Metamath C technical appendix

MARIO CARNEIRO, Carnegie Mellon University

1 INTRODUCTION

This is an informal development of the theory behind the Metamath C language: the syntax and separation logic, as well as the lowering map to x86. For now, this is just a set of notes for the actual compiler. (Informal is a relative word, of course, and this is quite formally precise from a mathematician's point of view. But it is not mechanized.)

2 SYNTAX

The syntax of MMC programs, after type inference, is given by the following (incomplete) grammar:

$\alpha, x, h, k \in$ Ident ::=	identifiers	
$s \in$ Size ::=	$8 \mid 16 \mid 32 \mid 64 \mid \infty$	integer bit size
$t \in$ TuplePattern ::=	$\_ \mid x \mid \boxed{x}$	ignored, variable, ghost variable
	$\mid t : \tau \mid \langle \overline{t} \rangle$	type ascription, tuple
$R \in$ Arg ::=	$x : \tau \mid \boxed{x} : \tau \mid h : A$	regular/ghost/proof argument
$\tau \in$ Type ::=	α	type variable reference
	$\mid \lvert\alpha\rvert$	moved type variable
	$\mid \mathbf{0} \mid \mathbf{1} \mid$ bool	void, unit, booleans
	$\mid \mathbb{N}_s \mid \mathbb{Z}_s$	unsigned and signed integers of different sizes
	$\mid \bigcap \overline{\tau} \mid \bigcup \overline{\tau}$	intersection type, (undiscriminated) union type
	$\mid \ast \overline{\tau} \mid \sum \overline{R}$	tuple type, structure (dependent tuple) type
	$\mid S(\overline{\tau}, \overline{pe})$	user-defined type

$A \in$ Prop ::=	pe	assert that a boolean value is true
	$\mid \top \mid \bot \mid$ emp	true, false, empty heap
	$\mid \forall x : \tau,\ A \mid \exists x : \tau,\ A$	universal, existential quantification
	$\mid A_1 \to A_2 \mid \neg A$	implication, negation
	$\mid A_1 \wedge A_2 \mid A_1 \vee A_2$	conjunction, disjunction
	$\mid A_1 * A_2 \mid A_1 \mathrel{-\!\!*} A_2$	separating conjunction and implication
	$\mid pe \mapsto pe'$	points-to assertion
	$\mid \boxed{x : \tau}$	typing assertion

Author's address: Mario Carneiro, Carnegie Mellon University.

$pe \in \text{PureExpr} ::=$	(the first half of Expr below)	pure expressions
$e \in \text{Expr} ::=$	x	variable reference
	$\mid () \mid \mathsf{true} \mid \mathsf{false} \mid n$	constants
	$\mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e$	logical AND, OR, NOT
	$\mid e_1 \,\&\, e_2 \mid e_1 \mid e_2 \mid !_s\, e$	bitwise AND, OR, NOT
	$\mid e_1 + e_2 \mid e_1 * e_2 \mid -e$	addition, multiplication, negation
	$\mid e_1 < e_2 \mid e_1 \le e_2 \mid e_1 = e_2$	equalities and inequalities
	$\mid \mathsf{if}\ h^? : e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	conditionals
	$\mid \langle \overline{e} \rangle$	tuple
	$\mid f(\overline{e})$	(pure) function call
	$\mid \mathsf{let}\ h^? := t := e_1\ \mathsf{in}\ e_2$	assignment to a regular variable
	$\mid \mathsf{let}\ t := p\ \mathsf{in}\ e$	assignment to a hypothesis
	$\mid \eta \leftarrow pe;\ e$	move assignment
	$\mid F(\overline{e})$	procedure call
	$\mid \mathsf{unreachable}\ p$	unreachable statement
	$\mid \mathsf{return}\ \overline{e}$	procedure return
	$\mid \mathsf{label}\ \overline{k(\overline{R}) := e}\ \mathsf{in}\ e'$	local mutual tail recursion
	$\mid \mathsf{goto}\ k(\overline{e})$	local tail call
$\eta \in \text{Place} ::=$	x	variable reference

$p \in \text{Proof} ::=$	$\mathsf{entail}\ \overline{p}\ q$	entailment proof
	$\mid \mathsf{assert}\ pe$	assertion
	$\mid \mathsf{typeof}\ pe$	take the type of a variable
$q \in \text{PureProof} ::=$	$\ldots$	MM0 proofs
$it \in \text{Item} ::=$	$\mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau$	type declaration
	$\mid \mathsf{const}\ t := e$	constant declaration
	$\mid \mathsf{global}\ t := e$	global variable declaration
	$\mid \mathsf{func}\ f(\overline{R}) : \overline{R} := e$	function declaration
	$\mid \mathsf{proc}\ f(\overline{R}) : \overline{R} := e$	procedure declaration

Missing elements of the grammar include:

- Switch statements, which are desugared to if statements.
- Raw MM0 formulas can be lifted to the 'Prop' type.
- Raw MM0 values can be lifted into $\mathbb{N}_\infty$ and $\mathbb{Z}_\infty$.
- There are more operations for working with pointers and arrays. These are discussed in section 3.8.

- There are operations for moving between typed values and hypotheses, which will be discussed later.
- There are also while loops and for loops, but we will focus on the general control flow of label and goto.

Language items that are considered but not present (yet) in the language include:

- Functions and procedures cannot be generic over type and propositional variables. (In fact there are no propositional variables in the language, only the type Prop of propositional expressions.) A generic propositional variable is used internally to model the frame rule but it is not available to user code.
- Recursive and mutually recursive function support is currently very limited.

Most of the constructs are likely familiar from other languages. We will call some attention to the more unusual features:

- Ghost variables $\boxed{x}$ are used to represent computationally irrelevant data. They can be manipulated just like regular variables, but they must not appear on the data path during code generation. We will use x^γ to generalize over ghost and non-ghost variables, where $\gamma = \bot$ means this is a ghost variable and $\gamma = \top$ means it is not. We use $\gamma' \le \gamma$ to mean that γ is "more computationally relevant" than γ', i.e. if x^γ is ghost then $x^{\gamma'}$ is too.
- The $!_s\ n$ operation performs the mathematical function $2^s - n - 1$, taking $2^\infty = 0$ so that $!_\infty\ n = -n - 1$. $!_s\ n$ is used for bitwise negation of unsigned integers, and $!_\infty\ n$ is used for bitwise negation of signed integers (even those of finite width).
- The assignment operator $\mathsf{let}\ h^? := t := e_1\ \mathsf{in}\ e_2$ assigns the variables of t to the result of e_1, but here it should be understood as a new binding, or shadowing declaration, rather than a reassignment to an existing variable. Even array assignments will be desugared into pure-functional update operations.
  Here $h^?$ denotes that the hypothesis binding h is optional; we will write the version with no hypothesis binding as $\mathsf{let}\ t := e_1\ \mathsf{in}\ e_2$. The version that has a hypothesis binding requires that e_1 is pure.
  The concrete version of the assignment operator also contains a "with $x \to y$" clause, but this only renames variables in the source (which is to say, it changes the mapping of source names to internal names) and so is not relevant for the theoretical presentation here.
- The operator $x \leftarrow pe;\ e$ is the primitive for mutation of the variables in the context. Intuitively, it can be thought as moving pe into x, but it has no effect on the type context, and is only used to coordinate data flow. In the grammar the left hand side is generalized to a type of "places" (a.k.a lvalues), but for now these can only be variable references. For example,

this:	has the same effect as:	which we can α-rename to:
$\mathsf{let}\ x := 1\ \mathsf{in}$	$\mathsf{let}\ x := 1\ \mathsf{in}$	$\mathsf{let}\ x := 1\ \mathsf{in}$
$\mathsf{let}\ y :=$	$\mathsf{let}\ \langle x, y\rangle :=$	$\mathsf{let}\ \langle x', y\rangle :=$
$\quad x \leftarrow x + 1;$	$\quad \mathsf{let}\ x := x + 1\ \mathsf{in}$	$\quad \mathsf{let}\ x' := x + 1\ \mathsf{in}$
$\quad -x\ \mathsf{in}$	$\quad \langle x, -x\rangle\ \mathsf{in}$	$\quad \langle x', -x'\rangle\ \mathsf{in}$
$e(x, y)$	$e(x, y)$	$e(x', y)$

