Metamath Zero

From Logic, to Proof Assistant, to Verified Compilation

Mario Carneiro

Abstract

As the usage of theorem prover technology expands, so too does the reliance on correctness of the tools. Metamath Zero is a verification system that aims for simplicity of logic and implementation, as well as efficiency of verification. It is formally specified in its own language, and supports a number of translations to and from other proof languages. This dissertation describes the abstract logic of Metamath Zero, essentially a multi-sorted first order logic, as well as the binary proof format and the way in which it can ensure essentially linear time verification while still being concise and efficient at scale.

In addition, we use the logic as the foundation for a proof assistant called Metamath One, and use it to write a modest library of theorems; this library is then used to build a proof-producing compiler, the Metamath C compiler, which allows writing programs at a high level in a type system strong enough to state and prove functional correctness properties, and lowers the proofs all the way down to machine code and a specification of the hardware.

Ultimately, we intend to use Metamath C to implement a verifier for Metamath Zero which is implementation-correct down to the binary executable, so it can be used as a root of trust for more complex proof systems.

Contents

Introduction

This thesis is about the intersection of two fields, mathematics and computer science, right at the foundations. By pursuing an optimal solution to the problem of trusted computation, with minimal detours for half-measures, we get an architecture that is small (small enough to be understood end-to-end), efficient (competitive with the fastest verifiers in use today), and with every component subject to rigorous evaluation and proof.[1] The architecture, called Metamath Zero (MM0), is not uniquely defined by these features, and indeed I would like there to be more systems out there like it to explore the design space. But hopefully it can at least be used to set the baseline (or rather, a high standard) for what we can expect from a verification system in the limit.

[1] Of course, there are caveats to any statement, and we will get to them. In particular, Gödel's incompleteness theorem is often used to argue that a bootstrapping system like the one I will describe is a pointless endeavor, but I will argue that the limitation is of a technical nature, and with a suitably modified statement we can still construct something morally equivalent to a self-proving system.

A *verification system* is a computer program which validates the correctness of mathematical derivations. In other words, a proof-checker. Verification systems often come bundled together with a *proof assistant* or *interactive theorem prover* (ITP), which is a program to assist in the construction of derivations that can be verified by the system. In general, it is much harder to construct a good proof assistant than a good verification system, because as the name suggests, an ITP has to interact with the user, who is guiding the computer towards the proof, and an unbounded amount of work can be poured into the proof assistant to help it find proofs better or make the system easier to use. By contrast, a verification system has a rigidly defined input and output, and interacts minimally with the human operator, which makes it a much simpler object of study.

0.1 How to trust a proof checker

The basic idea behind using a computer to check proofs is that they can do so faster and with fewer errors than a human. This, in turn, means that they can be put to use to check proofs that would be infeasible for a human to check. The first major example of this was the proof of the four-color theorem by Appel and Haken in 1976, which has been simplified since but still involves an enumeration of 633 configurations. But we should consider how, exactly, we can come to believe that a proof containing a computer component in fact validates the theorem in question.

Let us consider a dialog between two humans Penny the Prover and Victor the Verifier. Penny has discovered a marvelous proof of a theorem *T*, and wishes to convince Victor that *T* is true. Victor is interested in *T* and cautiously optimistic about this news of Penny's, but would like to see for themself that *T* is true. Unfortunately, *T* has a long and complicated proof, and Penny realizes that Victor may get confused while reading it and start to doubt their own faculties, or give up altogether upon seeing the length of the proof.

At this point we will make another simplifying assumption, which is that Victor's time is much more valuable than Penny's, so Penny is willing to put in lots of extra work to make Victor's job as easy as possible.[2] If the proof, like Appel and Haken's, requires an enumeration of thousands or millions of cases, then Penny is almost forced to involve a computer in the proof, because it can go through all the cases tirelessly, while a human would not have a chance of making it through all the cases before dropping to sociological estimates of correctness like "Penny is conscientious and trustworthy so probably the next 100 cases are okay."

[2] This is not an entirely unreasonable assumption. Perhaps Penny is an early-career researcher while Victor is a busy professor. Most proofs are also read many more times than they are written, so it makes sense to optimize for the reader. Realistically, there are limits on the effort that Penny can put into the proof, and we will get back to this later.

We need a computer in the loop then, but what does that entail? Enter Robo-Victor. Robo-Victor is just like Victor, except it's a computer. It can read the million case proof much quicker than Victor, and since Penny knew that Robo-Victor was coming they prepared a proof in a format that Robo-Victor could read. But Victor is still not convinced, because they're not Robo-Victor, so we have not yet accomplished our goal. Penny now needs to convince Victor that Robo-Victor is a careful reader, and that they aren't partners in a confidence game to fool Victor.

The assertion that Robo-Victor is a careful reader (that is, that it doesn't validate bad proofs) is also a mathematical assertion, so Penny could try to use the same method to convince Victor of this fact. But Victor is a busy person, they don't have time for infinite regresses. Of course, the original method is still on the table: Penny can write a paper proof and give it to Victor who reads it and is convinced. It was only the size of the original proof that lead to the investigation of alternative methods, and we have reduced the problem to an assertion about Robo-Victor, which no longer involves the million-case proof. As long as it is easy for Victor to verify that Robo-Victor does what it was designed to do we can run it on the big proof, wait for the green light and then Victor will also be convinced.

We can do a bit better though: A formal proof of correctness of Robo-Victor (checked by Robo-Victor itself) is not really convincing on its own, any more than a suspect asserting their truthfulness, but the combination of the two methods is better than either one individually. Victor can read Robo-Victor's code to ensure it works as expected, then if Robo-Victor reads the code as well and agrees, Victor is reassured. Because Victor and Robo-Victor have completely different failure modes, Victor can be more confident with Robo-Victor's help, even though Robo-Victor is the suspect.

Of course, this is not even taking into account those sociological measures of correctness, which certainly apply to Robo-Victor. All the common proof assistants today rely on this mechanism. Coq, Isabelle, Lean are trustworthy because lots of people use them, they are maintained by teams of smart people, no one has proved false in them for a while, etc. Proofs in the real world are often a combination of factors like this.

We can extract a plan of action from this little allegory. Notice that the criteria going into the design of Robo-Victor do not really depend on the original million-case proof that motivated Penny. As long as Robo-Victor is fast enough, it is worthwhile to deploy on any large-enough proof.[3] Our goal is to solve the meta-problem, to make Victor's job as easy as possible, abstracted over possible target theorems and proofs. The main criteria are:

[3] For *very* large proofs, one might need a Super-Victor that is more complex and more efficient than Robo-Victor. Luckily, Robo-Victor can also help Victor to believe that Super-Victor works as designed.

- Victor needs to read Robo-Victor's code, so it should be short, clear, and well documented.
- Victor needs to run (or observe the run of) Robo-Victor and see that it is satisfied with the proof (both the original million-case proof as well as Robo-Victor's correctness proof), so Robo-Victor should be fast.[4]
- Victor needs to verify that Robo-Victor is in fact verifying Penny's proof, so the statement of the theorem should be expressed in either the output or the input in a clear and readable way, which can't be confused with a different theorem.[5]
- Penny wants the provability assertions coming from Robo-Victor to be relevant to the original proof, so Robo-Victor should be able to validate theorems in Penny's proof system.[6]

[4] The speed requirements are not extreme, because we can bootstrap faster versions if necessary, but we are lower bounded by the time that Robo-Victor takes to verify either its own code (for the bootstrap path) or the original proof (for the direct path).

[5] A surprisingly large number of proof assistants fail this test, which is also known as "Pollack-consistency" after:

Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012

[6] This is a tricky one, because we don't know what Penny wants to prove. However, we can get around this by using a "universal" formal system which can model other formal systems. As it happens, just about every practical formal system already meets this criterion.

0.2 *Bootstrap an existing proof language?*

The goal to have a self-verifying system with low dependencies puts certain constraints on the design of the language, and outright excludes the majority of proof languages in common use.

- Coq, Lean, and Isabelle, while having a strong tradition of program verification, sacrifice simplicity of the logical kernel for this, which makes the task of writing a verifier nontrivial and a verified verifier nearly impossible.
- HOL4 cannot be written off because the CakeML[7] system proves that a verified verifier is within reach, but it is still very complex and pays for this with a very expensive bootstrap. (It also isn't actually self-verifying, but rather HOL Light verifying at the moment.)
- HOL Light is a promising candidate because of its simple kernel, and the logical core has been verified in HOL Light. There is still a fair amount of interpretation required to close this bootstrap, since HOL Light relies on the OCaml type system for soundness and this is not checked. As mentioned, there is also a verification of HOL light in HOL4 that goes all the way to the binary, so if HOL light can be used to verify HOL4 code then this could also be a way to close the bootstrap.
- Metamath is also a simple kernel system, but it has no program verification support and no direct support even for programmability, relying on external programs to produce proofs.

[7] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, January 2014

The Metamath Zero system consists of three separate components to address our constraints.

- Metamath Zero (MM0), the logic, is very closely related to Metamath and retains its simple kernel, adding the minimum features required to allow for trusted specifications of large scale developments.
- Metamath One (MM1), the proof assistant, addresses the problems with Metamath's lack of programmability by layering a tactic language on top of a live user interface modeled after Lean. This makes it easy to write proofs that meet MM0's stringent requirements while keeping these features out of the kernel.
- Metamath C (MMC), the programming language, is a language for writing programs that are intended for formal verification. It is similar to languages like Dafny or Why3, but with C/Rust-like semantics with native support for refinement types and ghost variables. The MMC compiler will read programs in this language and produce MM0 proof objects asserting the correctness of a block of bytes representing a binary executable.

They relate to the allegory of Penny and Victor in the following way:

- Penny is a mathematician or proof engineer who uses MM1 to write MM0 proofs;

- Victor is an outsider or proof auditor who would like to know that MM0 proofs actually establish a target claim;
- Robo-Victor is the MM0 verifier;
- I am "Meta-Penny," playing the role of Penny as regards the one-step-removed goal to prove the correctness of Robo-Victor, and MMC is a tool in that proof.

The next few chapters roughly follow the structure of these components:

- Chapter 1 describes MM0, starting from some examples, to the formal logic, and then to the low level implementation of the kernel and some of its characteristics.
- Chapter 2 describes MM1, with some examples and an overview of the features, some examples on how to use the language, and then shows the actual library of mathematical theorems developed in MM1 in service of the other parts of the project.
- Chapter 3 describes MMC, going through some examples and then showing all the interacting constructs that make it a useful verified programming language.
- Chapter 4 describes the MMC compiler, treating Chapter 3 as the problem statement and explaining how the compiler can take that whole language and reduce everything down to formal proofs in Peano arithmetic.
- Chapter 5 discusses some of the things that we can do with these languages, including some current projects as well as future directions.

1 Metamath Zero: The Logic

At the core of a verification system is a logic that can be checked mechanically. We require a language that has a simple metatheory, is fast to execute, scales at least to program correctness proofs, is automation-friendly, and clearly expresses result theorems and specifications such that the result can be audited by humans.

1.1 Metamath

Note: Readers uninterested in Metamath and the way MM0 relates to it can safely skip to section 1.3 without loss of context.

As its name suggests, Metamath Zero is based on Metamath,[1] a formal system developed by Norman Megill in 1990. Its largest database, `set.mm`, is the home of over 30000 proofs in ZFC set theory. In the space of theorem prover languages, it is one of the simplest, by design.

[1] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, 2019

The name "Metamath" comes from "metavariable mathematics," because the core concept is the pervasive use of metavariables over an object logic. An example theorem statement in Metamath is

$$\vdash (\forall x\, (\varphi \to \psi) \to (\forall x\, \varphi \to \forall x\, \psi))$$

which has three "free metavariables:" x, φ, and ψ. φ and ψ range over formulas of the object logic (let us say first order logic formulas like $\forall v_0\, v_0 = v_1$), and x ranges over variables of the object logic (that is, x can be $v_0, v_1, \ldots$).

However, this object logic never appears in actual usage. Rather, a theorem is proved with these metavariables, and then it is later applied with the metavariables (simultaneously) substituted for expressions that will contain more metavariables. For example one could apply the above theorem with the substitution $\{x \mapsto y,\ \varphi \mapsto \forall y\, \varphi,\ \psi \mapsto x = y\}$

to get:

$$\vdash (\forall y\, (\forall y\, \varphi \to x = y) \to (\forall y\, \forall y\, \varphi \to \forall y\, x = y))$$

which again contains metavariables (in this case x, y, φ) that can be further substituted later.

One consequence of the fact that variables like x are themselves "variables ranging over variables" is that in a statement like $\forall x\, x = y$, the variable y may or may not be bound by the $\forall x$ quantifier, because x and y may be the same variable. In order to express that two variables are different, the language includes "disjoint variable provisos" $A \mathbin{\#} B$, which may be used as preconditions in theorems and assert that variables A and B may not be substituted for expressions containing a common variable. This is usually seen in the special cases $x \mathbin{\#} y$, asserting that x and y are not the same variable, and $x \mathbin{\#} \varphi$, asserting that the substitution to φ does not contain the variable that x is substituted to.

When a theorem is applied, a substitution σ of all the variables is provided, and for each pair of variables $A \mathbin{\#} B$, it is checked that for every pair of variables $v \in \sigma(A), w \in \sigma(B)$, the disjoint variable condition $v \mathbin{\#} w$ is in the context. (This is why the relation is termed a "disjoint variable condition": if $A \mathbin{\#} B$ then the set of variables in substitutions to A and B must be disjoint.)

This is essentially the whole algorithm. There is no built in notion of free and bound variable, proper substitution, or alpha renaming — these can all be defined in the logic itself. It turns out that this is not only straightforward to implement (which explains why there are 17 known Metamath verifiers written in almost as many languages), but the fundamental operation, substitution, is effectively string interpolation in the sense of `printf`, which can be done very efficiently on modern computers. As a result, Metamath boasts some of the fastest checking times of any theorem prover library; the reference implementation, `metamath.exe`, can check the `set.mm` database of ZFC mathematics in about 8 seconds, and the fastest checker, `smm`, has performed the same feat in 0.7 seconds (on a 2-core Intel i5 1.6GHz). (This is a reported number from an older machine on an older and smaller `set.mm`. We reran `smm` single threaded on a Intel i7 3.9 GHz and the latest version of `set.mm`, and obtained 927 ± 28 ms.)

1.2 *Shortcomings of Metamath*

The primary differences between Metamath and Metamath Zero lie in the handling of first order variables ("variables over variables" from the previous section), expression parsing, and definitions, so some attention is merited to the way these are handled. In each case, Metamath chooses the simplest course of action, possibly at the cost of not making a statement as strong as one would like.

1.2.1 *Bundling*

As has been mentioned already, variables can alias, which leads to a phenomenon known as "bundling" in which a theorem might mean many different things depending on how the variables are substituted. For example, $\vdash \exists x\, x = y$ is an axiom in `set.mm` with no disjointness assumptions on x and y. There are essentially two different kinds of object language assertions encoded here. If $i \neq j$, then $\vdash \exists v_i\, v_i = v_j$ asserts that there exists an element equal to v_j, and when the indices are the same, $\vdash \exists v_i\, v_i = v_i$ asserts that there exists an element that is equal to itself. As it happens, in FOL both of these statements are true, so we are comfortable asserting this axiom.

Nevertheless, there is no easy way to render this as a single theorem of FOL, except by taking the conjunction of the two statements, and this generalizes to more variables – a bundled theorem containing n variables with no disjointness condition is equivalent (in the intended semantics) to B_n shadow copies of that theorem in FOL, where B_n is the nth Bell number, counting the number of ways that n elements can be partitioned into groups, depending on whether they are mapped to the same or different variables. The Bell numbers grow exponentially, $B_n = e^{O(n \ln n)}$, so this is at least a theoretical problem.

From the point of view of the Metamath user, this is not actually a problem – this says that Metamath in theory achieves *exponential compression* over more traditional variable handling methods, in which variables with different names are always distinct. However, it is a barrier to translations out of Metamath, because of the resulting exponential explosion.

However, this is not a problem in practice, because the theoretically predicted intricately bundled theorems aren't written. Usually all or almost all first order variables will be distinct from each other, in which case there is exactly one corresponding FOL theorem (up to alpha renaming). In order to ease translations, MM0 requires that all first order variables be distinct, and shoulders the burden of unbundling in the translation from Metamath to MM0 (see section 5.1.1).

1.2.2 *Strings vs trees*

Metamath uses strings of tokens in order to represent expressions. That is, the theorem $\vdash (\varphi \to \varphi)$ is talking about the provability of the expression consisting of five tokens [`(`, `ph`, `->`, `ph`, `)`], with the initial constant `|-` distinguishing this judgment from other judgments (for example $\vdash \varphi$ asserts that φ is provable, while wff φ asserts that φ is a well formed formula (wff)). The upshot of this is that parsing is trivial; spaces between tokens are mandatory so it is often as simple as `tokens = mm_file.split(" ")`. This makes correctness of the verifier simpler because the Metamath specification lines up closely with the internal data representation.

However, this leads to a problem when interpreting expressions as formulas of FOL. The axioms that define the wff φ judgment can be interpreted as clauses of a context-free grammar, and when that grammar is unambiguous there is a one-to-one relationship between strings and their parse trees, which are identified with the proofs of wff φ judgments [2]. So in effect, parsing is not required because the parses are provided with the proof. But unambiguity of a context-free grammar, though true for `set.mm` [3], is undecidable in general, yet is soundness critical — if parentheses were omitted in the definition of wff $\varphi \to \psi$ (that is, if the formation rule for wffs included the clause "if the strings u and v are wffs then u,'$\to$',v is a wff"), there would be two parses of $\bot \to \bot \to \bot$, and by conflating them it is not difficult to prove a contradiction.[4]

[2] Mario Carneiro. Models for Metamath. presented at CICM 2016, 2016

[3] Mario Carneiro. Grammar ambiguity in `set.mm`. 2013

[4] Appendix A contains an example of a contradiction stemming from an ambiguous grammar.

Metamath Zero uses trees (or more accurately dags, directed acyclic graphs) to represent expressions, which has some other side benefits for the proof format (see section 1.6). This on its own is enough to prevent ambiguity from leading to unsoundness. However, this means that an MM0 verifier requires a dynamic parser for its operation, which we will discuss in more detail in section 1.5.

1.2.3 *Definitions*

In Metamath, a definition is no more or less than an axiom. Generally a new definition begins with an axiom defining a new syntax constructor, for example wff $\exists! x\, \varphi$, and an axiom that uses the $\leftrightarrow$ symbol to relate this syntax constructor with its "definition," for example

$$y \mathrel{\#} x,\ y \mathrel{\#} \varphi \quad \vdash \exists! x\, \varphi \leftrightarrow \exists y\, \forall x\, (\varphi \leftrightarrow x = y).$$

Once again, the correctness of these definitional axioms is soundness critical but not checked by the verifier. Definitions such as the above definition of $\exists!$ are conservative and eliminable (this is a metatheorem that can be proved outside Metamath), and by convention almost all definitions in `set.mm` have a syntactic form like this, that is, a new constructor $P(\bar{x})$ is introduced together with an axiom $y_i \mathrel{\#} x_j,\ y_i \mathrel{\#} y_j \vdash P(\bar{x}) \leftrightarrow \varphi(\bar{x}, \bar{y})$, where the additional variables $\bar{y}$ (disjoint from $\bar{x}$ and each other) are all bound in the FOL sense.

This convention is sufficiently precise that there is a tool that checks these criteria, but this goes beyond the official Metamath specification, and only one of the 17 verifiers (the `mmj2` verifier) supports this check. This effectively means that MM verification in practice extends beyond the narrow definition of MM verification laid out in the standard.

Metamath Zero bakes in a concept of definition, which necessitates a simple convertibility judgment. It also requires an identification of variables that are "bound in the FOL sense," which means that it cannot completely ignore the notion of free and bound variables, at least when checking definitions.

1.3 MM0 primer

Before we get to the formal definition, it will help to have some simple examples of `.mm0` files to get a sense for the language. Figure 1.1 shows a MM0 file which declares the basics of propositional logic.

```
delimiter $ ( ~ $  $ ) $;
strict provable sort wff;
term im (a b: wff): wff; infixr im: $->$ prec 25;
term not (a: wff): wff; prefix not: $~$ prec 40;

-- The Lukasiewicz axioms for propositional logic
axiom ax_1 (a b: wff): $ a -> b -> a $;
axiom ax_2 (a b c: wff):
  $ (a -> b -> c) -> (a -> b) -> a -> c $;
axiom ax_3 (a b: wff):
  $ (~a -> ~b) -> b -> a $;
axiom ax_mp (a b: wff):
  $ a -> b $ >
  $ a $ >
  $ b $;

-- Assert that ‘P -> P‘ is provable
theorem id (P: wff): $ P -> P $;
```

Figure 1.1: An example file `prop.mm0`, which defines propositional calculus and asserts that $a \to a$ is provable from these axioms.

All mathematical text is enclosed in `$` characters. The `delimiter` keyword sets up the lexer to allow no space following a '`(`' or '$\sim$' character and before a '`)`'; spaces are otherwise required to separate tokens inside a math string.

The declaration `strict provable sort wff;` creates a new sort named `wff`, which can be used as the type of axioms and theorems (`provable`) and which does not admit binding variables (`strict`).

The next two lines declare two term constructors called `im` and `not`, along with notations '`a -> b`' and '`∼a`' to denote them.[5]

[5] Note that unlike Metamath, we do not need to include parentheses in this definition, because the mathematics parser explicitly includes a precedence system. We declared `im` as right associative using the `infixr` keyword.

The next group of lines declares several *axioms*. This consists of a sequence of binders like `(a b: wff)` which declare that *a* and *b* are schematic variables ranging over sort `wff`, and this is followed by the expression which is asserted, like '`a -> b -> a`'. In the case of the `ax_mp` axiom (modus ponens), because this is an inference rule we have two *hypotheses* '`a -> b`' and '`a`' from which we derive '`b`'. These are separated from the conclusion by a short arrow '`>`' (*outside* the math quotations).

The final line is a `theorem`, which has exactly the same syntax as `axiom`. A MM0 file will generally contain some axioms and then some theorems, and the meaning of the specification is that the theorems follow from the axioms. Crucially, since `.mm0` is a *specification* file format, it does not contain proofs for the theorems that are asserted. These are provided separately, and the job of a MM0 verifier is to use the proof file to validate the `.mm0` specification file. This is a separation of concerns, so that the `.mm0` file contains only those things that must be validated by a human while the proof file can focus on being good for computer validation.

Two other features of MM0 that will be important later which are not represented in this example are definitions and first order (binding) variables. To demonstrate this, we extend Figure 1.1 to encompass predicate calculus as well in Figure 1.2.

```
-- Predicate logic (on nat)
sort nat;
term al {x: nat} (P: wff x): wff; prefix al: $A.$ prec 41;

def ex {x: nat} (P: wff x): wff = $ ∼(A. x ∼P) $;
prefix ex: $E.$ prec 41;

term eq (a b: nat): wff; infixl eq: $=$ prec 50;

axiom ax_gen {x: nat} (P: wff x): $ P $ > $ A. x P $;
axiom ax_4 {x: nat} (P Q: wff x):
  $ A. x (P -> Q) -> A. x P -> A. x Q $;
axiom ax_5 {x: nat} (p: wff): $ p -> A. x p $;
axiom ax_6 (a: nat) {x: nat}: $ E. x x = a $;
axiom ax_7 (a b c: nat): $ a = b -> a = c -> b = c $;
axiom ax_11 {x y: nat} (P: wff x y):
  $ A. x A. y P -> A. y A. x P $;
```

Figure 1.2: Extension of Figure 1.1 to include first order logic operators and a selection of axioms about them.

We introduce another sort `nat`, but this time it is neither `provable` (because only propositions can be proven, not terms) nor `strict` (because we will be making use of binding variables of sort `nat`).

Binders using braces like `{x: wff}` denote first order or binding variables. The declaration `term al {x: nat} (P: wff x): wff;` says that `al` takes a name x and an expression $P(x)$ with possibly free occurrences of x of sort `wff`, and binds those occurrences to produce another expression of sort `wff`. The x appearing in `(P: wff x)` says that P can depend on x here.

In axioms, we use similar binders to express constraints on the legal substitution instances of the axioms. For example `ax_gen` can be used with the substitution $\{x \mapsto y, P \mapsto y = z\}$ to assert that if $\vdash y = z$ is derivable then so is $\vdash \forall y\ y = z$. On the contrary, in `ax_5`, which has binders `{x: wff} (p: wff)`, because there is no x in the binder for p, we are not permitted to use the substitution $\{x \mapsto y, p \mapsto y = z\}$ like before, because p must not contain a free occurrence of x.

1.4 *The MM0 formal system*

MM0 is intended to act as a schematic metatheory over multi-sorted first order logic. This means that it contains *sorts*, two kinds of *variables*, *expressions* constructed from *term constructors* and *definitions*, and *axioms* and *theorems* using expressions for their hypotheses and conclusion. Theorems have *proofs*, which involve applications of other theorems and axioms.

1.4.1 *Sorts*

An MM0 file declares a (finite) collection of sorts, as given by `sort` declarations. Every expression has a unique sort, and an expression can only be substituted for a variable of the same sort. There are no type constructors or function types, so the type system is finite. (Higher order functions are mimicked using open terms, see section 1.4.2.) We use s to denote sorts in the grammar.

1.4.2 *Variables*

MM0 distinguishes between two different kinds of variables. One may variously be called names, first order variables or bound/binding variables. They will be denoted in this chapter with letters $x, y, z, \dots$. They are essentially names that may be bound by quantifiers internal to the logic. "Substitution" of names is α-conversion; expressions cannot be substituted directly for names, although axioms may be used to implement this action indirectly.

The other kind of variable may be called a (schematic) metavariable or second order variable, and these may *not* be bound by quantifiers;

they are always implicitly universally quantified and held fixed within a single theorem, but unlike names, they may be directly substituted for an expression. We use $\varphi, \psi, \chi, \dots$ to denote schematic metavariables in the grammar.

In FOL, the notation $\varphi(\bar{x})$ is used to indicate that a metavariable is permitted to depend on the variables $\bar{x}$, and sometimes but not always additional "parameter" variables not under consideration. In MM0, we use a binder $\varphi : s\,\overline{x}$, where s is the sort and $\overline{x}$ are the *dependencies* of φ, to indicate that φ represents an open term that may reference the variables $\overline{x}$ declared in the context. This is opposite the Metamath convention which requires mentioning all pairs of variables that are *not* dependent, but it is otherwise a merely cosmetic change. Such a variable may also be glossed as a pre-applied higher order variable; for example a variable of type $\varphi : \mathsf{wff}\ x$ can be interpreted as a predicate $P : U \to \mathsf{bool}$ where every occurrence of φ in the statement is replaced with $P\ x$.

1.4.3 Abstract syntax

The expression grammar in MM0 is quite simple:

$$e ::= x \mid \varphi \mid f\,\overline{e}$$

That is to say, an expression is either a first order variable, a second order variable (note we do not write $\varphi(x)$ as in FOL, it is just φ), or a term constructor f (introduced by a previous `term` or `def` declaration) applied to zero or more subexpressions $\overline{e}$ (and we prefer the Haskell/Lisp-style adjacency notation for function application here).

This applies even to binding notations like $\mathsf{al}\ x\ p$, which has two arguments, a name x and a subexpression p which can contain x. So for example the expression $\forall y\, y = z$ is represented as $(\mathsf{al}\ y\ (\mathsf{eq}\ y\ z))$.

Expressions are interpreted in a context Γ, which declares the first and second order variables to be used in the expression:

$$\Gamma ::= \cdot \mid \Gamma,\ x : s \mid \Gamma,\ \varphi : s\,\overline{x}$$

This reflects the concrete syntax `{x: s}` and `(ph: s x)` for binders that we saw in section 1.3.

A *statement* is an expression which has a `provable` sort. We will use A, B to denote statements. $\Delta ::= \overline{A}$ is a list of hypotheses; in the concrete syntax these can either be unnamed, as in `$ a -> b $ > ...`, or named using a binder `(major_premise: $ a -> b $)`. (The names are never referenced within an `.mm0` file because the proofs of theorems are not given, but they are more useful in `.mm1` files, see Chapter 2.) In the abstract syntax we will have no need for the names.

The valid declarations are:

$\delta ::=$ sort s	sorts
$\mid$ term $f(\Gamma) : s\,\overline{x}$	axiomatic term constructors
$\mid$ def $f(\Gamma) : s\,\overline{x}$	abstract definitions
$\mid$ def $f(\Gamma) : s\,\overline{x} = \overline{y : s'}.\, e$	definitions
$\mid$ axiom $(\Gamma; \Delta \vdash A)$	axioms
$\mid$ theorem $(\Gamma; \Delta \vdash A)$	theorems

The syntax for a full file is then $E ::= \overline{\delta}$; this is also called an *environment*. We have already discussed sort declarations.[6]

[6] We omit sort modifiers to simplify the presentation, and assume the most permissive settings.

`term` and `def` introduce new term constructors which can be used in expressions. We supply a context Γ (a sequence of binders) and a result type (possibly including dependencies) to create a maybe binding term constructor. A simple definition like ‘and’ would look like:

$$\text{def and}(\varphi : \mathtt{wff}, \psi : \mathtt{wff}) : \mathtt{wff} = \mathtt{not}\,(\mathtt{im}\,\varphi\,(\mathtt{not}\,\psi)).$$

The $\overline{y : s'}$ part of def has not been used thus far. This is a list of first order “dummy” variables that are permitted to appear bound in e. All variables appearing in e must be declared, even bound variables. An example of a definition with a bound variable is the unique existence predicate, written here as both concrete and abstract syntax:

```
def eu {x: nat} (p: wff x) {.y: nat}: wff =
  $ E. y A. x (p <-> x = y) $;
```

$$\text{def eu}(x : \mathtt{nat}, p : \mathtt{wff}\ x) : \mathtt{wff} = y : \mathtt{nat}.$$
$$\mathtt{ex}\,y\,(\mathtt{al}\,x\,(\mathtt{iff}\,p\,(\mathtt{eq}\,x\,y)))$$

In the concrete syntax, these variables are denoted with a prefix dot as in `{.y: nat}`, and they can appear anywhere in the binder list but regular variables cannot depend on them.

Definitions can also be “abstract,” meaning that the definition itself is not specified, but there must exist a concrete definition making all theorems to follow true. This can be used to specify an operation by axioms. For example, we can define `and` axiomatically by asserting that some term exists that has the properties of conjunction:

```
def an (a b: wff): wff; infixl an: $/\$ prec 34;
theorem anl (a b: wff): $ a /\ b -> a $;
theorem anr (a b: wff): $ a /\ b -> b $;
theorem ian (a b: wff): $ a -> b -> a /\ b $;
```

We could prove this specification by using `~(a -> ~b)` as a witness and then proving the three theorems. Abstract definitions can still be unfolded like regular definitions within the proof file, which must supply values for all definitions.

Finally we have axiom and theorem, which assert, within context Γ, that statement A follows from hypotheses Δ. A general axiom or theorem is really an inference rule $\Gamma; \Delta \vdash A$, where Δ is a list of hypotheses and A is a conclusion, and Γ contains the variable declarations used in Δ and A. For example, the Łukasiewicz axioms for propositional logic in this notation are:

$$\begin{aligned}
\varphi\ \psi : \mathsf{wff}; \cdot &\vdash \varphi \to \psi \to \varphi \\
\varphi\ \psi\ \chi : \mathsf{wff}; \cdot &\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi) \\
\varphi\ \psi : \mathsf{wff}; \cdot &\vdash (\neg\varphi \to \neg\psi) \to (\psi \to \varphi) \\
\varphi\ \psi : \mathsf{wff};\ \varphi \to \psi,\ \varphi &\vdash \psi
\end{aligned}$$

Things get more interesting with the FOL axioms:

$$\begin{aligned}
x : \mathsf{var}, \varphi\ \psi : \mathsf{wff}\ x; \cdot &\vdash \forall x\, (\varphi \to \psi) \to (\forall x\, \varphi \to \forall x\, \psi) \\
x : \mathsf{var}, \varphi : \mathsf{wff}; \cdot &\vdash \varphi \to \forall x\, \varphi
\end{aligned}$$

Notice that φ has type $\mathsf{wff}\ x$ in the first theorem and wff in the second, even though x appears in both statements. This indicates that in the first theorem φ may be substituted with an open term such as $x < 2$, while in the second theorem φ must not contain a free occurrence of x.

