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
	$\mid$ array τ pe	arrays of known length
	$\mid$ own $\tau \mid \&\tau \mid \&^{\mathbf{mut}}\tau$	owned, borrowed, mutable pointers
	$\mid \bigcap \overline{\tau} \mid \bigcup \overline{\tau}$	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
	$\mid \ldots$	

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

$A \in \text{Prop} ::= pe$		assert that a boolean value is true
	$\mid \top \mid \bot \mid \mathsf{emp}$	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
	$\mid \ldots$	

$pe \in \text{PureExpr} ::=$ (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
	$\mid x \leftarrow pe;\ e$	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
	$\mid \ldots$	

$p \in \text{Proof} ::=$	$\mathsf{entail}\ \overline{p}\ q$	entailment proof
	$\mid \mathsf{assert}\ pe$	assertion
	$\mid \mathsf{typeof}\ pe$	take the type of a variable
	$\mid \ldots$	
$q \in \text{PureProof} ::=$	$\ldots$	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 indexing and slicing array references, as well as assigning to parts of an array. These will become important later but we will ignore them for now.
- 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 $\gamma = \top$ means this is a ghost variable and $\gamma = \bot$ means it is not.
- 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. For example,

this:	has the same effect as:	which we can α-rename to:
let $x := 1$ in	let $x := 1$ in	let $x := 1$ in
let $y :=$	let $\langle x, y\rangle :=$	let $\langle x', y\rangle :=$
$\quad x \leftarrow x + 1;$	$\quad$ let $x := x + 1$ in	$\quad$ let $x' := x + 1$ in
$\quad -x$ in	$\quad \langle x, -x\rangle$ in	$\quad \langle x', -x'\rangle$ 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.
- $\Gamma \Rightarrow \Gamma'$ is 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.

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 $\overline{k(\bar{R})}$, including a special jump target $\mathsf{return}(\bar{R})$ where $\bar{R}$ is the declared return type.
- 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:

$$
\begin{aligned}
&|\alpha|, \mathbf{0}, \mathbf{1}, \mathsf{bool}, \mathbb{N}_s, \mathbb{Z}_s, \&\tau \text{ copy} \\
&\qquad |\alpha| = |\alpha| \qquad \text{(that is, } \alpha \text{ maps to } |\alpha|) \\
&\qquad |\mathsf{array}\ \tau\ n| = \mathsf{array}\ |\tau|\ n \\
&|\mathsf{own}\ \tau| = |\&^{\mathbf{mut}}\tau| = \mathbb{N}_{64} \\
&\qquad |\bigcap \overline{\tau}| = \bigcap \overline{|\tau|} \\
&\qquad |\bigcup \overline{\tau}| = \bigcup \overline{|\tau|} \\
&\qquad |\ast \overline{\tau}| = \ast \overline{|\tau|} \\
&\qquad |\sum \overline{\tau}| = \sum \overline{|\tau|} \\
&\qquad |S(\overline{\tau}, \overline{pe})| = [S](\overline{\tau}, \overline{pe}) \qquad \text{(that is, the effect of moving } S \text{ is precalculated)}
\end{aligned}
$$

The only interesting case is for owned and mutable pointers, which 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 $|\mathsf{own}\ \tau|$ 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:

$$pe, \top, \bot, \mathsf{emp} \text{ copy}$$

$$|\forall x : \tau,\ A| = \begin{cases} \forall x : \tau,\ |A| & \text{if } \tau \text{ copy} \\ \mathsf{emp} & o.w. \end{cases}$$

$$|\exists x : \tau,\ A| = \exists x : |\tau|,\ |A|$$

$$|A_1 \wedge A_2| = |A_1| \wedge |A_2|$$

$$|A_1 \vee A_2| = |A_1| \vee |A_2|$$

$$|A_1 * A_2| = |A_1| * |A_2|$$

$$|A \to A'| = \begin{cases} A \to |A'| & \text{if } A \text{ copy} \\ \mathsf{emp} & o.w. \end{cases}$$

$$|A \mathrel{-\!\!*} A'| = \begin{cases} A \mathrel{-\!\!*} |A'| & \text{if } A \text{ copy} \\ \mathsf{emp} & o.w. \end{cases}$$

$$|\neg A| = \begin{cases} A & \text{if } A \text{ copy} \\ \mathsf{emp} & o.w. \end{cases}$$

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
$$\Gamma \vdash x^\gamma : \tau \Rightarrow x^\gamma : \tau$$

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

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

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

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

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

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

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^y represents either x or $\boxed{x}$ in this rule.