  The surface syntax uses a combination of mut declarations and shadowing let bindings to signal that a move assignment is desired; move is the desugared form. One can view this as one part of a phi expression in SSA form.

- The expression label $\overline{k(\overline{R}) := e \text{ in } e'}$ is similar in behavior to a recursive let binding such as those found in functional languages, but the $\overline{k}$ are all continuations, which is to say they do not return to the caller when using goto $l(\overline{e})$, which is how we ensure that they can be compiled to plain label and goto at the machine code level.
- The typeof *pe* operator "moves" a value $x : \tau$ and returns a fact $\boxed{x : \tau}$ that asserts ownership of the resources of x. See 3.2.

3 TYPING

3.1 Overview

The main typing judgments are:

- $\Gamma \vdash t : \tau \Rightarrow \overline{R}$ and $\Gamma \vdash t : A \Rightarrow \overline{R}$
  type a tuple pattern against a value of type τ or A, producing additional hypotheses $\overline{R}$ that will enter the context
- $\Gamma \vdash \tau$ type
  determines that a type τ is a valid type in the current context
- $\Gamma \vdash A$ prop
  determines that A is a valid separating proposition in the current context
- $\Gamma \vdash R$ arg
  determines that R is a valid argument extending the current context
- $\Gamma \vdash e : \tau \dashv \Gamma'$
  determines that e is a valid expression of type τ, which modifies the context to Γ'. In the special case where $\Gamma' = \Gamma$, we will write $\Gamma \vdash e : \tau$ instead.
- $\Gamma \vdash pe : \tau$
  This is only a specialization of the $\Gamma \vdash e : \tau$ judgment, because pure expressions do not change the context.
- $\Gamma \vdash p : A \dashv \Gamma'$
  determines that p is a proof of A, which modifies the context to Γ'. In the special case where $\Gamma' = \Gamma$, we will write $\Gamma \vdash p : A$ instead.
- $\Gamma \Rightarrow \Gamma'$
  an auxiliary judgment for applying pending mutations to the context.
- $\Gamma \vdash e : \tau \dashv\!\Rightarrow \Gamma'$ is defined to mean $\Gamma \vdash e : \tau \dashv \Gamma_1 \ \wedge \ \Gamma_1 \Rightarrow \Gamma'$ for some Γ_1.
- $\Gamma \vdash it$ ok
  The top level item typing judgment

Central to all of these judgments is the context Γ, which consists of:

- The global environment of previously declared items, including in particular a record $\mathsf{self}(\bar{R}) : \bar{S}$ recording the type of the function being typechecked (if a function/procedure is being checked). This doesn't change during expression typing.
- A list of type variables $\overline{\alpha}$. This is only nonempty when type checking a type declaration.
- A list of declared jump targets $\overline{k(\Gamma', \bar{R})}$, including a special jump target $\mathsf{return}(\bar{R})$ where $\bar{R}$ is the declared return type. The Γ' in each jump target is the context required for that jump to typecheck; it lies somewhere between the initial context Γ at the point of the label, and the moved-out context $|\Gamma|$.
- A list of regular variables, ghost variables, and hypotheses $\overline{R}$ with their types.
- (Used only in non-pure expression and proofs:) An unordered map of mutation records of the form $x \leftarrow pe : \tau$ where x is already in the context and $\Gamma \vdash pe : \tau$. These represent mutations to elements of the context that need to be propagated forward along the control flow.

The type variables don't depend on anything and cannot be introduced in the middle of an item, so these can be assumed to come first, but jump targets can depend on regular variables. We use the notation $\Gamma, \overline{k(\bar{R})}$ and $\Gamma, \overline{R}$ to denote extension of the context with a list of jump targets or variables, respectively, and $\Gamma, x \leftarrow pe : \tau$ to denote the insertion of $x \leftarrow pe : \tau$ into the list of mutations, replacing $x \leftarrow pe' : \tau'$ if it is present.

3.2 Moving types

The last essential element to understand the typing rules is the "moved" modality on types and propositions, denoted $|\tau|$ or $|A|$. For separating propositions this is also known as the persistence modality, and it represents what is left of a proposition after all the "ownership" is removed from it. We use moved types to represent a value that has been accessed. This satisfies the axioms $||\tau|| = |\tau|$ and $A \Leftrightarrow A * |A|$. We extend this to arbitrary arguments and contexts $|R|$ and $|\Gamma|$ by applying the modality to all contained types and propositions.

A type/proposition is called "copy" or persistent if $|\tau| = \tau$, and is denoted τ copy.

The moved modality is defined like so:

$$
\begin{aligned}
&|\alpha|, \mathbf{0}, \mathbf{1}, \text{bool}, \mathbb{N}_s, \mathbb{Z}_s \text{ copy} \\
&|\alpha| = |\alpha| \qquad \text{(that is, } \alpha \text{ maps to } |\alpha|) \\
&|\bigcap \overline{\tau}| = \bigcap \overline{|\tau|} \\
&|\bigcup \overline{\tau}| = \bigcup \overline{|\tau|} \\
&|\ast \overline{\tau}| = \ast \overline{|\tau|} \\
&|\sum \overline{\tau}| = \sum \overline{|\tau|} \\
&\left|S(\overline{\tau}, \overline{pe})\right| = [S](\overline{\tau}, \overline{pe}) \qquad \text{(that is, the effect of moving } S \text{ is precalculated)}
\end{aligned}
$$

There are no interesting cases among the types presented here. When we get to pointer types in section 3.8 we will see that $|\&^{\mathbf{own}}\tau| = \left|\&^{\mathbf{mut}}\tau\right| = \mathbb{N}_{64}$, so pointers become "mere integers" after they are moved away. (Note, however, that they actually retain their original types for type inference purposes; that is, the typechecker remembers that they have type $|\&^{\mathbf{own}}\tau|$ in order to determine the type that would result from dereferencing the pointer, if it were still valid.)

