$ becomes a `mov` to put the translation of v into `r1` and then the instruction **halt**$[\tau]$.

Material taken from *From System F to Typed Assembly Language* by Morrisett, Walker, Crary, and Glew, TOPLAS 1999

9. Appendix: Definitions

9.1. λ^F.

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

$$\boxed{\Delta \vdash_F \tau} \quad \frac{\mathrm{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_F \tau} \qquad\qquad \boxed{\Delta; \Gamma \vdash_F e \colon \tau} \quad \frac{\Delta; \Gamma \vdash_F u \colon \tau}{\Delta; \Gamma \vdash_F u^\tau \colon \tau}$$

$$\boxed{\Delta; \Gamma \vdash_F u \colon \tau}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_F x \colon \tau} \qquad\qquad \frac{}{\Delta; \Gamma \vdash_F i \colon \mathrm{int}}$$

$$\frac{\Delta \vdash_F \tau_1 \qquad \Delta \vdash_F \tau_1 \qquad \Delta; \Gamma, x \colon \tau_1 \to \tau_2, x_1 \colon \tau_1 \vdash_F e \colon \tau_2}{\Delta; \Gamma \vdash_F (\mathrm{fix}\, x(x_1 \colon \tau_1) \colon \tau_2.e) \colon \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Gamma \vdash_F e_1 \colon \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash_F e_2 \colon \tau_1}{\Delta; \Gamma \vdash_F e_1\ e_2 \colon \tau_2} \qquad\qquad \frac{\Delta, \alpha; \Gamma \vdash_F e \colon \tau}{\Delta; \Gamma \vdash_F \Lambda\alpha.e \colon \forall\alpha.\tau}$$

$$\frac{\Delta \vdash_F \tau \qquad \Delta; \Gamma \vdash_F e \colon \forall\alpha.\tau'}{\Delta; \Gamma \vdash_F e\ [\tau] \colon \tau'[\tau/\alpha]} \qquad\qquad \frac{\forall i.\ \Delta; \Gamma \vdash_F e_i \colon \tau_i}{\Delta; \Gamma \vdash_F \langle e_1, \dots, e_n \rangle \colon \langle \tau_1, \dots, \tau_n \rangle}$$

$$\frac{\Delta; \Gamma \vdash_F e \colon \langle \tau_1, \dots, \tau_n \rangle \qquad 1 \le i \le n}{\Delta; \Gamma \vdash_F \pi_i(e) \colon \tau_i} \qquad\qquad \frac{\Delta; \Gamma \vdash_F e_1 \colon \mathrm{int} \qquad \Delta; \Gamma \vdash_F e_2 \colon \mathrm{int}}{\Delta; \Gamma \vdash_F e_1\ p\ e_2 \colon \mathrm{int}}$$

$$\frac{\Delta; \Gamma \vdash_F e_1 \colon \mathrm{int} \qquad \Delta; \Gamma \vdash_F e_2 \colon \tau \qquad \Delta; \Gamma \vdash_F e_3 \colon \tau}{\Delta; \Gamma \vdash_F \mathrm{if0}(e_1, e_2, e_3) \colon \tau}$$

Translation from λ^F to λ^K. (variables introduced by the translation must be fresh)

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

$$\begin{array}{ll}
\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket & \stackrel{\text{def}}{=} \forall[].(\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket) \to \mathrm{void}
\end{array}$$

$$\begin{array}{ll}
\mathcal{K}_{\mathrm{prog}}\llbracket u^\tau\rrbracket & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u^\tau\rrbracket(\mathrm{fix}\, x_f[](x\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket).\mathbf{halt}[\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket]x^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket})^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket}
\end{array}$$

$$\begin{array}{ll}
\mathcal{K}_{\mathrm{exp}}\llbracket y^\tau\rrbracket k & \stackrel{\text{def}}{=} k[](y^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}) \\
\mathcal{K}_{\mathrm{exp}}\llbracket i^\tau\rrbracket k & \stackrel{\text{def}}{=} k[](i^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}) \\
\mathcal{K}_{\mathrm{exp}}\llbracket(\mathrm{fix}\, x(x_1\colon\tau_1)\colon\tau_2.e)^\tau\rrbracket k & \stackrel{\text{def}}{=} k[]((\mathrm{fix}\, x[](x_1\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket, x_k\colon \mathcal{K}_{\mathrm{cont}}\llbracket\tau_2\rrbracket). \\
& \qquad\qquad \mathcal{K}_{\mathrm{exp}}\llbracket e\rrbracket x_k^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_2\rrbracket})^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}) \\
\mathcal{K}_{\mathrm{exp}}\llbracket(u_1^{\tau_1}\ u_2^{\tau_2})^\tau\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u_1^{\tau_1}\rrbracket(\mathrm{fix}\, x_f[](x_1\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket). \\
& \quad \mathcal{K}_{\mathrm{exp}}\llbracket u_2^{\tau_2}\rrbracket(\mathrm{fix}\, x_f[](x_2\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_2\rrbracket). \\
& \qquad x_1^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket}[](x_2^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau_2\rrbracket}, k))^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_2\rrbracket})^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_1\rrbracket} \\
\mathcal{K}_{\mathrm{exp}}\llbracket(\Lambda\alpha.u^\tau)^{\tau'}\rrbracket k & \stackrel{\text{def}}{=} k[]((\mathrm{fix}[\alpha](x_k\colon \mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket). \\
& \qquad\qquad \mathcal{K}_{\mathrm{exp}}\llbracket u^\tau\rrbracket x_k^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket})^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau'\rrbracket}) \\
\mathcal{K}_{\mathrm{exp}}\llbracket(u^\tau\ [\sigma])^{\tau'}\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u^\tau\rrbracket(\mathrm{fix}\, x_f[](x\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket). \\
& \quad x^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}[\mathcal{K}_{\mathrm{typ}}\llbracket\sigma\rrbracket](k))^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket} \\
\mathcal{K}_{\mathrm{exp}}\llbracket\langle u_1^{\tau_1},\ldots,u_n^{\tau_n}\rangle^\tau\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u_1^{\tau_1}\rrbracket(\mathrm{fix}\, x_f[](x_1\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket).\cdots \\
& \quad \mathcal{K}_{\mathrm{exp}}\llbracket u_n^{\tau_n}\rrbracket(\mathrm{fix}\, x_f[](x_n\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_n\rrbracket). \\
& \qquad k[](\langle x_1^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket},\ldots,x_n^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau_n\rrbracket}\rangle^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}))^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_n\rrbracket}\cdots)^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_1\rrbracket} \\
\mathcal{K}_{\mathrm{exp}}\llbracket\pi_i(u^\tau)^{\tau'}\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u^\tau\rrbracket(\mathrm{fix}\, x_f[](x\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket). \\
& \quad \mathtt{let}\ y = \pi_i\ x\ \mathtt{in}\ k[](y^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau'\rrbracket}))^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket} \\
\mathcal{K}_{\mathrm{exp}}\llbracket(u_1^{\tau_1}\ p\ u_2^{\tau_2})^\tau\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u_1^{\tau_1}\rrbracket(\mathrm{fix}\, x_f[](x_1\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket). \\
& \quad \mathcal{K}_{\mathrm{exp}}\llbracket u_2^{\tau_2}\rrbracket(\mathrm{fix}\, x_f[](x_2\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_2\rrbracket). \\
& \qquad \mathtt{let}\ y = x_1\ p\ x_2\ \mathtt{in}\ k[](y^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau\rrbracket}))^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_2\rrbracket})^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_1\rrbracket} \\
\mathcal{K}_{\mathrm{exp}}\llbracket\mathrm{if0}(u_1^{\tau_1}, e_2, e_3)^\tau\rrbracket k & \stackrel{\text{def}}{=} \mathcal{K}_{\mathrm{exp}}\llbracket u_1^{\tau_1}\rrbracket(\mathrm{fix}\, x_f[](x\colon \mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket). \\
& \quad \mathrm{if0}(x^{\mathcal{K}_{\mathrm{typ}}\llbracket\tau_1\rrbracket}, \mathcal{K}_{\mathrm{exp}}\llbracket e_2\rrbracket k, \mathcal{K}_{\mathrm{exp}}\llbracket e_3\rrbracket k)^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau\rrbracket})^{\mathcal{K}_{\mathrm{cont}}\llbracket\tau_1\rrbracket}
\end{array}$$

9.2. λ^K.

