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:

$\alpha, x, h, k \in \text{Ident} ::= \text{identifiers}$	
$s \in \text{Size} ::= 8 \mid 16 \mid 32 \mid 64 \mid \infty$	integer bit size
$t \in \text{TuplePattern} ::= \_ \mid x \mid \boxed{x}$	ignored, variable, ghost variable
$\mid t : \tau \mid \langle \overline{t} \rangle$	type ascription, tuple
$R \in \text{Arg} ::= x : \tau \mid \boxed{x} : \tau$	regular/ghost argument
$\tau \in \text{Type} ::= \alpha$	type variable reference
$\mid \lvert\alpha\rvert$	moved type variable
$\mid \mathbf{1} \mid \text{bool}$	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
$\mid \mathop{\ast} \overline{\tau} \mid \sum \overline{R}$	tuple type, structure (dependent tuple) type
$\mid A$	proposition
$\mid S(\overline{\tau}, \overline{pe})$	user-defined type

$A \in \text{Prop} ::= pe$	assert that a boolean value is true
$\mid \top \mid \bot \mid \text{emp}$	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
$\mid A_1 \land A_2 \mid A_1 \lor A_2$	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
	$\mid e_1 \,\&\, e_2 \mid e_1 \,\mathbf{\mid}\, e_2 \mid \,!_s\, e$	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
	$\mid \mathsf{let}\ t := e_1\ \mathsf{in}\ e_2$	assignment to a variable
	$\mid \eta \leftarrow pe;\ e \mid \boxed{\eta \leftarrow pe};\ e$	move assignment
	$\mid F(\overline{e})$	procedure call
	$\mid \mathsf{unreachable}\ e$	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
	$\mid \mathsf{entail}\ \overline{e}\ p$	entailment proof
	$\mid \mathsf{assert}\ pe$	assertion
	$\mid \mathsf{typeof}\ pe$	take the type of a variable
$p \in \text{PureProof}$	$::= \ldots$	MM0 proofs
$\eta \in \text{Place}$	$::= x$	variable reference

$it \in \text{Item}$	$::= \mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau$	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 $\mathsf{let}\ t := e_1\ \mathsf{in}\ e_2$ 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.
  The concrete version of the assignment operator also contains a "with $x \longrightarrow y$" 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 $x^\gamma \leftarrow pe;\ e$ is the primitive for mutation of the variables in the context (where, as with ghost variables, we use γ to generalize over the ghost and non-ghost versions of the operator). 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 expression $\mathsf{label}\ \overline{k(\overline{R}) := e}\ \mathsf{in}\ e'$ 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 $\mathsf{goto}\ l(\overline{e})$, which is how we ensure that they can be compiled to plain label and goto at the machine code level.
- The $\mathsf{typeof}\ pe$ 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:

- $\Gamma \vdash t : \tau \Rightarrow \overline{R}$
  types a tuple pattern against a value of type τ, 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
- $\Gamma; \delta \vdash e : \tau \dashv \delta'$
  determines that e is a valid expression of type τ, which modifies the value context from δ to δ'. In the special case where $\delta' = \delta$, we will write $\Gamma; \delta \vdash e : \tau$ instead.
- $\Gamma; \delta \vdash e \Rightarrow pe : \tau \dashv \delta'$
  is the same as the previous, but additionally says that the returned value can be expressed as the pure expression pe in context Γ.
- $\Gamma \vdash \delta$ means that δ is a valid value context. It is defined as: if $(x^\gamma := pe : \tau) \in \delta$ then $\Gamma \vdash pe : \tau$ and $x \in \mathrm{Dom}(\Gamma)$, and if $(x \longrightarrow y) \in \delta$ then $x, y \in \mathrm{Dom}(\Gamma)$.
- $\Gamma \vdash pe : \tau$
  The typing rule for pure expressions, which does not depend on the value context.
- $\Gamma \Rightarrow \Gamma'$
  an auxiliary judgment for applying pending mutations to the context.
- $\Gamma; \delta \vdash e : \tau \dashv\!\!\Rightarrow \delta'$ is defined to mean $\Gamma; \delta \vdash e : \tau \dashv \delta_1 \ \wedge\ \delta_1 \Rightarrow \delta'$ for some δ_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 $\overline{k(\delta, \bar{R})}$, including a special jump target $\mathsf{return}(\bar{R})$ where $\bar{R}$ is the declared return type. The δ in each jump target is the context required for that jump to typecheck; it lies somewhere between the initial context δ at the point of the label, and the moved-out context $|\delta|$.
- A list of logical variables $x : |\tau|$ with their types. Here $|\tau|$ is used to indicate that while the type τ itself is recorded, it is only accessible in "moved" form.