Type validity

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

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

$$\textsc{ty-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \text{ type}} \qquad \textsc{ty-core-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash |\alpha| \text{ type}} \qquad \textsc{ty-array}\ \frac{\Gamma \vdash \tau \text{ type} \quad \Gamma \vdash n : \mathbb{N}_s}{\Gamma \vdash \text{array } \tau\ n \text{ type}} \qquad \textsc{ty-own}\ \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \text{own } \tau \text{ type}} \qquad \textsc{ty-ref}\ \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \&\tau \text{ type}}$$

$$\textsc{ty-mut}\ \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \&^{\mathbf{mut}}\tau \text{ type}} \qquad \textsc{ty-inter}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \bigcap \tau \text{ type}} \qquad \textsc{ty-union}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \bigcup \tau \text{ type}} \qquad \textsc{ty-list}\ \frac{\forall i,\ \Gamma \vdash \tau_i \text{ type}}{\Gamma \vdash \circledast \tau \text{ type}}$$

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

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

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

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

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

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

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

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 $x : \mathsf{own}\ \tau$, then x is itself just a number, but $\boxed{x : \mathsf{own}\ \tau}$ 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.
- $\Gamma' - x$ removes variable x from the context. The primary complication here is We can ensure that variables of Γ' that are not in Γ, but retains the types from Γ'. This is used only when Γ' is an extension of Γ, with some variables moved; this operator ensures that they remain moved.

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

$$\textsc{tye-var-ref}\ \frac{(x^Y : \tau) \in \Gamma}{\Gamma \vdash x : |\tau|} \qquad \textsc{tye-unit}\ \Gamma \vdash () : \mathbf{1} \qquad \textsc{tye-true}\ \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \textsc{tye-false}\ \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \textsc{tye-nat}\ \frac{0 \le n \quad s < \infty \to n < 2^s}{\Gamma \vdash n : \mathbb{N}_s}$$

$$\textsc{tye-int}\ \frac{s < \infty \to -2^{s-1} \le n < 2^{s-1}}{\Gamma \vdash n : \mathbb{Z}_s} \qquad \textsc{tye-tuple}\ \frac{\forall i < n,\ \ \Gamma_i \vdash e_i : \tau \dashv \Gamma_{i+1}}{\Gamma_0 \vdash \langle \overline{e} \rangle : \mathbin{\divideontimes} \tau \dashv \Gamma_n} \qquad \textsc{tye-not}\ \frac{\Gamma \vdash e : \mathsf{bool} \dashv \Gamma'}{\Gamma \vdash \neg e : \mathsf{bool} \dashv \Gamma'}$$

$$\textsc{tye-and, tye-or}\ \frac{\Gamma \vdash e_1 : \mathsf{bool} \dashv \Gamma_1 \quad \Gamma_1 \vdash e_2 : \mathsf{bool} \dashv \Gamma_2}{\Gamma \vdash e_1 \wedge e_2 : \mathsf{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 \vee e_2 : \mathsf{bool} \dashv \Gamma_2} \qquad \textsc{tye-band, tye-bor}\ \frac{\tau \in \{\mathbb{N}_s, \mathbb{Z}_{s'}\} \quad \Gamma \vdash e_1 : \tau \dashv \Gamma_1 \quad \Gamma_1 \vdash e_2 : \tau \dashv \Gamma_2}{\Gamma \vdash e_1\ \&\ e_2 : \tau \dashv \Gamma_2 \quad \Gamma \vdash e_1\ |\ e_2 : \tau \dashv \Gamma_2}$$

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

$$\textsc{tye-lt, tye-le, tye-eq}\ \frac{\tau, \tau' \in \{\mathbb{N}_s, \mathbb{Z}_s\} \qquad \Gamma \vdash e_1 : \tau \dashv \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \tau' \dashv \Gamma_2}{\Gamma \vdash e_1 < e_2 : \text{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 \le e_2 : \text{bool} \dashv \Gamma_2 \quad \Gamma \vdash e_1 = e_2 : \text{bool} \dashv \Gamma_2}$$

$$\textsc{tye-dep-if}\ \frac{c \text{ pure} \quad \Gamma \vdash c : \text{bool} \quad \Gamma, h : c \vdash e_1 : \tau \dashv\vdash \Gamma', h : c \quad \Gamma, h : \neg c \vdash e_2 : \tau \dashv\vdash \Gamma', h : \neg c}{\Gamma \vdash (\text{if } h : c \text{ then } e_1 \text{ else } e_2) : \tau \dashv \Gamma'}$$