$$\begin{array}{rcll}
\tau, \sigma & ::= & \alpha \mid \text{int} \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void} & \textit{types} \\
v, k & ::= & u^\tau & \textit{annotated values} \\
u & ::= & x \mid i \mid \langle v_1, \ldots, v_n \rangle \mid \text{fix}\, x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e & \textit{values} \\
p & ::= & + \mid - \mid \times & \textit{primitives} \\
d & ::= & x = v \mid x = \pi_i\ v \mid x = v_1\ p\ v_2 & \textit{declarations} \\
e & ::= & \texttt{let}\ d\ \texttt{in}\ e \mid v[\tau_1, \ldots, \tau_n](v_1, \ldots, v_m) \mid \text{if0}(v, e_1, e_2) \mid \mathbf{halt}[\tau]v & \textit{terms} \\
\Delta & ::= & \alpha_1 \ldots, \alpha_n & \textit{type contexts} \\
\Gamma & ::= & x_1 \colon \tau_1, \ldots, x_n \colon \tau_n & \textit{value contexts}
\end{array}$$

$$\boxed{\Delta \vdash_K \tau} \quad \frac{\text{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_K \tau} \qquad\qquad \boxed{\Delta; \Gamma \vdash_K v \colon \tau} \quad \frac{\Delta; \Gamma \vdash_K u \colon \tau}{\Delta; \Gamma \vdash_K u^\tau \colon \tau}$$

$$\boxed{\Delta; \Gamma \vdash_K u \colon \tau}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_K x \colon \tau} \qquad \frac{}{\Delta; \Gamma \vdash_K i \colon \text{int}} \qquad \frac{\forall i.\ \Delta; \Gamma \vdash_K v_i \colon \tau_i}{\Delta; \Gamma \vdash_K \langle v_1, \ldots, v_n \rangle \colon \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\begin{array}{c}\forall i.\ \Delta, \alpha_1, \ldots, \alpha_n \vdash_K \tau_i \\ (\Delta, \alpha_1, \ldots, \alpha_n); (\Gamma, x \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void}, x_1 \colon \tau_1, \ldots, x_n \colon \tau_n) \vdash_K e\end{array}}{\Delta; \Gamma \vdash_K (\text{fix}\, x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e) \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void}}$$

$$\boxed{\Delta; \Gamma \vdash_K e}$$

$$\frac{\Delta; \Gamma \vdash_K v \colon \tau \qquad \Delta; \Gamma, x \colon \tau \vdash_K e}{\Delta; \Gamma \vdash_K \texttt{let}\ x = v\ \texttt{in}\ e}$$

$$\frac{\Delta; \Gamma \vdash_K v \colon \langle \tau_1, \ldots, \tau_n \rangle \qquad \Delta; \Gamma, x \colon \tau_i \vdash_K e \qquad 1 \le i \le n}{\Delta; \Gamma \vdash_K \texttt{let}\ x = \pi_i\ v\ \texttt{in}\ e}$$

$$\frac{\Delta; \Gamma \vdash_K v_1 \colon \text{int} \qquad \Delta; \Gamma \vdash_K v_2 \colon \text{int} \qquad \Delta; \Gamma, x \colon \text{int} \vdash_K e}{\Delta; \Gamma \vdash_K \texttt{let}\ x = v_1\ p\ v_2\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\forall i.\ \Delta \vdash_K \sigma_i \qquad \forall i.\ \Delta; \Gamma \vdash_K v_i \colon \tau_i[\sigma_1/\alpha_1] \cdots [\sigma_n/\alpha_n] \\ \Delta; \Gamma \vdash_K v \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void}\end{array}}{\Delta; \Gamma \vdash_K v[\sigma_1, \ldots, \sigma_n](v_1, \ldots, v_m)}$$

$$\frac{\Delta; \Gamma \vdash_K e_1 \colon \text{int} \qquad \Delta; \Gamma \vdash_K e_2 \qquad \Delta; \Gamma \vdash_K e_3}{\Delta; \Gamma \vdash_K \text{if0}(v, e_2, e_3)} \qquad \frac{\Delta; \Gamma \vdash_K v \colon \tau}{\Delta; \Gamma \vdash_K \mathbf{halt}[\tau]v}$$

Translation from λ^K to λ^C. (variables introduced by the translation must be fresh)

$$\mathcal{C}_{\text{typ}}[\![\alpha]\!] \overset{\text{def}}{=} \alpha$$

$$\mathcal{C}_{\text{typ}}[\![\text{int}]\!] \overset{\text{def}}{=} \text{int}$$

$$\mathcal{C}_{\text{typ}}[\![\forall[\alpha_1,\ldots,\alpha_n](\tau_1,\ldots,\tau_n) \to \text{void}]\!] \overset{\text{def}}{=} \exists\beta.\langle\forall[\alpha_1,\ldots,\alpha_n](\beta,\mathcal{C}_{\text{typ}}[\![\tau_1]\!],\ldots,\mathcal{C}_{\text{typ}}[\![\tau_n]\!]) \to \text{void},\beta\rangle$$

$$\mathcal{C}_{\text{typ}}[\![\langle\tau_1,\ldots,\tau_n\rangle]\!] \overset{\text{def}}{=} \langle\mathcal{C}_{\text{typ}}[\![\tau_1]\!],\ldots,\mathcal{C}_{\text{typ}}[\![\tau_n]\!]\rangle$$

$$\mathcal{C}_{\text{prog}}[\![e]\!] \overset{\text{def}}{=} \mathcal{C}_{\text{exp}}[\![e]\!]$$

$$\mathcal{C}_{\text{exp}}[\![\texttt{let}\ d\ \texttt{in}\ e]\!] \overset{\text{def}}{=} \texttt{let}\ \mathcal{C}_{\text{dec}}[\![d]\!]\ \texttt{in}\ \mathcal{C}_{\text{exp}}[\![e]\!]$$

$$\begin{aligned}\mathcal{C}_{\text{exp}}[\![u^\tau[\sigma_1,\ldots,\sigma_n](v_1,\ldots,v_m)]\!] \overset{\text{def}}{=}\ & \texttt{let}\ [\alpha,x] = \text{unpack}\,\mathcal{C}_{\text{val}}[\![u^\tau]\!]\ \texttt{in}\\ & \texttt{let}\ x_{\text{code}} = \pi_1\ x^{\langle\tau_{\text{code}},\alpha\rangle}\ \texttt{in}\\ & \texttt{let}\ x_{\text{env}} = \pi_2\ x^{\langle\tau_{\text{env}},\alpha\rangle}\ \texttt{in}\\ & {x_{\text{code}}}^{\tau_{\text{code}}}[\mathcal{C}_{\text{typ}}[\![\sigma_1]\!]]\cdots[\mathcal{C}_{\text{typ}}[\![\sigma_n]\!]]\\ & \qquad (x^\alpha_{\text{env}},\mathcal{C}_{\text{val}}[\![v_1]\!],\ldots,\mathcal{C}_{\text{val}}[\![v_n]\!])\\ & \quad \text{where } \mathcal{C}_{\text{val}}[\![\tau]\!] = \exists\alpha.\langle\tau_{\text{code}},\alpha\rangle\end{aligned}$$

$$\mathcal{C}_{\text{exp}}[\![\text{if0}(v,e_1,e_2)]\!] \overset{\text{def}}{=} \text{if0}(\mathcal{C}_{\text{val}}[\![v]\!],\mathcal{C}_{\text{exp}}[\![e_1]\!],\mathcal{C}_{\text{exp}}[\![e_2]\!])$$

$$\mathcal{C}_{\text{exp}}[\![\mathbf{halt}[\tau]v]\!] \overset{\text{def}}{=} \mathbf{halt}[\mathcal{C}_{\text{typ}}[\![\tau]\!]]\mathcal{C}_{\text{val}}[\![v]\!]$$

$$\mathcal{C}_{\text{dec}}[\![x = v]\!] \overset{\text{def}}{=} x = \mathcal{C}_{\text{val}}[\![v]\!]$$

$$\mathcal{C}_{\text{dec}}[\![x = \pi_i\ v]\!] \overset{\text{def}}{=} x = \pi_i\ \mathcal{C}_{\text{val}}[\![v]\!]$$

$$\mathcal{C}_{\text{dec}}[\![x = v_1\ p\ v_2]\!] \overset{\text{def}}{=} x = \mathcal{C}_{\text{val}}[\![v_1]\!]\ p\ \mathcal{C}_{\text{val}}[\![v_2]\!]$$

$$\mathcal{C}_{\text{val}}[\![x^\tau]\!] \overset{\text{def}}{=} x^{\mathcal{C}_{\text{typ}}[\![\tau]\!]}$$

$$\mathcal{C}_{\text{val}}[\![i^\tau]\!] \overset{\text{def}}{=} i^{\mathcal{C}_{\text{typ}}[\![\tau]\!]}$$

$$\mathcal{C}_{\text{val}}[\![\langle v_1,\ldots,v_n\rangle^\tau]\!] \overset{\text{def}}{=} \langle\mathcal{C}_{\text{val}}[\![v_1]\!],\ldots,\mathcal{C}_{\text{val}}[\![v_n]\!]\rangle^{\mathcal{C}_{\text{typ}}[\![\tau]\!]}$$

$$\begin{aligned}\mathcal{C}_{\text{val}}[\![(\text{fix}\,x[\alpha_1,\ldots,\alpha_n](x_1\colon\tau_1,\ldots,x_m\colon\tau_m).e)^\tau]\!] \overset{\text{def}}{=}\ & (\text{pack}\,[\tau_{\text{env}},\langle(v_{\text{code}}[\beta_1]\cdots[\beta_n])^{\tau_{\text{code}}},v_{\text{env}}\rangle^{\langle\tau_{\text{code}},\tau_{\text{env}}\rangle}]\\ & \text{as}\ \mathcal{C}_{\text{typ}}[\![\tau]\!])^{\mathcal{C}_{\text{typ}}[\![\tau]\!]}\end{aligned}$$