1.4.4 Well-formedness

Now we turn to the typing rules of the logic. Each typing judgment will be introduced with a box showing the syntax of the judgment. The first one is $\boxed{(E)\ \Gamma\ \mathsf{ctx}}$, which asserts that Γ is a valid context. The (E) syntax indicates that the judgment has a parameter E that is held fixed in the definition and is elided in the typing rules. In this case we need it in the $\mathsf{sort}\, s \in E$ hypotheses.

$$\boxed{(E)\ \Gamma\ \mathsf{ctx}} \qquad \frac{}{\cdot\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad x \notin \mathrm{Dom}(\Gamma) \quad \mathsf{sort}\, s \in E}{\Gamma, x : s\ \mathsf{ctx}}$$

$$\frac{\Gamma\ \mathsf{ctx} \quad \varphi \notin \mathrm{Dom}(\Gamma) \quad \mathsf{sort}\, s \in E \quad \overline{x \in \Gamma}}{\Gamma, \varphi : s\, \overline{x}\ \mathsf{ctx}}$$

This just says that a context is well formed when every variable is in a defined sort, and all dependencies of second order variables are first order variables that appear earlier in the context.

We define typing of expressions and expression lists by mutual recursion.

$$\boxed{(E)\ \Gamma \vdash e : s} \qquad \frac{(x : s) \in \Gamma}{\Gamma \vdash x : s} \qquad \frac{(\varphi : s\,\overline{x}) \in \Gamma}{\Gamma \vdash \varphi : s} \qquad \frac{(f(\Gamma') : s\,\overline{x}) \in E \quad \Gamma \vdash \overline{e} :: \Gamma'}{\Gamma \vdash f\,\overline{e} : s}$$

$$\boxed{(E)\ \Gamma \vdash \overline{e} :: \Gamma'} \qquad \frac{}{\Gamma \vdash \cdot :: \cdot} \qquad \frac{\Gamma \vdash \overline{e} :: \Gamma' \quad (y : s) \in \Gamma}{\Gamma \vdash (\overline{e}, y) :: (\Gamma', x : s)} \qquad \frac{\Gamma \vdash \overline{e} :: \Gamma' \quad \Gamma \vdash e' : s}{\Gamma \vdash (\overline{e}, e') :: (\Gamma', \varphi : s\,\overline{x})}$$

For both first and second order variables, we say that the variable has the sort it is declared with in the context. For functions, we require $(f(\Gamma') : s\,\overline{x}) \in E$, which is to say, there is a term or def declaration with this signature in the environment, and then the list of arguments must match the types declared in the function, using the $\Gamma \vdash \overline{e} :: \Gamma'$ judgment.

The expression list judgment just matches each expression against each variable in the target context Γ'. The only interesting case is the one for first order variables, which requires that when matching against a $x : s$ binder the expression must be some y such that $(y : s) \in \Gamma$. That is, a name can only be substituted for another name, not a full expression. This is what prevents us from substituting $0 : \texttt{nat}$ for the variable $x : \texttt{nat}$ in $\forall x,\ x < x + 1$ to get $\forall 0,\ 0 < 0 + 1$ which is nonsense.

We also will need two definitions of the variables in an expression, denoted $\mathrm{V}_\Gamma(e)$ and $\mathrm{FV}_\Gamma(e)$, read "variables in e" and "free variables in e" respectively.

$$\begin{array}{lll} \mathrm{V}(x) = \{x\} & \mathrm{FV}(x) = \{x\} & \\ \mathrm{V}_\Gamma(\varphi) = \overline{x} & \mathrm{FV}_\Gamma(\varphi) = \overline{x} & \text{where } (\varphi : s\,\overline{x}) \in \Gamma \\ \mathrm{V}(f\,\overline{e}) = \bigcup_i \mathrm{V}(e_i) & \mathrm{FV}(f\,\overline{e}) = \underline{\mathrm{FV}}(\overline{e} :: \Gamma') \cup \{e_i \mid \Gamma'_i \in \overline{x}\} & \text{where } f(\Gamma') : s\,\overline{x} \end{array}$$

$$\begin{aligned} \underline{\mathrm{FV}}(\cdot :: \cdot) &= \emptyset \\ \underline{\mathrm{FV}}((\overline{e}, y) :: (\Gamma', x : s)) &= \underline{\mathrm{FV}}(\overline{e} :: \Gamma') \\ \underline{\mathrm{FV}}((\overline{e}, e') :: (\Gamma', \varphi : s\,\overline{x})) &= \underline{\mathrm{FV}}(\overline{e} :: \Gamma') \cup (\mathrm{FV}(e') \setminus \{e_i \mid \Gamma'_i \in \overline{x}\}) \end{aligned}$$

$\mathrm{V}(e)$ just collects all (first order) variables that are mentioned in the expression, whether they are binding occurrences or not. Second order variables yield their free variable list. So for example if $\varphi : \mathsf{wff}\ x\ y$, then $\mathrm{V}(\forall x, \varphi) = \{x, y\}$ and $\mathrm{V}(\forall z, \varphi) = \{x, y, z\}$.

$\mathrm{FV}(e)$ respects binders, which is to say it removes variables that are bound from the set. So $\mathrm{FV}(\forall x, \varphi) = \{y\}$ and $\mathrm{FV}(\forall z, \varphi) = \{x, y\}$. The $\underline{\mathrm{FV}}(\overline{e} :: \Gamma')$ definition is an auxiliary used to define FV on function applications, by recursing on the type of f along with the subexpressions $\overline{e}$. First order variables are skipped, and for each second order variable with a dependency on a first order variable, that first order variable is removed from the FV set. This is the same binding structure that we would get in higher order logic if each second order variable was lambda-bound over the declared dependency variables before getting passed to f. (We will explain the HOL interpretation in more detail in section 5.3.)

Finally, we have the rules for environment well formedness. An environment is composed of well formed declarations:

$$\boxed{E\ \mathsf{env}} \qquad \frac{}{\cdot\ \mathsf{env}} \qquad \frac{E\ \mathsf{env} \quad E \vdash \delta\ \mathsf{decl}}{E, \delta\ \mathsf{env}}$$

$$\boxed{(E)\ \delta\ \mathsf{decl}} \qquad \frac{}{\mathsf{sort}\ s\ \mathsf{decl}} \qquad \frac{\mathsf{sort}\ s \in E \quad \Gamma\ \mathsf{ctx} \quad \overline{x \in \Gamma}}{\mathsf{term}\ f(\Gamma) : s\,\overline{x}\ \mathsf{decl}}$$

$$\frac{\mathsf{sort}\ s \in E \quad \Gamma, \overline{y : s'}\ \mathsf{ctx} \quad \overline{x \in \Gamma} \quad \left(\Gamma, \overline{y : s'} \vdash e : s \quad \mathrm{FV}_{\Gamma, \overline{y:s'}}(e) \subseteq \overline{x}\right)^?}{\mathsf{def}\ f(\Gamma) : s\,\overline{x}\ (= \overline{y : s'}.\ e)^?\ \mathsf{decl}}$$

$$\frac{\Gamma\ \mathsf{ctx} \quad \overline{\Gamma \vdash A : s} \quad \Gamma \vdash B : s'}{\mathsf{axiom}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{decl}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \overline{\Gamma \vdash A : s} \quad \Gamma \vdash B : s'}{\mathsf{thm}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{decl}}$$

These are mostly as expected:

- For a term to be well formed it needs to have a well formed context and a well formed resulting type $s\,\overline{x}$.
- For a definition to be well formed, in addition to the term rules, e should be well formed of type s, and the free variables of e must be listed in $\overline{x}$. (In particular, since $\overline{y}$ is disjoint from $\mathrm{Dom}(\Gamma) \supseteq \overline{x}$, all the dummy variables are bound, that is, $\mathrm{FV}(e) \cap \overline{y} = \emptyset$.)
- An axiom or theorem must have well formed hypotheses and conclusion statements.

This is only the baseline checking of the theory contents – it does not actually check theorems. So we have a second judgment for the correctness of a declaration:

$$\boxed{(E)\ \delta\ \mathsf{ok}} \qquad \frac{\mathsf{sort}\ s\ \mathsf{decl}}{\mathsf{sort}\ s\ \mathsf{ok}} \qquad \frac{\mathsf{term}\ f(\Gamma) : s\,\overline{x}\ \mathsf{decl}}{\mathsf{term}\ f(\Gamma) : s\,\overline{x}\ \mathsf{ok}}$$

$$\frac{\mathsf{def}\ f(\Gamma) : s\,\overline{x} = \overline{y : s'}.\ e\ \mathsf{decl}}{\mathsf{def}\ f(\Gamma) : s\,\overline{x} = \overline{y : s'}.\ e\ \mathsf{ok}} \qquad \frac{\mathsf{axiom}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{decl}}{\mathsf{axiom}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{ok}}$$

$$\frac{\mathsf{thm}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{decl} \quad \Gamma, \overline{y : s''}\ \mathsf{ok} \quad \Gamma, \overline{y : s''}; \overline{A} \vdash B}{\mathsf{thm}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{ok}}$$

There are two relevant changes from the δ decl judgment:

- Abstract definitions are not allowed: $\mathsf{def}\ f(\Gamma) : s\,\overline{x}\ \mathsf{ok}$ is false.
- Theorems must have proofs. The $\Gamma; \overline{A} \vdash B$ judgment will be defined in the next section (section 1.4.5).

We will say that an environment is "full" if it is provable and has no abstract definitions:

$$\boxed{E\ \mathsf{full}} \qquad \frac{}{\cdot\ \mathsf{full}} \qquad \frac{E\ \mathsf{full} \quad E \vdash \delta\ \mathsf{ok}}{E, \delta\ \mathsf{full}}$$

To define provability on environments containing abstract definitions, we define what it means for an environment to be refined and extended with additional definitions and theorems:

$$\boxed{E \Rightarrow E'} \qquad \frac{}{E \Rightarrow E} \qquad \frac{E \Rightarrow E' \quad E' \Rightarrow E''}{E \Rightarrow E''} \qquad \frac{E \Rightarrow E'}{E, \delta \Rightarrow E', \delta}$$

$$\frac{}{E \Rightarrow E, (\mathsf{def}\ f(\Gamma) : s\, \overline{x})} \qquad \frac{}{\begin{array}{l} E, (\mathsf{def}\ f(\Gamma) : s\, \overline{x}) \Rightarrow \\ \qquad E, (\mathsf{def}\ f(\Gamma) : s\, \overline{x} = \overline{y : s'}.\ e) \end{array}}$$

$$\frac{}{E \Rightarrow E, \mathsf{thm}\ (\Gamma; \overline{A} \vdash B)}$$

Then, an environment is provable if it has a full extension:

$$\boxed{E\ \mathsf{ok}} \qquad \frac{E\ \mathsf{env} \quad E \Rightarrow E' \quad E'\ \mathsf{env} \quad E'\ \mathsf{full}}{E\ \mathsf{ok}}$$

The purpose of a MM0 verifier is to validate a claimed proof of E ok, given a `.mm0` file with the concrete syntax of E. This claimed proof may include additional definitions and theorems beyond those included in the original `.mm0` file, so it effectively encodes E', as well as proof objects for each $\mathsf{thm}\ (\Gamma; \overline{A} \vdash B) \in E'$.

1.4.5 *The MM0 proof judgment*

The proof judgment $(E; \Gamma; \Delta)\ \vdash A$ [7] asserts that statement A is derivable from hypotheses Δ, with a context Γ of variables and dummy variables that are available for use in the proof, and an environment E that provides the lemmas and definitions that can be used.

[7] As a reminder, the parentheses in $(E; \Gamma; \Delta)$ indicate that these are parameters of the judgment that do not change in the inductive rules.

It is notable that Δ is a parameter, by contrast to HOL or natural deduction systems where hypotheses can be introduced and discharged. MM0 models a "Hilbert system" in which axioms are used rather than structural rules, although it is possible to encode structural rules in the target logic, if A can itself be an implication or sequent. For example, if the logic has a sequent operator $a \mid\!\sim b$, then the $\rightarrow I$ rule of natural deduction would look like $\mathsf{axiom}\ ((a, b \mid\!\sim c) \vdash (a \mid\!\sim b \rightarrow c))$.

Proof judgment $\boxed{(E; \Gamma; \Delta)\ \vdash A}$

$$\text{P-HYP}\ \frac{A \in \Delta}{\vdash A} \qquad \text{P-CONV}\ \frac{\vdash A \equiv B \quad \vdash A}{\vdash B}$$

$$\text{P-THM}\ \frac{\begin{array}{c} (\Gamma'; \overline{A} \vdash B) \in E \qquad \Gamma \vdash \overline{e} :: \Gamma' \qquad \forall i,\ \vdash A_i[\Gamma' \mapsto \overline{e}] \\ \forall i\, j\, x,\ \Gamma'_i = x \notin \mathrm{V}_{\Gamma'}(\Gamma'_j) \rightarrow e_i \notin \mathrm{FV}_{\Gamma}(e_j) \end{array}}{\vdash B[\Gamma' \mapsto \overline{e}]}$$

There are three ways to prove a theorem (up from two in Metamath[8]):

[8] Metamath does not have a concept of definitions as distinct from axioms, so there is no convertibility rule or $A \equiv B$ judgment.

- The hypothesis rule P-HYP: if $A \in \Delta$, then $\vdash A$.
- The conversion rule P-CONV: If A and B are convertible (defined in section 1.4.6 below), then $\vdash A$ iff $\vdash B$.
- Theorem application P-THM. If $(\Gamma'; \overline{A} \vdash B)$ is an assertion in E (which is to say, $\texttt{axiom}\ (\Gamma'; \overline{A} \vdash B) \in E$ or $\texttt{thm}\ (\Gamma'; \overline{A} \vdash B) \in E$), then we can substitute $\overline{e}$ for the variables of Γ' to deduce that $\Gamma; \overline{A[\Gamma' \mapsto \overline{e}]} \vdash B[\Gamma' \mapsto \overline{e}]$, so if each $A_i[\Gamma' \mapsto \overline{e}]$ is derivable, then $B[\Gamma' \mapsto \overline{e}]$ is also derivable.

  Here the substitution operation $A[\Gamma \mapsto \overline{e}]$ is defined as:

$$\begin{aligned} x[\Gamma' \mapsto \overline{e}] &= e_i && \text{where } x = \Gamma'_i \\ \varphi[\Gamma' \mapsto \overline{e}] &= e_i && \text{where } \varphi = \Gamma'_i \\ (f\,\overline{e'})[\Gamma' \mapsto \overline{e}] &= f\,\overline{e'[\Gamma' \mapsto \overline{e}]} \end{aligned}$$

In other words, substitution $A[\Gamma' \mapsto \overline{e}]$ entails finding and replacing all occurrences of each variable in A (which must be an element of Γ' if A is well typed) with the corresponding element of $\overline{e}$. This does no special handling around binders, and in fact even binder variables themselves can be targeted by this "substitution" operation, making it also play the role of α-renaming.

The theorem application rule is the core of the logic, and it is complex so it deserves some attention. First, we find some theorem that is declared in the environment, $\Gamma'; \overline{A} \vdash B$. For example, modus ponens:

$$\varphi : \texttt{wff},\ \psi : \texttt{wff};\ \ \varphi,\ \varphi \to \psi \vdash \psi$$

Here $\Gamma' = (\varphi : \texttt{wff},\ \psi : \texttt{wff})$ and $A_1 = \varphi$, $A_2 = (\varphi \to \psi)$, $B = \psi$. Then, we take some substitution $\overline{e}$ which is valid in the current context, for example $e_1 = (0 < x)$, $e_2 = (1 < x)$. We have to check that $\Gamma \vdash \overline{e} :: \Gamma'$, which is to say, $(0 < x) : \texttt{wff}$ and $(1 < x) : \texttt{wff}$. (There is a side condition, but in this case it is trivially satisfied.) We then apply the substitution $\Gamma' \mapsto \overline{e}$ to A_1, A_2, B, resulting in the expressions:

$$\begin{aligned} A_1[\Gamma' \mapsto \overline{e}] &= \varphi\,[\varphi \mapsto 0 < x,\ \psi \mapsto 1 < x] = (0 < x) \\ A_2[\Gamma' \mapsto \overline{e}] &= (\varphi \to \psi)[\varphi \mapsto 0 < x,\ \psi \mapsto 1 < x] = (0 < x \to 1 < x) \\ B[\Gamma' \mapsto \overline{e}] &= \psi\,[\varphi \mapsto 0 < x,\ \psi \mapsto 1 < x] = (1 < x) \end{aligned}$$

Therefore, if $\vdash 0 < x$ and $\vdash 0 < x \to 1 < x$ are derivable, then we deduce $\vdash 1 < x$.[9]

The final component of the theorem application rule is the side condition $\forall i\,j\,x,\ \Gamma'_i = x \notin \mathrm{V}_{\Gamma'}(\Gamma'_j) \to e_i \notin \mathrm{FV}_{\Gamma}(e_j)$, which we will read as "$\Gamma' \mapsto \overline{e}$ is an admissible substitution." In words, it says that for any two variables Γ'_i and Γ'_j in the context Γ' where $\Gamma'_i = x$ is a first order variable which is not in $\mathrm{V}_{\Gamma'}(\Gamma'_j)$ (if $\Gamma'_j = y$ is a first order variable

[9] Of course, written this way, it seems like we made a whole lot of fuss for such a triviality. But what we have gained by doing this is that the system does not actually need to know anything at all about modus ponens or even implication to make this inference: it simply applied a user-specified axiom that had the form of modus ponens. The object logic need not even have modus ponens or an equivalent, which makes it usable for studying logics which express the rules in a different way, and furthermore user can prove theorems as inference rules and the system will use them just as effectively as core logical axioms: modus ponens is in no way "special" here.

then this means $x \neq y$, and if $\Gamma'_j = \varphi$ is a second order variable then this means x is not one of the dependencies of φ), then e_i (which is necessarily a first order variable x') is not among the free variables of e_j.

In our previous example, this condition was vacuous because $\Gamma' = (\varphi : \texttt{wff},\ \psi : \texttt{wff})$ consists only of second order variables. Here are two more examples from the FOL axioms:

$$x : \texttt{nat},\ \varphi : \texttt{wff}\ x;\ \ \varphi \vdash \forall x,\ \varphi \quad \text{vs.} \quad x : \texttt{nat},\ \varphi : \texttt{wff};\ \ \vdash \varphi \to \forall x,\ \varphi$$

The first theorem is called the axiom of generalization or `ax_gen`, and says that we can introduce $\forall x$ in front of any provable theorem $\vdash \varphi$. This applies even if φ contains a free occurrence of x, as indicated by the x in "$\varphi : \texttt{wff}\ x$". By contrast, the second theorem (called `ax_5` in `peano.mm0`) says that φ implies $\forall x,\ \varphi$ which does *not* hold if you allow for free occurrences of x (for example $x = 0 \to \forall x,\ x = 0$ does not hold on the natural numbers).

Consider the effect of the admissibility side condition on these two theorems. In the first case, we consider all pairs of variables from the context such that one of them is a first order variable; this means (x, x) and (x, φ). Then we restrict to the case that V of the second variable does not contain x, but since $\mathrm{V}(x) = \{x\}$ and $\mathrm{V}(\varphi) = \{x\}$ there is no such example and hence the side condition is trivial.

In the second case, we have $\mathrm{V}(x) = \{x\}$ and $\mathrm{V}(\varphi) = \emptyset$ so (x, φ) matches the preconditions, and therefore the admissibility criterion requires that if you substitute $(x \mapsto y, \varphi \mapsto p)$ in this theorem we must have $y \notin \mathrm{V}(p)$, which is to say, p does not contain a free occurrence of y.

The notion of "strictly admissible substitution" is obtained by replacing FV with V on the right, that is:

$$\forall i\, j\, x,\ \Gamma'_i = x \notin \mathrm{V}_{\Gamma'}(\Gamma'_j) \to e_i \notin \mathrm{V}_{\Gamma}(e_j)$$

(Using V or FV on the left makes no difference.) This is an underapproximation of admissible substitutions which is a bit easier to check, so for performance and portability reasons the MM0 verifiers in fact check strict admissibility in the theorem application rule. (We will use P-THM-S to refer to the strictified version of the P-THM rule.) It is not obvious, but the two yield an equivalent theory.

1.4.6 The MM0 convertibility judgment

The convertibility judgment $(E; \Gamma) \vdash e \equiv e'$ is used in the conversion rule, and asserts that two expressions are "equivalent" from the point of view of the logic. This is a new addition of MM0 compared to Metamath, which is necessary for unfolding definitions.

Convertibility $\boxed{(E;\Gamma) \vdash e \equiv e'}$

$$\textsc{c-refl}\ \frac{\Gamma \vdash e : s}{\vdash e \equiv e} \qquad \textsc{c-symm}\ \frac{\vdash e \equiv e'}{\vdash e' \equiv e} \qquad \textsc{c-trans}\ \frac{\vdash e_1 \equiv e_2 \quad \vdash e_2 \equiv e_3}{\vdash e_1 \equiv e_3}$$

$$\textsc{c-cong}\ \frac{\Gamma \vdash f\,\overline{e} : s \quad \Gamma \vdash f\,\overline{e'} : s \quad \forall i,\ \vdash e_i \equiv e'_i}{\vdash f\,\overline{e} \equiv f\,\overline{e'}} \qquad \textsc{c-alpha}\ \frac{\Gamma \vdash e =_\alpha e'}{\vdash e \equiv e'}$$

$$\textsc{c-unfold}\ \frac{(\mathsf{def}\, f(\Gamma') : s\,\overline{x} = \overline{y : s'}.\, e') \in E \qquad \Gamma \vdash (\overline{e}, \overline{z}) :: (\Gamma', \overline{y : s'}) \qquad \forall i\,j,\ z_i \notin \mathrm{FV}_\Gamma(e_j) \qquad \forall i\,j,\ i \neq j \to z_i \neq z_j}{\vdash f\,\overline{e} \equiv e'[\Gamma', \overline{y : s'} \mapsto \overline{e}, \overline{z}]}$$

Most of the rules are fairly standard:

- C-REFL, C-SYMM, C-TRANS say that convertibility is an equivalence relation (on well typed terms).
- C-CONG says that it descends into arbitrary term structure: every term constructor respects convertibility.
- The C-ALPHA rule says that two terms that are α-equivalent are convertible. (We include this in the formal theory for now but we will later show that it is not needed, and MM0 verifiers do not have this rule.)
- The only really interesting rule is C-UNFOLD, which says that if $f(\Gamma') := \overline{y}.\ e'$ is defined in the environment, then we are permitted to replace $f\ \overline{e}$ with $e'[\Gamma' \mapsto \overline{e},\ \overline{y} \mapsto \overline{z}]$ where $\overline{z}$ are new dummy variables that are mutually distinct and do not interfere with the arguments $\overline{e}$.

We say that e and e' are α-equivalent and write $\Gamma \vdash e =_\alpha e'$ if e' is equal to $e[\Gamma \mapsto \overline{v}]$ where $v_i = \Gamma_i$ for all $v_i \notin S$, where S is a set of first order variables such that $\mathrm{FV}(e) \cap S = \emptyset$, and also $\mathrm{V}(\varphi) \cap S = \emptyset$ for every second order variable φ appearing in e. In other words, e' is obtained from e by replacing *only* bound variables. As in the C-UNFOLD rule, the variables in S must be mutually distinct and free from anything else in Γ.

For example, $\forall x,\ x = y$ is α-equivalent to $\forall z,\ z = y$ because we can replace $x \mapsto z$ in the expression (since $x \notin \mathrm{FV}(\forall x,\ x = y)$), but it is not α-equivalent to $\forall x,\ x = z$ because y is free in the expression. This notion is weaker than the standard FOL α-equivalence relation because it only considers renaming all binders globally in the expression "in one go," which for example excludes α-equivalence of $\forall x,\ \forall x,\ x = y$ and $\forall x,\ \forall z,\ z = y$.

It is also weak in the presence of second order variables: if φ depends on x then there is no expression p such that $\forall x,\ \varphi$ is α-equivalent to $\forall y,\ p$. We can overcome this limitation with a bit of help from the object logic, however: it is possible to define an explicit substitution operator in the logic and then prove $\vdash (\forall x,\ \varphi) \leftrightarrow (\forall y,\ \varphi[y/x])$.

1.5 *The `.mm0` specification format*[10]

[10] https://github.com/digama0/mm0/blob/master/mm0.md

The `.mm0` file is responsible for explaining to the reader what the statement of all relevant theorems is. It closely resembles the axiomatic description of section 1.4, but with a concrete syntax.

1.5.1 *Sort modifiers*

Sorts have modifiers that limit what roles they can play. These are enforced by the verifier but not strictly necessary for expressivity.

- Every statement is required to have a provable sort, so that one can assert that if x : nat then $\vdash x$ is nonsense and not permitted.
- The free modifier asserts that a sort cannot be used as a dummy variable, in which case the sort may possibly be empty.
- The strict modifier asserts that the sort cannot be used as a name. This is useful for metavariable-only sorts like wff.
- The pure modifier asserts that the sort has no expression constructors (terms or defs). This is useful for name-only sorts like var.

1.5.2 *No proofs*

As its name implies, the `.mm0` specification file is only about specifying axioms and theorems, so it does not contain any proofs. This is an unusual choice for a theorem prover, although some systems like Mizar and Isabelle support exporting an "abstract" of the development, with proofs omitted.

The reason for this comes back to our breakdown of the purpose of the different components of the architecture as described in the introduction. The correctness theorem we are aiming for here is akin to security with respect to an extreme threat model. As an illustration, suppose you are trying to encourage formalization of some theorem of interest, let's say Fermat's Last Theorem (FLT), and you organize a competition. You write FLT as a `.mm0` file, and open it up for the world to submit proof attempts as corresponding `.mmb` files. *Even in* *the face of bad faith proof attempts*, even if you are receiving gigabytes-long machine learned proofs, you want the assurance that if the verifier accepts it, then the theorem is proved from the axioms you defined. (It would also be nice to know that the verifier cannot misbehave in other ways, such as leaking your data to the internet, and in practice a proof that the verifier is correct will have to establish the lack of general misbehavior anyway, since most kinds of misbehavior such as buffer overflows can potentially be exploited to trick the verifier to accept bad proofs.)

This may seem obvious, but it is surprisingly common for proof systems to contain convenience features that lead to security flaws, or proofs of contradiction, that "no reasonable person would use," making the tacit assumption that the proof author is reasonable. In practice this means that the proof itself must be inspected to ensure that none of these exploits have been used, or else a social system must be in place so that we can trust the authors; but this defeats the point of formalizing in the first place.

Since an `.mm0` file is a *formalization target* or *problem statement*, it does not require or even accept proofs of its statements directly inline. Axioms and theorems look exactly the same except for the keyword used to introduce them.

1.5.3 Abstract definitions

We can do something similar with definitions: we can write a definition with no definiens, so that it looks just like a term declaration. This allows us to assert *the existence* of a term constructor which satisfies any theorems that follow, which gives us a kind of abstraction. Sometimes it is easier to write down characteristic equations for a function rather than an explicit definition, especially in the case of recursive functions.

If we view the entire `.mm0` file as a single theorem statement of the metalogic, then this construction corresponds to a second order (constructive) existential quantifier, complementing the second order universal quantifiers that are associated to theorems with free metavariables.

1.5.4 Local theorems and definitions

Once one is committed to not proving theorems in the specification file, most dependencies go away. Theorems never reference each other, and only reference terms and definitions involved in their statements. So if focus is given to one theorem, then almost everything else goes away, and even in extreme cases it becomes quite feasible to write down everything up to and including the axiomatic framework in a few thousand lines. In the above example of FLT, the specification file must define the natural numbers and exponentiation, but certainly not modular forms. These are properly the domain of the proof file. (The degree to which this statement is accurate depends to some extent on the theorem. FLT is something of an extreme case in that the statement requires much less technology than the proof. However, there is usually at least a 10 to 1 factor between definitions and proofs and often much more; `set.mm` for example contains 29 times as many theorems as definitions, and the theorems are on average 4.7 times longer than definitions, so removing all the theorems amounts to a 99.27% reduction in formal content.)

But that means that the proof file must have license to introduce its own definitions and theorems, beyond the ones described in the specification file (but *not* sorts, term constructors, or axioms). And this is exactly the piece that is missing in Metamath: Forbidding new axioms is necessary in order to prevent a malicious proof author from assuming false things, but in MM that also means no new definitions, and that is an untenable expressivity limitation.

1.5.5 Notation

In the abstract characterization, we did not concern ourselves with notation, presuming that terms were constructed inductively as trees, but early testing of the concrete syntax revealed that no one likes to read piles of s-expressions, and readability was significantly impacted. The notation system was crafted so as to make parsing as simple as possible to implement, while still ensuring unambiguity, and allowing some simple infix and bracketing notations. Notations are enclosed in `$` sentinels (as in LaTeX) so that parsing can be separated into a static part (containing the top level syntax of the language) and a dynamic part (containing user notations for mathematical operations that have been defined).

The dynamic parser is a precedence parser, with a numeric hierarchy of precedence levels $0, 1, 2, \dots$ with an additional level max, forming the order $\mathbb{N} \cup \{\infty\}$. ($\mathsf{max}$ is the precedence of atoms and parenthesized expressions.) Infix constants are declared with a precedence, and left/right associativity. (An earlier version of MM0 used nonassociative operators and a partial order for precedence levels, but this complicated the parser for no added expressivity. We recognize that overuse of precedence ordering can lead to miscommunication, but this is in the trusted specification file anyway, so the drafter must take care to be clear and use parentheses responsibly.)

General notations are also permitted; these have an arbitrary sequence of constants and variables, and can be used to make composite notations like `sum_ i < n ai` as an approximation of $\sum_{i<n} a_i$. The only restriction on general notations to make them unambiguous is that they must begin with a unique constant, in this case `sum_`. This is restrictive, but usually one can get away with a subscript or similar disambiguating mark without significantly hampering readability. (This may be relaxed in higher level languages, but recall that we are still in the base of the bootstrap here, so every bit of simplicity matters.)

Coercions are functions from one sort to another that have no notation. For example, if we have a sort of set expressions and another sort of class expressions, we might register a coercion set $\to$ class so that $x \in y$ makes sense even if x and y are sets and $x \in A$ is a relation between a set and a class. For unambiguity, the verifier requires that the coercion graph have at most one path from any sort to any other.

1.6 The `.mmb` binary proof file[11]

[11] https://github.com/digama0/mm0/tree/master/mm0-c

Having a precise language for specifying formal statements is nice, but it is most powerful when coupled with a method for proving those formal statements. We have indicated several times now design decisions that were made for efficiency reasons. How does MM0 achieve these goals?

The only constraint on the `.mmb` format is that it somehow guides the verifier to validate that the input `.mm0` specification is provable. A useful model to keep in mind is that of a powerful but untrustworthy oracle providing hints whenever the verifier needs one, or a nondeterministic Turing machine that receives its nondeterminism from external input.

There are two fundamental principles that guide the design: "avoid search," and "don't repeat yourself." By spoon-feeding the verifier a very explicit proof, we end up doing a lot less computation, and by pervasively deduplicating, we can avoid all the exponential blowups that happen in unification. Using these techniques, we managed to translate `set.mm` into MM0 (see section 5.1.1) and verify the resulting binary proof file in 195 ± 5 ms (Intel i7 3.9 GHz, single threaded). (This is not a fair comparison in that we are not checking `set.mm` as is, we are adding a bunch of information and rearranging it to be faster to check, and observing that the result is faster to check. But in a sense that's the point.) While `set.mm` is formidable, at 34 MB / 590 kLOC, we are planning to scale up to larger or less optimized formal libraries to see if it is competitive even on more adversarial inputs.

1.7 High level structure

The proof file is designed to be manipulated in situ; it does not need to be processed into memory structures, as it is already organized like one. It contains a header that declares the sorts, and the number of terms/defs and axioms/theorems, and then links to the beginning of the term table and the theorem table, and the declaration list.