For propositions, the effect is more dramatic:

$$
\begin{aligned}
&pe, \top, \bot, \text{emp} \text{ copy} \\
&|\forall x : \tau,\ A| = \begin{cases} \forall x : \tau,\ |A| & \text{if } \tau \text{ copy} \\ \text{emp} & o.w. \end{cases} \\
&|\exists x : \tau,\ A| = \exists x : |\tau|,\ |A| \\
&|A_1 \wedge A_2| = |A_1| \wedge |A_2| \\
&|A_1 \vee A_2| = |A_1| \vee |A_2| \\
&|A \to A'| = \begin{cases} A \to |A'| & \text{if } A \text{ copy} \\ \text{emp} & o.w. \end{cases} \\
&|\neg A| = \begin{cases} \neg A & \text{if } A \text{ copy} \\ \text{emp} & o.w. \end{cases}
\end{aligned}
$$

$$|A_1 * A_2| = |A_1| * |A_2|$$
$$|A \mathrel{-\!\!*} A'| = \begin{cases} A \mathrel{-\!\!*} |A'| & \text{if } A \text{ copy} \\ \mathsf{emp} & o.w. \end{cases}$$
$$|pe \mapsto pe'| = \mathsf{emp}$$
$$\left|\boxed{x : A}\right| = \boxed{x : |A|}$$

Because moving is monotonic, that is $A \to |A|$ but not the other way around, negative uses of a non-persistent proposition cause it to completely collapse to emp when moved.

3.3 The Typing Rules

We now give the main typing rules for the logic. This corresponds roughly to the `typeck` phase of the compiler. Note that ghost variable markings are ignored during this phase; they will come back during the layout phase.

Tuple pattern typing $\boxed{\Gamma \vdash t : \tau \Rightarrow \overline{R}}$ $\boxed{\Gamma \vdash t : A \Rightarrow \overline{R}}$

TPT-IGNORE
$$\Gamma \vdash \_ : \tau \Rightarrow \cdot$$

TPP-IGNORE
$$\Gamma \vdash \_ : A \Rightarrow \cdot$$

TPT-VAR
$$\Gamma \vdash x^Y : \tau \Rightarrow x^Y : \tau$$

TPP-VAR
$$\Gamma \vdash h : A \Rightarrow h : A$$

TPT-TYPED
$$\frac{\Gamma \vdash t : \tau \Rightarrow \overline{R}}{\Gamma \vdash (t : \tau) : \tau \Rightarrow \overline{R}}$$

TPP-TYPED
$$\frac{\Gamma \vdash t : A \Rightarrow \overline{R}}{\Gamma \vdash (t : A) : A \Rightarrow \overline{R}}$$

TPT-SUM
$$\frac{\Gamma \vdash t : \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash \langle \overline{t'} \rangle : \overline{R}[t/x] \Rightarrow \bar{S}'}{\Gamma \vdash \langle t, \overline{t'} \rangle : \sum x : \tau, \overline{R} \Rightarrow \bar{S}, \bar{S}'}$$

TPP-EX
$$\frac{\Gamma \vdash t : \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash \langle \overline{t'} \rangle : \overline{R}[t/x] \Rightarrow \bar{S}'}{\Gamma, \bar{S} \vdash \langle t, \overline{t'} \rangle : \exists x : \tau, \overline{R} \Rightarrow \bar{S}, \bar{S}'}$$

TPT-LIST
$$\frac{\forall i, \quad \Gamma \vdash t_i : \tau_i \Rightarrow (\bar{R})_i}{\Gamma \vdash \langle \bar{t} \rangle : \ast\, \overline{\tau} \Rightarrow \overline{\bar{R}}}$$

TPT-AND
$$\frac{\forall i, \ \tau_i \text{ copy} \quad \forall i, \ \Gamma \vdash t_i : \tau_i \Rightarrow (\bar{R})_i}{\Gamma \vdash \langle \bar{t} \rangle : \bigwedge \overline{\tau} \Rightarrow \overline{\bar{R}}}$$

TPP-SEP
$$\frac{\Gamma \vdash t : A \Rightarrow \bar{R} \quad \Gamma \vdash t' : A' \Rightarrow \bar{R}'}{\Gamma \vdash \langle t, t' \rangle : A * A' \Rightarrow \bar{R}, \bar{R}'}$$

TPP-AND
$$\frac{\Gamma \vdash t : A \Rightarrow \bar{R} \quad \Gamma \vdash t' : |A'| \Rightarrow \bar{R}'}{\Gamma \vdash \langle t, t' \rangle : A \wedge A' \Rightarrow \bar{R}, \bar{R}'}$$

The only really relevant rules here for expressiveness are the TPT-VAR and TPP-VAR rules; the rest are convenience rules for being able to destructure a type or proposition into components using the tuple pattern. For notational simplicity we show the TPT-SUM rule in iterative form, but it actually matches an n-ary tuple against an n-ary struct type in one go.

In the TPT-SUM and TPP-EX rules, we use $\overline{R}[t/x]$ to denote the result of substituting t for x in R. For this to work, t must be reified as a tuple of variables rather than simply a destructuring pattern, which in particular means that '_' ignore patterns are interpreted as inserting internal variables with no user-specified name rather than being omitted from the context entirely as the TPT-IGNORE rule would suggest.

Argument typing

$\boxed{\Gamma \vdash R \text{ arg}}$

$$\textsc{arg-type}\ \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash x^\gamma : \tau \text{ arg}} \qquad \textsc{arg-prop}\ \frac{\Gamma \vdash A \text{ prop}}{\Gamma \vdash h : A \text{ arg}}$$

This one is simple so we get it out of the way first. We will avoid dealing with variable shadowing rules here; suffice it to say that variables in the context must always be distinct, and we will perform renaming from the surface syntax to ensure this property when necessary. Also remember that x^γ represents either x or $\boxed{x}$ in this rule.

Type validity

$\boxed{\Gamma \vdash \tau \text{ type}}$

$$\textsc{ty-void}\ \Gamma \vdash \mathbf{0} \text{ type} \qquad \textsc{ty-unit}\ \Gamma \vdash \mathbf{1} \text{ type} \qquad \textsc{ty-bool}\ \Gamma \vdash \text{bool type} \qquad \textsc{ty-nat}\ \Gamma \vdash \mathbb{N}_s \text{ type} \qquad \textsc{ty-int}\ \Gamma \vdash \mathbb{Z}_s \text{ type}$$

$$\textsc{ty-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \text{ type}} \qquad \textsc{ty-core-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash |\alpha| \text{ type}} \qquad \textsc{ty-inter}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \bigcap \tau \text{ type}} \qquad \textsc{ty-union}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \bigcup \tau \text{ type}} \qquad \textsc{ty-list}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \circledast \tau \text{ type}}$$

$$\textsc{ty-struct-1}\ \frac{\Gamma \vdash R_0 \text{ arg}}{\Gamma \vdash \sum R_0 \text{ type}} \qquad \textsc{ty-struct-2}\ \frac{\Gamma \vdash R_0 \text{ arg} \quad \Gamma, R_0 \vdash \sum \overline{R} \text{ type}}{\Gamma \vdash \sum R_0, \overline{R} \text{ type}}$$

$$\textsc{ty-user}\ \frac{\text{type } S(\overline{\alpha}, \overline{R}) \quad \forall i,\ \Gamma \vdash \tau_i \text{ type} \quad \Gamma \vdash \langle \overline{pe} \rangle : \sum \overline{R}[\overline{\tau}/\overline{\alpha}]}{\Gamma \vdash S(\overline{\tau}, \overline{pe}) \text{ type}}$$

Type validity is also relatively straightforward. Type variables are looked up in the context, and structs can have dependent types, but the only way dependencies can appear is through TY-ARRAY, which can have a natural number size bound, and in hypotheses that appear in struct declarations.