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 $\Gamma, x \longleftarrow pe : \tau$ to denote the insertion of $x \longleftarrow pe : \tau$ into the list of mutations, replacing $x \longleftarrow pe' : \tau'$ if it is present.

The secondary context used in the typing rule $\Gamma, \delta \vdash e : \tau \dashv \delta'$ for expressions is the "value context", which contains the actual current value of variables in the context. It has two components:

- A list of records of the form $x^\gamma := pe : \tau$, which represent the "actual resources" associated to a variable x. Note that x need not be in the context, but $\Gamma \vdash pe : \tau$ so all variables in pe must be in the context. For function arguments and other variables with no known value, we use $x^\gamma : \tau$, a shorthand for $x^\gamma := x : \tau$, where $(x : |\tau|) \in \Gamma$.
- A rename map, which is a list of records of the form $x \longrightarrow y$ where x and y are variables which are either in the context or in the value context. This keeps track of what a variable's "current name" is, after some number of renames. When a block ends, the values associated to renamed variables become the initial values of variable names in the code following the block.
  A variable can only be renamed once, and it is always renamed to a fresh variable; this means that the rename map is an injective partial function, i.e., if $x \longrightarrow y, y'$ then $y = y'$ and if $x, x' \longrightarrow y$ then $x = x'$.

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:

$$|\alpha|, \mathbf{1}, \mathrm{bool}, \mathbb{N}_s, \mathbb{Z}_s \text{ copy}$$

$$\begin{aligned}
|\alpha| &= |\alpha| \qquad \text{(that is, } \alpha \text{ maps to } |\alpha|\text{)} \\
|\textstyle\bigcap \overline{\tau}| &= \textstyle\bigcap \overline{|\tau|} \\
|\textstyle\bigcup \overline{\tau}| &= \textstyle\bigcup \overline{|\tau|} \\
|\textstyle\mathop{\ast} \overline{\tau}| &= \textstyle\mathop{\ast} \overline{|\tau|} \\
|\textstyle\sum \overline{\tau}| &= \textstyle\sum \overline{|\tau|} \\
\left|S(\overline{\tau}, \overline{pe})\right| &= [S](\overline{\tau}, \overline{pe}) \qquad \text{(that is, the effect of moving } S \text{ is precalculated)}
\end{aligned}$$

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:

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

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

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 $\boxed{\Gamma \vdash t : \tau \Rightarrow \overline{R}}$

$$\textsc{tp-ignore}\quad \Gamma \vdash \_ : \tau \Rightarrow \cdot \qquad \textsc{tp-var}\quad \Gamma \vdash x^{\gamma} : \tau \Rightarrow x^{\gamma} : \tau \qquad \textsc{tp-typed}\quad \frac{\Gamma \vdash t : \tau \Rightarrow \overline{R}}{\Gamma \vdash (t : \tau) : \tau \Rightarrow \overline{R}}$$

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

$$\textsc{tp-list}\quad \frac{\forall i,\ \Gamma \vdash t_i : \tau_i \Rightarrow (\bar{R})_i}{\Gamma \vdash \langle \bar{t} \rangle : \mathop{\Large *} \bar{\tau} \Rightarrow \bar{\bar{R}}} \qquad \textsc{tp-and}\quad \frac{\forall i,\ \tau_i\ \mathsf{copy} \quad \forall i,\ \Gamma \vdash t_i : \tau_i \Rightarrow (\bar{R})_i}{\Gamma \vdash \langle \bar{t} \rangle : \bigwedge \bar{\tau} \Rightarrow \bar{\bar{R}}}$$

The only really relevant rules here for expressiveness are the TP-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 TP-SUM rule in iterative form, but it actually matches an n-ary tuple against an n-ary struct type in one go.

In the TP-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 TP-IGNORE rule would suggest.