$$\begin{aligned}\text{where } y_1^{\sigma_1},\ldots,y_i^{\sigma_i} &= \text{fv}(\text{fix}\,x[\alpha_1,\ldots,\alpha_n](x_1\colon\tau_1,\ldots,x_m\colon\tau_m).e)\\ \beta_1,\ldots,\beta_j &= \text{ftv}(\text{fix}\,x[\alpha_1,\ldots,\alpha_n](x_1\colon\tau_1,\ldots,x_m\colon\tau_m).e)\\ \tau_{\text{env}} &= \mathcal{C}_{\text{typ}}[\![\langle\sigma_1,\ldots,\sigma_i\rangle]\!]\\ \tau_{\text{rawcode}} &= \forall[\beta_1,\ldots,\beta_j,\alpha_1,\ldots,\alpha_n].(\tau_{\text{env}},\mathcal{C}_{\text{typ}}[\![\tau_1]\!],\ldots,\mathcal{C}_{\text{typ}}[\![\tau_m]\!]) \to \text{void}\\ \tau_{\text{code}} &= \forall[\alpha_1,\ldots,\alpha_n].(\tau_{\text{env}},\mathcal{C}_{\text{typ}}[\![\tau_1]\!],\ldots,\mathcal{C}_{\text{typ}}[\![\tau_m]\!]) \to \text{void}\\ v_{\text{env}} &= \langle y_1^{\mathcal{C}_{\text{typ}}[\![\sigma_1]\!]},\ldots,y_i^{\mathcal{C}_{\text{typ}}[\![\sigma_i]\!]}\rangle^\tau_{\text{env}}\\ v_{\text{code}} &= (\text{fix}\,x_{\text{code}}[\beta_1,\ldots,\beta_j,\alpha_1,\ldots,\alpha_n](x_{\text{env}}\colon\tau_{\text{env}},x_1\colon\mathcal{C}_{\text{typ}}[\![\tau_1]\!],\ldots,x_m\colon\mathcal{C}_{\text{typ}}[\![\tau_m]\!]).\\ &\qquad \texttt{let}\ x = (\text{pack}\,[\tau_{\text{env}},\langle {x_{\text{code}}}^{\tau_{\text{rawcode}}}[\beta_1]\cdots[\beta_j]^{\tau_{\text{code}}},x_{\text{env}}^{\tau_{\text{env}}}\rangle^{\langle\tau_{\text{code}},\tau_{\text{env}}\rangle}]\,\text{as}\,\mathcal{C}_{\text{typ}}[\![\tau]\!])^{\mathcal{C}_{\text{typ}}[\![\tau]\!]}\ \texttt{in}\\ &\qquad \texttt{let}\ y_1 = \pi_1\ {x_{\text{env}}}^{\tau_{\text{env}}}\ \texttt{in}\ \cdots\texttt{let}\ y_i = \pi_i\ {x_{\text{env}}}^{\tau_{\text{env}}}\ \texttt{in}\ \mathcal{C}_{\text{exp}}[\![e]\!])^{\tau_{\text{rawcode}}}\end{aligned}$$

9.3. λ^C. (differences from λ^K are highlighted)

$$\begin{array}{rlr}
\tau, \sigma & ::= \alpha \mid \mathrm{int} \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void} \mid \exists \alpha.\tau & \textit{types} \\
v & ::= u^\tau & \textit{annotated values} \\
u & ::= x \mid i \mid \langle v_1, \ldots, v_n \rangle \mid \mathrm{fix}\, x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e \mid v[\tau] & \textit{values} \\
& \quad \mid \mathrm{pack}\, [\tau_1, v]\, \mathrm{as}\, \tau_2 & \\
p & ::= + \mid - \mid \times & \textit{primitives} \\
d & ::= x = v \mid x = \pi_i\ v \mid x = v_1\ p\ v_2 \mid [\alpha, x] = \mathrm{unpack}\, v & \textit{declarations} \\
e & ::= \mathtt{let}\ d\ \mathtt{in}\ e \mid v(v_1, \ldots, v_m) \mid \mathrm{if0}(v, e_1, e_2) \mid \mathbf{halt}[\tau]v & \textit{terms} \\
\Delta & ::= \alpha_1 \ldots, \alpha_n & \textit{type contexts} \\
\Gamma & ::= x_1 \colon \tau_1, \ldots, x_n \colon \tau_n & \textit{value contexts}
\end{array}$$

$$\boxed{\Delta \vdash_C \tau} \quad \frac{\mathrm{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_C \tau} \qquad \boxed{\Delta; \Gamma \vdash_C v \colon \tau} \quad \frac{\Delta; \Gamma \vdash_C u \colon \tau}{\Delta; \Gamma \vdash_C u^\tau \colon \tau}$$

$$\boxed{\Delta; \Gamma \vdash_C u \colon \tau}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_C x \colon \tau} \qquad \frac{}{\Delta; \Gamma \vdash_C i \colon \mathrm{int}} \qquad \frac{\forall i.\ \Delta; \Gamma \vdash_C v_i \colon \tau_i}{\Delta; \Gamma \vdash_C \langle v_1, \ldots, v_n \rangle \colon \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\begin{array}{c}\forall i.\ \alpha_1, \ldots, \alpha_n \vdash_C \tau_i \\ \alpha_1, \ldots, \alpha_n; x \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void}, x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash_C e\end{array}}{\Delta; \Gamma \vdash_C (\mathrm{fix}\, x[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e) \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \mathrm{void}}$$

$$\frac{\Delta \vdash_C \sigma \qquad \Delta; \Gamma \vdash_C v \colon \forall[\alpha, \beta_1, \ldots, \beta_n].(\tau_1, \ldots, \tau_n) \to \mathrm{void}}{\Delta; \Gamma \vdash_C v[\sigma] \colon \forall[\beta_1, \ldots, \beta_n].(\tau_1[\sigma/\alpha], \ldots, \tau_n[\sigma/\alpha]) \to \mathrm{void}}$$

$$\frac{\Delta \vdash_C \tau_1 \qquad \Delta; \Gamma \vdash_C v \colon \tau_2[\tau_1/\alpha]}{\Delta; \Gamma \vdash_C \mathrm{pack}\, [\tau_1, v]\, \mathrm{as}\, \exists \alpha.\tau_2 \colon \exists \alpha.\tau_2}$$

$$\boxed{\Delta; \Gamma \vdash_C e}$$

$$\frac{\Delta; \Gamma \vdash_C v \colon \tau \qquad \Delta; \Gamma, x \colon \tau \vdash_C e}{\Delta; \Gamma \vdash_C \mathtt{let}\ x = v\ \mathtt{in}\ e} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash_C v \colon \langle \tau_1, \ldots, \tau_n \rangle \\ \Delta; \Gamma, x \colon \tau_i \vdash_C e \qquad 1 \le i \le n\end{array}}{\Delta; \Gamma \vdash_C \mathtt{let}\ x = \pi_i\ v\ \mathtt{in}\ e}$$

$$\frac{\Delta; \Gamma \vdash_C v_1 \colon \mathrm{int} \qquad \Delta; \Gamma \vdash_C v_2 \colon \mathrm{int} \qquad \Delta; \Gamma, x \colon \mathrm{int} \vdash_C e}{\Delta; \Gamma \vdash_C \mathtt{let}\ x = v_1\ p\ v_2\ \mathtt{in}\ e}$$

$$\frac{\forall i.\ \Delta; \Gamma \vdash_C v_i \colon \tau_i \qquad \Delta; \Gamma \vdash_C v \colon \forall[].(\tau_1, \ldots, \tau_m) \to \mathrm{void}}{\Delta; \Gamma \vdash_C v(v_1, \ldots, v_m)}$$

$$\frac{\Delta; \Gamma \vdash_C e_1 \colon \mathrm{int} \qquad \Delta; \Gamma \vdash_C e_2 \qquad \Delta; \Gamma \vdash_C e_3}{\Delta; \Gamma \vdash_C \mathrm{if0}(v, e_2, e_3)} \qquad \frac{\Delta; \Gamma \vdash_C v \colon \tau}{\Delta; \Gamma \vdash_C \mathbf{halt}[\tau]v}$$

$$\frac{\Delta; \Gamma \vdash_C v \colon \exists \alpha.\tau \qquad \Delta, \alpha; \Gamma, x \colon \tau \vdash_C e}{\Delta; \Gamma \vdash_C \mathtt{let}\ [\alpha, x] = \mathrm{unpack}\, v\ \mathtt{in}\ e}$$

Translation from λ^C to λ^H. (variables introduced by the translation must be fresh)

We use the following shorthand for heaps:

$$H ::= x_1 \mapsto h_1, \ldots, x_n \mapsto h_n$$

$\mathcal{H}_{\text{typ}}[\![\tau]\!] \overset{\text{def}}{=} \tau$

$\mathcal{H}_{\text{prog}}[\![e]\!] \overset{\text{def}}{=} \texttt{letrec}\ H\ \texttt{in}\ e'$ where $(e', H) = \mathcal{H}_{\text{exp}}[\![e]\!]$

$\mathcal{H}_{\text{exp}}[\![\texttt{let}\ d\ \texttt{in}\ e]\!] \overset{\text{def}}{=} (\texttt{let}\ d'\ \texttt{in}\ e', H, H')$

where $(d', H) = \mathcal{H}_{\text{dec}}[\![d]\!]$ and $(e', H') = \mathcal{H}_{\text{exp}}[\![e]\!]$

$\mathcal{H}_{\text{exp}}[\![v(v_1, \ldots, v_n)]\!] \overset{\text{def}}{=} (v'(v'_1, \ldots, v'_n), H, H_1, \ldots, H_n)$

where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$ and $(v'_1, H_1) = \mathcal{H}_{\text{val}}[\![v_1]\!]$, $\ldots$, $(v'_n, H_n) = \mathcal{H}_{\text{val}}[\![v_n]\!]$