Proposition validity

$\boxed{\Gamma \vdash A \text{ prop}}$

$$\textsc{typ-true}\ \Gamma \vdash \top \text{ prop} \qquad \textsc{typ-false}\ \Gamma \vdash \bot \text{ prop} \qquad \textsc{typ-emp}\ \Gamma \vdash \text{emp prop} \qquad \textsc{typ-bool}\ \frac{\Gamma \vdash pe : \text{bool}}{\Gamma \vdash pe \text{ prop}} \qquad \textsc{typ-not}\ \frac{\Gamma \vdash A \text{ prop}}{\Gamma \vdash \neg A \text{ prop}}$$

$$\textsc{typ-and}\ \frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \wedge B \text{ prop}} \qquad \textsc{typ-or}\ \frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \vee B \text{ prop}} \qquad \textsc{typ-sep}\ \frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A * B \text{ prop}}$$

$$\textsc{typ-wand}\ \frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \mathrel{-\!\!*} B \text{ prop}} \qquad \textsc{typ-forall}\ \frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash A \text{ prop}}{\Gamma \vdash \forall x : \tau,\ A \text{ prop}}$$

$$\textsc{typ-exists}\ \frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash A \text{ prop}}{\Gamma \vdash \exists x : \tau,\ A \text{ prop}} \qquad \textsc{typ-points-to}\ \frac{\Gamma \vdash \ell : \mathbb{N}_{64} \quad \Gamma \vdash v : |\tau|}{\Gamma \vdash \ell \mapsto v \text{ prop}} \qquad \textsc{typ-typing}\ \frac{\Gamma \vdash x : |\tau| \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \boxed{x : \tau} \text{ prop}}$$

There is nothing non-standard in these rules, except perhaps the requirement in the TYP-FORALL and TYP-EXISTS rules that the types are moved (needed because the assertion language itself should not be able to take ownership of variables used in the assertions).

The most interesting rule is TYP-TYPING, which describes the typing assertion $\boxed{x : \tau}$. One should think of $x : \tau$ in the context as a separating conjunction of $x : |\tau|$ (which asserts, roughly, that x is a reference to some data in the stack frame that is a valid bit-pattern for type τ), plus the "fact" $h : \boxed{x : \tau}$, which represents ownership of all the resources that x may point to. For example, if $x : \&^{\mathbf{own}}\tau$, then x is itself just a number, but $\boxed{x : \&^{\mathbf{own}}\tau}$ is equal to $\exists v : \tau,\ x \mapsto v$, saying that x points to some data v, and $v : \tau$ may itself own some portion of the heap.

3.4 Expression typing

The typing rules for expressions make use of the following operators on contexts:

- $\Gamma_{|x|}$ "moves" x out of the context, by replacing $x : \tau$ with $x : |\tau|$ or $x : A$ with $x : |A|$. This does not invalidate the well formedness of any type, proposition, or pure expression.

The rules for pure expression typing are the same as for regular expression typing, although since all the pure expression constructors do not change the context, they are all of the form $\Gamma \vdash pe : \tau \dashv \Gamma$, which we abbreviate as $\Gamma \vdash pe : \tau$.

Note that the TYE-VAR-REF rule ignores the effect of mutations. This is necessary so that new mutations do not cause the context to become ill-typed. Instead, mutations are applied in the translation from surface syntax, so that "$x \leftarrow 1;\ x + x$" is elaborated into "move $x \leftarrow 1$ in $1 + 1$".

Expression validity (pure expressions) $\boxed{\Gamma \vdash e : \tau \dashv \Gamma'}$

TYE-VAR-REF
$$\frac{(x^Y : \tau) \in \Gamma}{\Gamma \vdash x : |\tau|}$$

TYE-UNIT
$$\Gamma \vdash () : \mathbf{1}$$

TYE-TRUE
$$\Gamma \vdash \mathsf{true} : \mathsf{bool}$$

TYE-FALSE
$$\Gamma \vdash \mathsf{false} : \mathsf{bool}$$

TYE-NAT
$$\frac{0 \le n \quad s < \infty \to n < 2^s}{\Gamma \vdash n : \mathbb{N}_s}$$

TYE-INT
$$\frac{s < \infty \to -2^{s-1} \le n < 2^{s-1}}{\Gamma \vdash n : \mathbb{Z}_s}$$

TYE-TUPLE
$$\frac{\forall i < n,\ \ \Gamma_i \vdash e_i : \tau \dashv \Gamma_{i+1}}{\Gamma_0 \vdash \langle \overline{e} \rangle : \circledast\, \tau \dashv \Gamma_n}$$

TYE-NOT
$$\frac{\Gamma \vdash e : \mathsf{bool} \dashv \Gamma'}{\Gamma \vdash \neg e : \mathsf{bool} \dashv \Gamma'}$$

TYE-AND, TYE-OR
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \dashv \Gamma_1 \quad \Gamma_1 \vdash e_2 : \mathsf{bool} \dashv \Gamma_2}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 \vee e_2 : \mathsf{bool} \dashv \Gamma_2}$$

TYE-BAND, TYE-BOR
$$\frac{\tau \in \{\mathbb{N}_s, \mathbb{Z}_s\} \quad \Gamma \vdash e_1 : \tau \dashv \Gamma_1 \quad \Gamma_1 \vdash e_2 : \tau \dashv \Gamma_2}{\Gamma \vdash e_1 \,\&\, e_2 : \tau \dashv \Gamma_2 \quad \Gamma \vdash e_1 \mid e_2 : \tau \dashv \Gamma_2}$$

TYE-BNOT
$$\frac{\tau = \mathbb{N}_s \vee (\tau = \mathbb{Z}_{s'} \wedge s = \infty) \quad \Gamma \vdash e : \tau \dashv \Gamma'}{\Gamma \vdash !_s\, e : \tau \dashv \Gamma'}$$

TYE-LT, TYE-LE, TYE-EQ
$$\frac{\tau, \tau' \in \{\mathbb{N}_s, \mathbb{Z}_s\} \qquad \Gamma \vdash e_1 : \tau \dashv \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \tau' \dashv \Gamma_2}{\Gamma \vdash e_1 < e_2 : \mathsf{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 \le e_2 : \mathsf{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 = e_2 : \mathsf{bool} \dashv \Gamma_2}$$

TYE-DEP-IF
$$\frac{c \text{ pure} \quad \Gamma \vdash c : \mathsf{bool} \quad \Gamma, h : c \vdash e_1 : \tau \dashv\vdash \Gamma', h : c \quad \Gamma, h : \neg c \vdash e_2 : \tau \dashv\vdash \Gamma', h : \neg c}{\Gamma \vdash (\mathsf{if}\ h : c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) : \tau \dashv \Gamma'}$$

$$\textsc{tye-if}\ \frac{\Gamma \vdash c : \mathsf{bool} \dashv \Gamma' \quad \Gamma' \vdash e_1 : \tau \dashv\!\Longrightarrow \Gamma'' \quad \Gamma' \vdash e_2 : \tau \dashv\!\Longrightarrow \Gamma''}{\Gamma \vdash (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) : \tau \dashv \Gamma''}$$

$$\textsc{tye-struct-var}^1\ \frac{\Gamma_0 \vdash e : \tau \dashv \Gamma_1 \quad \Gamma_1 \vdash \langle \overline{e} \rangle : \sum \bar{R}[e/x] \dashv \Gamma_n}{\Gamma_0 \vdash \langle e, \overline{e} \rangle : \sum x^\gamma : \tau, \bar{R} \dashv \Gamma_n} \qquad \textsc{tye-struct-prop}\ \frac{\Gamma_0 \vdash p : A \dashv \Gamma_1 \quad \Gamma_1 \vdash \langle \overline{e} \rangle : \sum \bar{R} \dashv \Gamma_n}{\Gamma_0 \vdash \langle p, \overline{e} \rangle : \sum h : A, \bar{R} \dashv \Gamma_n}$$

$$\textsc{tye-func-call}\ \frac{\mathsf{func}\ f(\overline{R}) : \overline{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R} \dashv \Gamma'}{\Gamma \vdash f(\overline{e}) : \sum \bar{S} \dashv \Gamma'}$$

The rules above are the only ones that apply to pure expressions. General expressions have additional typing rules for the other constructions, continued below.