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

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

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

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

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

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

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

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

$$\textsc{tye-let}\ \frac{\Gamma \vdash e_1 : \tau \dashv \Gamma' \quad \Gamma' \vdash t : \tau \Rightarrow \overline{R} \quad \Gamma'' \vdash \tau' \text{ type} \qquad \Gamma', \overline{R} \vdash e_2 : \tau' \dashv\vdash \Gamma'', \overline{R}}{\Gamma \vdash (\text{let } t := e_1 \text{ in } e_2) : \tau' \dashv \Gamma''} \qquad \textsc{tye-proc-call}\ \frac{\text{proc } F(\overline{R}) : \overline{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R} \dashv \Gamma'}{\Gamma \vdash F(\overline{e}) : \sum \bar{S} \dashv \Gamma'}$$

[1] In TYE-STRUCT-VAR, e must be pure or $x \notin \text{Var}(\bar{R})$, 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.

TYE-LET-PR
$$\frac{(\Gamma \vdash p : A) \in \Gamma \dashv \Gamma' \quad \Gamma' \vdash t : A \Rightarrow \overline{R} \quad \Gamma'' \vdash t : \tau \text{ type} \quad \Gamma', \overline{R} \vdash e : \tau \dashv\Longrightarrow \Gamma'', \overline{R}}{\Gamma \vdash (\text{let } t := p \text{ in } e) : \tau \dashv \Gamma''}$$

TYE-RETURN
$$\frac{\text{self}(\bar{R}) : \bar{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{S} \dashv\Longrightarrow |\bar{R}|}{\Gamma \vdash \text{return } \overline{e} : \tau}$$

TYE-LABEL
$$\frac{\forall i,\ |\Gamma|, \overline{k(\bar{R})}, (\bar{R})_i \vdash e_i : \mathbf{0} \dashv \Gamma_i'' \quad \Gamma, \overline{k(\bar{R})} \vdash e' : \tau \dashv \Gamma'}{\Gamma \vdash (\text{label } \overline{k(\bar{R}) := e} \text{ in } e') : \tau \dashv \Gamma'}$$

TYE-GOTO
$$\frac{\Gamma \vdash \langle \overline{e} \rangle : \sum (\bar{R})_i \dashv\Longrightarrow \Gamma', \overline{k(\bar{R})}}{\Gamma \vdash \text{goto } k_i(\overline{e}) : \tau}$$

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

TPR-ASSERT
$$\frac{\Gamma \vdash pe : \text{bool} \dashv \Gamma'}{\Gamma \vdash \text{assert } pe : pe \dashv \Gamma'}$$

TPR-TYPEOF
$$\frac{\Gamma \vdash pe : \tau \dashv \Gamma'}{\Gamma \vdash \text{typeof } pe : \boxed{pe : \tau} \dashv \Gamma'}$$

TPR-ENTAIL
$$\frac{\Gamma \vdash \langle \overline{p} \rangle : \ast\, \overline{A} \dashv \Gamma' \quad \vdash q : \ast\, \overline{A} \mathrel{-\!\!*} B}{\Gamma \vdash \text{entail } \overline{p}\ q : B \dashv \Gamma'}$$

3.5 Mutation application

The role of the $\Gamma \vdash e : \tau \dashv\Longrightarrow \Gamma'$ 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 $\Gamma \vdash e : \tau \dashv\Longrightarrow \Gamma'$ 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 $\boxed{\Gamma \Rightarrow \Gamma'}$

CS-REFL
$$\Gamma \Rightarrow \Gamma$$

CS-TRANS
$$\frac{\Gamma_1 \Rightarrow \Gamma_2 \quad \Gamma_2 \Rightarrow \Gamma_3}{\Gamma_1 \Rightarrow \Gamma_3}$$

CS-DROP
$$\frac{\text{for all } S, \text{ if } (x \leftarrow S) \in \Gamma \text{ then } \Gamma \vdash S}{\Gamma, \overline{R} \Rightarrow \Gamma}$$

CS-DROP-LABEL
$$\Gamma, \overline{k(\bar{R})} \Rightarrow \Gamma$$

CS-MOVE
$$\frac{R \in \Gamma}{\Gamma \Rightarrow \Gamma_{|R|}}$$

CS-MUT
$$\frac{\Gamma \vdash \langle \overline{x} \rangle : \sum |\overline{R}| \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \overline{R}}{\Gamma, \overline{x \leftarrow e} \Rightarrow \Gamma_{\overline{R}}}$$

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:

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

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 $|\bar{R}|$, the moved-out version of the original context, with all extra variables and labels dropped, and no active mutations.