The term table and theorem table contain the statements of all theorems and the types of all term constructors. These tables are consulted during typechecking, and the verifier uses a counter as a sliding window into the table to mark what part of the table has been verified (and thus is usable). This means that a term lookup is generally a single indexed memory access, usually in cache, which makes type checking for expressions ($\Gamma \vdash e : s$) extremely fast in practice.

Variable names, term names, and theorem names are all replaced as identifiers with indices into the relevant arrays. All strings are stored in an index that is placed at the end of the file, linked to from the header, and not touched by the verifier except when it wants to report an error. It is analogous to debugging data stored in executables — it can be stripped without affecting anything except the quality of error reporting.

A term entry contains a table of variable declarations (the context Γ and the target type $s\,\overline{x}$) followed by a unify stream for definitions, and a theorem entry contains a table of variable declarations (the context Γ), followed by a unify stream (section 1.8).

1.8 The declaration list

After the term and theorem tables is the the declaration list, which validates each declaration in the `.mm0` file, possibly interspersed with additional definitions and theorems. This data is processed in a single pass, and contains in particular proofs of theorems. The global state of the verifier is very small; it need only keep track of how many terms, theorems, and sorts have been verified so far, treating some initial segment of the input tables as available for use and the rest as inaccessible. Because terms and theorems are numbered in the same order they appear in the file, when a theorem appears in the declaration list it is always the one just after the current end of the theorem table.

There are two kinds of opcode streams, proof streams and unify streams. Unify streams appear only in the term and theorem tables, and are used when a theorem is referenced or a definition is unfolded. Proof streams appear in the declaration list and provide proofs for

$$\begin{array}{rll}
\sigma ::= e \mid \vdash A \mid e \equiv e' \mid e \overset{?}{\equiv} e' & & \text{stack element} \\
H, S, U, K ::= \overline{\sigma} & & \text{heap, stack, unify heap, unify stack} \\
\Delta ::= \overline{A} & & \text{hypothesis list}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{Save:} & H; S, \sigma \hookrightarrow H, \sigma; S, \sigma & \\
\mathsf{Term}\ f\mathsf{:} & S, \overline{e} \hookrightarrow S, e' & \begin{pmatrix} f : \Gamma' \Rightarrow s\,\overline{x}, \quad \vdash \overline{e} :: \Gamma' \\ e' := \mathrm{alloc}(f\,\overline{e} : s) \end{pmatrix} \\
\mathsf{Ref}\ i\mathsf{:} & S \hookrightarrow S, e & (e := H[i]) \\
\mathsf{Dummy}\ s\mathsf{:} & H; S \hookrightarrow H, e; S, e & (e := \mathrm{alloc}(x : s),\ x \text{ fresh}) \\
\mathsf{Thm}\ T\mathsf{:} & S, \overline{e}^*, A \hookrightarrow S', \vdash A & (\mathrm{Unify}(T)\text{: } S; \overline{e}; A \hookrightarrow_{\mathsf{u}} S') \\
\mathsf{Hyp:} & \Delta; H; S, A \hookrightarrow \Delta, A; H, \vdash A; S & \\
\mathsf{Conv:} & S, A, \vdash B \hookrightarrow S, \vdash A, A \overset{?}{\equiv} B & \\
\mathsf{Refl:} & S, e \overset{?}{\equiv} e' \hookrightarrow S & (e = e') \\
\mathsf{Symm:} & S, e \overset{?}{\equiv} e' \hookrightarrow S, e' \overset{?}{\equiv} e & \\
\mathsf{Cong:} & S, f\,\overline{e} \overset{?}{\equiv} f\,\overline{e'} \hookrightarrow S, \overline{e \overset{?}{\equiv} e'}^* & \\
\mathsf{Unfold:} & S, f\,\overline{e} \overset{?}{\equiv} e'', e' \hookrightarrow S', e' \overset{?}{\equiv} e'' & (\mathrm{Unify}(f)\text{: } S; \overline{e}; e' \hookrightarrow_{\mathsf{u}} S') \\
\mathsf{Sorry:}^{12} & S, A \hookrightarrow S, \vdash A & \\
\mathsf{ConvCut:} & S, e, e' \hookrightarrow S, e \equiv e', e \overset{?}{\equiv} e' & \\
\mathsf{ConvRef}\ i\mathsf{:} & S, e \overset{?}{\equiv} e' \hookrightarrow S & (H[i] = e \equiv e') \\
\mathsf{ConvSave:} & H; S, e \equiv e' \hookrightarrow H, e \equiv e'; S & \\
\mathsf{ConvSorry:}^{12} & S, e \overset{?}{\equiv} e' \hookrightarrow S &
\end{array}$$

$$\begin{array}{lcl}
\mathsf{USave:} & U; K, \sigma \hookrightarrow_{\mathsf{u}} U, \sigma; K, \sigma & \\
\mathsf{UTerm}\ f\mathsf{:} & K, f\,\overline{e} \hookrightarrow_{\mathsf{u}} K, \overline{e} & \\
\mathsf{URef}\ i\mathsf{:} & U; K, e \hookrightarrow_{\mathsf{u}} U; K & (U[i] = e) \\
\mathsf{UDummy}\ s\mathsf{:} & U; K, x \hookrightarrow_{\mathsf{u}} U, x; K & (x : s) \\
\mathsf{UHyp:} & S, \vdash A; K \hookrightarrow_{\mathsf{u}} S; K, A &
\end{array}$$

Figure 1.3: Proof stream and unify stream opcodes and their operational semantics. Proof steps have the form $C : \Delta; H; S \hookrightarrow \Delta'; H'; S'$, and unify steps have the form $C : S; U; K \hookrightarrow S'; U'; K'$, but values that do not change are suppressed. $\overline{e}^*$ denotes the reverse of $\overline{e}$.

[12] The Sorry and ConvSorry opcodes are optionally supported. They are intended for checking of partial proofs. They cause a "uses sorry" flag to be set in the verifier, which causes the overall run to fail.

theorems.

During a proof, the verifier state consists of a store (a write-once memory arena that is cleared after each proof) which builds up pointer data structures for constructed expressions, a heap H, and a stack S. A stack element σ can be either an expression e or a proof $\vdash A$, both of which are simply pointers into the store where the relevant expression is stored. The nodes themselves store the head and sort of the expression: $x : s$, $\varphi : s$, or $f\ \overline{e} : s$, as well as precalculating $\mathrm{V}(e)$ ($\mathrm{FV}(e)$ when constructing definition expressions). (There are also two kinds of convertibility proof that can be on the stack, discussed below.)

A declaration in the list is an opcode for the kind of declaration, a pointer to the next declaration (for fast scanning and parallelization), and some data depending on what kind of declaration it is:

- Sorts and terms just mark the next item in the declaration table as valid for use.
- A definition $\mathsf{def}\ f(\Gamma) : s\,\overline{x} = \overline{y : s'}.\ e$ reads a proof stream $\mathrm{Proof}(f)$: $\cdot;\Gamma;\cdot \hookrightarrow \cdot;\ \Gamma,\overline{y : s'};\ e$ (which is to say, it initializes the heap with Γ and an empty stack, and expects a single expression e on the stack after executing $\mathrm{Proof}(f)$), and then checks that $\mathrm{Unify}(f)$, the corresponding element of the declaration table, satisfies $\mathrm{Unify}(f)$: $\cdot;\overline{y : s'}^*;e \hookrightarrow_{\mathsf{u}} \cdot;\Gamma';\cdot$.
- A theorem or axiom $T : (\Gamma, \Delta \vdash A)$ reads a proof stream $\mathrm{Proof}(T)$: $\cdot;\Gamma;\cdot \hookrightarrow \Delta^*;\ \Gamma,\overline{y : s'};\ \vdash A$ (for axioms, the stack at the end holds A instead of $\vdash A$), and then checks $\mathrm{Unify}(T)$: $\cdot;\vdash\Delta;\ \Gamma;\ A \hookrightarrow_{\mathsf{u}} \cdot;\Gamma';\cdot$.

In short, we build up an expression using the $\mathrm{Proof}(f)$ proof stream, and then check it against the expression that is in the global space using $\mathrm{Unify}(f)$, so that we can safely reread it later.

At the beginning of a proof, the heap is initialized with expressions for all the variables. An opcode like $\mathsf{Term}\ f$ will pop n elements $\overline{e}$ from the stack, and push $f\ \overline{e}$, while $\mathsf{Ref}\ i$ will push $H[i]$ to the stack. The verifier is arranged such that no expression is always accessed via backreference if it is constructed more than once, so equality testing is always $O(1)$.

The opcode $\mathsf{Thm}\ T$ pops $\overline{e}$ from the stack (the number of variables in the theorem), pops B' from the stack (the substituted conclusion of the theorem), then calls a *unifier* for T, stored in the theorem table for T, which is another sequence of opcodes. This will pop some number of additional $\overline{\vdash A'}$ assumptions from the stack, and then $\vdash B'$ is pushed on the stack.

The unifier is responsible for deconstructing B' and proving that $B[\Gamma \mapsto \overline{e}] = B'$, where B and Γ are fixed from the definition of T, and

$\overline{e}$ and B' are provided by the theorem application. It has its own stack K and heap U; the unify heap is the incoming substitution, and the unify stack is the list of unification obligations. For example URef i pops e from the stack and checks that $U[i] = e$, while UTerm f pops an expression e from the unify stack, checks that $e = f\ \overline{e'}$, and then pushes $\overline{e'}$ on the stack (in reverse order). The appropriate list of opcodes can be easily constructed for a given expression by reading the term in prefix order, with UTerm at each term constructor and URef for variables. The UHyp instruction pops $\vdash A'$ from the main stack S and pushes A' to the unify stack K; this is how the theorem signals that it needs a hypothesis.

Convertibility is handled slightly differently than in the abstract formalism. Most of the convertibility rules are inverted, working with a co-convertibility hypothetical $e \overset{?}{\equiv} e'$. In the absence of $e \overset{?}{\equiv} e'$ judgments on the stack, the meaning of the stack is that all $\vdash A$ statements in it are provable under the hypotheses Δ, but $S, e \overset{?}{\equiv} e'$ means that if $e \equiv e'$ is provable, then the meaning of S holds. So for instance, the Conv rule $S, A, {\vdash}B \ \hookrightarrow\ S, {\vdash}A, A \overset{?}{\equiv} B$ says that from $\vdash B$, we can deduce that if $\vdash A \equiv B$ is provable, then $\vdash A$ holds, which is indeed the conversion rule.

The reason for this inversion is that it makes most unfolding proofs much terser, since all the terms needed in the proof have already been constructed, and the Refl and Cong rules only need to deconstruct those terms.

The ConvCut rule is not strictly necessary, but is available in accordance with the "don't repeat yourself" principle. It allows for an unfolding proof to be stored and replayed multiple times, which might be useful if it is a frequently appearing subterm.

The handling of memory is interesting in that all allocations are "controlled by the user" in the sense that they happen only on Term f and Dummy s steps. (Note that "the user" here is really the compiler, since the `.mmb` format is how the compiler communicates to the verifier.) Because proof streams are processed in one pass, that means that every allocation in the verifier can be identified with a particular opcode in the file.

But the biggest upshot of letting the user control allocation is that they have complete control over the result of pointer equality. That is, whenever a statement contains a subterm multiple times, for example $g(f(x), f(x))$, the user can arrange the proof such that these subterms are always pointers to the same element on the heap (in this example, Ref x, Term f, Save, Ref 1, Term g, assuming that the Save puts $f(x)$ at index 1). This would not be possible without hash-consing if the verifier "built expressions on its own volition" in the course of performing substitution or applying theorems. As such, the verifier can simply *require* that every term be constructed at most once (or at least, any expressions that participate in an equality test should be identified), and then expression equality testing in steps like URef and Refl is always constant time.

An earlier version of the verifier actually put the data that would otherwise need to be allocated into the instruction itself (i.e. the instruction might be $\mathsf{Term}(f, \overline{e}, \overline{x})$, and the verifier is responsible for checking that $\mathrm{V}(f\ \overline{e}) = \overline{x}$). However, this wastes a lot of space (the $\mathrm{V}(e)$ slots are typically 8 bytes) for ephemeral data. Putting too much data into the proof file means more IO to read it, which can cancel the performance benefits of not having to allocate memory. The memory high-water is under 1 megabyte even after reading the largest proofs in `set.mm` (which deliberately includes a few stress test theorems), so memory usage doesn't seem to be a major issue. Nevertheless, it is useful to note that by encoding the heap and stack in the instruction stream, it is possible to perform verification with $O(1)$ writable memory, streaming almost all of the proof.

Verification is not quite linear time, because each `Thm` T instruction causes the verifier to read $\mathrm{Unify}(T)$, which is approximately as large as the (deduplicated) statement of T. It is $O(mn)$ where n is the length of the proof and m is the length of the longest theorem statement, but the statements that exercise the quadratic worst case are rather contrived; they require *one theorem statement* to be on the same order as the whole file. An example would be if we have a theorem $T : (a : \mathsf{nat} \vdash a \cdot \bar{n} = 0)$, where $\bar{n}$ is a large unary numeral $S^n(0)$, and we have a proof that constructs and discards $T(0), \ldots, T(\bar{n})$, and then proves a triviality. This requires only $O(n)$ to state (because there are $O(n)$ expression subterms in the full proof), but each application of $T(i)$ requires $O(n)$ to verify because it must match the large term $\bar{n}$ in the statement of T, resulting in $O(n^2)$ overall. However, it should be emphasized that this is not a realistic workload; large theorem statements are very rare, and the large theorems that are used are rarely referenced multiple times in one proof, so for almost all reasonable proof libraries this achieves linear time verification.

1.9 *Compilation*

Fairly obviously, the `.mmb` format is not meant to be written by humans; instead it is "compiled" from source in some other human readable language. (The design is similar to, and indeed inspired by, high entropy machine code encodings.) The details of compilation depend on the form of this language, but the backend will probably be similar regardless. For the MM1 compiler (see Chapter 2), after executing the high level tactics and programs, the result is an environment object in memory. Here expressions are stored in the usual functional way as trees (pointer data structures) with possible but not mandatory subexpression sharing, and proofs contain similar subproof sharing. For example, consider the following short MM1 file and proof:

```
provable sort wff;
term im: wff > wff > wff; infixr im: $->$ prec 1;
axiom a1 (a b: wff): $ a -> b -> a $;
axiom mp (a b: wff): $ a -> b $ > $ a $ > $ b $;
theorem a1i (a b: wff) (h: $ a $): $ b -> a $ = '(mp a1 h);
```

The proof (mp a1 h) is elaborated using first order unification to determine the necessary substitutions for term arguments, resulting in the elaborated proof (mp a (im b a) (a1 a b) h). We also add an argument with the theorem statement of each intermediate step to obtain

$$\text{(mp a (im b a) (a1 a b (im a (im b a))) h (im b a)).}$$

(Already at this point we will have a lot of "accidental" subterm sharing, not shown in the printed s-expression, since it naturally appears as a result of first order unification and substitution.) We then perform hash-consing to ensure that every subterm has at most one index, also throwing in the expressions for the hypotheses by using a synthetic root node $\underline{\text{root}}$. We end up with a structure like so:

$$\begin{aligned}&\mathsf{let}\ 0 := \mathsf{a},\ 1 := \mathsf{b},\ 2 := \mathsf{h},\ 3 := (\mathsf{im}\ 1\ 0),\ 4 := (\mathsf{im}\ 0\ 3),\\ &\quad 5 := (\mathsf{a1}\ 0\ 1\ 4),\ 6 := (\mathsf{mp}\ 0\ 3\ 5\ 2\ 3)\ \mathsf{in}\ (\underline{\mathsf{root}}\ 0\ 6)\end{aligned}$$

We inline all references that appear at most once:

$$\begin{aligned}&\mathsf{let}\ 0 := \mathsf{a},\ 1 := \mathsf{b},\ 2 := \mathsf{h},\ 3 := (\mathsf{im}\ 1\ 0)\\ &\mathsf{in}\ (\underline{\mathsf{root}}\ 0\ (\mathsf{mp}\ 0\ 3\ (\mathsf{a1}\ 0\ 1\ (\mathsf{im}\ 0\ 3))\ 2\ 3))\end{aligned}$$

And now we can produce a proof stream by traversing this expression in postfix order, recursing into a numbered reference if it is the first appearance of the number. The numbers 0-1 in this case are already on the heap at the beginning of the stream because they are in the context.[13]

$$\begin{aligned}&\mathsf{Ref}\ 0, \mathsf{Hyp}^2,\\ &\quad \mathsf{Ref}\ 0,\\ &\quad \mathsf{Ref}\ 0, \mathsf{Ref}\ 1, \mathsf{Term\ im}, \mathsf{Save}^3,\\ &\quad \mathsf{Ref}\ 0, \mathsf{Ref}\ 1, (\mathsf{Ref}\ 0, \mathsf{Ref}\ 3, \mathsf{Term\ im}), \mathsf{Thm\ a1},\\ &\quad \mathsf{Ref}\ 2,\\ &\quad \mathsf{Ref}\ 3,\\ &\mathsf{Thm\ mp}.\end{aligned}$$

[13] The indentation and grouping is used here to indicate the tree structure, but the actual output is a plain list. The superscripts on Hyp and Save indicate what heap ID was associated to them. This is known by the compiler because the heap size goes up by one on each heap-modifying instruction and is initialized to 2 at the start, because there are two variables a, b in the theorem statement.

The unify stream is similarly obtained by hash-consing the expression ($\underline{\mathsf{root}}$ (im b a) a) containing the statement of the theorem (note that the hypotheses come in reverse order, after the conclusion) and writing the result in prefix order:

$$\mathsf{UTerm\ im},\ \mathsf{URef}\ 1,\ \mathsf{URef}\ 0,\ \mathsf{UHyp},\ \mathsf{URef}\ 0.$$

The overall `.mmb` file is thus produced by serializing the header, then the term and definition statements, then the theorem statements, and finally the declaration list which contains all statements in the order they were declared, with proof streams for terms, defs, axioms and proofs. While certainly more work than verification, the cost is not significantly different from compilation from regular programming languages.

2

Metamath One

So far, we have talked about the MM0 verifier, which checks a very explicit proof from some untrusted source. But in some sense checking such proofs is the easy problem, when compared with the problem of getting proofs in any kind of formally specified language in the first place. In order to make this pipeline useful, we need a way to produce formal proofs, and that means a front end to complement the MM0 back end.

There are two principal methods for producing `.mm0`/`.mmb` pairs: Translate them from another language, or write in a language that is specifically intended for compilation to MM0. (Translations are discussed in section 5.1.1.)

The MM1 language[1] has a syntax that is mostly an extension of MM0, but allows providing proofs of theorems. There are currently two MM1 compilers, `mm0-hs` written in Haskell and `mm0-rs` written in Rust, both of which provide verification, parsing and translation for all the MM0 family languages (the three formats mentioned in this paper, plus some debugging formats), and compilation of MM1 files to MMB. Furthermore, they provide a server compliant with the Language Server Protocol to provide editing support (syntax highlighting, live diagnostics, go-to-definition, hovers, etc.) for Visual Studio Code, extensible to other editors in the future.

[1] `https://github.com/digama0/mm0/blob/master/mm0-hs/mm1.md`

If one takes the MM0 file from Figure 1.1 and changes it to have extension `.mm1`, the VSCode editor mode will apply MM1 language rules to it, resulting in a warning saying that `id` is not proven. We can start the proof like so:

```
theorem id (P: wff): $ P -> P $ = '_;
                                -- ^ error here: |- P -> P
```

We can apply theorems to get subgoals:

```
theorem id (P: wff): $ P -> P $ =
'(ax_mp _ _);
     -- ^   |- ?a -> P -> P
       -- ^ |- ?a
```

Here `?a` is a *metavariable*, a placeholder for a term that is not yet known. They are usually resolved by *unification*, where applying theorems in sequence causes the conclusion for one theorem to match the hypothesis of the next and constraining the metavariables to be particular terms. This process is effective enough that the term arguments of a theorem usually do not have to be given, but in some cases the theorem can be fully constructed even though all metavariables are not yet solved:

```
theorem id (P: wff): $ P -> P $ =
'(ax_mp (ax_mp ax_2 ax_1) ax_1);
                       -- ^^^^ ?a: wff
```

The reason for this error is that it corresponds to the following proof:

1.	$(P \to (?a \to P) \to P) \to (P \to ?a \to P) \to P \to P$	ax_2
2.	$P \to (?a \to P) \to P$	ax_1
3.	$(P \to ?a \to P) \to P \to P$	ax_mp 1, 2
4.	$P \to ?a \to P$	ax_1
5.	$P \to P$	ax_mp 3, 4

This is a valid proof for any choice of $?a$, but we still have to provide a choice. We can use `!` before a theorem application to turn all implicit term arguments into explicit arguments so that we can provide a value:

```
theorem id (P: wff): $ P -> P $ =
'(ax_mp (ax_mp ax_2 ax_1) (! ax_1 _ _));
                                 -- ^ ?a: wff

theorem id (P: wff): $ P -> P $ =
'(ax_mp (ax_mp ax_2 ax_1) (! ax_1 _ P));
-- OK
```

There is one more step we need to do to make this `.mm1` function as a proof of the `.mm0` file from Figure 1.1, which is to mark the theorem as `pub`:

```
pub theorem id (P: wff): $ P -> P $ =
'(ax_mp (ax_mp ax_2 ax_1) (! ax_1 _ P));
```

Theorems are considered `local` by default, which means that they can be used as lemmas in further proofs but they do not themselves count toward the proofs in the `.mm0` file, which consists of the exported theorems, which depending on the nature of the development could be almost nothing except for the “final theorem” and the prerequisites for its statement, or it could be almost all the theorems for a reusable library file.

2.1 MM1 syntax

The syntax of MM1 is very similar to MM0, and in fact they share the same parser in `mm0-rs`. It allows the following extensions to the MM0 syntax:

- `import "foo.mm1";` statements can be used to have multi-file developments.[2]
- `def` and `theorem` accept visibility modifiers `pub`, `abstract`, `local`.
  - `pub` is for definitions or theorems that appear in the corresponding `.mm0` file.
  - `abstract` is for definitions only, when the corresponding definition in the `.mm0` file has an omitted body. An `abstract def` can reference `local` definitions.
- Type inference for variables is allowed:

  ```
  def and2 (a b) = $ a /\ b = 0 $;
  ```

  acts like

  ```
  def and2 (a: wff) (b: nat): wff = $ a /\ b = 0 $;
  ```

  assuming that `/\` and `=` have been defined with the natural types.
- Anonymous theorems like `theorem _: $ P -> P $ = ...;` are legal and will typecheck the proof but not add the theorem to the environment.
- Definitions (whether `abstract` or not) and theorems require proofs. This has the form `theorem foo: $ stmt $ = proof;` where `proof` is an s-expression that evaluates to a proof of `$ stmt $`.
- Statements of all kinds can have annotations like `@annot theorem ...` where `annot` is an s-expression. These work similar to Python decorators: the annotation value is passed to a hook along with the theorem that was added. These can be used to maintain data structures like simplification sets, or create additional automatically generated theorems based on the definition or theorem.
- `do { ... };` blocks can be written at the top level in order to evaluate an arbitrary sequence of s-expressions.
- Formulas can contain antiquotation, for example using `$ 2 + 2 = ,{2 + 2} $` to construct the formula `$ 2 + 2 = 4 $`.

The last four cases make use of the second layer of syntax in MM1, the MM1 metaprogramming language. This is a simple interpreted language with similar syntax and semantics to Lisp, or more precisely Scheme.[3] Here are a number of examples showing basic behavior of

[2] `mm0-rs` actually allows import statements for `.mm0` files as well as `.mm1` files, but they are not an official part of the specification, and the `mm0-rs join` tool can be used to concatenate `.mm0` import networks into a single file.

[3] R. Kent Dybvig. *The SCHEME programming language*. Mit Press, 2009

the primitives. In each case, the comment on the right shows what is printed when the line is executed.

```
do {
  "hello world"                -- "hello world"
  (display "hello world")      -- hello world
  (print (null? ()))           -- #t
  (print @ null? ())           -- #t
  (if (null? '(1)) 0 1)        -- 1
  (if (null? ()) 0 1)          -- 0
  {2 + 2}                      -- 4
  '{2 + 2}                     -- (+ 2 2)
  {1 < 2 < 3 < 4}              -- #t
  {1 < 2 < 3 < 3}              -- #f
  {1 * 2 * 3 * 4}              -- 24
  (max 1 2 3 4)                -- 4
  (min 1 2 3 4)                -- 1
  (hd '(* 1 2 3 4))            -- *
  (tl '(* 1 2 3 4))            -- (1 2 3 4)
  (list 1 2 3 4)               -- (1 2 3 4)
  (def x 5)
  {x + x}                      -- 10
  (def (x) 5)
  x                            -- #<closure>
  (x)                          -- 5
  (def (fact x)
    (if {x = 0}
      1
      {x * (fact {x - 1})}))
  (fact 5)                     -- 120
  ((fn (a) ''a) 1)             -- (quote a)
  ((fn (a) '',a) 1)            -- (quote 1)
  ((fn (a) '(,a . ,a)) 1)      -- (1 . 1)
  (cons (list (def x 2) x) (x))-- ((2) . 5)
};
```

Most of the syntax should be familiar to those with a Lisp background.[4] Some notes on the syntax:

[4] Why Lisp? Because it is one of the simplest languages to implement an interpreter for. We can really use any metaprogramming language here as long as it is Turing complete.

- `(f a b)` is a function call to `f` with arguments a, b, i.e. the equivalent of `f(a, b)` in C-like languages.
- `'expr` is *quotation*, an operation that causes each expression to yield itself instead of being evaluated. As shown above, `(+ 2 2)` evaluates to `4`, but `'(+ 2 2)` evaluates to the expression `(+ 2 2)`, a list with the three elements `+`, `2`, `2`.
- Inside a quotation, `,expr` is *antiquotation*, which evaluates `expr` normally and interpolates it into the expression. `'(+ 2 ,(+ 2 2))` evaluates to `(+ 2 4)`.
- `{a R b}` means the same as `(R a b)`. It is conventionally used for infix operators like `+` so that they don't need to be written in polish

notation as is common in lisp.

- Square brackets `[e1 ... en]` are interchangeable with `(e1 ... en)`.
- `(f a @ g b)` means the same as `(f a (g b))`. This is similar to the `$` operator in Haskell for decreasing parentheses when there are many nested tail expressions (which is very common in Lisp).
- `(fn (x y) {x + y})` is a *closure*, in this case a function that takes two arguments and adds them together. Closures can capture variables in an outer scope, for example `(fn (x) (fn (y) {x + y}))` is a function of one argument x returning a function of one argument y which will add x to y.
- `(def x {2 + 2})` can be used to define and assign a new variable. `(def (f x y) {x + y})` is shorthand for `(def f (fn (x y) {x + y}))`.
- `(begin e1 e2 ... en)` evaluates each of `e1`, `e2`, ... in order, returning the result of `en`.

2.1.1 *Tactics*

In addition to basic language features, there are also a number of features that more directly tie in to the proof assistant:

- Formulas like `$ P -> P $` use the math parser (including all declared notations) to parse the expression into a parse tree. This example is equivalent to writing `'(im P P)`. It also supports antiquotation, so `$ 2 + 2 = ,{2 + 2} $` evaluates to `(eq (add 2 2) 4)`.
- The `(match e [pat1 val1] ... [patn valn])` expression matches an expression `e` against a number of patterns, which can also be formulas. This is useful for defining proof automation.

A *tactic* is a metaprogram used to derive a proof. In order to facilitate tactic programming, these proofs are expressed as s-expressions, where each theorem application is applied to its list of substitutions and subproofs. For example, the proof of $(a \to b) \to (a \to b)$ using `id` would be written as `(id (im a b))`, and if we apply this to $h : a \to b$ to redundantly prove $a \to b$, the proof term would be

```
(ax_mp (im a b) (im a b) (id (im a b)) h)
```

where we pass $a \to b$ for the substitution arguments, `(id (im a b))`: $(a \to b) \to (a \to b)$ for the first subproof argument, and `h`: $a \to b$ for the second subproof argument.

There are a few built-in tactics, and the most important and flexible built-in tactic is `(refine)`. This tactic is so important, in fact, that it is called by default whenever a proof yields a value other than `#undef`.

In a refine script, substitution arguments are omitted, so the same `id` example would instead be written `'(ax_mp id h)`. The arguments can be reinserted per-application using `(! ax_mp (im a b) _ id h)`, where `_` can be used to omit any part of the proof. If a part of the proof cannot be inferred, for example if the proof is `(ax_mp id _)`, the `_` will have an error highlight describing the expected type, and interactive proof primarily goes by filling such holes with more subproofs.

If a function is passed directly inline in a refine script as in `(ax_mp id ,f)`, the function will be called like `(f refine tgt)` where `refine` is a callback and `tgt` is the expected type of the subgoal. Quite often this is the most convenient way to call custom automation in the middle of a proof.

Here is an example that defines a general purpose refine script `conj-prove`, which will prove theorems like $a \wedge ((b \wedge c) \wedge d) \rightarrow b$ by searching in the tree of conjunctions for the expression on the right hand side. It brings together many of the features we have discussed so far.

```
delimiter $ ( ~ $  $ ) $;

provable sort wff;

term im: wff > wff > wff; infixr im: $->$ prec 25;
axiom id (a: wff): $ a -> a $;

term and: wff > wff > wff; infixl and: $/\$ prec 35;
axiom anwl (h: $ a -> c $): $ a /\ b -> c $; -- axiomatizing these theorems for brevity
axiom anwr (h: $ b -> c $): $ a /\ b -> c $;

do {
  -- (find lhs rhs) returns a proof of lhs -> rhs where lhs is a tree of conjunctions
  -- with rhs in one of the leaves, or #undef if rhs is not found
  (def (find lhs rhs)
    (match lhs
      [$ ,l /\ ,r $
        (match (find l rhs) -- if it is a conjunction, then look on the left
          [#undef
            (match (find r rhs) -- or on the right
              [#undef #undef] -- if not found, return #undef
              -- if proof of r -> rhs found, apply anwr to prove l /\ r -> rhs
              [result '(anwr ,result)])]
          -- if proof of l -> rhs found, apply anwl to prove l /\ r -> rhs
          [result '(anwl ,result)])]
      -- if it's not a conjunction, but it is rhs, then id proves lhs -> rhs, else fail
      [_ (if {lhs == rhs} 'id #undef)]))

  -- tactic to prove theorems like: |- a /\ ((b /\ c) /\ d) -> b
  -- It receives two arguments, a callback 'refine' to pass the constructed proof script,
  -- and 'tgt' which is the goal theorem
  (def (conj-prove refine tgt)
    -- match on the target type, e.g. a /\ (b /\ c) -> d
```

```
      (match tgt
        [$ ,lhs -> ,rhs $
          -- call find and return the proof
          (refine tgt (find lhs rhs))]))
  };

  -- Examples of using ‘conj-prove‘
  theorem _: $ a /\ b /\ c /\ (d /\ e) -> d $ = conj-prove;
  theorem _: $ a /\ b /\ (d /\ e) -> d $ = conj-prove;
  theorem _: $ a /\ b /\ (e /\ d) -> d $ = conj-prove;
  theorem _: $ d /\ b /\ (e /\ d) -> d $ = conj-prove;
  theorem _: $ d /\ b /\ (e /\ d) -> b $ = conj-prove;
  theorem _: $ b -> b $ = conj-prove;
  theorem _: $ d /\ b /\ (e /\ d) -> d $ = conj-prove;
  -- This starts the proof with anwr and finishes the proof with conj-prove,
  -- so it finds a different proof for the same theorem
  theorem _: $ d /\ b /\ (e /\ d) -> d $ = '(anwr ,conj-prove);
```

Other than simply calling `refine`, it is also possible to use the *tactic mode* to prove a theorem. Every theorem starts with an initial tactic state consisting of one goal, the theorem statement. The tactic state in general consists of a list of subgoals that remain to be proven, and `refine` will create subgoals for any `_` that appears in the provided script. Here are some other builtin tactics:

- `(focus e1 e2 ... en)` first stashes all goals other than the first one and sets the goal state to consist only of the first goal. It then evaluates each of the `ei`, and if it does not evaluate to `#undef` it is passed to `(refine)`. At the end, all subgoals should have been solved, else there is an error, and the stashed goals are restored after the `focus` block. This tactic is used mainly for block-structured proofs, since everything related to solving the first goal appears in this block, and later goals can be handled by subsequent `focus` blocks.