$\mathcal{H}_{\text{exp}}[\![\text{if0}(v, e_1, e_2)]\!] \overset{\text{def}}{=} (\text{if0}(v', e'_1, e'_2), H, H_1, H_2)$

where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$, $(e'_1, H_1) = \mathcal{H}_{\text{exp}}[\![e_1]\!]$, and $(e'_2, H_2) = \mathcal{H}_{\text{exp}}[\![e_2]\!]$

$\mathcal{H}_{\text{exp}}[\![\textbf{halt}[\tau]v]\!] \overset{\text{def}}{=} (\textbf{halt}[\tau]v', H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{dec}}[\![x = v]\!] \overset{\text{def}}{=} (x = v', H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{dec}}[\![x = \pi_i\ v]\!] \overset{\text{def}}{=} (x = \pi_i\ v', H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{dec}}[\![x = v_1\ p\ v_2]\!] \overset{\text{def}}{=} (x = v'_1\ p\ v'_2, H_1, H_2)$

where $(v'_1, H_1) = \mathcal{H}_{\text{val}}[\![v_1]\!]$ and $(v'_2, H_2) = \mathcal{H}_{\text{val}}[\![v_2]\!]$

$\mathcal{H}_{\text{dec}}[\![[\alpha, x] = \text{unpack}\, v]\!] \overset{\text{def}}{=} ([\alpha, x] = \text{unpack}\, v', H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{val}}[\![x^\tau]\!] \overset{\text{def}}{=} (x^\tau, \cdot)$

$\mathcal{H}_{\text{val}}[\![i^\tau]\!] \overset{\text{def}}{=} (i^\tau, \cdot)$

$\mathcal{H}_{\text{val}}[\![\langle v_1, \ldots, v_n\rangle^\tau]\!] \overset{\text{def}}{=} (\langle v'_1, \ldots, v'_n\rangle^\tau, H_1, \ldots, H_n)$

where $(v'_1, H_1) = \mathcal{H}_{\text{val}}[\![v_1]\!]$, $\ldots$, $(v'_n, H_n) = \mathcal{H}_{\text{val}}[\![v_n]\!]$

$\mathcal{H}_{\text{val}}[\![v[\sigma]^\tau]\!] \overset{\text{def}}{=} (v'[\sigma]^\tau, H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{val}}[\![(\text{pack}\,[\tau_1, v]\,\text{as}\,\tau_2)^\tau]\!] \overset{\text{def}}{=} (\text{pack}\,[\tau_1, v']\,\text{as}\,\tau_2)^\tau, H)$ where $(v', H) = \mathcal{H}_{\text{val}}[\![v]\!]$

$\mathcal{H}_{\text{val}}[\![(\text{fix}\,x[\alpha_1, \ldots, \alpha_n](x_1\colon \tau_1, \ldots, x_m\colon \tau_m).e)^\tau]\!] \overset{\text{def}}{=} ({x_{\text{clos}}}^\tau, H, x_{\text{clos}} \mapsto h)$

where $(e', H) = \mathcal{H}_{\text{exp}}[\![e[x_{\text{clos}}/x]]\!]$ and $h = \text{code}[\alpha_1, \ldots, \alpha_n](x_1\colon \tau_1, \ldots, x_m\colon \tau_m).e'\}$

9.4. λ^H. (differences from λ^C are highlighted)

τ, σ	$::=$	$\alpha \mid \text{int} \mid \langle \tau_1, \dots, \tau_n \rangle \mid \forall[\alpha_1, \dots, \alpha_n].(\tau_1, \dots, \tau_m) \to \text{void} \mid \exists \alpha.\tau$	*types*
v	$::=$	u^τ	*annotated values*
u	$::=$	$x \mid i \mid \langle v_1, \dots, v_n \rangle \mid v[\tau] \mid \text{pack}\,[\tau_1, v]\,\text{as}\,\tau_2$	*values*
h	$::=$	$\text{code}[\alpha_1, \dots, \alpha_n](x_1 \colon \tau_1, \dots, x_m \colon \tau_m).e$	*heap values*
p	$::=$	$+ \mid - \mid \times$	*primitives*
d	$::=$	$x = v \mid x = \pi_i v \mid x = v_1\ p\ v_2 \mid [\alpha, x] = \text{unpack}\,v$	*declarations*
e	$::=$	$\texttt{let}\ d\ \texttt{in}\ e \mid v(v_1, \dots, v_m) \mid \text{if0}(v, e_1, e_2) \mid \textbf{halt}[\tau]v$	*terms*
P	$::=$	$\texttt{letrec}\ x_1 \mapsto h_1, \dots, x_n \mapsto h_n\ \texttt{in}\ e$	*programs*
Δ	$::=$	$\alpha_1 \dots, \alpha_n$	*type contexts*
Γ	$::=$	$x_1 \colon \tau_1, \dots, x_n \colon \tau_n$	*value contexts*

$$\boxed{\Delta \vdash_H \tau} \quad \frac{\text{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_H \tau} \qquad\qquad \boxed{\Delta; \Gamma \vdash_H v \colon \tau} \quad \frac{\Delta; \Gamma \vdash_H u \colon \tau}{\Delta; \Gamma \vdash_H u^\tau \colon \tau}$$

$$\boxed{\Delta; \Gamma \vdash_H u \colon \tau}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_H x \colon \tau} \qquad \frac{}{\Delta; \Gamma \vdash_H i \colon \text{int}} \qquad \frac{\forall i.\ \Delta; \Gamma \vdash_H v_i \colon \tau_i}{\Delta; \Gamma \vdash_H \langle v_1, \dots, v_n \rangle \colon \langle \tau_1, \dots, \tau_n \rangle}$$

$$\frac{\Delta \vdash_H \sigma \qquad \Delta; \Gamma \vdash_H v \colon \forall[\alpha, \beta_1, \dots, \beta_n].(\tau_1, \dots, \tau_n) \to \text{void}}{\Delta; \Gamma \vdash_H v[\sigma] \colon \forall[\beta_1, \dots, \beta_n].(\tau_1[\sigma/\alpha], \dots, \tau_n[\sigma/\alpha]) \to \text{void}}$$

$$\frac{\Delta \vdash_H \tau_1 \qquad \Delta; \Gamma \vdash_H v \colon \tau_2[\tau_1/\alpha]}{\Delta; \Gamma \vdash_H \text{pack}\,[\tau_1, v]\,\text{as}\,\exists\alpha.\tau_2 \colon \exists\alpha.\tau_2}$$

$$\boxed{\Gamma \vdash_H h \colon \tau\ \text{hval}}$$

$$\frac{\forall i.\ \alpha_1, \dots, \alpha_n \vdash_H \tau_i \qquad \alpha_1, \dots, \alpha_n; \Gamma, x_1 \colon \tau_1, \dots, x_n \colon \tau_n \vdash_H e}{\Gamma \vdash_H \text{code}[\alpha_1, \dots, \alpha_n](x_1 \colon \tau_1, \dots, x_m \colon \tau_m).e \colon \forall[\alpha_1, \dots, \alpha_n].(\tau_1, \dots, \tau_m) \to \text{void hval}}$$

$$\boxed{\Delta; \Gamma \vdash_H e}$$

$$\frac{\Delta; \Gamma \vdash_H v \colon \tau \qquad \Delta; \Gamma, x \colon \tau \vdash_H e}{\Delta; \Gamma \vdash_H \texttt{let}\ x = v\ \texttt{in}\ e} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash_H v \colon \langle \tau_1, \dots, \tau_n \rangle \\ \Delta; \Gamma, x \colon \tau_i \vdash_H e \qquad 1 \le i \le n\end{array}}{\Delta; \Gamma \vdash_H \texttt{let}\ x = \pi_i\ v\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_H v_1 \colon \text{int} \\ \Delta; \Gamma \vdash_H v_2 \colon \text{int} \qquad \Delta; \Gamma, x \colon \text{int} \vdash_H e\end{array}}{\Delta; \Gamma \vdash_H \texttt{let}\ x = v_1\ p\ v_2\ \texttt{in}\ e} \qquad \frac{\begin{array}{c}\forall i.\ \Delta; \Gamma \vdash_H v_i \colon \tau_i \\ \Delta; \Gamma \vdash_H v \colon \forall[].(\tau_1, \dots, \tau_m) \to \text{void}\end{array}}{\Delta; \Gamma \vdash_H v(v_1, \dots, v_m)}$$

$$\frac{\Delta; \Gamma \vdash_H e_1 \colon \text{int} \qquad \Delta; \Gamma \vdash_H e_2 \qquad \Delta; \Gamma \vdash_H e_3}{\Delta; \Gamma \vdash_H \text{if0}(v, e_2, e_3)} \qquad \frac{\Delta; \Gamma \vdash_H v \colon \tau}{\Delta; \Gamma \vdash_H \textbf{halt}[\tau]v}$$

$$\frac{\Delta; \Gamma \vdash_H v \colon \exists\alpha.\tau \qquad \Delta, \alpha; \Gamma, x \colon \tau \vdash_H e}{\Delta; \Gamma \vdash_H \texttt{let}\ [\alpha, x] = \text{unpack}\,v\ \texttt{in}\ e}$$

$$\boxed{\vdash_H P} \quad \frac{\begin{array}{c}\forall i.\ \cdot \vdash_H \tau_i \qquad \forall i.\ x_1 \colon \tau_1, \dots, x_n \colon \tau_n \vdash_H h_i \colon \tau_i\ \text{hval} \\ \cdot; x_1 \colon \tau_1, \dots, x_n \colon \tau_n \vdash_H e \qquad x_i \neq x_j\ \text{for}\ i \neq j\end{array}}{\vdash_H \texttt{letrec}\ x_1 \mapsto h_1, \dots, x_n \mapsto h_n\ \texttt{in}\ e}$$