For general expressions, we must worry about the following additional effects:

- Variables in the context can be moved by their being referenced (in the TYE-VAR-MOVE rule).
- Variables can be mutated, resulting in contexts with unapplied mutations. We will return to this in section 3.5.

Expression validity $\boxed{\Gamma \vdash e : \tau \dashv \Gamma'}$

$$\textsc{tye-var-move}\ \frac{(x^\gamma : \tau) \in \Gamma}{\Gamma \vdash x : \tau \dashv \Gamma_{|x|}} \qquad \textsc{tye-mut}\ \frac{\Gamma \vdash pe : \tau \dashv \Gamma' \qquad (x^\gamma : \tau'') \in \Gamma' \qquad \Gamma', x \leftarrow pe : \tau \vdash e : \tau' \dashv \Gamma''}{\Gamma \vdash (x \leftarrow pe;\ e) : \tau' \dashv \Gamma''}$$

$$\textsc{tye-mut-pr}\ \frac{\Gamma \vdash p : A \dashv \Gamma' \qquad (h : A') \in \Gamma' \qquad \Gamma', h \leftarrow p : A \vdash e : \tau \dashv \Gamma''}{\Gamma \vdash (h \leftarrow p;\ e) : \tau \dashv \Gamma''}$$

$$\textsc{tye-let-hyp}^2\ \frac{\Gamma \vdash pe : \tau \qquad \Gamma \vdash t : \tau \Rightarrow \overline{R} \qquad \Gamma' \vdash \tau'\ \mathsf{type} \qquad \Gamma, \overline{R}, h : t = pe \vdash e : \tau' \dashv\!\Longrightarrow \Gamma', \overline{R}, h : t = pe}{\Gamma \vdash (\mathsf{let}\ h := t := pe\ \mathsf{in}\ e) : \tau' \dashv \Gamma'}$$

$$\textsc{tye-unreachable}\ \frac{\Gamma \vdash p : \bot \dashv \Gamma'}{\Gamma \vdash \mathsf{unreachable}\ p : \tau}$$

$$\textsc{tye-let}\ \frac{\Gamma \vdash e_1 : \tau \dashv \Gamma' \quad \Gamma' \vdash t : \tau \Rightarrow \overline{R} \quad \Gamma'' \vdash \tau'\ \mathsf{type} \qquad \Gamma', \overline{R} \vdash e_2 : \tau' \dashv\!\Longrightarrow \Gamma'', \overline{R}}{\Gamma \vdash (\mathsf{let}\ t := e_1\ \mathsf{in}\ e_2) : \tau' \dashv \Gamma''}$$

$$\textsc{tye-proc-call}\ \frac{\mathsf{proc}\ F(\overline{R}) : \overline{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R} \dashv \Gamma'}{\Gamma \vdash F(\overline{e}) : \sum \bar{S} \dashv \Gamma'}$$

$$\textsc{tye-let-pr}\ \frac{\Gamma \vdash p : A \dashv \Gamma' \quad \Gamma' \vdash t : A \Rightarrow \overline{R} \quad \Gamma'' \vdash t : \tau\ \mathsf{type} \qquad \Gamma', \overline{R} \vdash e : \tau \dashv\!\Longrightarrow \Gamma'', \overline{R}}{\Gamma \vdash (\mathsf{let}\ t := p\ \mathsf{in}\ e) : \tau \dashv \Gamma''}$$

$$\textsc{tye-return}\ \frac{\mathsf{self}(\bar{R}) : \bar{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{S} \dashv\!\Longrightarrow |\bar{R}|}{\Gamma \vdash \mathsf{return}\ \overline{e} : \tau}$$

$$\textsc{tye-label}\ \frac{\forall i,\ \Gamma_i', \overline{k(\Gamma', \bar{R})}, (\bar{R})_i \vdash e_i : \mathbf{0} \dashv \Gamma_i''' \quad \Gamma, \overline{k(\Gamma', \bar{R})} \vdash e' : \tau \dashv \Gamma''}{\Gamma \vdash (\mathsf{label}\ \overline{k(\bar{R}) := e}\ \mathsf{in}\ e') : \tau \dashv \Gamma''} \qquad \textsc{tye-goto}\ \frac{\Gamma \vdash \langle \overline{e} \rangle : \sum (\bar{R})_i \dashv\!\Longrightarrow \Gamma_i', \overline{k(\Gamma', \bar{R})}}{\Gamma \vdash \mathsf{goto}\ k_i(\overline{e}) : \tau}$$

Note that the unreachable, return, goto functions do not return to the calling context, so they return an arbitrary type τ, and also roll back the context to the initial state Γ. A more complex

[1] In TYE-STRUCT-VAR, e must be pure or $x \notin \mathrm{Var}(\bar{R})$, so that $\bar{R}[e/x]$ is well defined.

[2] In TYE-LET-HYP, t must not contain ignore patterns so that t is a pure expression and $t = pe$ is well defined.

model here would allow the final context to be a special context $\Gamma_\bot$, which can step to any context with the same variables as Γ.

Proofs are essentially (effectful) expressions with proposition type, so the rules look much the same. Pure proofs are simply imported from the MM0 logical enironment so we do not discuss them here. The main job of metamath C is to make sure that these pure proofs have simple types, not using the entire context, since the user will be directly interacting with them.

Proof validity $\boxed{\Gamma \vdash p : A \dashv \Gamma'}$

$$\textsc{tpr-assert}\ \frac{\Gamma \vdash pe : \mathsf{bool} \dashv \Gamma'}{\Gamma \vdash \mathsf{assert}\ pe : pe \dashv \Gamma'} \qquad \textsc{tpr-typeof}\ \frac{\Gamma \vdash pe : \tau \dashv \Gamma'}{\Gamma \vdash \mathsf{typeof}\ pe : \boxed{pe : \tau} \dashv \Gamma'} \qquad \textsc{tpr-entail}\ \frac{\Gamma \vdash \langle \overline{p} \rangle : \ast\, \overline{A} \dashv \Gamma' \quad \vdash q : \ast\, \overline{A} \mathrel{-\!\!*} B}{\Gamma \vdash \mathsf{entail}\ \overline{p}\ q : B \dashv \Gamma'}$$

3.5 Mutation application

The role of the $\Gamma \vdash e : \tau \mathrel{+\!\!\!\Longrightarrow} \Gamma'$ judgment is to clean up the context at the terminator of a basic block in the control flow graph: after the branches of an if statement, and at a return and goto. It is also used whenever the context has to drop a variable, such as after a let expression completes.