- `(have 'h $ty$ '(pr))` will introduce a new named subproof named `h` of statement `$ty$`, where `'(pr)` is the proof of the statement.

Most tactics beyond this are written directly in MM1; we will discuss this in section 2.2.1.

2.1.2 *MM1 tooling features*

The program `mm0-rs` is a Rust application with a number of tools for working with MM1 files. It is approximately 26 000 lines of code, with an additional 29 000 lines for the MMC compiler. Here is an incomplete list of the things it can do:

- The `mm0-rs join` command will concatenate `.mm0` files which use `import "file.mm0";` lines, which are accepted even though they are not technically part of the MM0 specification. This allows the use of `import` for structuring a development, while still being able to pack everything in one file for "distribution" or to interface with a simple (perhaps formally verified!) verifier which does not have a full understanding of the filesystem.

- The `mm0-rs server` command will start an LSP[5] server which supports many LSP features: auto-completion, hover info, go-to-definition, list all definitions, find references, syntax-aware rename, semantic highlighting.
- The `mm0-rs compile` command will compile a `.mm1` file and report any warnings or errors. This is essentially the command-line version of the server mode. This is also used for generating `.mmb` files for completed developments.
- The `mm0-rs doc` command will generate a static website for specified theorems.

[5] As the name suggests, the Language Server Protocol is a standard communication protocol for language servers. This means that an LSP server will work on almost any editor, although the majority of testing has been done on Visual Studio Code.

Most other features are exposed through the MM1 language itself:

- It is possible to `import "foo.mmb";` from an MM1 file to import external developments.
- The metaprogramming language has decent performance: it is compiled down to bytecode and evaluated, similar to Python.[6]
- There is support for the `output string` command, which will generate strings either on standard out or back into the metaprogramming language.

[6] We have plans to use Just-in-time compilation (similar to Java or Javascript) to execute even more efficiently, although at the moment the cost is not too heavy.

2.2 *Proof developments using MM1*

2.2.1 *The `peano.mm1` metaprogramming library*

The file `peano.mm1` which acts as the axiomatic basis for almost all subsequent work in this project is also home to a small (≈400 line) library of extensions of the builtin facilities of the language.

- `append`, `filter`, `len`, `range`, `for`, `iterate`, `find` are basic list utilities.
- There are also basic tactics like `exact`, `swap`, `suffices`.
- There are utilities for debugging and error reporting.
- The `(named)` tactic wraps a proof script to fill all remaining variables with fresh names. This solves the issue we saw with the `id` proof requiring a term argument, in the case where the missing arguments are variable names (which is a common scenario in FOL proofs).
- An annotation hook is added so that `@f def foo` evaluates `(f 'foo)` if it is a function, and `@_ def foo` evaluates `(default-annotate 'foo)`.

- The default annotation runs a metaprogram `(derive-eq)`, which generates equality theorems for every definition in terms of equality theorems on all the arguments. For example, for a definition like

  ```
  @_ def bool (n: nat): wff = $ n < 2 $;
  ```

  it generates the theorem

  ```
  theorem booleq (_n1 _n2: nat):
    $ _n1 = _n2 -> (bool _n1 <-> bool _n2) $;
  ```

  by unfolding each side and then applying the corresponding equality theorem for "<".

- The `eqtac` refine script automates proofs of $a = b \to P(a) = P(b)$, which forms the core of most beta reduction / substitution theorems in `peano.mm1`. For example applications of the theorem $(\forall x\, P) \to P[a/x]$ will usually use `eqtac` to evaluate $P[a/x]$.

- An extensible general purpose evaluator `(eval)` is implemented by attaching evaluation functions to some definitions and automatically deriving other evaluation functions by unfolding other definitions. Since most things are defined over `nat` many things are computable in this way. For example, we tell it that the term `d0` is implemented by `0` and `suc x` is implemented by `(fn (x) {x + 1})` and `add x y` is implemented by `+`, from which we can automatically derive that `(eval $2 + 2$)` $= 4$.[7]

[7] This is similar to the `#eval` command from Lean: it is not proof-producing and is only used to reflect numerical assertions to actual Lisp numerical operations where they can be executed.
Even without proof production this can be useful: for example, the first step in a primality prover is usually to decide whether the number is in fact prime or not, because if it is not then there is generally a much better proof than a failure in the middle of a primality test.

2.2.2 *`peano.mm1`: Peano arithmetic*

Peano arithmetic (PA) is a first order axiomatization of the theory of natural numbers. The axiomatization[8] consists of the following components:

[8] `https://github.com/digama0/mm0/blob/master/examples/peano.mm1`

- Classical propositional logic, using essentially the Łukasiewicz axioms shown in Figure 1.1, with the addition of a truth constant and `axiom itru: $ T. $`.

- Predicate logic implemented with the axioms from Figure 1.2.

- The non-logical axioms of PA: a term `d0: nat` (denoted 0) and `suc: nat > nat`, axioms that say `suc` is injective and not equal to 0, and the induction axiom:[9]

  ```
  axiom peano5 {x: nat} (p: wff x):
    $ [0 / x] p -> A. x (p -> [suc x / x] p) -> A. x p $;
  ```

[9] It is worth pointing out that although PA is "not finitely axiomatizable,' this is clearly a finite axiomatization. The ability for MM0, like Metamath, to natively support schemes through its use of open formula variables like `(p: nat x)` is key to this – MM0 axioms and theorems correspond to axiom and theorem schemes in the traditional reckoning.

- Class theory over `nat`. This introduces a new sort `set` which represents possibly infinite subsets of $\mathbb{N}$, together with the $x \in A$ and $\{x \mid p(x)\}$ operations and the axiom $a \in \{x \mid p(x)\} \leftrightarrow p(a)$.

- A definite description operator `the: set > nat` such that `the` $\{x\} = x$, and `the` $A = 0$ if A is not a singleton.

The last two points are not traditionally a part of PA, but they are a conservative extension: any theorem about classes can be converted to a theorem (scheme) about wff predicates with one designated free variable, and any theorem about `the A` is equivalent to one without it by rewriting using the theorem:[10]

```
theorem eqtheb: $ a = the A <->
  (A == {x | x = a} \/ ~E. y A == {x | x = y} /\ a = 0) $;
```

[10] `E. y p` is the notation for $\exists y\ p(y)$, and `A == B` is equality of classes, $(A = B) := \forall z\ (z \in A \leftrightarrow z \in B)$.

In future work, we would like to formalize the metatheory of this axiom system, and prove that the provable formulas in this system correspond to (schemes of) provable formulas in "textbook PA."

Once the axiom system is set, we can start to prove some theorems. As of this writing, `peano.mm1` contains 2687 theorems, of which 668 are automatically generated.

- The first 700 or so come before the non-logical axioms of PA and concern general FOL, class theory, and the definite description operator.
- The next 100 theorems are about $+, -, *, \leq, <$ and induction.
- There are several other basic operations like `finite A`, `if`, `bool`, `min`, `max`.
- The operators `x // y` (flooring division), `x % y`, `x || y` (divisibility), `mod(n): x = y` add another hundred theorems, including the quotient-remainder theorem.
- The theory of the definitions `b0 x := 2 * x`, `b1 x := 2 * x + 1`, and `odd x`, in addition to being useful for the obvious parity applications, is also leveraged to implement disjoint sums: `Sum A B`[11] can be defined such that the even numbers are doubles of `A` and the odd numbers are double-plus-one of elements of `B`. This construct even plays triple duty as a pairing operator for classes, since `A` and `B` are uniquely recoverable from `Sum A B`.

[11] We will color `Sum A B` like a type even though it is a regular definition, because the MM0 built-in type system becomes less relevant around this point now that we have a more powerful (dependent) type system expressed using expressions of sort `set`.

- The pairing function `(a, b)` is defined as

  ```
  def pr (a b: nat): nat = $ (a + b) * suc (a + b) // 2 + b $;
  ```

  which is a direct transcription of the cantor pairing function:

  $$(a, b) := \frac{(a+b)(a+b+1)}{2} + b$$

  We prove that this function is a bijection, and so we also get functions `fst` and `snd` which invert it.
- Using ordered pairs, we can build a few "lambda operators:"
  - The "regular" lambda operator we define when $a(x)$: `nat` as $\lambda x.\ a(x) = \{p \mid \exists x.\ p = (x, a)\}$. This is always a proper class, since it has domain `nat`.

- We define $\lambda_S\, x.\ A(x) := \{z \mid \pi_2(z) \in A(\pi_1(z))\}$, so $(a, b) \in \lambda_S\, x.\ A(x) \leftrightarrow b \in A(a)$. Because $(-, -)$ is treated as a right associative operator, we can build up n-ary relations such that $\lambda_S\, x.\ \lambda_S\, y.\ \{z \mid p(x, y, z)\}$ is notation for $\{(x, y, z) \mid p(x, y, z)\}$.
- We define $\lambda_f\, x.\ A(x) := \{((a, b), y) \mid (b, y) \in A(a)\}$. This is useful for building up n-ary functions, since $\lambda_f\, x.\ \lambda_f\, y.\ \lambda z.\ a(x, y, z)$ expresses the equivalent of $\lambda(x, y, z).\ a(x, y, z)$.
- $\prod x \in A.\ B(x) := \{(x, y) \mid x \in A \wedge y \in B(x)\}$. This is the dependent version of the cartesian product $A \times B$. Notably, we can prove that $\prod x \in A.\ B(x)$ is finite if A is finite and $B(x)$ is finite for all $x \in A$, which is analogous to the replacement axiom of ZF (but for finite set theory).

There are also corresponding application operators:

- $F \mathbin{@_S} a := \{x \mid (a, x) \in F\}$ is complementary to λ_S, in the sense $(\lambda_S\, x.\ A(x)) \mathbin{@_S} a = A(a)$.
- $F \mathbin{@} a :=$ `the` $(F \mathbin{@_S} a)$ is complementary to λ, in the sense $(\lambda\, x.\ v(x)) \mathbin{@} a = v(a)$.
- $F \mathbin{@_f} a := \lambda_S\, y.\ F \mathbin{@_S} (a, y)$ is complementary to λ_f, in the sense $(\lambda_f\, x.\ F(x)) \mathbin{@_f} a = F(a)$.

By combining all of these we get a fairly flexible way of defining functions and predicates with arbitrary arity encoded as elements of `set` (i.e. subsets of $\mathbb{N}$).

- Using the aforementioned encoding using `b0` and `b1`, we can treat $\mathbb{N}$ as a disjoint sum of two copies of itself, and use that to interpret $\mathbb{Z}$, where `b0 x` means x and `b1 x` means $-x - 1$.
  - $x -_{\mathbb{Z}\mathbb{N}} y$ is the $\mathbb{N} \to \mathbb{N} \to \mathbb{Z}$ function which takes the difference of two natural numbers as an integer
  - `zfst x` $= \max(x, 0)$ is the "positive part", and `zsnd x` $= \max(-x, 0)$ is the "negative part" of an integer
  - $x + y$, $-x$, $x - y$, $x * y$, $x \le y$, $x < y$, $|x|$, $x \mid y$, $x \bmod y$, and the $x = y \pmod{n}$ relation are defined on $\mathbb{Z}$ by operating on positive and negative parts
- The $\gcd(x, y)$ function is defined, along with its key properties, as well as $\mathrm{coprime}(x, y)$ and the modular inverse function $\mathrm{inv}_n(x)$.
- This builds up to the definition (due to Gödel):

$$\mathsf{pset}(m, v) := \{n \mid (\forall x.\ 0 < x \le n \to x \mid m) \ \wedge\ m(n+1) + 1 \mid v\}$$

The key point is that the numbers $m(n+1)+1$ are coprime for different choices of n, so if we fix m to be a number that divides all the numbers up to N, then for any subset $A \subseteq \{0, \ldots, N\}$ we can take the product $v := \prod_{n \in A}(m(n+1)+1)$ and so prove $\mathsf{pset}(m, v) =$

A. The upshot is that every finite set A is encoded by a natural number (in this case (m, v)).

- We prove the equivalent of the separation axiom and the replacement axiom:
  - $\exists a.\ \mathrm{pset}(a) = \{x \mid x < n \wedge p(x)\}$
  - $\mathrm{finite}(A) \to \exists a. \forall x \in A.\ \mathrm{pset}(a)\ @\ x = v(x)$
- This suffices to bootstrap a recursion operator satisfying:
  - $\mathrm{rec}_{z,S}(0) = z$
  - $\mathrm{rec}_{z,S}(n+1) = S\ @\ \mathrm{rec}_{z,S}(n)$

  and we can include the index of recursion as well:
  - $\mathrm{recn}_{z,S}(0) = z$
  - $\mathrm{recn}_{z,S}(n+1) = S(n, \mathrm{recn}_{z,S}(n))$ [12]
- Lots of useful functions can now be defined:
  - x^y, $\mathrm{shl}(x, y) := x \cdot 2^y$ and $\mathrm{shr}(x, y) := \lfloor x/2^y \rfloor$;
  - $\mathrm{ns}(a) := \{x \mid \mathrm{odd}(\mathrm{shr(x,y)})\}$ is registered as a coercion `nat` $\to$ `set`, so that $(x \in y) \leftrightarrow \mathrm{odd}(\mathrm{shr(x,y)})$. The $\mathrm{ns}(x)$ function also has the same separation and replacement properties as $\mathrm{pset}(x)$, and it is a bit easier to reason about, but we needed $\mathrm{pset}(x)$ to prove the existence of these functions in the first place.
  - The inverse of $\mathrm{ns}(a)$ is $\mathrm{lower}(A) :=$ `the`$\{n \mid \mathrm{ns}(n) = A\}$. Since every finite set is in the image of ns, we have that $\mathrm{lower}(\mathrm{ns}(x)) = x$, and $\mathrm{ns}(\mathrm{lower}(A)) = A \leftrightarrow \mathrm{finite}(A)$.
- We finish off finite set theory with singleton sets and insertion, the set $\mathrm{upto}(n) = \{0, \ldots, n-1\} = 2^n - 1$, and some sets like bool, $\mathcal{P}(A)$, and $\bigcup A$.[13]
  - We also add another lambda: $\lambda(x \in a).\ F(x)$ is $(\lambda x.\ F(x)) \upharpoonright a$, which is finite since a : `nat`. So this is useful when we want a function to be syntactically a finite set when it has a finite domain.
- We can now define strong recursion:
  - $\mathrm{srec}_S(n) := S(\lambda(x \in \mathrm{upto}(n)).\ \mathrm{srec}_S(x))$
- We can also have generalized recursion which depends on a varying parameter:
  - $\mathrm{grec}_{z,K,S}(0, k) = z$
  - $\mathrm{grec}_{z,K,S}(n+1, k) = S(n, k, \mathrm{grec}_{z,K,S}(n, K(n, k)))$
- Lists are defined by iterated application of the cons operator $(a : b) := \mathrm{suc}(a,\ b)$. So for example $a : (b : (c : 0)) = [a, b, c]$. The list recursion operator is defined using strong recursion:
  - $\mathrm{lrec}_{z,S}([]) = z$
  - $\mathrm{lrec}_{z,S}(a : l) = S(a, l, \mathrm{lrec}_{z,S}(l))$

[12] We will omit the @ for brevity in the following descriptions: when S : `set`, $S(a)$ denotes $S\ @\ a$.

[13] These are still subject to the usual constraints that every element of a `set` is a `nat` and hence can only encode a finite set.

- Many simple list operations can now be defined: length, membership, append, nth-element, repeat, reverse, map, join, filter, zip, take, drop, sublist.

These are all fairly basic results, but they make everything to follow go more smoothly since PA is a "generous arena" for doing almost anything with natural numbers or other countable sets, and is also powerful enough to work with infinite sets, as long as they can be constructed by a formula.

2.2.3 *`peano_hex.mm1`: Hexadecimal arithmetic*

This file[14] builds on `peano.mm1` with a definition of hexadecimal numbers and an arithmetic evaluator. This file is much more automatic: there are 1129 more theorems, but only 138 of them are literally introduced by the `theorem` keyword.

[14] https://github.com/digama0/mm0/blob/master/examples/peano_hex.mm1

- It starts with the string axioms, which is an axiomatic extension of `peano.mm1` to include:
  - A sort `hex` and terms `x0`, ..., `xf`: `hex` which are the only inhabitants of the sort
  - A sort `char` and term `ch` h_1 h_2 : `char` where h_1, h_2 : `hex`
  - A sort `string` with terms:
    * `s0`: `string` representing the empty string
    * `s1` c : `string` where c : `char`
    * `sadd` s_1 s_2 : `string` where s_1, s_2 : `string` for string append

  MM0 has no mechanism for adding new sorts except by axiomatization, so these are proper axiomatic extensions. See section 4.8.2 for more information on why we want literal axioms here even though these types can all obviously be modeled using `nat`, and hence this is a conservative extension.

- On their own, these sorts cannot really be reasoned about because we have no operations to do anything other than construct strings. So we axiomatize some more functions:
  - `h2n: hex > nat`, `c2n: char > nat`, `s2n: string > nat` which embed the values as elements of `nat`
  - The functions evaluate as expected on all the term constructors:
    * `h2n x0 = 0,...,h2n xf = 15`
    * `c2n (ch hi lo) = h2n hi * 16 + h2n lo`
    * `s2n s0 = []`
      `s2n (s1 c) = [c2n c]`
      `s2n (sadd s t) = s2n s ++ s2n t`

  – Since we know that there are no other constructors of the `hex` sort, we can also assert that `h2n h < 16` for *any* `h: hex`. This cannot be proved but we add it as an axiom.
  – Similarly, `c2n h < 256` and `s2n h e. List (upto 256)`, where `List A` is the set of lists of elements of `A`.

All of this is clearly seen to be a conservative extension, where we interpret an element of `hex` as a `nat` less than 16, a `char` as a `nat` less than 256, and a `string` as a list of `char`s. There are some facts like $13 < 16$ that can be derived by combining these axioms, so we prove these in advance of the axiomatization just in case.

The `hex` sort turns out to be convenient for representing arbitrary numbers in hexadecimal, and base 16 is a good sweet spot between the size of the initial times tables and the number of digits to work through for doing arithmetic. We start by introducing `hex: nat > hex > nat` such that `hex` $n\ h := 16n + h$. A number is represented as a list of `hex` applications applied to `h2n`, for example `hex (hex (h2n xa) x1) xe` is the hexadecimal number `0xa1e` or 2590. Writing `hex` as an infix operator `:x`, this would be written `xa :x x1 :x xe` using MM0 notations, but by adding a hook for numeric parsing the same term can be written in MM1 as `,0xa1e` or `,2590` (using antiquotation to insert a literal number into the term, which is translated by a `refine` hook into a hexadecimal expression).

Because it will be relevant for Chapter 4, let us look in more detail at how the successor algorithm works. We have the following theorems:

```
theorem decsuc_lem (h1: $ h2n a = d $) (h2: $ h2n b = suc d $):
  $ suc a = b $ = '(eqtr4 (suceq h1) h2);
theorem decsucf:
  $ suc xf = x1 :x x0 $ = '(eqtr4 suc_xf hex10);
theorem decsucx (h: $ suc b = c $):
  $ suc (a :x b) = a :x c $ = '(eqtr3 addS2 (addeq2 h));
theorem decsucxf (h: $ suc a = b $):
  $ suc (a :x xf) = b :x x0 $ = '(eqtr suchexf (hexeq1 h));
```

(The proofs are not important, but included just to give a sense of how they get discharged.) We use `decsuc_lem` and some one-shot automation to prove that, since we know `h2n x5 = 5` and `h2n x6 = 6` (these are axioms) and `suc 5 = 6` by definition, we can apply `decsuc_lem` to prove `theorem decsuc5: $ suc x5 = x6 $`. We work out this theorem for all 16 digits, except for `xf` which has a different statement because of the carry.

The successor algorithm is a metaprogram `(mksuc a)` returns a pair `(b p)` where `p: suc a = b`.

• If the input has the form `a :x xf`, then let `(b p) := (mksuc a)` and then return `b :x x0` as the successor term and `(decsucxf p)` as the proof.

- If the input has the form `a :x b` for `b` $\neq$ `xf`, then let `(c p) := (mksuc b)` and then return `a :x c` as the successor term and `(decsucx p)` as the proof.
- If the input has the form `h2n xf`, then `x1 :x x0` is the term and `decsucf` as the proof.
- If the input has the form `h2n` xi for $i \neq 15$, then `h2n` x$(i+1)$ is the term and `decsuc`i is the proof.

In short, we need two things for each proof:

- A collection of "pro-forma" theorems which have been prepared in order to cover all of the cases of the algorithm. These are usually single-purpose lemmas, and may be obtained by a secondary automation, in this case for the 15 `decsuc`i lemmas which all have the same form.
- An algorithm which encodes the actual execution pattern to construct the proof. This execution pattern need not have anything to do with the way the definitions are written (e.g. evaluating unary natural numbers because Peano arithmetic is defined that way).

The large number of automatic theorems in this section is primarily due to things like the addition table, which requires theorems for all $16 \times 16 = 256$ combinations in the single digit case.[15]

[15] There are ways to make the precomputation smaller at the cost of having longer proofs "at runtime," for example by only keeping the table of $a+b$ for $a \leq b$ and applying a commutativity step otherwise. But these proofs are relatively low cost, and it is difficult to estimate how much arithmetic will be done using them during compilation and assembly, but it seems safe to err on the side of optimizing for runtime.

The file contains:

- Proofs of $a < b$ for all the digits for which this is true
- An algorithm which decides $a < b$, $a = b$, or $b < a$ given two numerals
- Proofs of $a + b = c$ and $\mathrm{suc}(a+b) = c$ for all digits a, b
- An algorithm to decide $a + b = c$ and $\mathrm{suc}(a+b) = c$ for numerals a, b by mutual recursion
- Proofs of $a \cdot b = c$ for all digits a, b; an algorithm deciding this for numerals

A meta-algorithm `norm_num` puts all these algorithms together to decide any closed statement involving digits, `suc`, $+$, $-$, $*$, $/$, %, `b0`, `b1`, or `c2n`. Some operations are deliberately not evaluated, like (x, y), because these are normally used for structural purposes and not as numerical operations.

2.2.4 *`mm0.mm1`: A formal specification of MM0*

The purpose of the file `mm0.mm0`[16] is to give a complete definition of what it means for a MM0 file to be "provable," starting from the string

[16] https://github.com/digama0/mm0/blob/master/examples/mm0.mm0

data of the file, through lexing, parsing, elaboration, and then the actual abstract syntactic rules in section 1.4.3.

For example, the rule saying that a theorem is well formed:

$$\frac{\Gamma\ \mathsf{ctx} \quad \overline{\Gamma \vdash A : s} \quad \Gamma \vdash B : s'}{\mathsf{thm}\ (\Gamma; \overline{A} \vdash B)\ \mathsf{decl}}$$

is expressed as:[17]

```
theorem DeclThm (env args hs ret: nat) {x: nat}:
  $ Decl env (DThm args hs ret) <->
    Ctx env args /\ all {x | ExprProv env args x} (ret : hs) $;
```

[17] Note that the MM0 type system is not particularly useful at this point: literally everything has type `nat`. This is expected; types are instead expressed using logical theorems. In this case, `Decl:  Env -> Decl -> wff`.

And the rule for deriving that two term applications are definitionally equal:

$$\text{C-CONG}\quad \frac{\Gamma \vdash f\,\overline{e} : s \quad \Gamma \vdash f\,\overline{e'} : s \quad \forall i, \vdash e_i \equiv e_i'}{\vdash f\,\overline{e} \equiv f\,\overline{e'}}$$

is expressed as:

```
--| VerifyConv (env: Env) (ctx: Ctx)
--|   (c: CExpr) (e1 e2: SExpr) (s: SortID): wff
--| means c is a proof of (env, ctx) |- e1 = e2 : s
def VerifyConv (env ctx c e1 e2 s: nat): wff;
...
theorem VerifyConvCong
  (env ctx f cs e1 e2 s: nat) {args ret o es1 es2: nat}:
  $ VerifyConv env ctx (CCong f cs) e1 e2 s <->
    E. args E. ret E. o E. es1 E. es2 (
      e1 = SApp f es1 /\ e2 = SApp f es2 /\
      getTerm env f args ret o /\
      VerifyConvs env ctx cs es1 es2 args /\
      s = fst ret) $;
```

Note that these are theorems asserting that a certain definition, `VerifyConv` in this case, which is left abstract, satisfies a certain definitional unfolding theorem. The purpose of the `mm0.mm1` file[18] is to prove that these theorems are provable, which is not completely trivial since the definitions can sometimes involve mutual recursion or induction, and MM0 has no native support for recursive `def`, so all definitions have to go through the various recursors defined in section 2.2.2.

[18] https://github.com/digama0/mm0/blob/master/examples/mm0.mm1

There is some rudimentary support for automatically generating pattern matching theorems by recognizing sum / product constructions like this:

```
def CRefl (e: nat): nat = $ b0 (b0 (b0 e)) $;
def CSymm (p: nat): nat = $ b0 (b0 (b1 p)) $;
def CTrans (p q: nat): nat = $ b0 (b1 (p, q)) $;
def CCong (f cs: nat): nat = $ b1 (b0 (f, cs)) $;
def CUnfold (f es zs: nat): nat = $ b1 (b1 (f, es, zs)) $;
```

in order to generate induction lemmas, but there is nothing at the level of e.g. the Isabelle inductive package yet, so most of the theorems are proven with explicit proofs.

2.2.5 `x86.mm1`: *A formal specification of the Intel x86 ISA*

The MMC compiler described in Chapter 4 produces proofs of program correctness relative to a model of the instruction set, so we first need a model of the target instruction set. We target Intel x86 for this initial implementation because it is practical and widespread, and extension to other architectures is future work. The `x86.mm0` file[19] consists of two main functions:

[19] https://github.com/digama0/mm0/blob/master/examples/x86.mm0

- The `decode ast l` relation, which is a single instruction disassembler: `ast e. XAST` represents an x86 instruction as an element of an inductive type / discriminated union, and `l e. List u8` is a list of bytes for the encoded instruction. Since it is a relation it is not inherently directed: it can be read either forwards or backwards to function as either an assembler or disassembler.
- The `execXAST k ast k2` relation, which expresses the result of stepping the machine state `k e. Config` on a given instruction `ast e. XAST` to yield a new state `k2 e. Config`.

These are combined to form the step relation for the machine:

```
def step (k k2: nat): wff =
$ k e. Config /\ readException k = 0 /\ E. l E. ast (
  readMemX k (readRIP k) l /\
  decode ast l /\
  execXAST (writeRIP k (readRIP k +_64 len l)) ast k2) $;
```

In words, this says that when the machine takes a step:

- The initial state should be a `k e. Config` (this is just a typing condition).
- The state can only take a step if the current exception state is `0`, meaning no error.
- The list of bytes `l` should be available in memory starting at `readRIP k`[20] with executable permissions. (The length of the read is nondeterministic and pinned down by the next rule.)
- The result of decoding `l` to an instruction should be `ast`.
- We increment `RIP` by the length of the instruction,[21] and then use `execXAST` to get the resulting state after executing `ast`.

[20] `RIP` is the instruction pointer register, also known as the program counter. It represents the location where code is currently being executed. The function `readRIP k` gets the value of `RIP` from the state, and `writeRIP k v` is the state that results after setting `RIP := v`.

[21] In x86, operations that depend on the current value of `RIP` like relative jumps or `RIP`-relative addressing will actually use the *end* of the current instruction when performing such calculations.

The definitions of `decode` and `execXAST` are large, handling approximately 89 instructions (although due to factoring out common parts there are effectively only 25 instructions that need separate handling.)

This is still far short of the approximately[22] 1503 instructions in the full specification. To ensure that this spec is an underapproximation of the real one, any byte sequence that does not encode to one of the known instructions is considered "undefined behavior" in the sense that it is a stuck state: `step k k2` is false for all `k2` if the instruction pointer in `k` points at an unknown instruction.

[22] It is surprisingly hard to get a straight answer to "how many instructions are there in x86." This is the number of instruction classes in the Intel XED disassembler library, which suffices to get a ballpark estimate of the complexity of the instruction set.

This suffices for "core x86" execution, but one instruction that we model is the `syscall` instruction, which allows for interaction with the environment. We need this instruction at least to exit the program, as well as to read the input and produce output. This is a very versatile instruction as is effectively a function call into the operating system, so to support this we need to model some of the operating system's state in addition to the program state.

There are multiple ways to do this, and we choose the simplest one which allows specification of one-shot IO programs. The basic idea is that we extend the program state to include two lists $i, o \in$ `List u8`, where i is the list of all input that has been read since the start of the program, and o is the list of all output that has been produced since the start of the program. With this modification, it is possible to execute through the `read()` and `write()` syscalls, such that the nondeterministic possible executions trace out the input-output graph of the program.

The last component we need is a specification of ELF, the executable and linker format, which is a file format that specifies how to load the program into memory and start execution. This is used to define the initial state of the program. We do not do a full specification (ELF is complex and very extensible), but instead only specify ELF files with one program segment which contains the executable data.[23]

[23] This is a *very* stripped down version of the ELF spec, just barely enough for the loader to use it to execute without errors. "Normal" executables have other sections which describe how the program is divided into code and data, not to mention debugging information. Without this tools like `objdump` will not work well, and anti-virus programs will be suspicious of the executable.

The net result is the following set of definitions:

```
--| Asserts that ‘ks: KernelState‘ is a possible initial state
--| for the ELF file ‘elf‘
def initialConfig (elf: string) (ks: nat): wff;
--| Asserts that ‘ks: KernelState‘ can step to ‘ks2: KernelState‘
def ksStep (ks ks2: nat): wff;
--| Asserts that ‘k: KernelState‘ is an exit state
--| with exit code ‘ret e. u32‘
def execExit (k ret: nat): wff;
--| Extracts the input that has been read so far
def ksIn (k: nat): nat;
theorem ksInT: $ k e. KernelState -> ksIn k e. List u8 $;
--| Extracts the output that has been produced so far
def ksOut (k: nat): nat;
theorem ksOutT: $ k e. KernelState -> ksOut k e. List u8 $;
```

This is the starting point for the MMC semantics described in section 3.3.4.

As with the `mm0.{mm0,mm1}` files, The file `x86.mm1`[24] covers all the same definitions as the specification but has to prove their existence. This one is logically simpler than `mm0.mm1` but technically more complex because the inductive types are much larger, so it uses more automation. There are also a few other theorems:

[24] https://github.com/digama0/mm0/blob/master/examples/x86.mm1

- The definitional theorems, mostly trivial or automatic.
- The typing theorems, like:

  ```
  theorem readModRM_T (rex rn rm l: nat):
    $ readModRM rex rn rm l ->
      rn e. Regs /\ rm e. RM /\ l e. List u8 $;
  ```

  While these seem like prime candidates for automation, the statements are not entirely regular, because some parameters are inputs and others are outputs. In this example, `rex` is an input, but its type (which is `rex e. REX`) is not required to establish that the three outputs `rn`, `rm`, `l` are well typed.
- A proof that `decode ast l -> len l <= 12` for all supported instructions.[25]
- Some automation lemmas for proving decode theorems, which are used by the assembler.
- There is another proof which is large enough to get its own file, which we cover in the next section.

[25] This is important because the specification of x86 says that any instruction which is longer than 15 bytes causes an illegal instruction exception even though it would otherwise be legal by the grammar. Luckily we don't target any instructions that can get that long, but if we did we would have to have additional side conditions in the assembler.