Translation from λ^H to λ^A. (variables introduced by the translation must be fresh)

We use the following abbreviations for sequences of declarations:

$$D ::= d_1, \ldots, d_n \qquad \texttt{let}\ d_1, \ldots, d_n\ \texttt{in}\ e \overset{\text{def}}{=} \texttt{let}\ d_1\ \texttt{in}\ \cdots \texttt{let}\ d_n\ \texttt{in}\ e$$

$$\mathcal{A}_{\text{typ}}[\![\alpha]\!] \overset{\text{def}}{=} \alpha$$

$$\mathcal{A}_{\text{typ}}[\![\text{int}]\!] \overset{\text{def}}{=} \text{int}$$

$$\mathcal{A}_{\text{typ}}[\![\forall[\alpha_1, \ldots, \alpha_n](\tau_1, \ldots, \tau_m) \to \text{void}]\!] \overset{\text{def}}{=} \forall[\alpha_1, \ldots, \alpha_n](\mathcal{A}_{\text{typ}}[\![\tau_1]\!], \ldots, \mathcal{A}_{\text{typ}}[\![\tau_m]\!]) \to \text{void}$$

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

$$\mathcal{A}_{\text{typ}}[\![\exists \alpha.\tau]\!] \overset{\text{def}}{=} \exists \alpha.\mathcal{A}_{\text{typ}}[\![\tau]\!]$$

$$\mathcal{A}_{\text{prog}}[\![\texttt{letrec}\ x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\ \texttt{in}\ e]\!]$$
$$\overset{\text{def}}{=} \texttt{letrec}\ x_1 \mapsto \mathcal{A}_{\text{hval}}[\![h_1]\!], \ldots, x_n \mapsto \mathcal{A}_{\text{hval}}[\![h_n]\!]\ \texttt{in}\ \mathcal{A}_{\text{exp}}[\![e]\!]$$

$$\mathcal{A}_{\text{hval}}[\![\text{code}[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e]\!]$$
$$\overset{\text{def}}{=} \text{code}[\alpha_1, \ldots, \alpha_n](x_1 \colon \mathcal{A}_{\text{typ}}[\![\tau_1]\!], \ldots, x_m \colon \mathcal{A}_{\text{typ}}[\![\tau_m]\!]).\mathcal{A}_{\text{exp}}[\![e]\!]$$

$$\mathcal{A}_{\text{exp}}[\![\texttt{let}\ d\ \texttt{in}\ e]\!] \overset{\text{def}}{=} \texttt{let}\ \mathcal{A}_{\text{dec}}[\![d]\!]\ \texttt{in}\ \mathcal{A}_{\text{exp}}[\![e]\!]$$

$$\mathcal{A}_{\text{exp}}[\![v(v_1, \ldots, v_n)]\!] \overset{\text{def}}{=} \texttt{let}\ D, D_1, \ldots D_n\ \texttt{in}\ v'(v'_1, \ldots, v'_n)$$

where $(v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$ and $(v'_1, D_n) = \mathcal{A}_{\text{val}}[\![v_1]\!]$, $\ldots$, $(v'_n, D_n) = \mathcal{A}_{\text{val}}[\![v_n]\!]$

$$\mathcal{A}_{\text{exp}}[\![\text{if0}(v, e_1, e_2)]\!] \overset{\text{def}}{=} \texttt{let}\ D\ \texttt{in}\ \text{if0}(v', \mathcal{A}_{\text{exp}}[\![e_1]\!], \mathcal{A}_{\text{exp}}[\![e_2]\!]) \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{exp}}[\![\textbf{halt}[\tau]v]\!] \overset{\text{def}}{=} \texttt{let}\ D\ \texttt{in}\ \textbf{halt}[\mathcal{A}_{\text{typ}}[\![\tau]\!]]v' \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{dec}}[\![x = v]\!] \overset{\text{def}}{=} D, x = v' \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{dec}}[\![x = \pi_i\ v]\!] \overset{\text{def}}{=} D, x = \pi_i\ v' \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{dec}}[\![x = v_1\ p\ v_2]\!] \overset{\text{def}}{=} D_1, D_2, x = v'_1\ p\ v'_2 \quad \text{where } (v'_i, D_i) = \mathcal{A}_{\text{val}}[\![v_i]\!]$$

$$\mathcal{A}_{\text{dec}}[\![[\alpha, x] = \text{unpack}\, v]\!] \overset{\text{def}}{=} D, [\alpha, x] = \text{unpack}\, v' \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{val}}[\![x^\tau]\!] \overset{\text{def}}{=} (x^\tau, \cdot)$$

$$\mathcal{A}_{\text{val}}[\![i^\tau]\!] \overset{\text{def}}{=} (i^\tau, \cdot)$$

$$\mathcal{A}_{\text{val}}[\![v[\sigma]^\tau]\!] \overset{\text{def}}{=} (v'[\mathcal{A}_{\text{typ}}[\![\sigma]\!]]^{\mathcal{A}_{\text{typ}}[\![\tau]\!]}, D) \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{val}}[\![(\text{pack}\, [\sigma, v]\, \text{as}\, \tau')^\tau]\!] \overset{\text{def}}{=} ((\text{pack}\, [\mathcal{A}_{\text{typ}}[\![\sigma]\!], v']\, \text{as}\, \mathcal{A}_{\text{typ}}[\![\tau']\!])^{\mathcal{A}_{\text{typ}}[\![\tau]\!]}, D) \quad \text{where } (v', D) = \mathcal{A}_{\text{val}}[\![v]\!]$$

$$\mathcal{A}_{\text{val}}[\![\langle u_1^{\tau_1}, \ldots, u_n^{\tau_n} \rangle^\tau]\!] \overset{\text{def}}{=} (y_n^{\mathcal{A}_{\text{typ}}[\![\tau]\!]}, D_1, \ldots, D_n, y_0 = \texttt{malloc}[\mathcal{A}_{\text{typ}}[\![\tau_1]\!], \ldots, \mathcal{A}_{\text{typ}}[\![\tau_n]\!]],$$
$$y_1 = y_0^{\sigma_0}[1] \leftarrow v'_1, \ldots, y_n = {y_{n-1}}^{\sigma_{n-1}}[n] \leftarrow v'_n)$$

where $(v'_i, D_i) = \mathcal{A}_{\text{val}}[\![u_i^{\tau_i}]\!]$ and $\sigma_i = \langle \mathcal{A}_{\text{typ}}[\![\tau_1]\!]^1, \ldots, \mathcal{A}_{\text{typ}}[\![\tau_i]\!]^1, \mathcal{A}_{\text{typ}}[\![\tau_{i+1}]\!]^0, \ldots, \mathcal{A}_{\text{typ}}[\![\tau_n]\!]^0 \rangle$

9.5. λ^A. (differences from λ^H are highlighted)

$$\begin{array}{rll}
\tau, \sigma & ::= \alpha \mid \text{int} \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void} \mid \exists \alpha.\tau & \textit{types} \\
\varphi & ::= 1 \mid 0 & \textit{initialization flags} \\
v & ::= u^\tau \mid ?\tau & \textit{annotated values} \\
u & ::= x \mid i \mid v[\tau] \mid \text{pack}\,[\tau_1, v]\,\text{as}\,\tau_2 & \textit{values} \\
h & ::= \text{code}[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e \mid \langle v_1, \ldots, v_n \rangle & \textit{heap values} \\
p & ::= + \mid - \mid \times & \textit{primitives} \\
d & ::= x = v \mid x = \pi_i\ v \mid x = v_1\ p\ v_2 \mid [\alpha, x] = \text{unpack}\,v & \textit{declarations} \\
 & \mid x = \mathtt{malloc}[\tau_1, \ldots, \tau_n] \mid x = v_1[i] \leftarrow v_2 & \\
e & ::= \mathtt{let}\ d\ \mathtt{in}\ e \mid v(v_1, \ldots, v_m) \mid \text{if0}(v, e_1, e_2) \mid \mathbf{halt}[\tau]v & \textit{terms} \\
P & ::= \mathtt{letrec}\ x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\ \mathtt{in}\ e & \textit{programs} \\
\Delta & ::= \alpha_1 \ldots, \alpha_n & \textit{type contexts} \\
\Gamma & ::= x_1 \colon \tau_1, \ldots, x_n \colon \tau_n & \textit{value contexts}
\end{array}$$

$$\boxed{\Delta \vdash_A \tau} \quad \frac{\text{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_A \tau} \qquad\qquad \boxed{\Delta; \Gamma \vdash_A v \colon \tau} \quad \frac{\Delta; \Gamma \vdash_A u \colon \tau}{\Delta; \Gamma \vdash_A u^\tau \colon \tau}$$

$$\boxed{\Delta; \Gamma \vdash_A u \colon \tau}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_A x \colon \tau} \qquad \frac{}{\Delta; \Gamma \vdash_A i \colon \text{int}} \qquad \frac{\Delta \vdash_A \tau_1 \qquad \Delta; \Gamma \vdash_A v \colon \tau_2[\tau_1/\alpha]}{\Delta; \Gamma \vdash_A \text{pack}\,[\tau_1, v]\,\text{as}\,\exists\alpha.\tau_2 \colon \exists \alpha.\tau_2}$$