Recall that $\Gamma \vdash e : \tau \mathrel{+\!\!\!\Longrightarrow} \Gamma'$ means $\Gamma \vdash e : \tau \dashv \Gamma_1$ and $\Gamma_1 \Rightarrow \Gamma'$ for some Γ_1. The reason we can't just use Γ_1 directly is because there may be pending mutations, whose values depend on variables we are about to drop. But mutations to variables are not required to be well typed at the target variable at the time of mutation, because for example structs may be written incrementally, with the half written structs being ill-typed. Instead, we delay committing these values as long as possible, even across CFG edges if we can. However, the rules given below are not deterministic, because CS-MUT steps can choose to apply any subset of outstanding mutations that are collectively well typed.

Mutation application $\boxed{\Gamma \Rightarrow \Gamma'}$

$$\textsc{cs-refl}\ \Gamma \Rightarrow \Gamma \qquad \textsc{cs-trans}\ \frac{\Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_2 \Rightarrow \Gamma_3}{\Gamma_1 \Rightarrow \Gamma_3} \qquad \textsc{cs-drop}\ \frac{\text{for all } S, \text{ if } (x \leftarrow S) \in \Gamma \text{ then } \Gamma \vdash S}{\Gamma, \overline{R} \Rightarrow \Gamma} \qquad \textsc{cs-drop-label}\ \Gamma, \overline{k(\bar{R})} \Rightarrow \Gamma$$

$$\textsc{cs-move}\ \frac{R \in \Gamma}{\Gamma \Rightarrow \Gamma_{|R|}} \qquad \textsc{cs-mut}\ \frac{\Gamma \vdash \langle \overline{x} \rangle : \sum \left|\overline{R}\right| \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \overline{R}}{\Gamma, \overline{x \leftarrow e} \Rightarrow \Gamma_{\overline{R}}}$$

Here $\Gamma_{|R|}$ is the context in which R has been moved away to nowhere (i.e. the memory is released), and $\Gamma_{\bar{R}}$ is the context which restores each variable in $\bar{R}$ to the specified status, reflecting that they are no longer moved away after the resource is transferred into the context.

To see how this plays out, recall the TYE-RETURN rule:

$$\textsc{tye-return}\ \frac{\mathsf{self}(\bar{R}) : \bar{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{S} \mathrel{+\!\!\!\Longrightarrow} \left|\bar{R}\right|}{\Gamma \vdash \mathsf{return}\ \overline{e} : \tau}$$

After executing $\langle \overline{e} \rangle$, we obtain the return value $\sum \bar{S}$ in some context Γ'. This context contains the same variables as Γ, but we are executing a return statement somewere deep in the function, so Γ will typically contain many variables that were not parameters to the function (the list $\bar{R}$), and these variables may appear in uncommitted mutations that are still in Γ, so to ensure consistency of the state at the return of the function, we have to commit all outstanding mutations, expressed

here by saying that the final state is simply $\left|\bar{R}\right|$, the moved-out version of the original context, with all extra variables and labels dropped, and no active mutations.

3.6 Top level typing

The full program consists of a list of top level items, which are typechecked incrementally:

AST typing $\boxed{\Gamma \vdash \overline{it} \dashv \Gamma'}$

$$\textsc{ok-zero}\ \frac{}{\Gamma \vdash \cdot \dashv \Gamma} \qquad \textsc{ok-append}\ \frac{\Gamma \vdash \overline{it} \dashv \Gamma' \quad \Gamma' \vdash it \dashv \Gamma''}{\Gamma \vdash \overline{it}, it' \dashv \Gamma''}$$

Individual items are typed as follows:

Item typing $\boxed{\Gamma \vdash it \dashv \Gamma'}$

$$\textsc{ok-type}\ \frac{\Gamma, \overline{\alpha} \vdash \sum \overline{R}\ \mathsf{type} \quad \Gamma, \overline{\alpha}, \overline{R} \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau \dashv \Gamma,\ \mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau} \qquad \textsc{ok-const}\ \frac{\Gamma \vdash pe : \tau \quad \Gamma \vdash t : \tau \Rightarrow \bar{R}}{\Gamma \vdash \mathsf{const}\ t := pe \dashv \Gamma, \bar{R}}$$

$$\textsc{ok-global}\ \frac{\Gamma \vdash e : \tau \dashv \Gamma' \quad \Gamma' \vdash t : \tau \Rightarrow \bar{R}}{\Gamma \vdash \mathsf{global}\ t := e \dashv \Gamma', \bar{R}}$$

$$\textsc{ok-func, ok-proc}\ \frac{\mathbf{kw} \in \{\mathsf{func}, \mathsf{proc}\} \quad \Gamma \vdash \sum \overline{R}\ \mathsf{type} \quad \Gamma, \overline{R} \vdash \sum \overline{S}\ \mathsf{type} \quad \Gamma, (\mathsf{self}(\overline{R}) : \overline{S}), \overline{R} \vdash e : \mathbf{0} \Longleftarrow \left|\bar{R}\right|}{\Gamma \vdash \mathbf{kw}\ f(\overline{R}) : \overline{S} := e \dashv \Gamma',\ \mathbf{kw}\ f(\overline{R}) : \overline{S}}$$

3.7 Uninitialized data

The approach for handling mutation also cleanly supports uninitialized data. We extend the language as follows:

$$\mathrm{Type} ::= \cdots \mid \tau^? \qquad \mathrm{Expr} ::= \cdots \mid \mathsf{uninit} \qquad \left|\tau^?\right| = |\tau|^? \qquad \boxed{x : \tau^?} = \top$$

$$\textsc{ty-maybe}\ \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \tau^?\ \mathsf{type}} \qquad \textsc{tye-uninit}\ \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathsf{uninit} : \tau^?}$$

We also modify the CS-MUT rule:

$$\textsc{cs-mut}\ \frac{\Gamma \vdash \langle \overline{x} \rangle : \sum \left|\overline{R^?}\right| \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \overline{R}}{\Gamma, \overline{x \leftarrow e} \Rightarrow \Gamma_{\overline{R}}}$$

such that the variables x are allowed to be potentially uninitialized versions of the types in e, meaning that a variable $x : \tau^?$ can change to $x : \tau$ as the result of a mutation.

3.8 Pointers

Thus far the rules have only talked about local variables and mutation of local variables, that we think of as being on the stack frame of the function. To understand the representation of pointers in the type system, it will help to understand the way contexts are modeled as separating propositions. The context is a large separating conjunction of $\boxed{x : \tau}$ assertions for every $(x^\gamma : \tau) \in \Gamma$ and A for

every $h : A$, plus additional "layout" information about the relation of non-ghost variables to the stack frame that will be calculated in the layout pass (see section 4).