Argument typing

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

$$\overset{\textsc{arg-type}}{\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash x^{Y} : \tau \text{ arg}}}$$

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

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

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

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

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

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

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

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

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

$$\overset{\textsc{typ-and}}{\frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \land B \text{ prop}}} \qquad \overset{\textsc{typ-or}}{\frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \lor B \text{ prop}}} \qquad \overset{\textsc{typ-sep}}{\frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A * B \text{ prop}}}$$

TYP-WAND
$$\frac{\Gamma \vdash A \text{ prop} \qquad \Gamma \vdash B \text{ prop}}{\Gamma \vdash A \mathrel{-\!\!*} B \text{ prop}}$$

TYP-FORALL
$$\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash A \text{ prop}}{\Gamma \vdash \forall x : \tau,\ A \text{ prop}}$$

TYP-EXISTS
$$\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash A \text{ prop}}{\Gamma \vdash \exists x : \tau,\ A \text{ prop}}$$

TYP-POINTS-TO
$$\frac{\Gamma \vdash \ell : \mathbb{N}_{64} \quad \Gamma \vdash v : |\tau|}{\Gamma \vdash \ell \mapsto v \text{ prop}}$$

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 $x : \&^{\text{own}}\tau$, then x is itself just a number, but $\boxed{x : \&^{\text{own}}\tau}$ 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 $x : |\tau|$. 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 <- 1; x + x`" is elaborated into "$x \leftarrow 1;\ 1 + 1$", while "$x \leftarrow 1;\ x + x$" in the core logic means that the x being referred to is the one before the mutation. The surface syntax uses "`with x -> y`" annotations on mutations to allow referencing both the old and new versions of the variable.

Expression validity (pure expressions) $\boxed{\Gamma \vdash pe : \tau}$

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

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

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

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

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
$$\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \neg e : \text{bool}}$$

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

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

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

TYE-LT, TYE-LE, TYE-EQ

$$\frac{\tau, \tau' \in \{\mathbb{N}_s, \mathbb{Z}_s\} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 < e_2 : \mathsf{bool} \quad \Gamma \vdash e_1 \le e_2 : \mathsf{bool} \quad \Gamma \vdash e_1 = e_2 : \mathsf{bool}}$$

TYE-IF

$$\frac{\Gamma \vdash c : \mathsf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) : \tau}$$

TYE-STRUCT

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R}[e/x]}{\Gamma \vdash \langle e, \overline{e} \rangle : \sum x^{\gamma} : \tau, \bar{R}}$$

TYE-FUNC-CALL

$$\frac{\mathsf{func}\ f(\overline{R}) : \overline{S} \quad \Gamma \vdash \langle \overline{e} \rangle : \sum \bar{R}}{\Gamma \vdash f(\overline{e}) : \sum \bar{S}}$$

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

$\boxed{\Gamma; \delta \vdash e : \tau \dashv \delta'}$ $\boxed{\Gamma; \delta \vdash e \Rightarrow pe : \tau \dashv \delta'}$

TYE-VAR-MOVE

$$\Gamma; \delta, x^{\gamma} := pe : \tau \vdash x \Rightarrow pe : \tau \dashv \delta, x^{\gamma} := pe : |\tau|$$

TYE-MUT

$$\frac{\begin{array}{c}\Gamma; \delta \vdash e \Rightarrow pe : \tau \dashv \delta_1 \quad \forall z,\ (x \longrightarrow z) \notin \delta_1 \quad \Gamma \vdash \delta_2 \\ \Gamma; \delta_1, (x \longrightarrow y), (y^{\gamma} := pe : \tau) \vdash e : \tau' \mathrel{+\!\!\Rightarrow} \delta_2 \quad y \notin \delta_2\end{array}}{\Gamma; \delta \vdash (x^{\gamma} \leftarrow e\ \mathsf{with}\ y \leftarrow x;\ e) : \tau' \dashv \delta_2}$$

TYE-LET-PURE

$$\frac{\begin{array}{c}\Gamma; \delta \vdash e_1 \Rightarrow pe : \tau \dashv \delta_1 \quad \Gamma \vdash \tau', \delta_2 \\ \Gamma, x : |\tau|\,; \delta_1, x^{\gamma} := pe : \tau \vdash e_2 : \tau' \mathrel{+\!\!\Rightarrow} \delta_2\end{array}}{\Gamma; \delta \vdash (\mathsf{let}\ x^{\gamma} := e_1\ \mathsf{in}\ e_2) : \tau' \dashv \delta_2}$$

TYE-UNREACHABLE

$$\frac{\Gamma; \delta \vdash e : \bot \dashv \delta_1 \quad \Gamma \vdash \delta_2}{\Gamma; \delta \vdash \mathsf{unreachable}\ e : \tau \dashv \delta_2}$$