$$\frac{\Delta \vdash_A \sigma \qquad \Delta; \Gamma \vdash_A v \colon \forall[\alpha, \beta_1, \ldots, \beta_n].(\tau_1, \ldots, \tau_n) \to \text{void}}{\Delta; \Gamma \vdash_A v[\sigma] \colon \forall[\beta_1, \ldots, \beta_n].(\tau_1[\sigma/\alpha], \ldots, \tau_n[\sigma/\alpha]) \to \text{void}}$$

$$\boxed{\Delta; \Gamma \vdash_A v \colon \tau^\varphi}$$

$$\frac{\Delta; \Gamma \vdash_A v \colon \tau}{\Delta; \Gamma \vdash_A v \colon \tau^\varphi} \qquad\qquad \frac{}{\Delta; \Gamma \vdash_A ?\tau \colon \tau^0}$$

$$\boxed{\Gamma \vdash_A h \colon \tau\ \text{hval}}$$

$$\frac{\forall i.\ \alpha_1, \ldots, \alpha_n \vdash_A \tau_i \qquad \alpha_1, \ldots, \alpha_n; \Gamma, x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash_A e}{\Gamma \vdash_A \text{code}[\alpha_1, \ldots, \alpha_n](x_1 \colon \tau_1, \ldots, x_m \colon \tau_m).e \colon \forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void hval}}$$

$$\frac{\forall i.\ \cdot; \Gamma \vdash_A v_i \colon \tau_i^{\varphi_i}}{\Gamma \vdash_A \langle v_1, \ldots, v_n \rangle \colon \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle\ \text{hval}}$$

$$\boxed{\vdash_A P}$$

$$\frac{\begin{array}{c}\forall i.\ \cdot \vdash_A \tau_i \qquad \forall i.\ x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash_A h_i \colon \tau_i\ \text{hval} \\ \cdot; x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash_A e \qquad x_i \neq x_j\ \text{for}\ i \neq j\end{array}}{\vdash_A \mathtt{letrec}\ x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\ \mathtt{in}\ e}$$

$\boxed{\Delta;\Gamma \vdash_A e}$

$$\frac{\Delta;\Gamma \vdash_A v\colon \tau \qquad \Delta;\Gamma, x\colon \tau \vdash_A e}{\Delta;\Gamma \vdash_A \texttt{let}\ x = v\ \texttt{in}\ e}$$

$$\frac{\Delta;\Gamma \vdash_A v\colon \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_1}\rangle \qquad \Delta;\Gamma, x\colon \tau_i \vdash_A e \qquad 1 \le i \le n \qquad \varphi_i = 1}{\Delta;\Gamma \vdash_A \texttt{let}\ x = \pi_i\ v\ \texttt{in}\ e}$$

$$\frac{\Delta;\Gamma \vdash_A v_1\colon \mathrm{int} \qquad \Delta;\Gamma \vdash_A v_2\colon \mathrm{int} \qquad \Delta;\Gamma, x\colon \mathrm{int} \vdash_A e}{\Delta;\Gamma \vdash_A \texttt{let}\ x = v_1\ p\ v_2\ \texttt{in}\ e}$$

$$\frac{\forall i.\ \Delta;\Gamma \vdash_A v_i\colon \tau_i \qquad \Delta;\Gamma \vdash_A v\colon \forall[].(\tau_1, \ldots, \tau_m) \to \mathrm{void}}{\Delta;\Gamma \vdash_A v(v_1, \ldots, v_m)}$$

$$\frac{\Delta;\Gamma \vdash_A e_1\colon \mathrm{int} \qquad \Delta;\Gamma \vdash_A e_2 \qquad \Delta;\Gamma \vdash_A e_3}{\Delta;\Gamma \vdash_A \mathrm{if0}(v, e_2, e_3)} \qquad \frac{\Delta;\Gamma \vdash_A v\colon \tau}{\Delta;\Gamma \vdash_A \mathbf{halt}[\tau]v}$$

$$\frac{\Delta;\Gamma \vdash_A v\colon \exists\alpha.\tau \qquad \Delta,\alpha;\Gamma, x\colon \tau \vdash_A e}{\Delta;\Gamma \vdash_A \texttt{let}\ [\alpha, x] = \mathrm{unpack}\, v\ \texttt{in}\ e}$$

$$\frac{\forall i.\ \Delta \vdash_A \tau_i \qquad \Delta;\Gamma, x\colon \langle \tau_1^0, \ldots, \tau_n^0\rangle \vdash_A e}{\Delta;\Gamma \vdash_A \texttt{let}\ x = \texttt{malloc}[\tau_1, \ldots, \tau_n]\ \texttt{in}\ e}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash_A v_1\colon \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n}\rangle \qquad \Delta;\Gamma \vdash_A v_2\colon \tau_i \\ \Delta;\Gamma, x\colon \langle \tau_1^{\varphi_1}, \ldots, \tau_i^1, \ldots, \tau_n^{\varphi_n}\rangle \vdash_A e \qquad 1 \le i \le n\end{array}}{\Delta;\Gamma \vdash_A \texttt{let}\ x = v_1[i] \leftarrow v_2\ \texttt{in}\ e}$$

Translation from λ^A to TAL.

We make use of mappings γ from λ^A variables to TAL registers and addresses:

$$\gamma ::= \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$$

$$
\begin{aligned}
\mathcal{T}_{\text{typ}}[\![\alpha]\!] &\overset{\text{def}}{=} \alpha \\
\mathcal{T}_{\text{typ}}[\![\text{int}]\!] &\overset{\text{def}}{=} \text{int} \\
\mathcal{T}_{\text{typ}}[\![\forall[\alpha_1, \ldots, \alpha_n].(\tau_1, \ldots, \tau_m) \to \text{void}]\!] &\overset{\text{def}}{=} \forall[\alpha_1, \ldots, \alpha_n].(\texttt{r1}\colon \mathcal{T}_{\text{typ}}[\![\tau_1]\!], \ldots, \texttt{r}n\colon \mathcal{T}_{\text{typ}}[\![\tau_m]\!]) \\
\mathcal{T}_{\text{typ}}[\![\langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle]\!] &\overset{\text{def}}{=} \langle \mathcal{T}_{\text{typ}}[\![\tau_1]\!]^{\varphi_1}, \ldots, \mathcal{T}_{\text{typ}}[\![\tau_n]\!]^{\varphi_n} \rangle \\
\mathcal{T}_{\text{typ}}[\![\exists \alpha.\tau]\!] &\overset{\text{def}}{=} \exists \alpha.\mathcal{T}_{\text{typ}}[\![\tau]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}_{\text{prog}}[\![\texttt{letrec}\ x_1 \mapsto h_1, \ldots, x_n \mapsto h_n\ \texttt{in}\ e]\!] \overset{\text{def}}{=} (H, \{\}, I) \\
&\quad \text{where}\quad \begin{aligned}[t]
\gamma &= \{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \\
(h_i', H_i) &= \mathcal{T}_{\text{hval}}[\![h_i]\!]\gamma \\
(I, H_{\text{exp}}) &= \mathcal{T}_{\text{exp}}[\![e]\!]\gamma; \cdot; \cdot \\
H_{\text{root}} &= \{\ell_1 \mapsto h_1', \ldots, \ell_n \mapsto h_n'\} \\
H &= H_{\text{root}} H_1 \cdots H_n H_{\text{exp}} \\
\ell_i &\ \text{distinct}
\end{aligned}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{T}_{\text{hval}}[\![\text{code}[\alpha_1, \ldots, \alpha_n](x_1\colon \tau_1, \ldots x_n\colon \tau_n).e]\!]\gamma \overset{\text{def}}{=} (\text{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I, H) \\
&\quad \text{where}\quad \begin{aligned}[t]
\Gamma &= \{\texttt{r1}\colon \mathcal{T}_{\text{typ}}[\![\tau_1]\!], \ldots, \texttt{r}n\colon \mathcal{T}_{\text{typ}}[\![\tau_n]\!]\} \\
\gamma' &= \gamma\{x_1 \mapsto \texttt{r1}, \ldots, x_n \mapsto \texttt{r}n\} \\
(I, H) &= \mathcal{T}_{\text{exp}}[\![e]\!]\gamma'; \alpha_1, \ldots, \alpha_n; \Gamma
\end{aligned}
\end{aligned}
$$

$$
\mathcal{T}_{\text{hval}}[\![\langle v_1, \ldots, v_n \rangle]\!]\gamma \overset{\text{def}}{=} (\langle \mathcal{T}_{\text{val}}[\![v_1]\!]\gamma, \ldots, \mathcal{T}_{\text{val}}[\![v_n]\!]\gamma \rangle, \{\})
$$

$$
\begin{aligned}
\mathcal{T}_{\text{val}}[\![x^\tau]\!]\gamma &\overset{\text{def}}{=} \gamma(x) \\
\mathcal{T}_{\text{val}}[\![i^\tau]\!]\gamma &\overset{\text{def}}{=} i \\
\mathcal{T}_{\text{val}}[\![v[\sigma]^\tau]\!]\gamma &\overset{\text{def}}{=} (\mathcal{T}_{\text{val}}[\![v]\!]\gamma)[\mathcal{T}_{\text{typ}}[\![\sigma]\!]] \\
\mathcal{T}_{\text{val}}[\![(\text{pack}\,[\sigma, v]\,\text{as}\,\tau')^\tau]\!] &\overset{\text{def}}{=} \text{pack}\,[\mathcal{T}_{\text{typ}}[\![\sigma]\!], \mathcal{T}_{\text{val}}[\![v]\!]\gamma]\,\text{as}\,\mathcal{T}_{\text{typ}}[\![\tau']\!]
\end{aligned}
$$