2.2.6 *`x86_determ.mm1`: Determinism of the decode function*

We mentioned that the `step k k2` function first nondeterministically reads some bytes starting at the instruction pointer, and then decodes the bytes, and the two steps together are deterministic. To show this, we need to know the following theorem:[26]

[26] https://github.com/digama0/mm0/blob/master/examples/x86_determ.mm1

```
theorem decode_determ2 (ast1 ast2 l l2: nat):
  $ decode ast1 l /\ decode ast2 (l ++ l2) ->
    ast1 = ast2 /\ l2 = 0 $;
```

or the simpler version:

```
theorem decode_determ (ast1 ast2 l: nat):
  $ decode ast1 l /\ decode ast2 l -> ast1 = ast2 $;
```

Now `decode` is by definition a large disjunction of cases like this one:

```
def decodeBinopHiReg (rex ast b l: nat): wff =
$ E. v E. x E. opc E. r (
  splitBits ((1, v) : (1, x) : (2, 0) : (4, 13) : 0) b /\
  readOpcodeModRM rex opc r l /\ opc != 6 /\
  ast = xastBinop (rex_reg 1 opc) (opSizeW rex v)
    (if (true x) (Rm_r r RCX) (Rm_i r 1))) $;
```

Most of this definition is not interesting, but in words, what this definition says is that to decode one particular instruction layout, called `BinopHiReg` here, the opcode byte `b` has the form `1101 10`xv in binary (that is, one of `0xD8`, `0xD9`, `0xDA`, `0xDB`), and then we read the Mod/RM byte(s) to get an opcode extension byte `opc` (which should not be 6, because this is an even more special instruction), and if all that goes well then we can assemble a `xastBinop` instruction where the second argument is either the immediate value 1 or the register `RCX`.

The challenge with proving `decode_determ` directly is that there are two `decode` hypotheses, each of which is a disjunction of about 46 instruction formats, and considering all the cases results in $46^2 = 2116$ possible ways all the instructions can interact. All the off-diagonal entries of this matrix should be impossible, and the on-diagonal entries reduce the problem to determinism lemmas for other definitions like `readOpcodeModRM`. We want to prove this using something closer to $O(n)$ or $O(n \log n)$ work as a function of the number of instructions.

The solution is to leverage the same mechanism the processor itself uses in order to distinguish all the cases, which is to branch on specific bits of the opcode byte. Most instruction formats are distinguished by the first byte; for example we saw `1101 10`xv above, and another instruction format has the form `1111` x`11`v. If we wanted to distinguish these two, we would note that bits 2 and 5 (counting from the right) are `0` in the first format and are `1` in the second format, so they are mutually exclusive. The bits marked x and v are variables so we can't use them for discriminating the patterns.

For the two patterns mentioned, we generate the following theorems:

```
theorem decodeBinopHiReg_bit (rex ast b l: nat):
  $ decodeBinopHiReg rex ast b l -> T. /\
      bit b 7 = 1 /\ bit b 6 = 1 /\ bit b 5 = 0 /\
      bit b 4 = 1 /\ bit b 3 = 0 /\ bit b 2 = 0 $;
theorem decodeHi_bit (rex ast b l: nat):
  $ decodeHi rex ast b l -> T. /\
      bit b 7 = 1 /\ bit b 6 = 1 /\ bit b 5 = 1 /\
      bit b 4 = 1 /\ bit b 2 = 1 /\ bit b 1 = 1 $;
```

These encode all the "fixed bits" of the opcode byte that we can potentially branch on.

So now, rather than doing case disjunction on `decode`, we instead do case disjunction on `bit b i = 0 \/ bit b i = 1` for some i such that all disjuncts satisfy either `bit b i = 0` or `bit b i = 1` (i.e. i is a fixed bit in every pattern) and the two groups are as evenly balanced as possible, and then prove that when `bit b i = 0` all the `bit b i = 1` disjuncts are excluded and vice versa. We are left with two smaller disjuncts and we pick another bit to split the group in half again, and continue until every group contains only one disjunct.[27]

[27] This procedure is not guaranteed to succeed; for example the three 3-bit patterns `01`a, `1`b`0`, c`01` are all mutually disjoint but there is no single bit that is fixed in every pattern. Luckily x86 does not have such cases.

The actual optimal case split tree was calculated offline, so the file contains the tree itself. Everything else is automatically generated, so adding a new instruction only requires adding an extra split to the tree. The resulting proof is essentially $O(n \log n)$ since it is doing a quicksort-like partition and recursion, although n is bounded by 8 since the splitting works at the level of bytes. For some instructions there is a secondary opcode byte and so we actually need to do this process twice.

2.2.7 *`separation_logic.mm1`: Separation logic*

This is a formalization of separation logic as required by the MMC compiler. See section 3.3.1 for the general background for separation logic.

- The heap join / disjoint union operation $h_1 \sqcup h_2 = h$
- The propositional lift $h \models \uparrow p \iff p$
- True and false lifted from propositions: $\bot := \uparrow\bot$, $\top := \uparrow\top$
- The empty heap: $h \models \mathsf{emp} \iff h = \emptyset$
- 'And', 'or', 'implies', 'exists', 'forall' lifted pointwise:

$$\begin{aligned} h \models P \wedge Q &\iff (h \models P) \wedge (h \models Q) \\ h \models P \vee Q &\iff (h \models P) \vee (h \models Q) \\ h \models P \to Q &\iff (h \models P) \to (h \models Q) \\ h \models \exists x \in A.\, P(x) &\iff \exists x \in A.\, h \models P(x) \\ h \models \forall x \in A.\, P(x) &\iff \forall x \in A.\, h \models P(x) \end{aligned}$$

- The empty lift $\uparrow^e p := \uparrow p \wedge \mathsf{emp}$
- The separating conjunction and separating implication:

$$\begin{aligned} h \models P * Q &\iff \exists h_1\, h_2.\, h_1 \sqcup h_2 = h \wedge (h_1 \models P) \wedge (h_2 \models Q) \\ h_1 \models P \mathrel{-\!\!*} Q &\iff \forall h_2\, h.\, h_1 \sqcup h_2 = h \to (h_2 \models P) \to (h \models Q) \end{aligned}$$

- The indexed separating conjunction $\bigast_{x \in A} P(x)$, which does not have a simple expression but satisfies lemmas such as:

$$\bigast_{x \in \emptyset} P(x) = \mathsf{emp}$$

$$\bigast_{x \in \{a\}} P(x) = P(a)$$

$$\bigast_{x \in A \cup B} P(x) = \bigast_{x \in A} P(x) * \bigast_{x \in B} P(x) \qquad (A, B \text{ disjoint})$$

- Strict weakening:[28] $(P \subseteq Q) := \forall h.\ (h \models P) \to (h \models Q)$
- Weakening: $(P \Rightarrow Q) := P \subseteq Q * \top$

[28] This one is notated as $\subseteq$ because it is literally the subset relation on sets of heaps.

2.2.8 `assembler-{old,new}.mm1`: *Assembler theorems (WIP)*

These theorems pertain to the proof producing assembler which is part of the MMC compiler. See section 4.3 for more information.

2.2.9 `compiler-{old,new}.mm1`: *Compiler theorems (WIP)*

These theorems pertain to the correctness proof for functions produced by the MMC compiler. See section 4.4 for more information.

2.2.10 `verifier.mm1`: *The bootstrap theorem*

This is the final goal of the bootstrap. The specification `verifier.mm0` is a simple combination of `mm0.mm0` and `x86.mm0` to assert that there exists (specified constructively via `abstract def`) a program which, when executed according to x86 semantics, will only validate provable MM0 files. In other words, it is a correct verifier.

This theorem is not yet complete, and the file primarily contains explorations in writing a program in MMC with an embedded specification and proof.

3 Metamath C

The requirements of verified programs are somewhat specific and not well addressed by either conventional programming languages such as C, Python, Rust, Haskell etc., or proof assistants like Isabelle, Coq, Lean, etc. On the one hand, for a low-level language we need ways to talk about imperative procedures, pointer manipulation, while loops, and the like, where every construct has a well defined lowering to machine instructions. On the other hand,we need the expressiveness to reason about the program inside an ambient logic, where infinite sets and undecidable predicates are common. These can sometimes be approximated by assertions, which have the advantage of being executable, but these can only be used for dynamic analysis, and in the context of a formal proof of correctness, executability of intermediate assertions is irrelevant and limiting (although it is nice to have when available).

Metamath C (abbreviated MMC) is a language that uses total functions (provably terminating mathematical functions as one would find in HOL or a dependent type theory) for its semantics, combined with inline separation logic through erased "hypothesis variables" for reasoning about heap structures and non-functional components. This is all on top of a C-like structure that is used to provide well defined and predictable lowering to machine code.

MMC code is currently written in what amount to string literals that are passed to the MMC compiler functions in MM1. As a result it inherits the same Scheme-like syntax used in MM1 tactics. (This may change in the future.) MMC has an extensible type system, and it produces MM0 proofs in the back-end. Because types are implemented as "type guards," they have an independent existence as well, and there are primitives for "casting" a variable to a type T given a proof that it has type T.

3.1 *On verified programming*

The problem that MMC is designed to solve is that of writing programs which compile to an executable with full functional correctness guarantees. There are several ways to approach this problem. Here are four representative examples which tackle the problem in different ways:

- Dafny[1] is a "verification-aware programming language." It has a syntax similar to Java, but augmented with `requires` and `ensures` clauses on functions, loop invariants and "lemmas" (functions whose preconditions and postconditions express a theorem). Verification conditions are checked by Boogie[2] and Z3[3]. Dafny code compiles to C#, Java, JavaScript and Go.
- Why3[4] is a "platform for deductive program verification." The user writes programs in a specification and programming language called WhyML, and uses a variety of automated and interactive theorem provers to discharge verification conditions. Programs are extracted to OCaml.
- CompCert C[5] is a compiler for the C language, which produces machine code for PowerPC, Arm, x86, and RISC-V. The correctness of the compiler with respect to the C specification down to assembly is formally verified in Coq.
- CakeML[6] is a compiler for a subset of Standard ML, which has been verified in HOL4 relative to a specification of machine code.

[1] https://dafny-lang.github.io/dafny/

[2] https://github.com/boogie-org/boogie

[3] https://github.com/Z3Prover/z3

[4] http://why3.lri.fr/

[5] https://compcert.org/compcert-C.html

[6] https://cakeml.org/

In the following sections, we will look at how MMC differs from each of them looking at different aspects of their operation.

The front end

Dafny and Why3 are both targeted at the front end: they provide a language with a syntax for expressing functional correctness properties at the level of the code itself. CompCert, however, is a C compiler: the input language is exactly C, which makes it essentially unsuitable for functional correctness proofs because C has no provision for design by contract or even a strict type system. Similarly CakeML is an ML compiler, and while ML has a strict type system it is not a proof language, and the types are not expressive enough for functional correctness.

Nevertheless, it is possible to overcome this modeling limitation, and both CompCert and CakeML have been used for verified programming. The key observation is that the compiler correctness proof takes place in a proof assistant (Coq for CompCert, HOL4 for CakeML), which means the semantics of the source language is expressed in this language. So rather than using the front end directly, we can write proofs in the proof assistant that a specific program syntax has the desired behavior, and then the compiler correctness proof ensures that the compiler will transform this program text into machine code with the same behavior. The Verified Software Toolchain[7] is a framework for doing this in Coq, and CakeML also provides this functionality when operated through HOL4.

[7] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg

However, this generally comes at a loss of front end quality, because Coq and HOL4 are general purpose proof assistants, and embedded programming languages and the specifications being proved get second-class notation.

Metamath C is embedded in the general purpose proof assistant MM1, but it is closer to the Dafny / Why3 approach in that the program syntax and type system includes embedded specifications.

The back end

The back end of a verified programming language is the lowest level about which theorems are proved. In this sense Dafny and Why3 are only skin-deep: the verification stops at the Dafny / WhyML language semantics, which are extracted into various unverified programming languages.

CompCert's correctness proof goes to assembly language, which is as far as it can reasonably go because of mismatch between the CompCert memory model and the memory model of the target architectures. (The company behind CompCert, AbsInt, also sells a separate tool called Valex for verifying assembly outputs from the CompCert compiler on some targets.) CakeML goes all the way to machine code, although it still takes a dependency on an assembler in order to handle linking to some tools from the C runtime library.

Metamath C produces ELF[8] binary files directly, and the specification of ELF describes how the files are loaded in memory and executed (on x86). There is no dependency on the C runtime library because the generated code makes system calls directly to the kernel.[9]

[8] Executable and Linker Format, the standard executable format on Linux

[9] This is only really possible on Linux; most operating systems do not have a stable syscall interface so compilers are generally advised to make use of the platform's `libc` implementation. To avoid the complication of dynamic linking this is not currently being done, but it would not be fundamentally harder to specify the behavior of the operating system interface or chosen functions from `libc`.

In principle, one can go even further than the ISA, by verifying the hardware itself. However, this has the drawback that the most users will not have the verified hardware, and this is unlikely to change until open source processors become more mainstream. In our allegory from the introduction, this amounts to Penny sending Victor a custom-built processor, and with hardware access comes a much higher risk that the processor is compromised (either because Penny is less qualified to produce a reliable processor than Intel, or because Penny has deliberately added a hardware backdoor in order to cheat). The social process leading to hardware reliability is strong, and it is difficult for a verified manufacturing process to overcome this head-start.

The proof mechanism

Different tools have different dispositions toward the proof of the compiler's inner workings, or the way in which the verified programming language avoids adding itself to the trust base of the verified program itself (if it does so at all).

Dafny and Why3 both rely on SAT solvers in the back end. Dafny lowers its programs to the Boogie intermediate language, which is then translated into SMT problems for Z3. Z3 does not currently produce proof objects which are suitable for reconstruction, so for the most part it must be trusted here, and the translation must also be trusted because the verification conditions are not knitted into an overall proof, although this is in principle possible.

Why3 has a similar story, although it uses a wide variety of interactive and automated provers in the back end, which, although convenient for the user, means that the trust base of the verified program involves the correctness of dozens of unverified tools: this kind of diversification is a double-edged sword for those concerned with end-to-end correctness.

As mentioned, CompCert is proven correct in Coq, which is in some sense the best possible result, although it is important to clarify what this means exactly. The proof in Coq shows that a certain function in dependent type theory computes a mapping from source AST to assembly syntax, which preserves execution behavior in a certain sense. The function must then be run on a given source AST (for the program the user is interested in) to produce a result, which may either be performed in the Coq proof assistant directly using kernel reduction, or the compiler can be extracted into a standalone executable which produces binaries. Some aspects of the proof, like the assembly-level verification, are delayed to this phase.

CakeML also uses this technique: most of the compiler is verified once and for all, which proves the correctness of a certain HOL function, and then the `EVAL` function is used to evaluate the HOL function and produce a proof of evaluation, which is then composed with the CakeML correctness proof to produce a proof of correctness of the target machine code.

Metamath C takes a different approach than either Dafny/Why3 or CompCert/CakeML here, although it is more similar to the CompCert approach. The critical observation is that proofs at "compiler run-time" are fundamentally more flexible than compile-time proofs.

Rather than proving the compiler itself correct, the compiler is interpreted as a "correctness proof generator," which is allowed to be fallible in a number of ways. Like tactics in Coq/Lean/Isabelle, the program is not verified, and it is not part of the trust base because the proofs it generates are passed to a simple and general purpose proof verifier.

This technique is also known as translation validation (TV) or proof-carrying code (PCC). The big advantage is that it allows us to freely incorporate unverified components like a register allocator, as long as the results of these components can be checked for correctness. Checking a solution is never harder than producing a solution, and often significantly easier (this roughly corresponds to the difference between the P and NP complexity classes). So this is a good way to get both good performance, as well as aiming more directly to the purpose of the compiler, which is to produce a proof that some machine code performs according to the user's specification.

Summary

The Metamath C compiler receives inputs in the MMC language, where the user writes:

- the specification (what the program should do);
- the code (how the program should do it);
- and the proofs (how the program's actions achieve its specification).

All this is at the level of the code itself – there is no reference to register values in the specification or the code (although this is a potential extension, analogous to `asm()` blocks in `gcc`). The compiler assists at this stage by producing type errors and prompts to help the user complete the proofs.

Once the proofs are type correct according to the MMC language, the compiler takes over, lowering the proofs through each intermediate language, all the way down to machine code. The lowest level proof language is then "code-generated" into a block of bytes and a MM0 proof object about that block of bytes, which is the final output.

The MM0 proof asserts that the block of bytes satisfies the translation of the high level MMC specification. The translation is trivial for pure propositions, so for example if the program produces a proof of $R(4) = 15$ then the final theorem would say:

> If the block of bytes P is interpreted as an ELF executable, loaded into memory, and executed according to the x86 specification, then it will not perform any undefined or unspecified actions and it will halt in a finite number of steps,[10] and if it halts with exit code 0 then $R(4) = 15$.

[10] The compiler is currently geared toward producing "total correctness" proofs, i.e. termination is checked via loop annotations. There is planned support for both total and partial correctness modes, which the user can select as appropriate.

Although the proof is natively written in MM0, it targets a relatively simple and portable logic (Peano Arithmetic) which means that the proof can also be translated to a wide variety of proof systems (see section 5.1.1).

3.2 *A tour of Metamath C*

MMC is a fairly complex language with many different interacting concepts, taking inspiration from imperative programming languages (especially C and Rust), dependent type theory, and separation logic, which can be overwhelming. So we will begin with some examples and introduce new concepts incrementally.

An aside on syntax: The MMC syntax is not yet finalized, and the current MMC parser is embedded in the MM1 programming language and as such uses lisp-style s-expressions. In this book we will instead use a concrete syntax which is closest to Rust.

3.2.1 *Procedures*

The top level syntax of a MMC program is similar to C: a list of function declarations, type declarations and global variable declarations. There are two kinds of functions, called "functions" and "procedures" and introduced by the `func` and `proc` keywords respectively.

A *procedure* is a piece of code that receives zero or more parameters and has zero or more return values. It is permitted to have *effects* like I/O behavior, but it is required to terminate in finitely many steps (see section 3.2.6). Here is an example of a procedure:

```
proc add2(x: u32, y: u32): u32 {
  return (x + y) as u32;
}
```

This function adds two 32 bit unsigned integers and returns the wrapped result. It is the equivalent of the following C definition:

```
unsigned int add2(unsigned int x, unsigned int y) {
  return x + y;
}
```

Similar to Rust and other ML-inspired languages, blocks are expressions and the last expression in a block is the return value, so the following procedure is equivalent:

```
proc add2(x: u32, y: u32): u32 { (x + y) as u32 }
```

Procedures can return multiple values:

```
proc numbers(): u32, u32 { 0, 1 }
```

Both function arguments and returns can depend on prior values:

```
proc deptypes(x: u32, _: x = 0): y: u32, sn((x + y) as u32) {
  1, sn((x + 1) as u32)
}
```

This is a function which accepts two parameters, a 32 bit integer x and an unused proof that $x = 0$, and returns two values: a value y (which we choose to be 1), and value which is equal to $x + y$. This makes use of the type `sn(`a`)` of values equal to a, and the expression `sn(`a`): sn(`a`)` which is its canonical constructor.

The `sn` type is a simple way to express precise preconditions / postconditions for a procedure. We will discuss looser and more logically complex preconditions / postconditions in section 3.2.12.

3.2.2 *Functions*

A procedure is *opaque*, which is to say the only externally visible properties of the procedure are given by its type signature. (As we have seen, dependent types allow for expressing precise preconditions and postconditions through the type signature, so this is not a significant limitation.) However, because procedures are not (necessarily) pure, they cannot appear inside types and pure expressions.

To address this issue, *functions* can be used instead. A function has the same syntax as a procedure, but uses the `func` keyword instead. Functions are compiled to both logical functions and machine code, and they can be used in later pure expressions.

```
func add(x: nat, y: nat): nat { x + y }
proc example(): sn(add(2, 2)) { sn(4) }
```

Currently, functions are mostly unsupported, but they should be able to support all language features except for non-termination and side-effecting operations.

3.2.3 *Variables*

Basic straight-line code works as one would expect, except that the type context (see section 3.3.3) keeps track of the values that variables have been assigned:

```
proc statements(): sn(1: u8) {
  let x: u8 := 1;
  // x: u8 := 1
  let y := x;
  // x: u8 := 1, y: u8 := 1
  sn(y)
}
```

(The comments after each line indicate the state of the type context after each line. Since the context knows that `y := 1`, the last line typechecks: `sn(y)` is equal to `sn(1)`.

Variables can also be shadowed, and overwritten:

```
proc statements(): sn(1) {
  let x := 1;
  // x: nat := 1
  let x := x + 1;
  // x*: nat := 1, x: nat := 1 + 1
  x <- x * 2;
  // x**: nat := 1, x*: nat := 1 + 1, x := (1 + 1) * 2
  sn(x)
}
```

The `x**` notation denotes a shadowed variable; shadowed variables are not accessible by name but still exist in the context because they may be used in the types of other variables. In straight line code there is no difference between mutating a variable and shadowing it, but it makes a difference in more complex control flow (see section 3.2.5).

3.2.4 *Tuples and destructuring*

A tuple type `(A, B)` represents a pair of values of types `A` and `B`. Tuples can be created and destructured in let binders:

```
proc tuples(): sn(1), sn(2) {
  let x: (nat, nat) := (1, 2);
  let (one, two) := x;
  sn(one), sn(two)
}
```

This program constructs a pair consisting of 1 and 2 and puts it in variable `x`, then destructures the pair into two variables `one` and `two`, and then returns them separately. The use of `sn` shows that the typechecker knows that `one := 1` and `two := 2` after this sequence of operations. Changing the `sn(2)` to `sn(5)` would result in a compiler error.

Tuples can also be dependent, and the fields of a tuple can be referred to by index or by name:

```
proc tuples2(): x: nat, sn(x + 1) {
  let s: (a: nat, b: nat, h: sn(a + b)) := (1, 1, sn(2));
  s.a, s.h
}
```

Procedures with multiple returns are also destructured at the call site.

```
proc send_many(): x: nat, sn(x) { 1, sn(1) }
proc recv_many() {
  let x, y := send_many();
```

```
    // x: nat, y: sn(x)
  }
```

3.2.5 *Control flow*

The usual control flow mechanisms, `if` and `while` are available, but with a dependent-type spin since we want to have access to information about the result of conditional expressions.

```
  proc if_statement(x: nat) {
    if h: x < 10 {
      // x: nat, h: x < 10
    } else {
      // x: nat, h: ~(x < 10)
    }
  }
```

If a value is assigned differently in two branches, its value is lost:

```
  proc if_statement2(b: bool) {
    let x := 1;
    let y := 1;
    // b: bool, x: nat := 1, y: nat := 1
    if b {
      // b: bool, x := 1, y := 1
      let x := 2;
      // b: bool, x* := 1, y := 1, x := 2
      y <- 2;
      // b: bool, x* := 1, y* := 1, x := 2, y := 2
    } else {
      // b: bool, x := 1, y := 1
    }
    // b: bool, x: nat := 1, y: nat
  }
```

Here we can also see the difference between shadowing and mutation. The variable `x` was shadowed in the if branch, so a new variable `x` was introduced and discarded, and after the `if` the old variable still exists and still has its old value. On the other hand, `y` was reassigned in the `if` statement, which has the same semantics as a reassignment in C. As a result, after the `if` statement `y` could have either value 1 or 2, so the fact `y := 1` is cleared from the type context and `y` retains only its type.

While loops also provide access to the positive fact inside the loop and the negative fact after it:

```
  proc while_loop() {
    let b := true;
    let h2 := while h: b {
      // h: b
      b <- false;
    };
    // h2: ~b
  }
```

Loops also support `break` and `continue`. In fact, `continue` is required at the end of loop bodies when termination proofs are used (see section 3.2.6).

The other mechanism for complex control flow is `label`, which is used to set up one or more mutually tail-recursive functions:

```
  proc fact(x: nat): nat {
    let result: nat := {
      label rec(x: nat, y: nat) {
        if x = 0 {
          finish(y)
        } else {
          rec(x - 1, y * x)
        }
      }
      label finish(y: nat) {
        y // assigns y to the 'let result' at the start
      }
      rec(x, 1)
    };
    result
  }
```

The use of `finish` is gratuitous here, we use it only to demonstrate that `label`s can call each other or themselves freely. These are not real functions in the sense that they do not get their own call frame, and that means that they can only be used tail-recursively: using `2 * finish(y)` in place of `finish(y)` in this example would be incorrect.

3.2.6 Termination

The MMC compiler is designed to be instantiated in one of two modes: *total* or *partial* correctness.

- In total correctness mode the final theorem about the code asserts that the program halts in a finite number of operations, it does not perform any undefined behaviors, and if it does not fail then a certain predicate of interest (defined by the return type of the `main()` function) holds.
- In partial correctness mode the final theorem asserts that the program does not perform any undefined behaviors, and *if* the program halts and does not fail then the predicate of interest holds.

The key difference, of course, is that halting is proven in the first formulation and assumed in the second formulation. For many applications partial correctness is sufficient: in particular, running the code is a requirement to observe the lack of failure,[11] and running the code also provides observational evidence of halting, so in this sense partial correctness is what we need to complement the facts acquired by running the code.

[11] If we have a program that performs some exhaustive search to establish a theorem, then even though we can prove that if the search succeeds then the theorem is true, this does not establish the theorem until we actually run the search and notice that it finds no counterexamples.

On the other hand, for logical functions we do need that the program halts for it to satisfy its definition, and it is also useful to have as a sanity check that the program is really working as intended (since non-termination can be used to provide well typed but useless programs).

In total correctness mode, all looping constructs (`while`, `label`, and recursive functions) accept a *variant* expression, which increases or decreases toward a bound.[12] For example, a desugared bounded for loop would look like this:

[12] Note that this only allows loops with order type at most ω. It is possible to do more complex recursion using nested loops, and the syntax may in the future be expanded to support larger ordinal loops, but because the proofs are conducted in Peano Arithmetic, the compiler proof can't support loops of order type ϵ_0 or higher.
In practice, ω recursion covers the vast majority of patterns in real programs, and by adding an explicit "fuel" parameter even general recursion can be handled, replacing the possibility of non-termination with the possibility of failure by timeout.

```
proc for_loop() {
  let x := 0;
  while h: x < 10, variant(x < 10 := h) {
    x <- x + 1;
    // x*: nat, x := x* + 1
    continue variant(p) // p: x* < x
  }
}
```

The variant declaration `variant(`e `<` n `:=` p`)` indicates that the expression e is increasing up to n, and p proves that inside the loop just after the condition, $e < n$ holds. It is complemented with the `variant(`p`)` at the end of the loop, which proves that `x* < x`, that is, the old value of the variant (in this case `x`) is less than the new value. Together, this ensures that the loop can only be entered at most $n - a$ times (where a is the initial value of the variant, in this case 0).

3.2.7 *Integral types and operations*

MMC supports several integral types:

$$\tau ::= \texttt{u8} \mid \texttt{u16} \mid \texttt{u32} \mid \texttt{u64} \mid \texttt{nat} \mid \texttt{i8} \mid \texttt{i16} \mid \texttt{i32} \mid \texttt{i64} \mid \texttt{int} \mid \dots$$

The types `u`N are the same as `u`N from Rust or `uint`N`_t` from C: they are unsigned N-bit integers, capable of storing any integer in the range $0, \dots, 2^N - 1$. Similarly, `i`N corresponds to `i`N in Rust and `int`N`_t` in C, the type of signed N-bit integers with the range $-2^{N-1}, \dots, 2^{N-1} - 1$.

The type `nat`, already used many times in previous examples, represents the type of *unbounded* natural numbers, that is $0, 1, \dots$. This is the "native" integer type of the language, in the sense that numeric literals

Operation	closed in uN	iN	nat	int	Meaning
`x + y`	−	−	+	+	addition
`x - y`	−	−	−	+	subtraction
`-x`	−	−	−	+	negation
`x * y`	−	−	+	+	multiplication
`x ^ y`	−	−	+	+	exponentiation
`div(x, y)`	+	−	+	+	floored division ($\lfloor x/y \rfloor$)[14]
`mod(x, y)`	+	+	+	+	modulo[14]
`max(x, y)`	+	+	+	+	maximum
`min(x, y)`	+	+	+	+	minimum
`band(x, y)`	+	+	+	+	bitwise AND
`bor(x, y)`	+	+	+	+	bitwise OR
`bxor(x, y)`	+	+	+	+	bitwise XOR
`bnot(x)`	+	+	−	+	bitwise NOT
`shl(x, y)`	−	−	+	+	left shift ($x \cdot 2^y$)
`shr(x, y)`	+	+	+	+	right shift ($\lfloor x/2^y \rfloor$)

Figure 3.1: Integer operations. An operation is "closed in uN" if the underlying mathematical operation yields values in uN when the inputs are in uN. In this case applying the operation will yield values in the type; otherwise it will be promoted to the next smallest type which is closed for the operation (so for example adding two `u32` values yields `nat`).

[14] Division by zero yields 0 (and `mod(x, 0) = x`), but there is also a three argument form `div(x, y, h)` that accepts a proof that $y \neq 0$.

have this type by default and most numeric operations return values of this type, but because MMC does not have an implementation of arbitrary-precision integers (a.k.a. bignums),[13] `nat` can only be used in limited ways in computationally relevant positions (see section 3.2.9). Similarly, `int` is the type of unbounded positive or negative integers.

[13] This is partially a deliberate decision to avoid unexpected and unpredictable compilation results, and partially simply because verified bignum implementations are a large project on their own. The language might gain support for this in the future.

A key property of MMC is that algebraic operations result in their exact untruncated values, which is why they often land in `nat` instead of a fixed width type, and additional actions must be taken to reduce the value to a fixed width type so that it can be implemented in the computer. For example, it is possible to use `nat` as an intermediate to calculate a wrapped result, like with the `add2()` example from section 3.2.1:

```
proc add_as(x: u32, y: u32): sn((x + y) as u32) {
  let z: nat := x + y;
  sn(z as u32)
}
```

It is also possible to avoid wrapping and instead prove that the value is in the target type:

```
proc add_cast(x: u32, y: u32, h: x + y < 2^32): sn(x + y: u32) {
  let z: u32 := cast(x + y, h);
  sn(z)
}
```

Finally, we can map it into the type by crashing on overflow (see section 3.2.8):

```
proc add_assert(x: u32, y: u32): sn(x + y: u32) {
  let h: x + y < 2^32 := assert(x + y < 2^32);
  let z: u32 := cast(x + y, h);
  sn(z)
}
```

The available integer operations are shown in Figure 3.1.

3.2.8 *Failure is always an option*

As mentioned previously in section 3.2.6, the correctness theorem for programs in MMC includes the possibility of failure. More specifically, a "failure" here means user-visible abnormal termination, and in the current implementation this means exiting with a non-zero exit code.[15]

It is important to note that it is unrealistic to have a theorem that asserts that failure cannot occur, because there are unfortunately many ways for a program to terminate abnormally that are not under control of the running process. For example the operating system can choose not to provide enough free memory for the program to conduct its activities,[16] or the program could get a `SIGINT` signal from the user pressing Ctrl-C at the console (which can be caught) or a `SIGKILL` signal from the process manager or OOM killer (which can't). Or the computer could simply lose power.

The current design of MMC exploits the fact that the correctness theorem allows for the possibility of failure to make the programming language more convenient. Specifically, there is an `assert(b): b` expression in the language which returns a proof that any executable boolean expression `b` is true. It is useful to supply facts to operations that have side conditions, like array indexing or casting such as in the `add_assert()` example from section 3.2.7. Its implementation is effectively `if h: b { h } else { fail() }` where `fail()` is a compiler-implemented function that crashes the process.[17]

One of the most important reasons to allow for the possibility of failure in any compiler, not just a proof producing one, is to make it possible to "ignore" resource limits in verification, which hugely simplifies verification tasks. For example, Coq and Lean have a `nat` type which purports to be a model of natural numbers; you can even prove it is infinite and has the properties you would expect of an unbounded natural number type. But if you try to evaluate the closed, well typed term $2^{2^{64}}$: `nat` the implementation limits will certainly bleed through – this will fail with an error condition not treated by the logic.

A potential future direction for the language would be to allow the user to prove that code does not fail except in force majeure situations, by banning the explicit or implicit use of `assert` and explicitly

[15] Some shells will indicate this automatically, and in vanilla Bash you can query the `$?` variable to see if the previous command terminated abnormally.

[16] It is even possible for the user or operating system to set the stack space so low that the environment variables don't fit on stack, which means that the program runs out of stack before it reaches the `_start` procedure (the program entry point), so even a completely trivial program that uses zero stack and exits immediately could still run out of stack space!