TYE-LET

$$\frac{\begin{array}{c}\Gamma; \delta \vdash e_1 : \tau \dashv \delta_1 \quad \Gamma \vdash t : \tau \Rightarrow \overline{R} \quad \Gamma \vdash \tau', \delta_2 \\ \Gamma, \overline{|R|}; \delta_1, \overline{R} \vdash e_2 : \tau' \mathrel{+\!\!\Rightarrow} \delta_2\end{array}}{\Gamma \vdash (\mathsf{let}\ t := e_1\ \mathsf{in}\ e_2) : \tau' \dashv \delta_2}$$

TYE-PROC-CALL

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

TYE-RETURN

$$\frac{\mathsf{self}(\bar{R}) : \bar{S} \quad \Gamma; \delta \vdash \langle \overline{e} \rangle : \sum \bar{S} \dashv \delta'}{\Gamma; \delta \vdash \mathsf{return}\ \overline{e} : \bot \dashv \delta'}$$

TYE-LABEL

$$\frac{\begin{array}{c}\forall i,\ \Gamma, \overline{k(\delta; \bar{R})}, (\bar{R})_i; \delta_i, (\bar{R})_i \vdash e_i : \bot \dashv \delta_i^2 \\ \Gamma, \overline{k(\delta; \bar{R})}; \delta^0 \vdash e' : \tau \dashv \delta^1\end{array}}{\Gamma; \delta^0 \vdash (\mathsf{label}\ \overline{k(\bar{R}) := e}\ \mathsf{in}\ e') : \tau \dashv \delta^1}$$

TYE-GOTO

$$\frac{\begin{array}{c}k(\delta'; \bar{R}) \in \Gamma \\ \Gamma; \delta \vdash \langle \overline{e} \rangle : \sum (\bar{R})_i \mathrel{+\!\!\Rightarrow} \delta'\end{array}}{\Gamma; \delta \vdash \mathsf{goto}\ k(\overline{e}) : \bot \vdash \delta'}$$

TYE-ASSERT

$$\frac{\Gamma \vdash e \Rightarrow pe : \mathsf{bool} \dashv \Gamma'}{\Gamma \vdash \mathsf{assert}\ e : pe \dashv \Gamma'}$$

TYE-TYPEOF

$$\frac{\Gamma \vdash e \Rightarrow pe : \tau \dashv \Gamma'}{\Gamma \vdash \mathsf{typeof}\ e : \boxed{pe : \tau} \dashv \Gamma'}$$

TYE-ENTAIL

$$\frac{\Gamma \vdash \langle \overline{e} \rangle : \mathop{\ast} \overline{A} \dashv \Gamma' \quad \vdash p : \mathop{\ast} \overline{A} \mathrel{-\!\!*} B}{\Gamma \vdash \mathsf{entail}\ \overline{e}\ p : B \dashv \Gamma'}$$

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.

3.5 Mutation application

The role of the $\Gamma; \delta \vdash e : \tau \dashv\!\!\Rightarrow \delta'$ 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 $\Gamma; \delta \vdash e : \tau \dashv\!\!\Rightarrow \delta'$ means $\Gamma; \delta \vdash e : \tau \dashv \delta_1$ and $\Gamma \vdash \delta_1 \Rightarrow \delta'$ for some δ_1. The reason we can't just use δ_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 $\boxed{\Gamma \vdash \delta \Rightarrow \delta'}$

CS-REFL
$$\Gamma \vdash \delta \Rightarrow \delta$$

CS-TRANS
$$\frac{\Gamma \vdash \delta_1 \Rightarrow \delta_2 \quad \Gamma \vdash \delta_2 \Rightarrow \delta_3}{\Gamma \vdash \delta_1 \Rightarrow \delta_3}$$

CS-DROP
$$\frac{\forall x, (x \longrightarrow y) \notin \delta}{\Gamma \vdash \delta, (y^Y := pe : \tau) \Rightarrow \delta}$$

CS-RENAME
$$\Gamma \vdash \delta, (x \longrightarrow y), (y^Y := pe : \tau) \Rightarrow \delta, (x^Y := pe : \tau)$$

CS-FORGET
$$\frac{\Gamma \vdash \Gamma[\overline{x \longrightarrow pe}]}{\Gamma \vdash \delta, \overline{x^Y := pe : \tau} \Rightarrow \delta, \overline{x^Y : \tau}}$$

This is a nondeterministic judgment, with the "goal" being to eliminate a particular variable and/or join with separate control flow which has assigned different values to the variables.