$$\mathcal{T}_{\text{exp}}\llbracket \texttt{let } x = u^{\tau} \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{mov}\, r, \mathcal{T}_{\text{val}}\llbracket u^{\tau}\rrbracket\gamma; I), H)$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta;\Gamma\{r\colon \mathcal{T}_{\text{typ}}\llbracket\tau\rrbracket\}$ and r is fresh

$$\mathcal{T}_{\text{exp}}\llbracket \texttt{let } x = \pi_i\ u^{\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle} \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{mov}\, r, \mathcal{T}_{\text{val}}\llbracket u^{\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}\rrbracket\gamma; \texttt{ld}\, r, r[i-1]; I), H)$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta;\Gamma\{r\colon \mathcal{T}_{\text{typ}}\llbracket\tau_i\rrbracket\}$ and r is fresh

$$\mathcal{T}_{\text{exp}}\llbracket \texttt{let } x = v_1\ p\ v_2 \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{mov}\, r, \mathcal{T}_{\text{val}}\llbracket v_1\rrbracket\gamma; \text{op}(p)\ r, r, \mathcal{T}_{\text{val}}\llbracket v_2\rrbracket\gamma; I), H)$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta;\Gamma\{r\colon \text{int}\}$,
$\text{op}(+) = \texttt{add}, \text{op}(-) = \texttt{sub}, \text{op}(\times) = \texttt{mul}$
and r is fresh

$$\mathcal{T}_{\text{exp}}\llbracket \texttt{let } [\alpha, x] = \text{unpack}\, u^{\exists\alpha.\tau} \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{unpack}[\alpha, r], \mathcal{T}_{\text{val}}\llbracket u^{\exists\alpha.\tau}\rrbracket\gamma; I), H)$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta\{\alpha\};\Gamma\{r\colon \mathcal{T}_{\text{typ}}\llbracket\tau\rrbracket\}$ and r is fresh

$$\mathcal{T}_{\text{exp}}\llbracket \texttt{let malloc}[\tau_1,\ldots,\tau_n] \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{malloc}\, r[\mathcal{T}_{\text{typ}}\llbracket\tau_1\rrbracket,\ldots,\mathcal{T}_{\text{typ}}\llbracket\tau_1\rrbracket]; I), H)$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta;\Gamma\{r\colon \langle\mathcal{T}_{\text{typ}}\llbracket\tau_1\rrbracket^0,\ldots,\mathcal{T}_{\text{typ}}\llbracket\tau_1\rrbracket^0\rangle\}$ and r is fresh

$$\begin{aligned}\mathcal{T}_{\text{exp}}\llbracket \texttt{let } x = u^{\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}[i] \leftarrow v \texttt{ in } e\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((&\texttt{mov}\, r, \mathcal{T}_{\text{val}}\llbracket u^{\langle\tau_1^{\varphi_1},\ldots,\tau_n^{\varphi_n}\rangle}\rrbracket\gamma;\\ &\texttt{mov}\, r', \mathcal{T}_{\text{val}}\llbracket v\rrbracket\gamma;\\ &\texttt{st}\, r[i-1], r'; I), H)\end{aligned}$$

where $(I, H) = \mathcal{T}_{\text{exp}}\llbracket e\rrbracket\gamma\{x \mapsto r\};\Delta;\Gamma\{r\colon \mathcal{T}_{\text{typ}}\llbracket\langle\tau_1^{\varphi_1},\ldots,\tau_{i-1}^{\varphi_{i-1}},\tau_i^1,\tau_{i+1}^{\varphi_{i+1}},\ldots\tau_n^{\varphi_n}\rangle\rrbracket\}$
and r, r' are fresh

$$\begin{aligned}\mathcal{T}_{\text{exp}}\llbracket v(v_1,\ldots,v_n)\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((&\texttt{mov}\, r'_0, \mathcal{T}_{\text{val}}\llbracket v\rrbracket\gamma;\\ &\texttt{mov}\, r'_1, \mathcal{T}_{\text{val}}\llbracket v_1\rrbracket\gamma;\ldots;\texttt{mov}\, r'_n, \mathcal{T}_{\text{val}}\llbracket v_n\rrbracket\gamma;\\ &\texttt{mov}\, \texttt{r1}, r'_1;\ldots;\texttt{mov}\, \texttt{r}n, r'_n; I), \{\})\end{aligned}$$

where the r'_i are fresh and disjoint from $\{\texttt{r1},\ldots,\texttt{r}n\}$.

$$\mathcal{T}_{\text{exp}}\llbracket \text{if0}(v, e_1, e_2)\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{mov}\, r, \mathcal{T}_{\text{val}}\llbracket v\rrbracket\gamma; \texttt{bnz}\, r, \ell[\Delta]; I_1), H_1 H_2\{\ell \mapsto h\})$$

where $(I_1, H_1) = \mathcal{T}_{\text{exp}}\llbracket e_1\rrbracket\gamma;\Delta;\Gamma$
$(I_2, H_2) = \mathcal{T}_{\text{exp}}\llbracket e_2\rrbracket\gamma;\Delta;\Gamma$
$h = \text{code}[\Delta]\Gamma.I_2$
and ℓ, r are fresh

$$\mathcal{T}_{\text{exp}}\llbracket \textbf{halt}[\tau]v\rrbracket\gamma;\Delta;\Gamma \quad \stackrel{\text{def}}{=} ((\texttt{mov}\, \texttt{r1}, \mathcal{T}_{\text{val}}\llbracket v\rrbracket\gamma; \textbf{halt}[\mathcal{T}_{\text{typ}}\llbracket\tau\rrbracket]), \{\})$$

9.6. **TAL.**

$$
\begin{array}{rcll}
\tau, \sigma & ::= & \alpha \mid \mathrm{int} \mid \forall[\alpha_1, \ldots, \alpha_n].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists \alpha.\tau & \textit{types} \\
\varphi & ::= & 1 \mid 0 & \textit{initialization flags} \\
\Psi & ::= & \{\ell_1 \colon \tau_1, \ldots, \ell_n \colon \tau_n\} & \textit{heap types} \\
\Gamma & ::= & \{r_1 \colon \tau_1, \ldots, r_n \colon \tau_n\} & \textit{register file types} \\
\Delta & ::= & \alpha_1 \ldots, \alpha_n & \textit{type contexts} \\
\\
r & ::= & \mathtt{r1} \mid \mathtt{r2} \mid \mathtt{r3} \mid \cdots & \textit{registers} \\
w & ::= & \ell \mid i \mid ?\tau \mid w[\tau] \mid \mathrm{pack}\,[\tau_1, w]\,\mathrm{as}\,\tau_2 & \textit{word values} \\
v & ::= & r \mid w \mid v[\tau] \mid \mathrm{pack}\,[\tau_1, v]\,\mathrm{as}\,\tau_2 & \textit{small values} \\
h & ::= & \langle v_1, \ldots, v_n \rangle \mid \mathrm{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I & \textit{heap values} \\
H & ::= & \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} & \textit{heaps} \\
R & ::= & \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\} & \textit{register files} \\
\\
\iota & ::= & \mathtt{add}\, r_d, r_s, v \mid \mathtt{mul}\, r_d, r_s, v \mid \mathtt{sub}\, r_d, r_s, v \mid \mathtt{bnz}\, r, v & \textit{instructions} \\
 & & \mid \mathtt{ld}\, r_d, r_s[i] \mid \mathtt{st}\, r_d[i].r_s \mid \mathtt{mov}\, r_d, v & \\
 & & \mid \mathtt{malloc}\, r_d[\tau_1, \ldots, \tau_n] \mid \mathtt{unpack}[\alpha, r_d], v & \\
I & ::= & \iota; I \mid \mathtt{jmp}\, v \mid \mathtt{halt}[\tau] & \textit{instruction sequences} \\
P & ::= & (H, R, I) & \textit{programs}
\end{array}
$$

$$
\boxed{\Delta \vdash_{\mathrm{TAL}} \tau} \quad \frac{\mathrm{ftv}(\tau) \subseteq \Delta}{\Delta \vdash_{\mathrm{TAL}} \tau} \qquad\qquad \boxed{\vdash_{\mathrm{TAL}} \Psi} \quad \frac{\forall i.\ \cdot \vdash_{\mathrm{TAL}} \tau_i}{\vdash_{\mathrm{TAL}} \{\ell_1 \colon \tau_1, \ldots, \ell_n \colon \tau_n\}}
$$

$$
\boxed{\Delta \vdash_{\mathrm{TAL}} \Gamma} \quad \frac{\forall i.\ \Delta \vdash_{\mathrm{TAL}} \tau_i}{\Delta \vdash_{\mathrm{TAL}} \{r_1 \colon \tau_1, \ldots, r_n \colon \tau_n\}}
$$

$$
\boxed{\Delta \vdash_{\mathrm{TAL}} \tau_1 \leq \tau_2}
$$

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

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

$$
\boxed{\Delta \vdash_{\mathrm{TAL}} \Gamma_1 \leq \Gamma_2} \quad \frac{\forall i \leq m.\ \Delta \vdash_{\mathrm{TAL}} \tau_i \qquad m \geq n}{\Delta \vdash_{\mathrm{TAL}} \{r_1 \colon \tau_1, \ldots, r_n \colon \tau_m\} \leq \{r_1 \colon \tau_1, \ldots, r_n \colon \tau_n\}}
$$