[17] The current implementation is simply a call to the x86 `ud2` instruction, which is a special instruction which is "defined to be undefined:" it causes the processor to throw an illegal instruction exception which results in the program crashing with the `SIGILL` signal, which appears as exit code 132 in bash, possibly accompanied by the text "`Illegal instruction (core dumped)`".
This is obviously not a very user-friendly way to abort the process; future work includes replacing this with a regular function which prints a panic message and exits normally with non-zero exit code.

bounding stack and memory usage.

3.2.9 *Ghost variables*

We say that a variable, expression or piece of code is *ghost*, or *computationally irrelevant*, if it does not exist in the generated code: it produces no output. From the perspective of the user, ghost code is executed just like other code, ghost variables can be mutated, stored in structures and passed around like regular variables, but any operations on ghost variables are eliminated by the compiler.

Variables can be marked ghost at the point of declaration:

```
let x: ghost u8 := 2;
let ghost x: u8 := 2; // same as above
```

The compiler tracks usage of ghost variables to ensure that they do not enter the data flow, but they can be used to perform computations that are needed for the proof but not for the program. They can be used in types (like the `n` in the type `[T; n]` of arrays of length `n`) as well as in loop variants and invariants.

For the most part, the MMC compiler will automatically make ghost determinations about variables: any variable that does not contribute to the return value or some IO operation will be made `ghost`. (This is analogous to dead code elimination in standard compilers.)[18] However, there are still two legitimate uses for an explicit `ghost` marking:

[18] In MMC there are actually two interpretations of "dead code elimination:" code can either be removed from the generated code (by marking more things as `ghost`), or they can be removed from the code and the proof (by removing them entirely).

Currently the compiler makes no attempt to do the second kind of dead code elimination, but it is fundamentally the same as ghost inference: anything that does not appear in the types or invariants in the return can be removed.

- The user may wish to document that some code is not executed and have the compiler uphold the property. Without the `ghost` marking, a future change could potentially cause the code to be executed, leading to either unexpectedly poor performance (if it is a reference implementation which is not intended for execution) or a ghost error somewhere else (because the ghost code is doing something that definitely can't be compiled.)
- In function signatures, it can be impossible to determine locally whether a certain variable is intended to be ghost or not. For example:

  ```
  proc id(x: u8): u8 { x }
  ```

  Is the variable x here ghost or not? That is, does the code generated from this function actually take a single value from the argument register and copy it to the return register, or does it take no arguments and do nothing? There is no way to tell by looking at this code alone. So MMC defaults to assuming that the variables are computationally relevant unless you explicitly mark variables as `ghost` in the signature:

```
proc id(x: ghost u8): ghost u8 { x }
```

The main utility of `ghost` markings is on types that would otherwise not already be ghost, like `u64`. Many types are explicitly zero-sized and so `ghost` makes no difference, like `()` (the unit type, returned by expressions that return "nothing") or `x = 1`. (Recall that `x = 1` is a boolean expression but also a type. For example, in `if h: x = 1 { ... }`, `h` is a variable whose type is `x = 1`, and the existence of this fact in the context allows us to know that `x = 1` is true inside the scope of the `if`.)

3.2.10 *Casting, type punning, and truncation*

Type-casting is the conversion of values from one type to another. This may entail a change of the logical value, the bit representation, or both, and the classic "C cast" operation `(T)x` is ambiguous. In MMC value-preserving type casts are performed with `cast(x)` or coercion (i.e. using a value of type `X` where a value of type `Y` is expected), bit-preserving and value-changing casts are performed with the `pun(x)` operation, and other kinds of casting (like truncating conversion) is performed with the `x as T` operation.

Conversion	`cast`	`pun`	`as`	coe.	Meaning	Logical value
`u`$\dot{M}$ → `u`$\dot{N}$, $\dot{M} \le \dot{N}$	$+$	$=$	$+$	$+$	widening (zero extend)	x
`i`$\dot{M}$ → `i`$\dot{N}$, $\dot{M} \le \dot{N}$	$+$	$=$	$+$	$+$	widening (sign extend)	x
`u`M → `i`$\dot{N}$, $M < \dot{N}$	$+$	$-$	$+$	$+$	widening (zero extend)	x
`i`M → `u`N, $M \le N$	$\vdash$	$=$	$+$	$-$	sign extend + wrapping	x `as` `u`N
`{u,i}`$\dot{M}$ → `u`N, $\dot{M} \ge N$	$\vdash$	$=$	$+$	$-$	truncation + wrapping	x `as` `u`N
`{u,i}`$\dot{M}$ → `i`N, $\dot{M} \ge N$	$\vdash$	$=$	$+$	$-$	truncation + wrapping	x `as` `i`N
`{u,i}`$\dot{M}$ → `bool`	$\vdash$	$=, \vdash$	$+$	$-$	is boolean / nonzero	$x \neq 0$
`bool` → `{u,i}`$\dot{M}$	$+$	$=$	$+$	$-$	convert to $\{0,1\}$	$\text{if}(x,1,0)$
`&sn x`, `&T`, `own T` → `u64`	$+$	$+$	$+$	$-$	pointer value	x
`X` → `Y`	$\vdash$	$=, \vdash$	$-$	$-$	treat as `Y` given proof	varies

Figure 3.2: Possible kinds of conversion. The entries for each kind of operator indicate whether the operation is allowed ($+$), disallowed ($-$), or allowed with a proof side condition ($\vdash$). The `pun` cases are only allowed ($=$) if the source and target have the same size.
A dot on a size variable means that ∞ is a valid option, i.e. `u`$\dot{N}$ includes both `u16` and `u`$\infty \equiv$ `nat`.

Figure 3.2 shows all the possible combinations and which of the conversion operators can be used.

`cast` is used for downcasting or upcasting integers, preserving the logical value.[19] It has the syntax `cast(x)` or `cast(x, h)`, and it takes a proof (indicated with $\vdash$ in the table) in some situations:

- If converting numeric types `X` → `Y` such that $X \not\subseteq Y$ as subsets of $\mathbb{Z}$, then `h` should be a proof that `x: Y`.
- If converting `{u,i}`$\dot{M}$ → `bool`, then `h` should be a proof that `x: bool`, that is, $x \in \{0,1\}$.

[19] This is C implicit conversion, or Rust `into()`.

- If converting `X` → `Y` in general, `h` should be a proof that `x: Y`, but the conversion may not always be possible if the compiler does not know how to remap the bit representation from `X` to `Y`.

For truncation and generally non-value-preserving type casts, we use the `x as Y` function.[20] This function does not take a proof, and can convert freely between numeric types in all cases except `int` → `nat` (where a truncation and wrapping conversion is not meaningful). Because this is generally a value-changing operation, this appears as an actual function applied to the value. The x `as {u,i}`N function in the logical value column is defined as:

$$x \texttt{ as } \texttt{u}N = x \bmod 2^N$$
$$x \texttt{ as } \texttt{i}N = ((x + 2^{N-1}) \bmod 2^N) - 2^{N-1}$$

[20] This is C explicit conversion, or Rust `x as Y`.

Type punning is the practice of reinterpreting a value without changing its bit-representation. For example, the type `u8` represents numbers $0, \dots, 255$ using a single byte of memory, while `i8` represents numbers $-128, \dots, 127$ also using a single byte of memory in two's complement. The representation of the number `200: u8` uses the bits `11001000`, while the representation of `-56: i8` is also `11001000`. Therefore, if we have the number `200: u8` and reinterpret the bits at type `i8`, we find that we have the value `-56` even though we have not actually run any code or changed memory in any way.

In MMC the `pun(x, h)` operation implements this kind of transformation.[21] It significantly overlaps with the `cast` and `as` functions, but it is more restrictive in that the source and target types need to have the same size (number of bytes in memory), and it is guaranteed to be a hardware no-op.

[21] This is the C++ `bit_cast()` operation, or Rust `transmute()`.

When the target type has nontrivial invariants (for example `bool` requires that the stored byte is 0 or 1), these must be proved in the provided proof argument. In particular, this is the method of choice for restoring the type of a value that was taken by the `typeof` operator (see section 3.2.12). For example, if `x: u64` and we provide a proof that `x` points to a value of type `T`, we can use `pun(x, h)` to construct an `own T` at the location of `x`.

3.2.11 *The empty type*

As mentioned in section 3.2.8, boolean values are also types, the type of proofs that the value is equal to `true`. So `false` is also a type – an empty type (since `false` is not equal to `true`). C does not have empty types, but Rust has the `!` ("never") type which serves this purpose, and many dependently typed languages also have an empty type of some kind.

The empty type, also known as the "bottom" type, is useful to encode in the type system that a certain position is statically unreachable, and a function that returns an empty type is a function that never returns. Logically, the main property of the empty type is that it has the "principle of explosion:" from `false` we can derive a value of any type. This is encoded in MMC as the `unreachable(`h`: false)` operator, which receives a proof of `false` in the current context and "cancels the current branch" of the code. This is equivalent to the `unreachable_unchecked()` operator in Rust or any immediate undefined behavior in C like `*NULL`. This has the effect of removing any branches that lead to the current code path, or possibly deleting the entire containing function because it can't be called. As with the earlier discussion on "ghost code" (see section 3.2.9), these blocks of code aren't really deleted completely, they are merely made "ghost," so they still exist for typechecking purposes but no code is emitted for them. For example:

```
// natural numbers are >= 0
// ignore the implementation of this theorem for now
func nat_nonneg(x: nat, h: ~(x >= 0)): false { ... }
proc more_code(x: nat, h: x >= 0) { ... }
proc deleted_branch(x: u8) {
  if h: x >= 0 {
    more_code(x, h);
  } else {
    let contradiction: false := nat_nonneg(x, h);
    // at this point we must be in dead code
    unreachable(contradiction)
  }
}
```

We have a branch on a condition `x >= 0`, but in the `else` branch we do some proof work and deduce that in fact the situation is impossible, at which point we call `unreachable` to tell the compiler about this. The compiler transforms this into the following:

```
proc deleted_branch(x: u8) {
  let h: x >= 0 := by_contra(λ h: ~(x >= 0). {
    let contradiction: false := nat_nonneg(x, h);
    contradiction
  });
  more_code(x, h);
}
```

The `by_contra(λ h. e)` here is not surface syntax (MMC does not currently have lambdas / closures either), but is intended to represent a proof method for $(\neg p \to \bot) \to p$ as a lambda term. Nothing in the block is executed, it is simply encoding a proof of $\neg p \to \bot$ that has been extracted from the code. As a result, the actual generated code just appears as a call to `more_code(x)` after `h` is removed and no branch is performed.

There are several constructs which return a proof of `false` (or rather, a value of any expected type, including `false`) as a way to signal that they perform non-local control flow and anything after the expression is dead:

- A `return` expression exits the function, so anything after a return is dead.
- `break` and `continue` exit the current loop scope.
- Jumping to a `label` returns `false`, even though label blocks typically evaluate to a value of some other type, because they are forced tail-calls: the label does not return to the caller but instead exits the containing block.

```
proc label_test(b: bool): u8 { // <- block returns u8
  label foo() { // <- label cannot specify return
    if b { return 1; } // return exits label_test(), not foo()
    2 // <- label block returns a value of type u8
  }
  label bar() {
    let h: false := foo(); // <- calling a label returns false
    let x := 2 + 2; // this is dead code
    unreachable(h) // <- satisfy the typechecker
  }
  bar() // <- this call also returns any type (in this case u8)
  // anything here is dead code, bar exits the block
}
```

- A `while true { ... }` without internal `break` returns the negation of the loop condition, so ∼`true` which is `false`. Infinite loops do not proceed past the loop.
- `assert(false)` also returns `false`, because it crashes the program so anything afterward is not executed.
- `unreachable(`h`)` also returns `false`. It might appear silly to produce `false` if we already have a proof of `false`, but this also has the effect of telling the compiler about this false proposition so that it doesn't have to search the context for false things all the time.[22]

[22] Most production compilers do actually watch the context for construction of uninhabited types, both because they want to catch the unreachability as early as possible, and also because they want to report dead code warnings. We will put this down as another avenue for future work.

3.2.12 *Separation logic types*

We have managed to avoid delving too deeply into separation logic in the types mentioned so far, but the underlying proof methodology is based on separation logic, and it is useful for many verification tasks to have explicit access to separation logic connectives in the assertion language when dealing more explicitly with "resources" in the machine state.

Type	Concrete syntax	Typehood predicate $\boxed{a : -}$	Meaning
$\exists x : \tau_1, \tau_2(x)$	`(ex x:` τ_1`,` $\tau_2(x)$`)`	$\exists x : \tau_1, \boxed{a : \tau_2(x)}$	Existential quantification
$\forall x : \tau_1, \tau_2(x)$	`all x:` τ_1`.` $\tau_2(x)$	$\forall x : \tau_1, \boxed{a : \tau_2(x)}$	Universal quantification
$\tau_1 \to \tau_2$	τ_1 `->` τ_2	$\boxed{a : \tau_1} \to \boxed{a : \tau_2}$	Non-separating implication
$\tau_1 \mathrel{-\!\!*} \tau_2$	τ_1 `-*` τ_2	$\boxed{a : \tau_1} \mathrel{-\!\!*} \boxed{a : \tau_1}$	Separating imp. (magic wand)
$\tau_1 \wedge \tau_2$	τ_1 `&&` τ_2	$\boxed{a : \tau_1} \wedge \boxed{a : \tau_2}$	Non-separating conjunction
$\tau_1 * \tau_2$	(τ_1, τ_2)	$\boxed{a.0 : \tau_1} * \boxed{a.1 : \tau_2}$	Separating conjunction
$\tau_1 \vee \tau_2$	τ_1 `||` τ_2	$\boxed{a : \tau_1} \vee \boxed{a : \tau_2}$	Disjunction
$\neg\tau$	$\sim\tau_1$	$\neg\boxed{a : \tau}$	Negation
$\ell \mapsto v$	ℓ `|->` v	$\ell \mapsto v$	Points-to assertion
$\boxed{e : \tau}$	`[`e`:` τ`]`	$\boxed{e : \tau}$	Typing assertion
$\lvert\tau\rvert$	`moved(`τ`)`	$\left\lvert\boxed{a : \tau}\right\rvert$	Persistent core of τ

Figure 3.3: Separation logic types. The "Typehood predicate" column shows the result of evaluating $\boxed{a : \tau}$ for the type τ shown in the first column.

A *type* is a function that maps values to separating propositions over machine states. That is, it is a true-or-false statement that refers to portions of the machine state (registers and memory). This is a very low level view, but it has the advantage that because it is so general, users can define types of arbitrary complexity, containing invariants and ownership semantics. Types also contain a size and an alignment, although for the x86 instantiation of MMC all types have alignment 1.

So the interpretation of an expression e is a "value," which we can model as natural numbers, and types τ are modeled as functions mapping values to separating propositions, so $e : \tau$ is a separating proposition. Conversely, we can map a separating proposition back to a type by using the $\boxed{e : \tau}$ type, which satisfies $h : \boxed{e : \tau}$ iff $e : \tau$.

The separation-logic-inspired types in MMC are shown in Figure 3.3.

- The $\exists x : \tau_1, \tau_2(x)$ type is expressed as part of the dependent tuple construction we have already seen as `(ex x:` τ_1`,` $\tau_2(x)$`)`. This is almost the same as `(ghost x:` τ_1`,` $\tau_2(x)$`)` but the `ex` modifier means that the value of x is not available at the logical level either: if `p: (ghost x:` τ_1`,` $\tau_2(x)$`)` then `p.0: ghost` τ_1 but if `p: (ex x:` τ_1`,` $\tau_2(x)$`)` then `p.0` is not well formed.
- Most of the binary operators operate "pointwise" on the value part a, which is generally trivial when using these types to encode pure separating propositions rather than types like `u8`. Nevertheless, things like $\sim$`u8` are well formed types which in this case would assert that `u8` is empty (so this is itself an empty type).
- The separating conjunction is expressed using the tuple type `(`τ_1`,` τ_2`)`. When the value part a is trivial this is equivalent to the usual sepa-

rating conjunction.

- The types $\ell \mapsto v$ and $\boxed{e : \tau}$ have trivial value representation, as evidenced by the fact that a does not appear in the typehood predicate.
- The type $|\tau|$ or `moved(`τ`)` represents "the part of τ that remains after removing everything that cannot be duplicated." This has a recursive definition over the types, but generally this has the effect of retaining pure propositions $x > 0$ while stubbing out parts of the type that cannot be copied like $\ell \mapsto v$ to $()$. It satisfies the fundamental property $\tau \Leftrightarrow \tau * |\tau|$.

There are not many operations to deal with these types; the compiler generally just preserves these types without interacting directly with them. But there are a few operations that work on the types:

- Pattern matching works on `let (x, y):` τ_1 `&&` τ_2 `:= ...`, but y gets the type `moved(`τ_2`)` if τ_2 is not a copyable type.
- Patten matching also works on existential types `(ex x:` τ_1`,` $\tau_2(x)$`)`.
- The `typeof(e): [e:` τ`]` operator will capture the typehood predicate for an expression and put it in a variable. This can be later used by the `pun` operator (see section 3.2.10) to reconstitute the expression.

The main operation for handling these hypothesis variables is `entail`. `entail[`e_1`, ...,` e_n`] {` p `}:` τ if $e_i : \tau_i$ and p is a proof of $\boxed{e_1 : \tau_1} * \cdots * \boxed{e_n : \tau_n} \Rightarrow \tau$. This essentially allows users to exit the MMC language in order to perform proofs of complex propositions.

The details of the proof language (the syntax for p) are still undetermined pending more data gathering regarding the ease of use of these proof blocks, but one proof method is to reference a theorem from the ambient logic in which the low level correctness proof is being conducted. This enables proving theorems in the MM1 proof assistant and then using the theorems in an MMC program.

3.2.13 *Pointers and arrays*

Type	Equivalent in: C	Rust	Typehood pred. $\boxed{x : -}$	Meaning
`&sn` η	τ`*`	`&'a mut` τ	$x = \&\eta$	Pointer to place $\eta : \tau$
`ref`^{a} τ	–	–	$\mathsf{ref}^a\ \boxed{x : \tau}$	A value of type τ stored in a
`&`$^a\tau$	τ`*`	`&'a` τ	$\exists v : \mathsf{ref}^a\ \tau.\ x = \&v$	Shared pointer to τ
`own` τ	τ`*`	`Box<`τ`>`	$\exists v : \tau.\ x \mapsto v$	Owned pointer to τ
`[`τ`;` n`]`	τ`[`n`]`	`[`τ`;` n`]`	–	Array of n values of type τ

Figure 3.4: Pointer and array types.

A pointer is a value that stores the location of another value. MMC has a number of different pointer types, summarized in Figure 3.4, for distinguishing between these different kinds of ownership:

- `&sn` η is a "singleton pointer to η," where η is a place or "lvalue" expression, usually a local variable like x. It is equivalent to `sn(&`η`)` where &η is a pure expression denoting "the memory location of η." This is the most common pointer type used in MMC. Other pointer types generally have to be destructured to get access to a singleton pointer which can then be used for reading and writing.
- `ref`^{a} τ is not actually a pointer type but is introduced here because it is used in the desugaring of the shared pointer type. The logical values in `ref`^{a} τ are the same as τ, but the data is "stored elsewhere," with the annotation a keeping track of what local variable(s) are affected by writes to this value. The a annotation is roughly analogous to the `'a` "lifetime" annotation on the corresponding Rust type.
- &$^a\tau$ is a shared pointer to an existentially quantified value of type τ, stored externally in a. To use such a pointer you can pattern match it to get `v: ref`^{a} τ and `ptr: &sn v`, at which point we know that `*ptr` yields `v`.
- `own` τ is also an existential package, for an owned pointer. It also can be pattern matched to get `v:` τ and `ptr: &sn v`.[23]
- `[`τ`;` n`]` is the type of arrays of size n and element type τ. This does not involve a pointer indirection, the array is stored directly in the value. Unlike the corresponding C and Rust types, n is a ghost expression which need not be fixed at compile time, so this is another example of a dependent type.[24]

[23] Currently MMC is not capable of creating values of type `own` τ, because it does not currently support `malloc()` or other means of dynamic allocation. This is future work, but at least the effect on the type system is known.

[24] Arrays which exist on the stack have to have known size, however, because the compiler does not use `alloca()` for dynamic allocations on the stack. So to take advantage of runtime values of n one has to put the array behind some kind of indirection or make it a ghost value.

The operations manipulating these types are:

- `*ptr:` τ works for pointers of type `&sn` η, &$^a\tau$, or `own` τ. In the first case the resulting expression has value η, and in the other two cases the expression has unknown value but known type τ.
- &e`: &sn` e produces a reference to e, which is either a place expression η or a regular expression e (in which case the expression is evaluated into a temporary variable and the location of that temporary is returned).
- `&sn` η coerces to &τ when $\eta : \tau$.
- Given an array `arr: [`τ`;` n`]`, the expression `arr[`i`,` h`]:` τ is a place expression (i.e. it can be used for reading and writing) when $h : i < n$. The h can also be omitted, in which case `assert(`$i < n$`)` is used.
- Similarly, `arr[`a`..+`b`,` h`]: [`τ`;` b`]` is a place expression that "slices"

`arr` to a smaller array, consisting of indices a through $a+b$ (exclusive) in the original array. Here h must be a proof of $a+b \le n$.

- To construct a new array, the operations `[`e_0`, ..., `e_{n-1}`]: [`τ`; `n`]` and `[`e`; `n`]: [`τ`; `n`]` work if the e_i have type τ.

3.2.14 *Mutable parameters*

For modeling purposes, we want to think of mutation as being desugared to a state monad, which is to say, every operation is passed the mutable values and returns the new versions of all mutable values in addition to the regular returns. However, we want to support mutation directly in the compiler so that spurious copies are avoided when possible.

The assignment operator `x <- y;` allows mutating variables inside a function, but to propagate mutations between function calls we have an additional `mut` annotation.

For example, here is a function that mutates a large array `arr` by parameter passing:

```
proc replace_value_copy(arr: [u8; 100]): u8, [u8; 100] {
  let old := arr[0];
  arr[0] <- 42;
  old, arr
}
proc caller() {
  let arr: [u8; 100] := [0; 100];
  let v, new_arr := replace_value_copy(arr);
  arr <- new_arr;
}
```

If the compiler was clever enough, it could conceivably avoid the copy of `arr` from the input to the output, but MMC is not that clever, and at the call site we have to put the new array in a temporary variable and reassign it to the original variable if we want to mutate `arr` in the caller.

To explicitly represent mutation, we can instead pass a reference to the array:

```
proc replace_value(mut arr: ref [u8; 100], ptr: &sn arr): u8 {
  let old := ptr[0];
  ptr[0] <- 42;
  old
}
proc caller() {
  let arr: [u8; 100] := [0; 100];
  let v := replace_value(arr, &arr); // mutates arr
}
```

The first argument to `replace_value` is a ghost parameter, which denotes the place that the `ptr` argument points to. As a result, the `(*ptr)[0] <- x;` line in `replace_value` is directly changing the array declared in `caller`'s stack frame.

Because of the use of dependent typing to express preconditions and postconditions, it is often the case that the incoming value of a `mut` parameter actually has a different type than the outgoing value. To express this we can use the `out` modifier on returns:

```
proc double(mut x: u8): out[x] sn((2 * x): u8) {
  x <- sn(cast(2 * x));
}
```

This example asserts that the input value x is mutated such that the value after the call is twice the input value. `out` arguments do not count as regular returns, so `return sn(cast(2 * x))` would not work here; the code must assign the specified returns to the function parameters using `x <- ...`. If a `out` argument is not specified, the `mut` argument is assumed to have the same type as it had in the parameter list.

3.2.15 *Global variables and constants*

Global variables are variables that are stored in the data segment of the executable, rather than on the stack frame of a function. Like with mutable parameters, a function which reads or writes a global variable actually takes it as an argument, except that the calling convention says that the value passed is at a particular fixed location in memory instead of in the argument registers as with normal arguments.

```
global CLOCK: u64 := 1;
const TICK_AMOUNT: u64 := 1;
proc tick(global mut CLOCK) {
  CLOCK <- cast(CLOCK + TICK_AMOUNT);
}
proc example() { // "global mut CLOCK: u64" elided
  tick();
  tick();
}
```

In this example, we have a global variable `CLOCK` that is incremented on every call to `tick()`. Every function implicitly has a list of globals that it uses or modifies in its parameters; this list is inferred if not specified, but it can be given explicitly if the function changes the type of the global or expects the global to have a different type than the one originally declared.

Constant declarations are similar to globals, but constants cannot be modified; they are simply shorthand for compile-time expressions.

Large constants will be stored in the read only portion of the binary and copied to local variables on each use.

3.2.16 *Type definitions*

```
type SmallishU8(n: nat) := (x: u8, h: x < n);
proc example() {
  let _: SmallishU8(14) := (0, assert(0 < 14));
}
```

Type declarations are abbreviations for types. They can have type or value parameters, and they will be unfolded as necessary.

3.3 *Modeling MMC*

A key difference between MMC and traditional programming languages is that the various features provided by the language are not merely chosen for their convenience, but also because they have a known semantics and we can lower them to a correctness proof. In fact, MMC has largely grown from the "bottom up," starting with very few conveniences[25] but a solid proof theory and growing in the direction of more syntactic sugar, instead of starting with the convenience and working toward a complete semantics and correctness proof.

[25] The astute reader will notice that the version of the language presented here still lacks a great many conveniences found in C and/or Rust such as `enum` and `union`, (bounded) `for` loops, uninitialized values, and lots and lots of library functions.

As a result, the proof context and Hoare logic interpretation of the code is very clearly visible in terms of the user's perspective on the type context, because they have direct access to manipulate much of it.

3.3.1 *Hoare logic primer*

Before getting to its implementation in MMC, it will help to have some context for Hoare logic in general, and its extension to separation logic. Hoare logic[26] is a methodology for reasoning about computer programs developed by Tony Hoare in 1969 and subsequently extended by many other researchers.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct 1969

The central notion is a predicate written $\{P\}\ C\ \{Q\}$, where P and Q are propositions about the machine state, and C is a subprogram. This asserts that if P holds of the machine state and C is executed, then the execution will not "go wrong," and furthermore if C terminates,[27] then Q will hold of the state after execution of C.

[27] Hoare logic has variations for doing either total or partial correctness, in a similar manner to section 3.2.6. Here we are considering the partial correctness variant; the HL-WHILE rule is more complex in the total correctness variant.

Hoare logic (example) $\boxed{\{P\}\ C\ \{Q\}}$

$$\textsc{hl-skip}\quad \{P\}\ \mathsf{skip}\ \{P\}$$

$$\textsc{hl-weak}\quad \frac{P \to P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \to Q}{\{P\}\ C\ \{Q\}}$$

$$\textsc{hl-seq}\quad \frac{\{P\}\ S\ \{Q\} \quad \{Q\}\ T\ \{R\}}{\{P\}\ (S;\ T)\ \{R\}}$$

$$\textsc{hl-assign}\quad \{P[E/x]\}\ x := E\ \{P\}$$

$$\textsc{hl-if}\quad \frac{\{P \land B\}\ C_1\ \{Q\} \quad \{P \land \neg B\}\ C_2\ \{Q\}}{\{P\}\ (\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)\ \{Q\}}$$

$$\textsc{hl-while}\quad \frac{\{P \land B\}\ C\ \{P\}}{\{P\}\ (\mathsf{while}\ B\ \mathsf{do}\ C)\ \{P \land \neg B\}}$$

This logic is already good enough to prove facts about simple programs: for example, the program $x := 0$ has the specification $\{\top\}\ x := 0\ \{x = 0\}$, which we can prove by applying the HL-ASSIGN rule to get $\{0 = 0\}\ x := 0\ \{x = 0\}$, and simplifying the precondition to $\top$ using the HL-WEAK rule.

One of the important properties about this logic is that it is *compositional* in the programs C: the proof of each program proceeds by recursion on the program, proving facts about the pieces and composing them to proofs about the whole program. However, there are some caveats:

- The HL-SEQ rule requires that we invent the intermediate predicate Q. This is okay if we have a way of working out the postcondition from the precondition or vice versa, but...
- The HL-ASSIGN rule needs to know the postcondition and derives the precondition
- The HL-WHILE rule needs to know the precondition and derives the postcondition
- The HL-WEAK rule can be used to glue together proofs where the precondition of one step doesn't match the postcondition required by the next step, but it provides no guidance regarding the intermediate variables P' and Q'.

Put together, these make it difficult to automatically derive the required predicates to construct a proof of $\{P\}\ C\ \{Q\}$ given only P, C, Q, and the full problem of inferring appropriate loop invariants is undecidable.[28]

[28] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. *ACM SIGPLAN Notices*, 51(1):217–231, 2016

For languages that have expressions, not just statements (and pure expressions), the Hoare triple is modified to read $\{P\}\ E\ \{v.\ Q(v)\}$, because the expression E evaluates to a value v and the postcondition may depend on that value. Here are some example rules:

hl-var

$$\{P(x)\}\ x\ \{v.\ P(v)\}$$

hl-assign′

$$\frac{\{P\}\ E\ \{v.\ Q[v/x]\}}{\{P\}\ x := E\ \{Q\}}$$

hl-add

$$\frac{\{P\}\ E_1\ \{v.\ Q(v)\} \quad \forall v.\ \{Q(v)\}\ E_2\ \{w.\ R(v+w)\}}{\{P\}\ (E_1 + E_2)\ \{x.\ R(x)\}}$$

3.3.2 Separation logic

Separation logic[29] is a more recent advance in program verification methodology that modifies the propositions P and Q to not merely be propositions referencing program variables, but to be *separating propositions*, or propositions about *subsets* of the program state.

[29] John C Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, 2(1):303–321, 2000

To see why this makes a difference, let us consider extending the toy language above with pointers. Suppose the operation $^*p \leftarrow E$ assigns the result of expression E to the memory location described by p. In Hoare logic, a reasonable logical rule for this would be:

hl-store

$$\{P[M[p := E]/M]\}\ {}^*p \leftarrow E\ \{P\}$$

Here M is a representation of memory as a function mapping addresses to values, and $M[p := E]$ is the function defined as:

$$M[p := E](x) := \begin{cases} E & \text{if } x = p \\ M(x) & \text{otherwise} \end{cases}$$

For example, suppose we know $M(a) = 1 \land M(b) = 1$ beforehand, and execute $^*a \leftarrow 2$. If we take P to be $M(a) = 2 \land M(b) = 1$, then by applying hl-store we obtain

$$\{M[a := 2](a) = 3 \land M[a := 1](b) = 1\}\ {}^*a \leftarrow 2\ \{M(a) = 2 \land M(b) = 1\}$$

so by applying hl-weak to change the precondition to the desired $M(a) = 1 \land M(b) = 1$, we are left with the goal:

$$M(a) = 1 \land M(b) = 1 \to M[a := 2](a) = 2 \land M[a := 1](b) = 1.$$

Now $M[a := 2](a) = 2$ is true by definition, but $M[a := 1](b) = M(b)$ only if $a \neq b$, so in fact our theorem is not provable. We need $a \neq b$ to be an additional precondition to prove the theorem.

This is a well known problem, the bane of C compilers everywhere, known as "pointer aliasing." A function which receives two arbitrary pointers does not know that writing to one does not affect the value of the other. It is possible to solve the problem in Hoare logic with enough disequality assumptions, but these assumptions grow quadratically with the number of pointers in the program: every part of the program potentially affects every other part of the program and we have to explicitly opt out of such entanglement.

This is where separation logic comes to the rescue. We extend the grammar of propositions with an operator $P * Q$, called "separating conjunction," which asserts that P and Q apply separately to disjoint pieces of the machine state. The assertion that a points to 1 is now expressed as $a \mapsto 1$, so in separation logic the Hoare triple we want to prove is instead:

$$\{a \mapsto 1 * b \mapsto 1\}\ {}^*a \leftarrow 2\ \{a \mapsto 2 * b \mapsto 1\}$$

The relevant rules we need to prove this are:

$$\text{SL-STORE}\quad \{p \mapsto a\}\ {}^*p \leftarrow E\ \{p \mapsto E\} \qquad\qquad \text{SL-FRAME}\quad \frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

There are two important differences here besides the change of notation.