- The simplest way to drop a variable is with the CS-DROP rule, which works as long as this is a variable that was not obtained from a mutation.
- For variables that are obtained by mutation, we have a $x \longrightarrow y$ in the context, and we can drop its value while storing the result back in the original variable using the CS-RENAME rule.
- In order to join control flow, we also need to "forget" the value associated with a variable. For example, if one branch of an if statement sets $x \longleftarrow 1$ and the other sets $x \longleftarrow 2$, we are allowed to use these settings inside the blocks of the if statement but at the end they must agree about the setting of the variable as well as its properties. For this we use the CS-FORGET rule, which erases the information that $x := pe$ for several variables at once. This existentially quantifies over the variables $\overline{x}$ and reintroduces them so that we no longer have access to the value. For this to be sound, we have a side condition that says that the context remains true if we replace $\overline{x}$ with $\overline{pe}$, because the actual assignments to the variables in Γ have changed even though we are keeping the same type.

To see how this plays out, consider the code

$$x := 0, h : x \geq 0 \vdash \text{if } b \; \{ \; x \longleftarrow 1 \; \},$$

which desugars to "if b then $x^\top \longleftarrow 1$; () else ()". After the mutation, we have $x \longrightarrow x', x' := 1$ so we can apply CS-RENAME to get $x := 1$. But the else branch has $x := 0$ so we can't merge just yet. We can apply CS-FORGET to forget x, because $x : \mathbb{N}, h : x \geq 0 \vdash 1 : \mathbb{N}, 1 \geq 0$, provided the compiler knows how to synthesize these proofs. (The proof of $1 : \mathbb{N}$ is already supplied by $x \longleftarrow 1 : \mathbb{N}$, but $1 \geq 0$ is not immediately available.) If the compiler cannot find this proof, it can be supplied by:

$$x := 0, h : x \geq 0 \vdash \text{if } b \; \{ \; x \longleftarrow 1; \; h \longleftarrow (p : 1 \geq 0) \; \},$$

where p is a proof of $1 \geq 0$. In this case, we are using CS-RENAME on x and h simultaneously, so the side goal is the same but we get the $1 \geq 0$ goal for free from the typing condition on $h := (p : 1 \geq 0)$.

3.6 Top level typing

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

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

$$\text{OK-ZERO}\quad \Gamma \vdash \cdot \dashv \Gamma \qquad\qquad \text{OK-APPEND}\quad \frac{\Gamma \vdash \overline{it} \dashv \Gamma' \quad \Gamma' \vdash it \dashv \Gamma''}{\Gamma \vdash \overline{it}, it' \dashv \Gamma''}$$

Individual items are typed as follows:

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

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

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

$$\text{OK-FUNC, OK-PROC}\quad \frac{\mathbf{kw} \in \{\mathsf{func}, \mathsf{proc}\} \quad \Gamma \vdash \sum \overline{R}\ \mathsf{type} \quad \Gamma, \overline{R} \vdash \sum \overline{S}\ \mathsf{type} \quad \Gamma, (\mathsf{self}(\overline{R}) : \overline{S}), \overline{R};\ \overline{R} \vdash e : \bot \dashv \delta}{\Gamma \vdash \mathbf{kw}\ f(\overline{R}) : \overline{S} := e \dashv \Gamma',\ \mathbf{kw}\ f(\overline{R}) : \overline{S}}$$

3.7 Uninitialized data

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

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

$$\text{TY-MAYBE}\quad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \tau^?\ \mathsf{type}} \qquad\qquad \text{TYE-UNINIT}\quad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma; \delta \vdash \mathsf{uninit} : \tau^? \dashv \delta}$$

That's it. Note that $\tau \leq \tau^?$ because the typing predicate of $\tau^?$ is $\top$, so we can always satisfy the side condition of CS-FORGET when performing a strong update of $x : \tau^?$ to τ when we initialize it.

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 $\&^{\mathbf{sn}}\eta$. $x : \&^{\mathbf{sn}}\eta$ simply means that x is a pointer that points to η, which is a "place", a writable location. $\boxed{x : \&^{\mathbf{sn}}\eta} = \eta \mathbin{@} x$, where $\eta \mathbin{@} x$ means that η is stored in memory at location x; see section 4. (This is not the same as $x \mapsto \eta$, because η is a place, i.e. a direct reference to a variable in the context, not a value.) This predicate is duplicable, so $\&^{\mathbf{sn}}\eta$ is copy (and coercible to $\mathbb{N}_{64}$). We add the following:

$$\text{Type} ::= \cdots \mid \&^{\mathbf{sn}}\eta \qquad \text{Expr} ::= \cdots \mid {}^*e \mid \&e \qquad \&^{\mathbf{sn}}\eta \text{ copy} \qquad \boxed{x : \&^{\mathbf{sn}}\eta} = \eta \mathbin{@} x$$

$$\frac{\Gamma \vdash \eta \text{ place}}{\Gamma \vdash \&^{\mathbf{sn}}\eta \text{ type}}\,\textsc{ty-snp} \qquad \frac{\Gamma;\delta \vdash e : \&^{\mathbf{sn}}\eta \dashv \delta'}{\Gamma;\delta \vdash {}^*e \Rightarrow \eta \dashv \delta' \text{ place}}\,\textsc{tye-deref} \qquad \frac{\Gamma;\delta \vdash e \Rightarrow \eta \dashv \delta' \text{ place}}{\Gamma;\delta \vdash \&e : \&^{\mathbf{sn}}\eta \dashv \delta'}\,\textsc{tye-ref}$$

To use these generalized lvalues, we need operations to read and write them:

$$\frac{\begin{array}{c}\Gamma;\delta \vdash e \Rightarrow \eta \dashv \delta_1 \text{ place}\\ \Gamma;\delta_1 \vdash \eta \Rightarrow pe : \tau \dashv \delta_2\end{array}}{\Gamma;\delta \vdash e \Rightarrow pe : \tau \dashv \delta_2}\,\textsc{tye-read} \qquad \frac{\begin{array}{c}\Gamma;\delta \vdash e \Rightarrow \eta \dashv \delta_1 \text{ place}\\ \Gamma;\delta_1 \vdash (\eta \leftarrow pe;\ e_2) : \tau \dashv \delta_2\end{array}}{\Gamma \vdash (e \leftarrow pe;\ e_2) : \tau \dashv \delta_2}\,\textsc{tye-write}$$

We needed two new judgments above, $\Gamma \vdash \eta$ place, which asserts that η is a place in the context, and $\Gamma;\delta \vdash e \Rightarrow \eta \dashv \delta'$ place which asserts that e evaluates as an lvalue to place η (which may require transforming the code to add a temporary variable). The simplest example of a place is a variable $x \in \Gamma$, but one can also take a subpart of a struct or a slice of an array. However, note that ${}^*e$ is a place expression but not a place value; it evaluates according to TYE-DEREF.

3.8.2 Owned pointers. An owned pointer is 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.

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

$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \&^{\mathbf{own}}\tau \text{ type}}\,\textsc{ty-own} \qquad \frac{\Gamma \vdash t : \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash t' : \&^{\mathbf{sn}}t \Rightarrow \bar{S}'}{\Gamma \vdash \langle t, t' \rangle : \&^{\mathbf{own}}\tau \Rightarrow \bar{S}, \bar{S}'}\,\textsc{tp-own}$$

By using destructuring, it is possible to obtain a pointer such as $t : \&^{\mathbf{sn}}(a, b)$; this type asserts that a and b are contiguous in memory such that a single pointer can access them both. This type can itself be destructured as if it were $\&^{\mathbf{sn}}a * \&^{\mathbf{sn}}b$.

3.8.3 Mutable pointers. Before we can explain mutable pointers, we need the concept of a mutable parameter. We have already seen that the $\leftarrow$ operator can mutate variables inside the value context δ, but currently return will drop all mutated values and return only the return values in the function signature. In order to allow variables to be mutated through the function, we add the ability to mark a variable in the returns $\overline{S}$ as $\mathrm{out}^x\ y : \tau$, if x is a function parameter (which is itself marked as mut $x : \tau$). This has the meaning that the variable x will be mutated so that $\delta \vdash x \longrightarrow^* y$ when the function reaches the return.

The rule TYE-RETURN is unchanged, but we have a new rule for fulfilling an $\mathrm{out}^x\ y$ argument:

$$\frac{\delta \vdash x \longrightarrow^* y \quad \Gamma;\delta \vdash y : \tau \dashv \delta_1 \quad \Gamma;\delta_1 \vdash \langle \overline{e} \rangle : \textstyle\sum \bar{R}[pe/y]}{\Gamma \vdash \langle \overline{e} \rangle : \textstyle\sum(\mathrm{out}^x\ y^\gamma : \tau), \bar{R}}\,\textsc{tye-struct-out}$$

Here $\delta \vdash x \longrightarrow^* y$ means that $x \longrightarrow \cdots \longrightarrow y \not\longrightarrow$ according to the rename map in δ.