3.8.1 Singleton pointers. The simplest pointer type is $\&^{\mathbf{sn}}(\eta : \tau)$. $x : \&^{\mathbf{sn}}(\eta : \tau)$ simply means that x is a pointer that points to η, which is a "place", a writable location. $|\&^{\mathbf{sn}}\eta| = \mathbb{N}_{64}$ and $\boxed{x : \&^{\mathbf{sn}}\eta} = x \mapsto \eta$. Supporting these requires no significant extensions to the language. We add the following:

$$\text{Type} ::= \cdots \mid \&^{\mathbf{sn}}(\eta : \tau) \qquad \text{Expr} ::= \cdots \mid {}^*e \mid \&e$$

$$|\&^{\mathbf{sn}}(\eta : \tau)| = \mathbb{N}_{64} \qquad \boxed{x : \&^{\mathbf{sn}}(\eta : \tau)} = x \mapsto \eta$$

TY-SNP
$$\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma \vdash \eta : |\tau|}{\Gamma \vdash \&^{\mathbf{sn}}(\eta : \tau) \text{ type}}$$

TYE-DEREF
$$\frac{\Gamma \vdash e : \&^{\mathbf{sn}}(\eta : \tau) \dashv \Gamma' \quad \Gamma' \vdash \eta : \tau \dashv \Gamma''}{\Gamma \vdash {}^*e : \tau \dashv \Gamma''}$$

TYE-REF
$$\frac{\Gamma \vdash \eta : |\tau|}{\Gamma \vdash \&\eta : \&^{\mathbf{sn}}(\eta : \tau)}$$

To allow for mutation, we must also extend the $x \leftarrow pe;\ e$ form to allow a pointer on the left:

$$\text{Place} ::= \cdots \mid {}^*e$$

TYE-MUT-DEREF
$$\frac{\Gamma \vdash e_1 : \&^{\mathbf{sn}}(\eta : \tau) \dashv \Gamma' \quad \Gamma' \vdash (\eta \leftarrow pe;\ e_2) : \tau' \dashv \Gamma''}{\Gamma \vdash ({}^*e_1 \leftarrow pe;\ e_2) : \tau' \dashv \Gamma''}$$

3.8.2 Owned pointers. An owned pointer is also fairly simple. We define $\boxed{x : \&^{\mathbf{own}}\tau}$ as $\exists v : \tau,\ x \mapsto v$, but we can't directly dereference an owned pointer as we must first have access to the variable v, so we require that it first be destructured to be used.

$$\text{Type} ::= \cdots \mid \&^{\mathbf{own}}\tau \qquad |\&^{\mathbf{own}}\tau| = \mathbb{N}_{64} \qquad \boxed{x : \&^{\mathbf{own}}\tau} = \exists v : \tau, x \mapsto v$$

TY-OWN
$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \&^{\mathbf{own}}\tau \text{ type}}$$

TPT-OWN
$$\frac{\Gamma \vdash t : \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash t' : \&^{\mathbf{sn}}t \Rightarrow \bar{S}'}{\Gamma \vdash \langle t, t' \rangle : \&^{\mathbf{own}}\tau \Rightarrow \bar{S}, \bar{S}'}$$

3.8.3 Mutable pointers. The type $\&^{\mathbf{mut}}\tau$ is not a true type, but is allowed in function signatures to indicate a $\&^{\mathbf{sn}}(\eta : \tau)$ value where η is external to the function. To support this we have to rewrite the provided function signature to include these η ghost values and then match them with provided expressions.

$$\rho \in \text{FArg} ::= R \mid x : \&^{\mathbf{mut}}\tau$$
$$\text{Item} ::= \cdots \mid \mathsf{func}\ f(\overline{\rho}) : \overline{R} := e \mid \mathsf{proc}\ f(\overline{\rho}) : \overline{R} := e$$

We have to modify ok-proc to accomodate the new arguments:

OK-FUNC, OK-PROC
$$\frac{\mathbf{kw} \in \{\mathsf{func}, \mathsf{proc}\} \quad \Gamma \vdash \overline{\rho}\ \text{args} \Rightarrow \overline{R} \quad \Gamma, \overline{R} \vdash \sum \overline{S} \text{ type} \quad \Gamma, (\mathsf{self}(\overline{\rho}) : \overline{S}), \overline{R} \vdash e : \sum \overline{S} \leftrightarrows |\bar{\rho}|}{\Gamma \vdash \mathbf{kw}\ f(\overline{\rho}) : \overline{S} := e \dashv \Gamma',\ \mathbf{kw}\ f(\overline{\rho}) : \overline{S}}$$

where we have a new judgment for elaborating function arguments, whose only interesting case elaborates $x : \&^{\mathbf{mut}}\tau$ to $\boxed{v} : \tau, x : \&^{\mathbf{sn}}v$:

Argument elaboration $\boxed{\Gamma \vdash \overline{\rho} \text{ args} \Rightarrow \overline{R}}$

$$\textsc{args-zero}\ \ \Gamma \vdash \cdot \Rightarrow \cdot \qquad \textsc{args-arg}\ \ \frac{\Gamma \vdash \overline{\rho} \Rightarrow \bar{R}}{\Gamma \vdash \overline{\rho}, R' \Rightarrow R, R'} \qquad \textsc{args-mut}\ \ \frac{\Gamma \vdash \overline{\rho} \Rightarrow \bar{R} \quad \Gamma, \bar{R} \vdash \tau\ \text{type}}{\Gamma \vdash \overline{\rho}, x : \&^{\mathbf{mut}}\tau \Rightarrow \bar{R}, \lceil \overline{v} \rceil : \tau, x : \&^{\mathbf{sn}} v}$$

The $|\bar{\rho}|$ that appears in OK-PROC denotes a final function context that moves all regular function arguments, but the $\lceil \overline{v} \rceil : \tau$ variables generated by $x : \&^{\mathbf{mut}}\tau$ elaboration are not moved. This ensures that they are still valid at function return.

Finally, we have to describe how a function is called. In short, e matches an argument of type $x : \&^{\mathbf{mut}}\tau$ if $e : \&^{\mathbf{sn}}\eta$ for some η, which is "captured" by the function and modified, generating a fresh $v' : \tau$ and a mutation record $\eta \leftarrow v'$.

3.8.4 Shared pointers. Shared pointers are the most complex, because they cannot be modeled by separating conjunctions, at least without techniques such as fractional ownership. This is not a problem until we get to the underlying separation logic. Here we only need to mark work that will be perfomed later on.

We introduce a new kind of variable modifier, a heap variable. $\hat{x} : \tau$ means that $x : \tau$, but x is not owned by the current context. Heap variables can overlap each other, but not other regular variables in the context.

Heap variables resemble shared references from Rust, and in particular they are annotated with a "lifetime". The difference is that the pointer-ness is separated out; a heap variable directly has the type of the pointee, and the pointer is just a $\&^{\mathbf{sn}}\eta$ where η is a heap variable.