- SL-STORE has a nontrivial precondition now. It requires that p be pointing to some value a (which is discarded), unlike the Hoare logic rule which allowed any choice of P, including $\top$.
- The SL-STORE rule does not contain any arbitrary predicate P in it. It doesn't need it, because a Hoare triple like $\{P\}\ C\ \{Q\}$ not only requires that P be true, it also asserts that any part of the machine state separate from P is untouched by C.
  This is what licenses the frame rule SL-FRAME, which allows us to screen out any part of the machine state irrelevant to this particular line of code. In the example, this is how we can carry $b \mapsto 1$ from the precondition to the postcondition without ever explicitly having to prove that $a \neq b$. There could be a hundred other variables in the context and there would be no quadratic growth of side conditions, so we have restored *spatial* compositionality.

Let us turn now to the task of applying this general framework to MMC in the subsequent sections.

3.3.3 *The type context*

The type context is the set of variables in scope, their types, and their values. At any given position in a program, there is a type context that determines the type correctness of the next expression to be executed. For example:

```
proc example(x: nat) {
  let y: nat := 1;
  // Here the type context is:
  // x: nat, y: nat := 1
  // so the following is well typed:
  let _: sn(1) = sn(y);
  // and the following is a compile error
  let _: sn(1) = sn(x); // ERROR
}
```

Evaluating an expression with side effects can cause the type context to change:

```
proc example(b: bool) {
  let y: nat := 1;
  // b: bool, y: nat := 1
  y <- 2;
  // b: bool, y: nat := 2
  if b {
    y <- y + 1;
    // b: bool, y: nat := 3
  }
  // b: bool, y: nat
}
```

Evaluating expressions can also cause separating propositions to be "moved out" of the type context:

```
proc free(p: own u64) { ... }
proc double_free(p: own u64) {
  // p: own u64
  free(p);
  // p: moved(own u64)
  free(p); // compile error, already moved p
}
```

Type contexts are very closely related to the preconditions that show up in Hoare logic. We saw in section 3.3.1 that if we want to not have to specify all intermediate propositions we need to either derive postconditions from preconditions or vice versa, and in MMC it is reasoning forwards, from preconditions to postconditions.

To forward reference a bit, the typing judgment for expressions looks like $\Gamma;\ \delta \vdash e : \tau \dashv \delta'$ where δ is the flow-sensitive part of the type context (which includes the current types and values of variables) and Γ contains the part that is only scope based (like the existence of variables, labels and loop labels, and other functions). This is translated to a Hoare triple of the form $\{\widetilde{\Gamma} * \widetilde{\delta}\}\ e\ \{v.\ \widetilde{\Gamma} * \widetilde{\delta'} * \widetilde{\tau}(v)\}$, where $\widetilde{\cdot}$ denotes the translation of Γ and δ into separating propositions (and τ into a function from values to separating propositions).

The simplest part of this is something we have already been taking advantage of in user code: A type τ is a function from values to separating propositions, so for each type we have to define what $\boxed{v : \tau}$ means, and in most cases we have already done so in section 3.2. To give some examples, $\boxed{v : \texttt{u64}}$ means $v < 2^{64}$, and $\boxed{v : \texttt{own u64}}$ means $\exists v < 2^{64}.\ x \mapsto v$.

The flow-sensitive context δ contains mainly the typing assertions for variables in the context. $x : \tau := e$ translates to the separating proposition $\boxed{e : \tau}$, and $x : \tau$ translates to $\boxed{x : \tau}$. These are conjoined with $*$ to form $\widetilde{\delta}$.

Finally, $\widetilde{\Gamma}$ contains assertions about how the variables are tied to the machine state itself. For example, that `x: u64` is currently being stored in register `RAX` (i.e. $\texttt{RAX} \mapsto x$), while `ghost z: u64` has no machine state associated to it (i.e. emp), and `y: u64` also has no machine state because it has been optimized out. The compiler generally retains control over how this mapping is performed as long as it adheres to some constraints based on the input code (for example, ghost variables must not have machine state), and as long as it can actually find code sequences satisfying the desired semantics.

3.3.4 *Toward a compositional program logic*

The parameters of the underlying machine semantics are as follows:

- A set S of valid machine states.
- A relation $\leadsto \subseteq S \times S$, where $s \leadsto s'$ denotes that s' is obtained in one primitive step from s.
- A relation $\mathsf{exits} \subseteq S \times \mathsf{u32}$, such that $\mathsf{exits}(s, c)$ indicates that s is an exit state with exit code c.
- A relation $\mathsf{init} : \mathsf{u8}^* \to \mathcal{P}(S)$, where $s \in \mathsf{init}(p)$ indicates that if p is the input program, s is a valid initial state ready to execute the code on the some input.
- A function $\mathsf{in} : S \to \mathsf{u8}^*$ which extracts the list of bytes on standard input that have been consumed since the start of the program.
- A function $\mathsf{out} : S \to \mathsf{u8}^*$ which extracts the log of all data printed on standard output since the start of the program.

The step relation is not assumed to be deterministic, and we have the following conventions:

- A state satisfying $\mathsf{exits}(s, 0)$ is a successful exit state
- A state satisfying $\mathsf{exits}(s, c)$ for $c \neq 0$ is a failure state
- A state satisfying $\neg\exists s'.\ s \leadsto s'$ (and $\neg\exists c.\ \mathsf{exits}(s, c)$) is a stuck state

We must avoid stuck states because they represent places where undefined behavior can occur: they may include running unspecified instructions or taking operations outside the model, which may invalidate the assumptions of the proof. Failure states are a permitted way to end execution (see section 3.2.8), and we promise nothing about the execution in case of failure.

The final theorem we are aiming to prove about program *prog*, given a user-given relation $T \subseteq \mathsf{u8}^* \times \mathsf{u8}^*$, is:

Theorem 3.3.1 (template)**.** *For any start state s_i after loading program prog, every state reachable from s_i can reach some other state s_f which is an exit state ($\mathsf{exits}(s_f, c)$), and furthermore if $c = 0$ then $T(i, o)$ holds, where i is the input consumed between s_i and s_f and o is the output at s_f. In symbols:*

$$\begin{aligned}\forall s_i \in \mathsf{init}(prog).\ \forall s_m.\ s_i \rightsquigarrow^* s_m \to\\ \exists s_f, c.\ s_m \rightsquigarrow^* s_f \land \mathsf{exits}(s_f, c) \land\\ (c = 0 \to T(\mathsf{in}(s_f), \mathsf{out}(s_f)))\end{aligned}$$

The $\forall\exists$ quantification here ensures that there are no reachable stuck states, because if $s_i \rightsquigarrow^* s_m$ and s_m was a stuck state then $s_f = s_m$ and there would not be any choice of c such that $\mathsf{exits}(s_f, c)$. It does not imply strong termination however, or any uniform bound on the number of steps of evaluation. (For example, the finite state machine with two states $\overset{\curvearrowleft}{s} \to f$ where f is final is not strongly terminating because one can loop s forever, but every path can be extended to end at f.) In the absence of non-determinism this does imply termination.[30]

We want to develop a program logic to assist in proving this theorem about the program that was given to us by the user. First, fix *prog* and R as global parameters of the logic.[31] We define:

- s is *terminal*, or $\mathsf{terminal}_T(s)$, if $\exists c.\ \mathsf{exits}(s, c) \land (c = 0 \to R(\mathsf{in}(s), \mathsf{out}(s)))$
- s *can finish*, or $\mathsf{canFinish}_T(s)$, if $\exists s'.\ s \rightsquigarrow^* s' \land \mathsf{terminal}_T(s')$
- s is *valid*, or $\mathsf{valid}_T(s)$, if $\forall s'.\ s \rightsquigarrow^* s' \to \mathsf{canFinish}_T(s')$

Then the target theorem asserts that $\forall s.\ s \in \mathsf{init}(prog) \to \mathsf{valid}_T(s)$: all initial states are valid. We prove this "from back to front": we first establish that if we reach the last line of the program then we are in a valid state, then prove that the second to last line is also valid, and so on until we reach the beginning of the program.

We want to get away from talking about machine states directly, so we introduce a satisfaction predicate $s \vDash P$ where P is a separating proposition. More precisely:

- A *place* is an abstraction of a "location" in the abstract machine

30 This is likely not as strong a theorem as it could be. A uniform bound on runtime seems possible and would close the strong termination loophole.

31 It may seem odd to fix *prog*, the final linked ELF file corresponding to the input program, since we will proceed by structural induction on the input program, and while we are constructing the proof we don't know yet what the program will look like concretely. But this kind of forward reference is fine as long as we eventually put in the real executable here; this is similar to leaving a metavariable in a proof and resolving it at the end.

state. The set of places is architecture-dependent, but for the x86 implementation, the places are:

  - The input
  - The output
  - The exception flags
  - The instruction pointer
  - The flags
  - One place per general-purpose register
  - One place per memory byte

- If p is a place, then $s(p)$ gets the value of a place out of the machine state s. (For example, the value of the register, or the byte of memory.)
- A *heap* is a finite partial function on places.
- If h is a heap, then $s \vDash h$ means that for all $(p, x) \in h$, $s(p) = x$.
- If h, h' are heaps, then $h \perp h'$ means $\mathrm{dom}(h) \cap \mathrm{dom}(h') = \emptyset$.
- A separating proposition is a set of *heaps*.
- If P is a separating proposition, then $s \vDash P$ means that there exists $h \in P$ such that $s \vDash h$.
- We define $\mathsf{ok}(P)$ to mean that $\forall s.\ s \vDash P \to \mathsf{valid}(s)$.

This is a fairly standard setup for defining a separation logic.[32] We can use this to construct some separating propositions: for example $p \mapsto v$ can be defined as $\{\{(p, v)\}\}$ (the singleton of a singleton heap mapping p to v), and

[32] There are much more sophisticated setups for separation logic, like Iris. A result of this simplicity is that certain kinds of higher order reasoning are not currently possible in MMC, but it is possible to change some parts of this foundational setup if more use cases arise.
Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018

$$P * Q := \{h \cup h' \mid h \in P \wedge h' \in Q \wedge h \perp h'\}.$$

$\mathsf{ok}(P)$ acts something like a one-sided version of a Hoare triple $\{P\}\ C\ \{Q\}$. We can define a two-sided version as

$$\{P\} \cdot \{Q\} := \forall R.\ \mathsf{ok}(Q * R) \to \mathsf{ok}(P * R)$$

but we don't really have an equivalent for C – the whole program is in the machine state already so we just need to step the state. (Note that we have built the frame rule into this definition.)

The form of $\{P\} \cdot \{Q\}$ leads to a natural generalization:

$$\{P\} \cdot \{Q_1, \ldots, Q_n\} := \forall R.\ \mathsf{ok}(Q_1 * R) \wedge \cdots \wedge \mathsf{ok}(Q_n * R) \to \mathsf{ok}(P * R)$$

This is useful for programs that have multiple exit points. The example programs in section 3.3.1 did not have any non-local control flow, but MMC notably has `return`, `break`, `continue`, and `l()` for jumping to a `label l()`. In general, when we are working through a program we will have some set of labels and exit points we can jump to, and in the proof we will have `ok`(Q) hypotheses for each of these exit points, either because they come later in the program (so we have already proven that they are `ok` since we work back-to-front), or because we have them as an inductive hypothesis (for a back edge in a loop, where the `variant` provides the proof that the inductive hypothesis applies).

In the next chapter, we will look into how compilation of MMC programs actually works. We will connect it back to separation logic in section 4.4.

4
The Metamath C Compiler

A *compiler* IS A PROGRAM that receives (generally) text input in some language with an execution semantics and produces machine code with the same execution behavior. The Metamath C compiler (MMCC) is similar, in that it takes code written in the MMC language (see Chapter 3) and produces executable (Intel x86) machine code, but unlike most compilers this is just one of the two outputs of the compiler. The other output is a proof that the machine code just generated satisfies a specification, which was written as part of the MMC input code. We call this a *proof-producing compiler*[1] because the generated proof verifies the behavior of the compiled code.

Architecturally, MMCC is fairly similar to a regular compiler. It consists of a sequence of *passes* over the code, transforming it into different forms suitable for different kinds of analysis, before producing machine code. The novelty is that most of the intermediate languages have a strong type system and the compiler preserves typing information through the translations.[2]

The compiler passes are summarized below, roughly in the order they are performed: from high level to low level.

- The first few steps are actually part of the MM1 language:
  - The MM1 file is parsed into an AST.
  - The s-expressions inside `do` blocks are further parsed into the MM1 metaprogramming language.
  - The MM1 code is executed, resulting in a call to the built-in `(mmc-compiler)` function on an s-expression literal containing the actual MMC code.

  This is because the MMC compiler is presently presented as a metaprogram which can be executed in the course of elaborating an MM1 file.[3]
- The lisp datum is parsed into an AST.

[1] There is a subtle distinction that it is not a *verified* compiler: the overall operation of the compiler itself is not proved correct. See section 3.1 for a discussion about this design decision.

[2] Even this is not unheard of in production compilers: Haskell GHC uses a strongly typed intermediate language with a static typechecker, and this is reportedly useful for unit-testing optimization passes. MMC simply takes this to the logical extreme by using a very strong type system capable of expressing functional correctness properties.

[3] This means that it gets some of the same niceties as regular use of MM1 like live diagnostics from the language server, but it means that MMC is written in lisp syntax instead of the C-like syntax presented in this dissertation. However this difference is only skin-deep; all the semantics are the same as presented here.

One nontrivial feature this arrangement provides is the ability to use MM1 as a macro language over MMC. This is difficult to mimic if MMC is parsed directly from text.

- The AST is desugared to a simpler AST. In this pass (which occurs concurrently with the previous step):
  - names are resolved;
  - label groups are identified;
  - label and loop label references are identified;
  - and some syntax sugar like `match` expressions and chained inequalities are reduced to primitives.
- The AST is typechecked and lowered to an intermediate language called HIR (high-level intermediate representation). This pass is responsible for the majority of user facing errors.
  - Type inference is performed.
  - Definitional reduction / type normalization is performed where necessary to make types match.
  - If the user uses an expression hole "_" to query the current type context, the error message is reported here.
  - If necessary, MM1 is called to resolve embedded proofs inside `entail`.

  The elaborated HIR is structurally similar to the AST but includes full type information and can be typechecked.
- The HIR is lowered to MIR (mid-level intermediate representation), which is no longer AST-structured but consists of a graph of "basic blocks" connected by jumps (also known as a control-flow graph or CFG).
  - Each basic block consists of a sequence of elementary operations that declare fresh variables (also known as `let`-normal form)
  - Variables are only declared once and mutation creates new names, also known as static-single assignment (SSA) form (with block parameters)
  - Basic blocks are connected into a tree reflecting the original AST structure[4]
  - All mutation is turned into explicit parameter passing
- The MIR is the most versatile IR for doing program analysis, and so a number of optimizations are performed at this level:
  - Reachability analysis is performed to detect dead blocks. A block is reachable if it is accessible from the entry block by forward edges, and also there exists some path from the block to the `return` or a side-effecting procedure or `assert`.[5]
  - Ghost analysis is performed to mark anything as `ghost` unless it must be computed to produce (a computationally relevant part of) the `return` or a side-effecting procedure or `assert`.
    - ∗ This pass is also responsible for the `if` → `by_contra` transformation mentioned in section 3.2.11.

[4] This is unusual for CFG representation; normally this is represented implicitly using the "dominator tree" instead. It helps here to have it be an explicit part of the IR because this gives us a place to put the `variant` annotations from the source and gives the structure of the correctness proof.

[5] In particular, an unguarded infinite loop is illegal because the entry is unreachable by this definition.

  - Legalization is a pass to remove infinite intermediates. For example:

    ```
    proc example(x: u64, y: u64) {
      let z: nat := x + y + y;
      return z as u64;
    }
    ```

    Here we cannot eliminate `z` as `ghost` because it is actually used to compute the return, so we work backwards from the `as u64` to work out a different way to compute the result without using `z`:

    ```
    proc example(x: u64, y: u64) {
      let z: nat := x + y + y;
      let z64: u64 := x +_64 y +_64 y;
      return z64;
    }
    ```

    Here `x +_64 y` is addition modulo 2^{64}, an operation not directly accessible in the source language. This will eventually be lowered to an `ADD` instruction in the hardware.
  - Because the previous step can produce dead values like `z`, we run ghost analysis again to mark `z` as `ghost`.

All of the above steps take place on each function as soon as it is added to the compiler state. Once we have the `main()` function we can do "link time analyses" that require all functions to be known:

- The collection pass determines which functions are accessible from `main()`, a global version of the reachability analysis. It also determines which type specializations of polymorphic functions should be instantiated,[6] as well as the constants that need to be placed in the read-only section of the binary and their addresses.
- The allocation pass determines the abstract layout of the stack frame for a function:
  - Which variables need to exist simultaneously and so must have disjoint storage
  - Which variables should have overlapped storage because they are SSA copies of the same source variable after "mutation"
  - Which variables are zero-sized and do not need physical storage

  Some more user-facing errors come from this step, if a variable that could not be eliminated up to this point turns out to be infinite-sized.
- The monomorphized MIR is lowered to another IR, called virtual-register code (VCode).
  - This IR exists primarily to be the input to register allocation.
  - It is architecture-specific, so the primitive operations here look like x86 instructions, except that they use an infinite pool of "vir-

[6] MMC monomorphizes generic functions similarly to C++ or Rust. That is, a function with a type argument has copies made for every type at which it is used, so the collector is used to determine how many copies need to be made.

  tual registers" instead of real registers
  - Even though MIR instructions are relatively simple, some need to be broken down into many operations, like array construction or accessing a nested place expression like `arr[1][2].f <- val;`
  - Block ordering is also optimized here: this attempts to make as many jumps into fallthroughs as possible by placing the jump target immediately after the jump.

- The code undergoes register allocation, which is the process of assigning concrete registers to the virtual registers, respecting which values need to be accessed later and "spilling" registers to the stack when there are not enough physical registers.[7]

[7] MMC uses an external register allocation library, because this is the only part of the compilation process that does not need to be proof preserving – the resulting register assignment can be checked for correctness after the fact.

- Applying the results of register allocation to VCode yields physical-register code (PCode). This is equivalent to assembly code, and we could pass this to an assembler at this point to get real machine code.
- Branch displacement optimization (BDO) is done at this point. This determines which jump instructions can use the short form that can only jump 128 bytes (which itself can cause more jumps to be shortened).
- The instructions are assembled into machine code (a sequence of bytes).
  - There are some minor optimizations performed in this process, like replacing `MOV reg, 0` with `XOR reg, reg` (which is smaller and faster).
- The functions are ordered and padded, the constants are laid out and the ELF header is added to produce the completed ELF file, ready for execution.

This is where a regular compiler would stop, and it is possible to run MMCC as a regular compiler to produce this output. But as we mentioned at the start, this is only the first of two outputs of the verifying compiler, and we have more work to do to get a proof out.

There are two important IRs for the proof generation stage: MIR and PCode.

- MIR has a good type system, a fairly lossless image of the user's proof annotations in the source language. In particular looking at this level means we do not need to verify anything about the higher level languages this was translated from, nor the optimizations that were performed on MIR.
- PCode has a defined deterministic mapping to the actual bytes that were emitted: we can give a precise accounting for everything that appears in the file. However it is not well suited to a strong type

system because it is tied up in details of the Intel x86 ISA.

- During the MIR → VCode translation, we kept a record of the mapping from one to the other, and we can compose that with the data coming from the register allocator to get a "justification" for every instruction in PCode relative to the source MIR.

So our strategy for producing the proof proceeds through those two IRs:

- For each function, we construct the assembly (a logical rendering of the PCode for the function) and prove that this assembly assembles to the subsequence of bytes that appeared in the final binary.

- (∗) For each function, we prove that it satisfies the MIR version of the type signature, by using the MIR for the broad strokes of the proof and the MIR → PCode map to show how the individual assembly lines combine to produce each MIR instruction.

- Therefore, looking at the type signature of `main(): R`, we conclude that if `main()` is run, then `R` holds afterwards, where `R` is the final property we wish to establish.

Currently, the MMC compiler performs all passes mentioned so far except for the MIR proof stage, marked (∗), which is incomplete. In the following sections we will go into more detail on some of these intermediate languages and passes, and the proof strategy for (∗).

4.1 *MIR in depth*

The first few intermediate languages are not particularly different from the input source. Typechecking occurs at the source level because it yields better error messages. But once most of the typechecking is done, the program is translated into *basic block* form in the Mid-level Intermediate Representation (MIR).

A basic block is a sequence of *statements* ending in a *terminator*. The characteristic property of a basic block is that you cannot jump into the middle of a block: all jumps go from the end of one block to the beginning of another.

MIR is also an interesting stage because it is the lowest level that actually has full proof information, translated from the higher levels. The levels below this one only have partial proof information and refer back to the MIR for the rest of the proof.

The syntax of MIR is as follows:

$prog ::= \lambda\ ctx \to ctx'.\ tail$	program
$block ::= \lambda\ ctx.\ tail$	basic block
$tail ::= term \mid stmt;\ tail$	block tail expression
$l ::= \texttt{return} \mid \langle\text{identifier}\rangle$	block label
$v ::= \langle\text{identifier}\rangle$	variable name
$n ::= \langle\text{integer}\rangle$	integer constant
$pr ::= \dots$	MM0 proof
$ctx ::= \overline{arg}$	block context
$arg ::= \texttt{ghost}^?\ v : \tau := e^?$	variable
$\tau, e ::= \dots$	types, pure expressions
$p ::= v \mid p.i \mid p[v,h] \mid p[v..^{+}v',h] \mid {}^*v$	place expressions
$o ::= \texttt{move}\ p \mid \texttt{ref}\ p \mid \texttt{copy}\ p \mid c$	operands
$c ::= () \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{uninit} \mid \texttt{sizeof}(\tau) \mid \texttt{proof}\ \{pr\}$	constants
$stmt ::= \texttt{let}\ (v : \tau := e^?) := rval;$	let-declaration
$\mid p : \tau \leftarrow o;$	mutating assignment
$\mid \texttt{labels}\ \overline{l := block};$	label group declaration
$\mid \texttt{pop\_labels}\ \overline{l};$	exit label group scope
$\mid \texttt{join}\ l := block;$	declare forward jump
$rval ::= o$	use operand
$\mid unop(o) \mid binop(o,o')$	apply unary/binary op.
$\mid \texttt{pun}(pk,\ p) \mid \texttt{cast}(ck,\ o : \tau)$	type conversion
$\mid \langle\overline{o}\rangle \mid [\overline{o}]$	construct tuple / array
$\mid \texttt{ghost}(o)$	force value as ghost
$\mid \&p$	address of place
$\mid \texttt{typeof}(o)$	take variable type
$term ::= l(\overline{v := o})$	jump to block
$\mid \texttt{if}\ o\ \{arg.\ tail\}\ \texttt{else}\ \{arg'.\ tail'\}$	branch
$\mid \texttt{assert}(o); \{arg.\ tail\}$	assert
$\mid \texttt{unreachable}(o)$	unreachable block
$\mid f(\overline{\tau}, \overline{o}); \{\overline{arg}.\ tail\}$	function call
$\mid \texttt{exit}(o)$	exit program

This is slightly simplified but still captures essentially everything in the MMC language. One interesting aspect of this grammar compared to traditional presentations of basic block form is that the blocks are not all declared at the top level and able to call each other freely.[8]

[8] For performance reasons, all the blocks are stored at the top level anyway, but the scoping constraints act as invariants on which blocks can call which others.

To illustrate the scoping constraints, let us consider a very basic typing rule which ensures that labels must be well scoped:

Program and block scoping $\boxed{\vdash \textit{prog}\ \mathsf{scoped}}$ $\boxed{\bar{l} \vdash \textit{block}\ \mathsf{scoped}}$

$$\frac{\texttt{return} \vdash \textit{tail}\ \mathsf{scoped}}{\vdash (\lambda\ \textit{ctx} \to \textit{ctx}'.\ \textit{tail})\ \mathsf{scoped}} \qquad \frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (\lambda\ \textit{ctx}'.\ \textit{tail})\ \mathsf{scoped}}$$

Statement / terminator scoping $\boxed{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}$

$$\frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (\texttt{let} \ldots;\ \textit{tail})\ \mathsf{scoped}} \qquad \frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (p : \tau \leftarrow o;\ \textit{tail})\ \mathsf{scoped}}$$

$$\frac{\forall i.\ \bar{l}, \overline{l'} \vdash \textit{tail}'_i\ \mathsf{scoped} \qquad \bar{l}, \overline{l'} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (\texttt{labels}\ \overline{l' := \lambda\ \textit{ctx}'.\ \textit{tail}'};\ \textit{tail})\ \mathsf{scoped}}$$

$$\frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l}, \overline{l'} \vdash (\texttt{pop\_labels}\ \overline{l'};\ \textit{tail})\ \mathsf{scoped}}$$

$$\frac{\bar{l} \vdash \textit{tail}'\ \mathsf{scoped} \qquad \bar{l}, l' \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (\texttt{join}\ l' := (\lambda\ \textit{ctx}'.\ \textit{tail}');\ \textit{tail})\ \mathsf{scoped}}$$

Because we are only tracking the list of labels in scope, `let` and assignment have no effect, but the latter three instructions (which are all ghost instructions, i.e. machine no-ops) do have an effect on the context.

- `labels` pushes a group of labels to the context, which are available both in the proof that the label blocks themselves are well scoped, as well as the remainder of the block.
- `pop_labels` undoes the effect of `labels`, removing a set of labels from the scope.[9]
- `join` is a forward jump declaration. We are not allowed to use the new label inside the label itself, but we can use it in the remainder of the block. This is appropriate to use for the merge point of an if statement, where the rest of the program after the `if` block goes in *tail*′, and *tail* has a branch followed by two jumps to l'.

[9] This is used because label scopes are supposed to follow lexical structure like the block of a loop, but the definition here is flow-based, which can sometimes lead to labels sticking around after they should not be accessible. For example the code after a `while` loop is dominated by the loop, but we should not still be able to jump back to the start of the loop after exiting it.

Scoping for terminators checks that the labels used are in the context:

$$\frac{l' \in \bar{l}}{\bar{l} \vdash l'(\overline{v := o})\ \mathsf{scoped}} \qquad \frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped} \qquad \bar{l} \vdash \textit{tail}'\ \mathsf{scoped}}{\bar{l} \vdash (\texttt{if}\ o\ \{\textit{arg}.\ \textit{tail}\}\ \texttt{else}\ \{\textit{arg}'.\ \textit{tail}'\})\ \mathsf{scoped}}$$

$$\frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (\texttt{assert}(o); \{\textit{arg}.\ \textit{tail}\})\ \mathsf{scoped}} \qquad \frac{\bar{l} \vdash \textit{tail}\ \mathsf{scoped}}{\bar{l} \vdash (f(\bar{\tau}, \bar{o}); \{\textit{arg}.\ \textit{tail}\})\ \mathsf{scoped}}$$

$\bar{l} \vdash$ `unreachable`(o) scoped $\qquad \bar{l} \vdash$ `exit`(o) scoped

The full typing rules complicate this by having a context of the form $ctx, \overline{l : ctx} \vdash \dots$. That is, we have both a local context (which is augmented every time we pass a `let` statement or a terminator like `if` or `assert` that adds the assumption about the relevant expression being true or false to the context), as well as a context in each label (which is formed from the context at the point the block was introduced, plus the declared context in the block). See Appendix B for a more complete display of the typing rules.

This IR strikes a balance between being logically coherent and also efficient for compiler optimizations and lowering to machine code. In the next section, we will talk about how to actually lower these constructs to primitive logical theorems.

4.2 *How proof generation works*

In any automated proof, there is usually a clear delineation between two different kinds of proof work:

1. The lemmas, which are manually proved generic theorems, possibly involving specialized definitions or written in a very specific way to fit the expectations of the automation.
2. The tactic, which is a metaprogram that decides "at runtime," having seen the particular instance of interest, how to combine the lemmas in order to prove the theorem.

For example, suppose we wish to implement an evaluator for addition on unary natural numbers.

1. The lemmas are:

```
theorem add0 (a: nat): $ a + 0 = a $;
theorem addS (a b c: nat): $ a + b = c $ > $ a + S b = S c $;
```

2. The tactic is a program with the following (MMC) pseudocode:[10]

```
proc eval_add(a: nat, b: nat): c: nat, proof(a + b = c) {
  if b = 0 {
    let pr: proof(a + 0 = a) := add0(a);
    return a, pr;
  } else {
    let b := b - 1;
    let c, (h: proof(a + b = c)) := eval_add(a, b);
    let pr: proof(a + S(b) = S(c)) := addS(a, b, c, h);
    return S(c), pr;
  }
}
```

[10] Note that the `proof(a + b = c)` argument is not ghost. We are not simply trying to return a value `c` for which `a + b = c`, we are actually constructing a tree-structured proof object witnessing `a + b = c` as part of the program output.

The characteristics of this scheme are:

- The lemmas are written in a particular form: `add0` is of the form `a + b = c`, and `addS` both takes and produces lemmas of this form.[11]
- The tactic works by structural induction on `b`, following essentially the same path as a normal recursive program computing `a + b`; we output exactly one lemma for each "decision" made in the computation.

The really interesting consequence of the scheme is that there is only a linear overhead over just running the corresponding functional program:

```
data Nat = Z | S Nat

eval_add :: Nat -> Nat -> Nat
eval_add a Z = a
eval_add a (S b) =
  let c = eval_add a b in
  S c
```

Thus, as long as we can craft a functional program to calculate the desired expression, we can also construct a proof object linear in the size of the computation to lead a completely generic MM0 verifier through the same computation.[12]

[11] The normal way to state `addS` would be `a + S b = S (a + b)`, and we could still do the job by applying transitivity and congruence of equality in the tactic. However, considering that lemmas are $O(1)$ proof work while tactics represent $O(n)$ computation, we are incentivised to move as much as possible out of the tactic and into the lemmas to keep the constant factor low.

[12] The ability to construct such a proof object is not specific to MM0, of course; just about any proof language has the same property. However, the *linearity* of the proof depends heavily on subterm sharing. For example, the proof objects generated by `eval_add` reuse `a` and the subterms of `b` many times.

4.3 The assembly proof

The proof will be manipulating syntactic objects for doing calculation, so we need some preliminaries for strings and numbers:

h : `hex` ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`	hexadecimal digits
| `8` | `9` | `a` | `b` | `c` | `d` | `e` | `f`	
c : `char` ::= $h_1\ h_2$	bytes
s : `string` ::= $\cdot$ | c | $s_1\ s_2$	strings
n, i, j : `nat` ::= h | $n\ h$	hexadecimal numbers

The terms of sort `nat` are defined in the obvious way, with $n\ h$ being defined as $16n + h$, so that `ab0` denotes the value `0xab0` or 2736. Arithmetic is defined over hexadecimal numbers in this representation.

4.3.1 Global assembly

An ELF file consists of a header which is always the same, followed by a number of procedure declarations.[13] The purpose of the global

[13] Constants that have been placed in the read-only section should also be in this definition; this is future work.

assembly theorem is to parse the part of the string after the header into these pieces.

$$\begin{array}{rll} \mathcal{A} ::= & \mathcal{A}_1;\ \mathcal{A}_2 & \text{sequential composition} \\ \mid & \mathsf{proc}\ n\ \{A\} & \text{procedure declaration} \end{array}$$

The judgment $s\ @\ i..j \Rightarrow \mathcal{A}$ asm asserts that assembly $\mathcal{A}$ can be assembled into the string s (or s can be disassembled to $\mathcal{A}$ depending on perspective), at locations $i..j$ (meaning that i is the address of the beginning of the string and j is the end of the string).

Global assembly $\boxed{s\ @\ i..j \Rightarrow \mathcal{A}\ \mathsf{asm}}$

ASM-SEQ

$$\frac{s_1\ @\ n_0..n_1 \Rightarrow \mathcal{A}_1\ \mathsf{asm} \quad s_2\ @\ n_1..n_2 \Rightarrow \mathcal{A}_2\ \mathsf{asm}}{s_1\ s_2\ @\ n_0..n_2 \Rightarrow \mathcal{A}_1;\ \mathcal{A}_2\ \mathsf{asm}}$$

ASM-PROC

$$\frac{s\ @\ \mathtt{0}..n \Rightarrow A\ \mathsf{lasm} \quad i+n=j}{s\ @\ i..j \Rightarrow \mathsf{proc}\ i\ \{A\}\ \mathsf{asm}}$$

ASM-PAD

$$\frac{s\ @\ i..j \Rightarrow \mathsf{proc}\ i\ \{A\}\ \mathsf{asm} \quad \mathsf{len}(s') = n \quad j+n=j'}{s\ s'\ @\ i..j' \Rightarrow \mathsf{proc}\ i\ \{A\}\ \mathsf{asm}}$$

The ASM-PAD theorem allows us to add additional garbage at the end of any procedure declaration without tracking it in the assembly. This is used for padding purposes, because the compiler will insert padding between procedure declarations so that they start on a 16-byte aligned address.