Conversely, when calling a function, the mut parameters get captured in the calling context, and changed to their out variants. Describing this is technically complicated so we will use a prose description. We define only the construct $\mathsf{let}\ \langle \overline{y}, t\rangle := F(\overline{e})\ \mathsf{in}\ e_2$ where t is a tuple pattern and $\overline{y}$ has the same length as the number of out parameters of F; that is, proc $F(\overline{R}) : \overline{\mathsf{out}^x\ y^Y : \tau}, \overline{S}$.

The arguments of F must be $e : \tau$ if $R = (x^Y : \tau)$, and must be $\eta : \tau$ place if $R = (\mathsf{mut}\ x^Y : \tau)$. If η is provided for argument x, and $\mathsf{out}^x\ y^Y : \tau'$ is among the out arguments of the function, and y is the corresponding element of the tuple in the let $\langle \overline{y}, t\rangle$ pattern match, then we perform an assignment $\eta \leftarrow y$ on return from the function. All these η places are disjoint because they were passed simultaneously to F, so there is no ambiguity about the order of writes. Finally, the result of the $F(\overline{e})$ invocation is pattern matched against the tuple pattern t and e_2 is executed.

The type $\&^{\mathbf{mut}}\tau$ is not a true type, but is allowed in function signatures to indicate a $\&^{\mathbf{sn}}\eta$ value where η is external to the function. The changes to η are a “side effect” and so we use the $\mathsf{out}^x\ y$ functionality from the previous section to support it.

In brief, if $x : \&^{\mathbf{mut}}\tau$ appears in the function arguments, we replace it by $\boxed{v} : \tau, x : \&^{\mathbf{sn}}v$ in the function arguments and add $\mathsf{out}_v\ \boxed{v'} : \tau$ at the beginning of the function returns. $\&^{\mathbf{mut}}\tau$ is not allowed to appear any other place than the top level of a function argument.

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 lifetime a is modeled roughly as a (precise, aka subsingleton) separating proposition P, with each $\mathsf{ref}^a\ x := pe : \tau$ being modeled as a place η for which $P \Rrightarrow (\eta := pe : \tau)$. That is, we can weaken P to obtain the fact that $\eta := pe : \tau$. (The relation $P \Rrightarrow Q$, which is a regular (not separating) proposition, is defined as $\vdash P \to (Q * \top)$.) Because P is a precise proposition, it satisfies $(P \Rrightarrow \exists x, Q) \to (\exists x, (P \Rrightarrow Q))$, which means we can pattern match on heap variables like regular variables, for example to obtain $\&\tau$ from $\&\&^{\mathbf{own}}\tau$. But this is only relevant for the semantic model; in the type checker we simply need some rules for how to manipulate these variables.

Syntactically, a lifetime can be either extern, referring to data outside the current context, or x, some variable in the context. These denote the scope of the borrow; a variable which is borrowed cannot be mutated. (Possible extensions include lifetimes with scope $\{x, y, z\}$ for creating data that spans multiple variables, and lifetimes with scope $x.\mathrm{field}$ in order to borrow only parts of a variable without locking the whole variable.) The proposition P from the previous paragraph is the implicit frame proposition in the extern case, and $x := pe : \tau$ from the value context at the time of the borrow in the case of x. (In the case of multiple variables, it is the separating conjunction of these $x := pe : \tau$ conditions and in the case of a subobject we destructure this proposition and pull out the $\eta := pe : \tau$ component.)

$$a \in \mathrm{Lft} ::= \mathsf{extern} \mid x \qquad \mathrm{Arg} ::= \cdots \mid \mathsf{ref}^a\ x : \tau \qquad \textsc{tp-sum-ref}\ \frac{\Gamma, \mathsf{ref}^a\ y : \tau \vdash \langle \bar{t} \rangle : \overline{R}[y/x] \Rightarrow \bar{S}}{\Gamma \vdash \langle y, \overline{t'} \rangle : \sum \mathsf{ref}^a\ x : \tau, \overline{R} \Rightarrow \mathsf{ref}^a\ y : \tau, \bar{S}}$$

$$\textsc{arg-ref}\ \frac{\mathrm{Var}(a) \subseteq \Gamma \quad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathsf{ref}^a\ x : \tau\ \mathsf{arg}} \qquad \textsc{tye-struct-ref}\ \frac{\Gamma; \delta \vdash e \Rightarrow (\eta := pe : \tau) \dashv\Rightarrow \delta_1\ \mathsf{read} \quad \Gamma; \delta_1 \vdash \langle \bar{e} \rangle : \sum \bar{R}[pe/x] \dashv \delta_2}{\Gamma; \delta \vdash \langle \eta, \bar{e} \rangle : \sum \mathsf{ref}^{\mathrm{Lft}(\eta)}\ x : \tau, \bar{R} \dashv \delta_2} \qquad \begin{aligned} \mathrm{Lft}(\mathsf{ref}^a\ x) &= a \\ \mathrm{Lft}(x) &= x \\ \mathrm{Lft}(\eta[pe]) &= \mathrm{Lft}(\eta) \end{aligned}$$