$$a \in \text{Lft} ::= \text{extern} \mid x \qquad \text{Arg} ::= \cdots \mid \text{ref}^a\ x : \tau \qquad \text{FArg} ::= \cdots \mid \text{ref}\ x$$

$$\text{TuplePattern} ::= \cdots \mid \text{ref}^a\ x \qquad \textsc{tpt-ref}\ \ \Gamma \vdash \text{ref}^a\ x : \tau \Rightarrow \text{ref}^a\ x : \tau \qquad \textsc{arg-ref}\ \ \frac{\text{Var}(a) \subseteq \Gamma \quad \Gamma \vdash \tau\ \text{type}}{\Gamma \vdash \text{ref}^a\ x : \tau\ \text{arg}}$$

$$\textsc{args-ref}\ \ \frac{\Gamma \vdash \overline{\rho} \Rightarrow \bar{R} \quad \Gamma, \bar{R} \vdash \tau\ \text{type}}{\Gamma \vdash \overline{\rho}, \text{ref}\ x : \tau \Rightarrow \bar{R}, \text{ref}^{\text{extern}}\ x : \tau} \qquad \textsc{tye-struct-ref}\ \ \frac{\Gamma \vdash \eta : |\tau| \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R}[\eta/x] \dashv \Gamma'}{\Gamma \vdash \langle \eta, \overline{e} \rangle : \sum \text{ref}^{\text{Lft}(\eta)}\ x : \tau, \bar{R} \dashv \Gamma'}$$

$$\text{Lft}(x) = x$$
$$\text{Lft}(\eta[pe]) = \text{Lft}(\eta)$$

Using heap variables, we can desugar shared references similarly to owned pointers:

$$\text{Type} ::= \cdots \mid \&^a\tau \qquad |\&^a\tau| = \mathbb{N}_{64} \qquad \boxed{x : \&^a\tau} = \exists v : \text{ref}^a\ \tau, x \mapsto v$$

$$\textsc{ty-shr}\ \ \frac{\text{Var}(a) \subseteq \Gamma \quad \Gamma \vdash \tau\ \text{type}}{\Gamma \vdash \&^a\tau\ \text{type}} \qquad \textsc{tpt-shr}\ \ \frac{\Gamma \vdash \text{ref}^a\ t : \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash t' : \&^{\mathbf{sn}} t \Rightarrow \bar{S}'}{\Gamma \vdash \langle t, t' \rangle : \&^a\tau \Rightarrow \bar{S}, \bar{S}'}$$

3.9 Arrays

Arrays here are fixed length, depending on another variable in the context.

$$\text{Type} ::= \cdots \mid \text{array } \tau\ pe \qquad\qquad |\text{array } \tau\ n| = \text{array } |\tau|\ n$$

$$\boxed{x : \text{array } \tau\ n} = (x : n \to |\tau|) * \text{✻}_{i<n} \boxed{x[i] : \tau}$$

$$\text{TY-ARRAY}\ \frac{\Gamma \vdash \tau\ \text{type} \quad \Gamma \vdash n : \mathbb{N}_s}{\Gamma \vdash \text{array } \tau\ n\ \text{type}}$$

TODO

4 THE LAYOUT PASS

The layout pass is responsible for assigning concrete memory locations to variables in the code. In particular, multiple variables may overlap the same memory location if they are never *live* at the same time, which is to say, the last use of one variable comes before the definition of the second. The analysis pass that determines these relations is considered part of the "nondeterministic" part of the compiler, meaning that it requires no proof. Instead, the analysis pass produces a satisfying layout, and the typing relation will validate that a layout puts variables in disjoint locations if they are live at the same time.

To that end, we introduce another syntactic category not present in the source language, a *machine place*, or M-place for short.

$$\mu ::= \text{Reg } r \mid \text{Stack } s$$

The registers r correspond to the registers on the machine, so there is one for every general-purpose register. (On x86-64 there are 16 general purpose registers, but RSP is the stack pointer, and one register is reserved by the compiler for spilling, so there are 14 registers available for use.)

The stack locations s correspond to an abstraction of the stack frame, optimized for disjointness proofs. A stack frame has a series-parallel layout:

$$\phi ::= \phi_0 * \phi_1 \mid \phi_0 \cup \phi_1 \mid |\tau|$$

and s is a path into the stack frame:

$$s ::= \text{id} \mid s.0 \mid s.1 \mid s.l \mid s.r$$

with the following typing rules:

Stack variable typing $\boxed{\phi \vdash s : \phi'}$

$$\text{STK-ID}\ \phi \vdash \text{id} : \phi \qquad \text{STK-FST}\ \frac{\phi \vdash s : \phi_1 * \phi_2}{\phi \vdash s.0 : \phi_1} \qquad \text{STK-SND}\ \frac{\phi \vdash s : \phi_1 * \phi_2}{\phi \vdash s.1 : \phi_2}$$

$$\text{STK-LEFT}\ \frac{\phi \vdash s : \phi_1 \cup \phi_2}{\phi \vdash s.l : \phi_1} \qquad \text{STK-RIGHT}\ \frac{\phi \vdash s : \phi_1 \cup \phi_2}{\phi \vdash s.r : \phi_2}$$

Intuitively, $\phi_1 * \phi_2$ is the stack layout consisting of the layout ϕ_1 followed by ϕ_2 in the bytes immediately after, while $\phi_1 \cup \phi_2$ consists of ϕ_1 and ϕ_2 superimposed on the same bytes (taking up size equal to the larger of the two).

At a given point in execution, each of the unions has one of its members "active" and the other "inactive", and a variable can only be accessed if it is active in all parent unions. A ghost variable is never assigned any stack location and hence it can never be accessed. More formally, we say that two stack paths are *incompatible*, written $s_1 \perp s_2$, if there exists s such that s_1 extends $s.l$ and s_2 extends $s.r$, or vice versa. We will maintain the invariant that if two variables in the context are represented by stack paths s_1 and s_2 then they are compatible.

4.1 Interpreting the context

The context Γ in the typing rules is ultimately compiled down to a separating proposition over machine states, and we need to interpret it in such a way that a validly typed expression corresponds to a valid theorem in separation logic.

Each variable in the context may or may not be associated with a component of the machine state which is currently storing the value of that variable. A ghost variable will never have machine state attached to it, and a variable may also not have machine state attached to it if it is past its last use, or if it is uninitialized. To express this, we will add a new kind of context, a machine context Δ which extends Γ with this information at each variable site.

- For each procedure in the global environment of declared items, we have a (persistent) proposition proc-ok$(\ell : \overline{R} \to \overline{S})$ which asserts that location ℓ (an actual machine location) is the entry point to a function f which, if called with arguments $\overline{R}$, will return values $\overline{S}$, according to the calling convention (which can be an additional parameter to proc-ok, but we can suppose that there is one fixed calling convention).
  Mutual recursions are more complex, as we may not be able to promise that they are safe to call without additional restrictions. Instead, for such functions we have proc-ok$(\ell : (\boxed{v : \mathbb{N}}, h : v < n, \overline{R}) \to \overline{S})$ where v is the variant, and n is a parameter, the value of the variant passed into this function. In other words, they must be called with a value of the variant less than the current one. We will not discuss the compilation of recursive functions here.
- Type declarations correspond to certain unfolding theorems so they have no representation in the context. We can ignore the type variables $\overline{\alpha}$ in Γ because we don't support generic functions.
- The jump targets $\overline{k(\Gamma', \bar{R})}$ in Γ become (persistent) propositions jump-ok$(\ell : (\Gamma', \bar{R}) \to \mathbf{0})$ asserting that if we jump to location ℓ with arguments $\bar{R}$ according to the calling convention of the jump, then this machine state is OK (will eventually reach a final termination with the desired global properties). The return$(\bar{R})$ continuation is also a jump target of this form (where the calling convention uses `ret` instead of `jump`).
- The regular variables, ghost variables, and hypotheses $\overline{R}$ become augmented to $\overline{\mathbf{R}}$ as follows:
  - a variable $x^\gamma : \tau$ becomes either $x^\gamma : \tau$ or $x^\top : \tau$ @ μ where μ is an M-place. The second form is only available for non-ghost variables, and the M-places of distinct variables in the context will always be compatible.
  - A mutation record $x \leftarrow pe : \tau$ becomes $x \leftarrow pe^\gamma : \tau$, where the ghost annotation indicates whether this mutation will leave x ghost or not.