$$
\boxed{\vdash_{\mathrm{TAL}} P} \quad \frac{\vdash_{\mathrm{TAL}} H \colon \Psi \qquad \Psi \vdash_{\mathrm{TAL}} R \colon \Gamma \qquad \Psi; \cdot; \Gamma \vdash_{\mathrm{TAL}} I}{\vdash_{\mathrm{TAL}} (H, R, I)}
$$

$$
\boxed{\vdash_{\mathrm{TAL}} H \colon \Psi} \quad \frac{\Psi = \{\ell_1 \colon \tau_1, \ldots, \ell_n \colon \tau_n\} \qquad \vdash_{\mathrm{TAL}} \Psi \qquad \forall i.\ \Psi \vdash_{\mathrm{TAL}} h_i \colon \tau_i\ \mathrm{hval}}{\vdash_{\mathrm{TAL}} \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \colon \Psi}
$$

$$
\boxed{\Psi \vdash_{\mathrm{TAL}} R \colon \Gamma} \quad \frac{\forall i \leq m.\ \Psi; \cdot \vdash_{\mathrm{TAL}} w_i \colon \tau_i\ \mathrm{wval} \qquad m \geq n}{\Psi \vdash_{\mathrm{TAL}} \{r_1 \mapsto w_1, \ldots, r_m \mapsto w_m\} \colon \{r_1 \colon \tau_1, \ldots, r_n \colon \tau_n\}}
$$

$\boxed{\Psi \vdash_{\text{TAL}} h \colon \tau \text{ hval}}$

$$\frac{\forall i.\ \Psi; \cdot \vdash_{\text{TAL}} w_i \colon \tau_i^{\varphi_i}}{\Psi \vdash_{\text{TAL}} \langle w_1, \ldots, w_n \rangle \colon \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \text{ hval}}$$

$$\frac{\alpha_1, \ldots, \alpha_n \vdash_{\text{TAL}} \Gamma \qquad \Psi; \alpha_1, \ldots, \alpha_n; \Gamma \vdash_{\text{TAL}} I}{\Psi \vdash_{\text{TAL}} \text{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I \colon \forall[\alpha_1, \ldots, \alpha_n].\Gamma \text{ hval}}$$

$\boxed{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau \text{ wval}}$

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

$$\frac{\Delta \vdash_{\text{TAL}} \tau \qquad \Psi; \Delta \vdash_{\text{TAL}} w \colon \forall[\alpha, \beta_1, \ldots, \beta_n].\Gamma \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} w[\tau] \colon \forall[\beta_1, \ldots, \beta_n].\Gamma[\tau/\alpha] \text{ wval}}$$

$$\frac{\Delta \vdash_{\text{TAL}} \tau \qquad \Psi; \Delta \vdash_{\text{TAL}} w \colon \tau'[\tau/\alpha] \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} \text{pack}\,[\tau, w]\,\text{as}\,\exists\alpha.\tau' \colon \exists\alpha.\tau' \text{ wval}}$$

$\boxed{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau^{\varphi}}$

$$\frac{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau \text{ wval}}{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau^{\varphi}} \qquad \frac{\Delta \vdash_{\text{TAL}} \tau}{\Psi; \Delta \vdash_{\text{TAL}} ?\tau \colon \tau^0}$$

$\boxed{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \tau}$

$$\frac{\Gamma(r) = \tau}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} r \colon \tau} \qquad \frac{\Psi; \Delta \vdash_{\text{TAL}} w \colon \tau \text{ wval}}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} w \colon \tau}$$

$$\frac{\Delta \vdash_{\text{TAL}} \tau \qquad \Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \forall[\alpha, \beta_1, \ldots, \beta_n].\Gamma'}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} v[\tau] \colon \forall[\beta_1, \ldots, \beta_n].\Gamma'[\tau/\alpha]}$$

$$\frac{\Delta \vdash_{\text{TAL}} \tau \qquad \Psi; \Delta; \Gamma \vdash_{\text{TAL}} v \colon \tau'[\tau/\alpha]}{\Psi; \Delta; \Gamma \vdash_{\text{TAL}} \text{pack}\,[\tau, v]\,\text{as}\,\exists\alpha.\tau' \colon \exists\alpha.\tau'}$$

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

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v \colon \mathrm{int} \qquad \begin{array}{c}\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_s \colon \mathrm{int} \\ \Psi; \Delta; \Gamma\{r_d \colon \mathrm{int}\} \vdash_{\mathrm{TAL}} I\end{array} \qquad \mathit{arith} \in \{\mathtt{add}, \mathtt{mul}, \mathtt{sub}\}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathit{arith}\, r_d, r_s, v; I}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v \colon \forall[].\Gamma' \qquad \begin{array}{c}\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r \colon \mathrm{int} \\ \Delta \vdash_{\mathrm{TAL}} \Gamma \leq \Gamma'\end{array} \qquad \Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} I}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{bnz}\, r, v; I}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_s \colon \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \Psi; \Delta; \Gamma\{r_d \colon \tau_i\} \vdash_{\mathrm{TAL}} I \qquad \varphi_i = 1 \qquad 0 \leq i \leq n\end{array}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{ld}\, r_d, r_s[i]; I}$$

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

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v \colon \tau \qquad \Psi; \Delta; \Gamma\{r_d \colon \tau\} \vdash_{\mathrm{TAL}} I}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{mov}\, r_d, v; I}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} r_d \colon \langle \tau_0^{\varphi_0}, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle \\ \Psi; \Delta; \vdash_{\mathrm{TAL}} r_s \colon \tau_i \qquad \Psi; \Delta; \Gamma\{r_d \colon \langle \tau_0^{\varphi_0}, \ldots, \tau_i^1, \ldots, \tau_{n-1}^{\varphi_{n-1}} \rangle\} \vdash_{\mathrm{TAL}} I\end{array}}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{st}\, r_d[i], r_s; I}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v \colon \exists \alpha.\tau \qquad \Psi; \Delta, \alpha; \Gamma\{r_d \colon \tau\} \vdash_{\mathrm{TAL}} I}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{unpack}[\alpha, r_d], v; I}$$

$$\frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} v \colon \forall[].\Gamma' \qquad \Delta \vdash_{\mathrm{TAL}} \Gamma \leq \Gamma'}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{jmp}\, v} \qquad \frac{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathtt{r1} \colon \tau}{\Psi; \Delta; \Gamma \vdash_{\mathrm{TAL}} \mathbf{halt}[\tau]}$$

Operational Semantics. $\boxed{P \longmapsto P}$

$$(H, R, \texttt{add}\, r_d, r_s, v; I) \longmapsto (H, R\{r_d \mapsto (R(r_s) + \hat{R}(v))\}, I)$$
$$(H, R, \texttt{mul}\, r_d, r_s, v; I) \longmapsto (H, R\{r_d \mapsto (R(r_s) \times \hat{R}(v))\}, I)$$
$$(H, R, \texttt{sub}\, r_d, r_s, v; I) \longmapsto (H, R\{r_d \mapsto (R(r_s) - \hat{R}(v))\}, I)$$

$$\begin{aligned}
&(H, R, \texttt{bnz}\, r, v; I) \longmapsto (H, R, I) \\
&\text{when } R(r) = 0 \\
&(H, R, \texttt{bnz}\, r, v; I) \longmapsto (H, R, I'[\tau_1/\alpha_1] \cdots [\tau_n/\alpha_n]) \\
&\qquad\qquad\qquad\qquad \text{where } \hat{R}(V) = \ell[\tau_1, \ldots, \tau_n] \\
&\qquad\qquad\qquad\qquad \text{and } H(\ell) = \text{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I' \\
&\text{when } R(r) = i \text{ and } i \neq 0
\end{aligned}$$

$$\begin{aligned}
(H, R, \texttt{jmp}\, v) &\longmapsto (H, R, I'[\tau_1/\alpha_1] \cdots [\tau_n/\alpha_n]) \\
&\text{where } \hat{R}(V) = \ell[\tau_1, \ldots, \tau_n] \\
&\text{and } H(\ell) = \text{code}[\alpha_1, \ldots, \alpha_n]\Gamma.I'
\end{aligned}$$

$$\begin{aligned}
(H, R, \texttt{ld}\, r_d, r_s[i]; I) &\longmapsto (H, R\{r_d \mapsto w_i\}, I) \\
&\text{where } R(r_s) = \ell \text{ and } H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle \\
&\text{with } 0 \le i < n \\
(H, R, \texttt{st}\, r_d[i], r_s; I) &\longmapsto (H\{\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, w_{n-1}\rangle\}, R, I) \\
&\text{where } R(r_d) = \ell \text{ and } H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle \\
&\text{with } 0 \le i < n
\end{aligned}$$

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

$$\begin{aligned}
(H, R, \texttt{malloc}\, r_d[\tau_1, \ldots, \tau_n]; I) &\longmapsto (H\{\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n\rangle\}, R\{r_d \mapsto \ell\}, I) \\
&\text{where } \ell \notin H
\end{aligned}$$

$$\begin{aligned}
(H, R, \texttt{unpack}[\alpha, r_d], v; I) &\longmapsto (H, R\{r_d \mapsto w\}, I[\tau/\alpha]) \\
&\text{where } \hat{R}(v) = \text{pack}\,[\tau, w]\,\text{as}\,\tau'
\end{aligned}$$

Where

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