Here the $\Gamma; \delta \vdash e \Rightarrow (\eta := pe : \tau) \dashv\Rightarrow \delta'$ read judgment is a conjunction of $\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta_1$ place followed by $\Gamma \vdash \delta_1 \Rightarrow \delta'$, such that $\Gamma; \delta' \vdash \eta := pe : \tau$ read. That is, first we evaluate the place expression, then we use $\Gamma \vdash \delta_1 \Rightarrow \delta'$ to ensure that η is locked and readable at type τ, and the final judgment asserts that in the result state we can in fact read $\eta : \tau$.

$$\delta \in \mathrm{VCtx} ::= \delta, (\mathsf{ref}^a\ x := pe : \tau)$$

$$\textsc{cs-lock}\ \Gamma \vdash \delta, (x := pe : \tau) \Rightarrow \delta, (\mathsf{ref}^x\ x := pe : \tau) \qquad \textsc{cs-unlock}\ \frac{\forall y, (\mathsf{ref}^x\ y := -) \notin \delta}{\Gamma \vdash \delta, (\mathsf{ref}^x\ x := pe : \tau) \Rightarrow \delta, (x := pe : \tau)}$$

$$\textsc{tyr-var}\ \frac{(\mathsf{ref}^a\ x := pe : \tau) \in \delta}{\Gamma; \delta \vdash x := pe : \tau\ \mathsf{read}} \qquad \textsc{tyr-place}\ \frac{(\mathsf{ref}^a\ x := pe : \tau) \in \delta}{\Gamma; \delta \vdash \eta := pe : \tau\ \mathsf{read}} \qquad \textsc{tye-read-ref}\ \frac{\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta'\ \mathsf{place} \quad \Gamma; \delta' \vdash \eta := pe : \tau\ \mathsf{read}}{\Gamma; \delta \vdash e \Rightarrow pe : |\tau| \dashv \delta'}$$

Note that we cannot move out a value from a ref variable, which is reflected in the use of $|\tau|$ in TYE-READ-REF. We also cannot mutate a ref, meaning that while a variable is locked (meaning that it is represented in the value context as a $\mathsf{ref}^x\ x$), mutation is not possible; however it is possible to mutate a variable that is currently locked by first unlocking it using the CS-UNLOCK rule, which requires first deleting all the heap variables that reference x using the CS-DROP rule.

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

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

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

3.9 Arrays

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

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

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

$$\textsc{ty-array}\ \frac{\Gamma \vdash \tau\ \mathsf{type} \quad \Gamma \vdash n : \mathbb{N}_s}{\Gamma \vdash \mathsf{array}\ \tau\ n\ \mathsf{type}}$$

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.

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

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:

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

with the following typing rules:

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

$$\textsc{stk-id}\quad \phi \vdash \mathsf{id} : \phi \qquad \textsc{stk-fst}\ \frac{\phi \vdash s : \phi_1 * \phi_2}{\phi \vdash s.0 : \phi_1} \qquad \textsc{stk-snd}\ \frac{\phi \vdash s : \phi_1 * \phi_2}{\phi \vdash s.1 : \phi_2} \qquad \textsc{stk-left}\ \frac{\phi \vdash s : \phi_1 \cup \phi_2}{\phi \vdash s.l : \phi_1} \qquad \textsc{stk-right}\ \frac{\phi \vdash s : \phi_1 \cup \phi_2}{\phi \vdash s.r : \phi_2}$$

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 $\overline{k(\delta, \bar{R})}$ in Γ become (persistent) propositions jump-ok$(\ell : (\delta, \bar{R}) \to \bot)$ 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`).
- Each variable $x^{\gamma} : |\tau|$ becomes either $x^{\gamma} : |\tau|$ or $x^{\top} : |\tau| \ @ \ \mu$ 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.

For the value context, we store no additional information regarding the rename map, but each $x^{\gamma} := pe : \tau$ corresponds to the separating proposition $\boxed{pe : \tau}$.