We use a trick here to avoid repeated bounds checks. Assembly requires everything to fit within 2^{64}, which is an extremely generous head-room but is still required for these lemmas to be correct. There is nothing about the lemmas just stated that would prevent their application to assemble more than 2^{64} procedures or very large procedures; it is even possible to do this with a small proof term by utilizing definitions and subterm sharing. So $s\ @\ i..j \Rightarrow \mathcal{A}$ asm is defined such that it is trivially satisfied when $j \geq 2^{64}$, and therefore we can assume during the main assembly proof that all numbers in sight are suitably bounded.

When we are done assembling all the procedures, we do exactly one bounds check on the final number we ended up with, and then we unpack the proof to obtain facts asserting that each individual procedure is where we put it.

The global assembly extraction judgment $\mathcal{G} \vdash \mathcal{A}$ asm says that $\mathcal{A}$ is assembled in the current global context $\mathcal{G}$, defined as follows:

$$\mathcal{G} \in \text{GCtx} ::= \{\mathsf{content} := s,\ \mathsf{filesz} := i,\ \mathsf{memsz} := j,\ \mathsf{result} := R\}$$

where $\mathcal{G}.\mathsf{content}$: `string` represents the body of the file (everything after the ELF header), and $\mathcal{G}.\mathsf{filesz}$: `u64` and $\mathcal{G}.\mathsf{memsz}$: `u64` represent the two fields from the ELF header that vary between compiler-produced executables. $\mathcal{G}.\mathsf{result} \subseteq \mathtt{u8}^* \times \mathtt{u8}^*$ is the final property we wish to establish, the $T(i, o)$ function from theorem 3.3.1.

Global assembly extraction $\boxed{\mathcal{G} \vdash \mathcal{A}\ \mathsf{asm}}$

ASMD-I

$$\frac{\mathcal{G}.\mathsf{content} \mathbin{@} \mathtt{0} \mathinner{..} \mathcal{G}.\mathsf{filesz} \Rightarrow \mathcal{A}\ \mathsf{asm} \qquad \mathcal{G}.\mathsf{filesz} \le \mathcal{G}.\mathsf{memsz} \qquad \mathtt{400078} + \mathcal{G}.\mathsf{memsz} = n \qquad n < 2^{64}}{\mathcal{G} \vdash \mathcal{A}\ \mathsf{asm}}$$

ASMD-LEFT

$$\frac{\mathcal{G} \vdash \mathcal{A}_1;\ \mathcal{A}_2\ \mathsf{asm}}{\mathcal{G} \vdash \mathcal{A}_1\ \mathsf{asm}}$$

ASMD-RIGHT

$$\frac{\mathcal{G} \vdash \mathcal{A}_1;\ \mathcal{A}_2\ \mathsf{asm}}{\mathcal{G} \vdash \mathcal{A}_2\ \mathsf{asm}}$$

The value `0x400078` appearing in ASMD-I is the hard-coded entry point of the program.[14] By using these theorems, we obtain a proof of $\mathcal{G} \vdash \mathsf{proc}\ p\ \{A\}\ \mathsf{asm}$ for each procedure in the program, which is the form we need for section 4.4.

[14] The ELF format lets you specify the entry point of the program, and the way we construct it the entry point always comes immediately after the header (which is `0x78` bytes). So the whole ELF file is being mapped starting at `0x400000` (we can't start too close to zero, and highly aligned pages are preferable for the loader).

4.3.2 *Local assembly*

We have to fill in the details on A and the lasm judgment from the previous section. Inside a procedure, we use local addressing: the $s \mathbin{@} p,\ i..j \Rightarrow A\ \mathsf{lasm}$ judgment is similar to $s \mathbin{@} i..j \Rightarrow \mathcal{A}\ \mathsf{asm}$ but asserts that the string is assembled at $p + i \mathinner{..} p + j$ instead, and A may also depend on p and i separately.[15] Local assembly sequences have a similar definition to global assembly:

[15] This is a slight simplification: in the real version the judgment also keeps track of whether s is empty or nonempty to avoid generating useless $\cdot + s$ expressions.

$A ::= A_1;\ A_2$	sequential composition
$\mid A \mathbin{@} i$	start a basic block
$\mid \mathsf{entry}(p, A)$	start the entry block
$\mid I$	instructions

Local assembly $\boxed{s \mathbin{@} p,\ i..j \Rightarrow A\ \mathsf{lasm}}$

LASM-SEQ

$$\frac{s_1 \mathbin{@} p,\ n_0..n_1 \Rightarrow A_1\ \mathsf{lasm} \qquad s_2 \mathbin{@} p,\ n_1..n_2 \Rightarrow A_2\ \mathsf{lasm}}{s_1\ s_2 \mathbin{@} p,\ n_0..n_2 \Rightarrow A_1;\ A_2\ \mathsf{lasm}}$$

LASM-AT

$$\frac{s \mathbin{@} p,\ i..j \Rightarrow A \ \mathsf{lasm}}{s \mathbin{@} p,\ i..j \Rightarrow A \mathbin{@} i \ \mathsf{lasm}}$$

LASM-ENTRY

$$\frac{s \mathbin{@} p,\ \mathtt{0}..j \Rightarrow A \ \mathsf{lasm}}{s \mathbin{@} p,\ \mathtt{0}..j \Rightarrow \mathsf{entry}(p, A) \ \mathsf{lasm}}$$

LASM-INST

$$\frac{s \mathbin{@} p, j \Rightarrow I \ \mathsf{inst} \quad \mathsf{len}(s) = n \quad i + n = j}{s \mathbin{@} p,\ i..j \Rightarrow I \ \mathsf{lasm}}$$

Local assembly extraction $\boxed{\mathcal{G} \vdash A \ \mathsf{lasm}}$

LASMD-I

$$\frac{\mathcal{G} \vdash \mathsf{proc}\ n\ \{A\}\ \mathsf{asm}}{\mathcal{G} \vdash A \ \mathsf{lasm}}$$

LASMD-LEFT

$$\frac{\mathcal{G} \vdash A_1;\ A_2 \ \mathsf{lasm}}{\mathcal{G} \vdash A_1 \ \mathsf{lasm}}$$

LASMD-RIGHT

$$\frac{\mathcal{G} \vdash A_1;\ A_2 \ \mathsf{lasm}}{\mathcal{G} \vdash A_2 \ \mathsf{lasm}}$$

$A \mathbin{@} i$ is used at the start of each basic block to record the current value of the local address so that it can be targeted by jumps inside the function, and $\mathsf{entry}(p, A)$ does the same thing but with the global address, to record the address of the entry block so that it can be targeted by call instructions in other functions.

The body of each block is also a local assembly sequence A, so that we can reuse LASM-SEQ for concatenating instructions, and we are left only with assembling individual instructions.

4.3.3 *Assembling instructions*

Assembling x86 instructions involves many lemmas because each lemma corresponds to one kind of instruction and x86 is a large and complicated architecture. We will show just a few of the rules.

$$\begin{aligned} I &::= \mathit{binop.sz}\ \mathit{dst}\ \mathit{src} \mid \mathsf{call}\ f \mid \mathsf{jump}\ \mathit{tgt} \mid \mathsf{ret} \mid \dots \\ \mathit{binop} &::= \mathsf{add} \mid \mathsf{xor} \mid \dots \\ \mathit{dst}, \mathit{src} &::= \mathit{reg} \mid \mathsf{M}[\mathit{si} \cdot \mathit{base} + \mathit{off}] \mid \mathsf{imm32}\ n \mid \mathsf{imm64}\ n \end{aligned}$$

Instruction parse (REX byte) $\boxed{s \mathbin{@} p, ip \Rightarrow I \ \mathsf{inst}}$

INST-REX

$$\frac{s \mathbin{@} p, ip, r \Rightarrow I \ \mathsf{inst}}{(4\ r)\ s \mathbin{@} p, ip \Rightarrow I \ \mathsf{inst}}$$

INST-NO-REX

$$\frac{s \mathbin{@} p, ip, \cdot \Rightarrow I \ \mathsf{inst}}{s \mathbin{@} p, ip \Rightarrow I \ \mathsf{inst}}$$

This handles the parsing of the REX byte, which can either be `0x4`r or omitted and affects the meaning of the rest of the instruction.

Instruction parse (core) $\boxed{s \mathbin{@} p, ip, rex^? \Rightarrow I \ \mathsf{inst}}$

INST-RET

$$\mathtt{c3} \mathbin{@} p, ip, rex^? \Rightarrow \mathsf{ret}\ \mathsf{inst}$$

This is the core judgment. Some instructions like INST-RET have a trivial parse (they are indicated by the byte `c3` and there are no arguments), but most instructions involve several subroutines, like this one:

INST-BINOP-IMM

$$\frac{\begin{array}{c}\mathsf{split}_{1,3}(y) = v, \mathtt{0} \quad \mathsf{opSizeW}(rex^?, v) = sz \\ \mathsf{parseModRM}(rex^?, s) \Rightarrow opc, \mathsf{reg}\ dst, s' \\ \mathsf{parseImm}(sz, s') \Rightarrow src \qquad \mathsf{parseBinop}(opc, sz, dst, \mathsf{imm32}\ src) \Rightarrow I\end{array}}{(\mathtt{8}\ y)\ s\ @\ p, ip, rex^? \Rightarrow I\ \mathsf{inst}}$$

This describes the parsing of a binary operation with an immediate argument. For example, `add rax, 1` would go through this lemma. The way this lemma is "executed" by the tactic is as follows:

- We are given the string s and $p, ip, rex^?$ as inputs, and want to produce I as output. (We also have I itself stored as a PCode instruction, which is useful to influence the choice of I when multiple instructions can be read from the same bytes.)
- We match s to the form $(\mathtt{8}\ y)\ s$ to get y and the substring s. (In other words, this rule applies when the opcode byte is `0x8`y for some hex digit y.)
- Call $\mathsf{split}_{1,3}(y) = v, z$ and assert that $z = \mathtt{0}$. This takes y which is a hex digit and splits it into digits of size 1 and 3 bits respectively. For example, $\mathsf{split}_{1,3}(\mathtt{d}) = \mathtt{1}, \mathtt{5}$ because `d` is `1101` in binary so the high bit is 1 and the low three bits are $\mathtt{101} = \mathtt{5}$. (This kind of bit packing and unpacking is common in the assembler because many values are bit-packed into the instruction bytes.)
- Call $\mathsf{opSizeW}(rex^?, v) = sz$. This is parsing the size of the instruction operands from the v argument and the REX byte.
- Call $\mathsf{parseModRM}(rex^?, s)$ to get opc, dst', s' and assert that $dst' = \mathsf{reg}\ dst$. This is parsing the next byte after the opcode byte, known as the Mod/RM byte in the Intel specification. It returns two fields, where in this case the first field is opc which encodes the binop, and dst must be a register for this form of the instruction (we are encoding a binary operation between a destination register and an immediate). The string s' is the remainder of the instruction after the Mod/RM byte and possible continuation bytes are parsed.
- Call $\mathsf{parseImm}(sz, s')$ to get src. This just reads all the remaining bytes in the instruction into the number src.
- Call $\mathsf{parseBinop}(opc.sz\ dst\ (\mathsf{imm32}\ src))$ to get the final instruction I. We will show this judgment below; this is mainly just assembling the binop.

Most instructions go along similar lines as this one.

BINOP-BINOP

$$\mathsf{parseBinop}(\mathit{binop.sz}\ \mathit{dst},\ \mathit{src}) \Rightarrow \mathit{binop.sz}\ \mathit{dst}\ \mathit{src}$$

BINOP-CLEAR-32

$$\mathsf{parseBinop}(\mathsf{xor.32}\ \mathit{dst}\ \mathit{dst}) \Rightarrow \mathsf{imm.32}\ \mathit{dst}\ 0$$

BINOP-CLEAR-64

$$\mathsf{parseBinop}(\mathsf{xor.32}\ \mathit{dst}\ \mathit{dst}) \Rightarrow \mathsf{imm.64}\ \mathit{dst}\ 0$$

This is a nondeterministic judgment which allows interpreting the bytes that normally would encode $\mathsf{xor.32}\ \mathit{dst}\ \mathit{dst}$ as either a 32 or 64 bit immediate move of the value 0 into a register. Read in reverse, it means that if the compiler has an $\mathsf{imm.64}\ \mathit{dst}\ 0$ it wants to encode, it produces the encoding of $\mathsf{xor.32}\ \mathit{dst}\ \mathit{dst}$ instead and hints to the assembler tactic that it wants it to be interpreted as $\mathsf{imm.64}\ \mathit{dst}\ 0$.

The instructions for jumping and calling are interesting for making use of the address parameters:

INST-CALL

$$\frac{p + ip = a \quad tgt -_{\mathbb{Z}} a = imm \quad \mathsf{parseImm32}(s) \Rightarrow imm}{\mathsf{e8}\ s\ @\ p, ip, rex^{?} \Rightarrow \mathsf{call}\ tgt\ \mathsf{inst}}$$

- First, we compute $p + ip = a$
- Call $\mathsf{parseImm32}(s)$ to get imm
- We compute $imm + a = tgt$ but prove $tgt -_{\mathbb{Z}} a = imm$ (note that imm can be positive or negative).
- The assembled instruction is $\mathsf{call}\ tgt$.

The semantics of the call instruction are that it offsets the current value of the instruction pointer by imm, so since the current instruction is at $p + ip$ and the absolute address of the target function is tgt, we want $tgt - a$ to be stored in the instruction.

INST-JUMP-8

$$\frac{tgt -_{\mathbb{Z}} ip = imm \quad \mathsf{parseImm8}(s) \Rightarrow imm}{\mathsf{eb}\ s\ @\ p, ip, rex^{?} \Rightarrow \mathsf{jump}\ tgt\ \mathsf{inst}}$$

jump is similar to call except that tgt is now a procedure-local address, the address of some other basic block in the function, so we do not need to involve p as both ip and tgt are local and their difference is imm which must be a signed 8 bit number. (There is another variant of this instruction for 32 bit offsets.)

We can also assemble ghost instructions using this approach. For example, the compiler will produce "fallthrough" terminators on basic blocks that jump to the immediately following block, which does not require any code to happen explicitly. But for uniformity of reasoning it is convenient to put a jump instruction there anyway, and this is done by adding another rule to assemble an instruction to the empty string:

$$\frac{\textsc{lasm-fallthrough}}{\cdot \mathbin{@} p, ip..ip \Rightarrow \mathsf{jump}\ ip\ \mathsf{lasm}}$$

When all is said and done, what we get out of this part of the proof is a theorem of the form $\mathcal{G} \vdash \mathsf{proc}\ i\ \{A\}\ \mathsf{asm}$ for each procedure, proved in the order these functions appear in the assembly. In the next section we will discuss the second and much larger part of the proof, where we take these assembly proofs and prove Hoare triples for each function (in the call graph dependency order).

4.4 The correctness proof

Warning: This is future work. Not all aspects of this part of the project are completely worked out.

The correctness proof is essentially the output of a proof-producing MIR typechecker, so to understand it we need to know the core concepts of MIR, starting with the contexts.

4.4.1 The type context

There are no less than five different kinds of contexts that play a role in the main part of the proof. The first three are "read-only" state, corresponding to the Γ in the $\Gamma, \delta \vdash e : \tau \dashv \delta'$ type judgment we mentioned in section 3.3.3.

- $\mathcal{G} \in$ GCtx: the global context. This contains information which is global to the entire executable, and every theorem depends on it.
  - $\mathcal{G}.\mathsf{content} : \mathtt{u8}^*$ – the ELF file data after the header
  - $\mathcal{G}.\mathsf{filesz} : \mathtt{u64}$ – the size of the ELF file
  - $\mathcal{G}.\mathsf{memsz} : \mathtt{u64}$ – the "size of the file in memory." This is generally larger than $\mathcal{G}.\mathsf{filesz}$, with the difference between the two being the number of bytes of read-write zeroed data for global variables.
  - $\mathcal{G}.\mathsf{result} \subseteq \mathtt{u8}^* \times \mathtt{u8}^*$ – the final property we wish to establish, the $T(i, o)$ function from theorem 3.3.1.
- $\mathcal{P} \in$ PCtx: The procedure context. PCtx extends GCtx with additional fields which are global to a specific procedure.
  - $\mathcal{P}.\mathsf{ret}$: Return – the return ABI, i.e. the number and types of return values and what registers they should be put in

  - $\mathcal{P}.\mathsf{epi}$: Epilogue – the epilogue sequence that is required to compensate for the prologue at the beginning of the function
  - $\mathcal{P}.\mathsf{se}$: `bool` – true if this function is permitted to perform side-effects

- $\mathcal{B} \in$ BCtx: The block context. BCtx extends PCtx with additional fields which are global to a specific basic block.
  - $\mathcal{B}.\mathsf{labs}$: LabelGroup* – the list of label groups that are legal to jump to at this point. This is used for back edges which correspond to `while` or `label` loops in the source code.

The δ in $\Gamma,\ \delta \vdash e : \tau \dashv \delta'$ is the flow-sensitive type context $\mathcal{T} \in$ TCtx, which changes between individual instructions and tracks the current values of variables. It consists of two parts, $\mathcal{T} ::= (\mathcal{V}, \mathcal{M})$, defined as:

- $\mathcal{V} \in$ VCtx: the variable context, which contains the information about which variables exist and their types and values. This closely resembles the type context from the user level view of the proof.

- $\mathcal{M} \in$ MCtx: the machine context, which keeps track of the values currently in specific registers or stack slots in the current stack frame. This is not user-visible, and primarily consists of information derived from register allocation. (So if the register allocator library was buggy, we would notice when working through the proof here.)

The variable context $\mathcal{V}$ has the structure:

$$\begin{aligned} V &::= V_1 * V_2 \mid \mathsf{v}_i : \tau \mid \tau \mid \cdot \\ \mathcal{V} &::= \{\mathsf{tree} := V, \mathsf{size} := n\} \end{aligned}$$

That is, there is a tree of $*$ applications (associated in an unspecified way) and at the leaves we have variable declarations $\mathsf{v}_i : \tau$, as well as anonymous declarations τ for simple (separating) propositional types. The $\mathcal{V}.\mathsf{size}$ field tracks the number of declared variables, and the v_i declarations in the tree all have distinct indices.

The machine context similarly has a tree structure:

$$\begin{aligned} \mathcal{M} ::= {} & (\mathcal{M}_1,\ \mathcal{M}_2) \mid \cdot \\ & \mid \mathsf{reg}_i \mapsto e \mid \mathsf{reg}_i \mapsto - \mid \mathsf{S}[i..j] \mapsto e \mid \mathsf{S}[i..j] \mapsto - \end{aligned}$$

This tracks the registers that are in use: they may be either storing an expression, or they may be "clobbered," i.e. they contain an arbitrary value but are available for use. Registers not listed in the context must not be touched. Similarly, stack slots can either be storing a value or they may be uninitialized.

4.4.2 *Block structure*

As mentioned, the proof structure is driven by the MIR representation of the program, which uses a CFG (control-flow graph) consisting of basic blocks which call each other. In the proof, this corresponds to a proof structure with lemmas proving $\mathcal{B} \vdash \mathsf{block}(\mathcal{T})\ @\ n$ for each possible jump target n.

Block correctness $\boxed{\mathcal{B} \vdash \mathsf{block}(\mathcal{T})\ @\ n}$

$$\textsc{block-i}\quad \frac{\mathcal{B} \vdash A\ @\ n\ \mathsf{lasm} \qquad \mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\bot\}}{\mathcal{B} \vdash \mathsf{block}(\mathcal{T}')\ @\ n} \qquad \textsc{block-weak}\quad \frac{\mathcal{B} \vdash \mathsf{block}(\mathcal{T})\ @\ n \qquad \mathcal{B} \vdash \{\mathcal{T}\} \cdot \{\mathcal{T}'\}}{\mathcal{B} \vdash \mathsf{block}(\mathcal{T}')\ @\ n}$$

The main rule is the first one, which uses the proof of $\mathcal{G} \vdash A\ \mathsf{lasm}$ from the previous section to get the assembly of a block and start proving it using the $\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}$ code correctness judgment. Here $\bot \in$ TCtx is the type context defined as $(\texttt{false}, \cdot)$, an impossible state. This expresses the fact that every basic block ends in a "terminator" which performs some kind of jump, so we do not fall off the end of the block.[16]

[16] Recall that even if this is a "fallthrough" block, we still put a ghost jump *tgt* instruction at the end, pointing to the next block.

The BLOCK-WEAK rule allows us to "weaken" the context using the idiom $\mathcal{B} \vdash \{\mathcal{T}\} \cdot \{\mathcal{T}'\}$, again using the $\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}$ judgment but with an empty instruction sequence. This holds if $\mathcal{T}'$ is logically entailed by $\mathcal{T}$, and it is useful when merging control flow into a block from a type context that does not exactly match.

Code correctness $\boxed{\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}}$

$$\textsc{code-0}\quad \mathcal{B} \vdash \{\bot\}\ A\ \{\bot\} \qquad \textsc{code-id}\quad \mathcal{B} \vdash \{\mathcal{T}\} \cdot \{\mathcal{T}\} \qquad \textsc{code-seq}\quad \frac{\mathcal{B} \vdash \{\mathcal{T}_0\}\ A_1\ \{\mathcal{T}_1\} \qquad \mathcal{B} \vdash \{\mathcal{T}_1\}\ A_2\ \{\mathcal{T}_2\}}{\mathcal{B} \vdash \{\mathcal{T}_0\}\ (A_1;\ A_2)\ \{\mathcal{T}_2\}}$$

This is the main judgment for proving properties of code fragments, and it has the general structure of a Hoare triple so we borrow the notation, although one should keep in mind that $\mathcal{T}$ is not literally a separating proposition, it is a TCtx (which encodes a separating proposition). Also A is not code at the source level, it is a sequence of assembly instructions.

As an aside, let us break down the actual *definition* of the predicate $\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}$. Recall that these rules like CODE-0 and CODE-ID are not true by definition; they are lemmas to be proved about a specific predicate. The reason we present the lemmas first and foremost is because they are the clauses used by the tactic, and the predicate is chosen to make the lemmas true and minimize the number of side conditions required in the lemmas. (For instance, the fact that CODE-0 has no assumptions implies that $\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}$ does not imply that $\mathcal{B}$ or A is well formed.)

$$\begin{aligned}&(\mathcal{B} \vdash \{\mathcal{T}\}\ A\ \{\mathcal{T}'\}) := \\ &\quad \forall p\ x\ y\ R.\ \mathsf{lasm}_{\mathcal{B}}(A, p, x..y)\ \wedge\ \mathsf{okScope}(\mathcal{B}, p, R)\ \to \\ &\qquad \mathsf{okT}_{\mathcal{B}}(\mathcal{T}', p+y, R)\ \to\ \mathsf{okT}_{\mathcal{B}}(\mathcal{T}, p+x, R)\end{aligned}$$

The definitional stack leading to this is somewhat deep so we won't peer into every part of this, but it is essentially a more elaborate version of the definition from section 3.3.4:

$$\{P\} \cdot \{Q_1, \ldots, Q_n\} := \forall R.\ \mathsf{ok}(Q_1 * R) \wedge \cdots \wedge \mathsf{ok}(Q_n * R) \to \mathsf{ok}(P * R)$$

The parts of this definition are:

- $\mathcal{B}$ is the block context, which relevantly contains the return information $\mathcal{B}.\mathsf{rets}$ and the label groups $\mathcal{B}.\mathsf{labs}$.
- A is the local assembly sequence, containing a sequence of instructions.
- $\mathcal{T}$ is the precondition and $\mathcal{T}'$ is the postcondition type context.
- p is the concrete procedure location, which has been abstracted in the definition so that explicit references to it are not needed.
- x is the address of the beginning of the assembly code A (relative to p), and y is the address of the end.
- R is the frame proposition, which ends up conjoined to all the ok assumptions as with $\mathsf{ok}(Q_i * R)$ in the simple version.
- $\mathsf{lasm}_{\mathcal{B}}(A, p, x..y)$ asserts that $s := \mathcal{B}.\mathsf{content}[p+x..p+y]$ is a string of bytes which assembles to $A\ @\ p, x$. (The definition of $s \in A\ @\ p, x$ depends on A, but proving facts about this relation was essentially the purpose of the $(s, p,\ i..j \Rightarrow A\ \mathsf{lasm})$ judgment from section 4.3.1.)
- $\mathsf{okT}_{\mathcal{B}}(\mathcal{T}, ip, R)$ asserts that if the instruction pointer is at ip, then the state is *valid* in the sense of section 3.3.4. We will break this proposition down some more below.
- $\mathsf{okScope}(\mathcal{B}, p, R)$ asserts that exiting via return is valid, and also exiting via one of the labels in the label groups is valid. Roughly:

  $$\begin{aligned}&\mathsf{okScope}(\mathcal{B}, p, R) := \\ &\quad \mathsf{ok}_{\mathcal{B}}(\mathsf{return}(\mathcal{B}) * R)\ \wedge\ \forall l \in \mathcal{B}.\mathsf{labs}.\ \mathsf{okT}_{\mathcal{B}}(\mathcal{T}, p+l, R),\end{aligned}$$

  where $\mathsf{return}(\mathcal{B})$ is a separating proposition asserting that the program has just returned from the function and values are in memory in accordance with the calling convention $\mathcal{B}$.ret. We will discuss label groups more in section 4.6.

The proposition $\mathsf{okT}_{\mathcal{B}}(\mathcal{T}, ip, R)$ defines the "main layout" of the stack frame. It has the following structure:

$$\begin{aligned}
&\mathsf{okT}_{\mathcal{B}}(\mathcal{T}, ip, R) := \mathsf{ok}_{\mathcal{B}}(\\
&\quad \mathsf{TEXT\_START} \mapsto \mathcal{B}.\mathsf{content} \ * \\
&\quad \mathsf{RIP} \mapsto \mathsf{TEXT\_START} + ip \ * \\
&\quad \mathsf{exception} \mapsto \mathsf{OK} \ * \\
&\quad (\exists f.\ \mathsf{flags} \mapsto f) \ * \\
&\quad (\exists sp.\ \mathsf{RSP} \mapsto sp \ * \ \mathsf{stackLayout}_{\mathcal{B}}(\mathcal{T}, sp)) \ * \\
&\quad \mathcal{T}.\mathsf{prop} \ * \ R)
\end{aligned}$$

This says that the code of the executable $\mathcal{B}$.content is loaded in memory at TEXT_START := `0x400078`, the instruction pointer is at the specified instruction, the exception flag is off (this is used for syscalls and processor signals), the flags are clobbered, the stack is set up according to the stack frame layout specified by $\mathcal{T}$, and any additional separating propositions in $\mathcal{T}$ (the variables) are assumed, and finally the frame proposition R as before. This is all wrapped in $\mathsf{ok}_{\mathcal{B}}(-)$ which means that any state satisfying this separating proposition will successfully (according to $\mathcal{B}$.result) execute to completion.

4.5 *Executing statements*

Some instructions, like register-register moves, are quite simple and just copy the expression v:

CODE-MOV-RR

$$\frac{\mathsf{read}(\mathcal{T}, \mathsf{reg}\ src) \Rightarrow v \qquad \mathsf{write}(\mathcal{T}, \mathsf{reg}\ dst, v) \Rightarrow \mathcal{T}'}{\mathcal{B} \vdash \{\mathcal{T}\}\ (\mathsf{mov.64}\ dst\ src)\ \{\mathcal{T}'\}}$$

These instructions are inserted by the register allocator, so we don't really analyze them too much, we just update the state as specified and if the register allocator has no bugs then the expressions we require will be in the appropriate register when we call read later to retrieve the value, and if not the proof construction will fail.

A more complex instruction is jcc *cond tgt* "jump to *tgt* if condition flags *cond* are set" which is used to implement `if` statements:

CODE-JCC

$$\frac{\mathsf{insert}(\mathcal{T}, \tau) \Rightarrow \mathcal{T}_1 \qquad \mathsf{insert}(\mathcal{T}, \neg\tau) \Rightarrow \mathcal{T}_2 \qquad \mathsf{flagCond}(f, cond) \Rightarrow \tau \qquad \mathcal{B} \vdash \mathsf{block}(\mathcal{T}_1) \mathbin{@} tgt}{\mathcal{B} \vdash \{\mathsf{withFlag}(f, \mathcal{T})\}\ (\mathsf{jcc}\ cond\ tgt)\ \{\mathcal{T}_2\}}$$

The $\mathsf{insert}(\mathcal{T},\tau)$ and $\mathsf{insert}(\mathcal{T},\neg\tau)$ calls construct the extended type context inside the branches of the if statement. The $\mathsf{withFlag}(f,\mathcal{T})$ contains an assignment of the flags f (overriding the "flags clobbered" in $\mathsf{okT}_{\mathcal{B}}$ we saw above), and the immediately prior instruction will be a cmp or other flag-setting command. $\mathsf{flagCond}(f,\mathit{cond})$ decodes this into an MMC type τ, which is pushed into the type contexts. If the condition is true, we jump to $\mathcal{B} \vdash \mathsf{block}(\mathcal{T}_1)\ @\ \mathit{tgt}$ and we should have previously proved that this block is safe to jump to; otherwise we continue and reach the postcondition $\mathcal{T}_2$.

4.6 *Label groups and proof by induction*

A label group is a collection of basic blocks which can call each other. For example, the following MMC program:

```
proc example() {
  let a := 1;
  label lab1(x: nat, y: nat, variant x + y) {
    lab2(0, variant assert(0 + a < x + y))
  }
  label lab2(z: nat, variant z + a) {
    lab1(1, 2, variant assert(1 + 2 < z + a))
  }
  if a = 1 {
    return;
  } else {
    lab2(1)
  }
}
```

compiles to the MIR shown in Figure 4.1.

```
proc example {
  entry():
    let a := 1;
    label_group(lab1, lab2);
    if a = 1 { goto iftrue(a); } else { goto iffalse(a); }
  lab1(a: nat, x: nat, y: nat, variant x + y):
    let h := assert(0 + a < x + y);
    goto lab2(a, 0, variant h);
  lab2(a: nat, z: nat, variant z + a):
    let h := assert(1 + 2 < z + a);
    goto lab1(a, 1, 2, variant h);
  iftrue(a: nat):
    return;
  iffalse(a: nat):
    goto lab2(a, 1);
}
```

Figure 4.1: The MIR translation of the `example()` function. The main difference is that `if` statements must jump to a new basic block, so there are two new explicit blocks `iftrue` and `iffalse`, and every block has `a: nat` in scope, which translates to a block parameter. (Technically, there would be additional lines to evaluate the subterms $0 + a < x + y$ and $a = 1$, since MIR only has elementary operations.) See Figure 4.2 for the translation of this function to logic.

1.	$\mathsf{ok}(\texttt{return})$	premise
2.	$\forall a.\ \mathsf{ok}(\texttt{iftrue}(a))$	fwd. sim. 1
3.	$\forall a\ x\ y.\ x+y<n \to \mathsf{ok}(\texttt{lab1}(a,x,y))$	assumption
4.	$\forall a\ z.\ z+a<n \to \mathsf{ok}(\texttt{lab2}(a,z))$	assumption
5.	$x+y=n$	assumption
6.	$0+a<x+y \to \mathsf{ok}(\texttt{lab2}(a,0))$	subst. 4, 5
7.	$\mathsf{ok}(\texttt{assert}(\cdots);\ \texttt{lab2}(a,0))$	fwd. sim. 6
8.	$\forall a\ x\ y.\ x+y=n \to \mathsf{ok}(\texttt{lab1}(a,x,y))$	fwd. sim. 5–7
9.	$z+a=n$	assumption
10.	$1+2<z+a \to \mathsf{ok}(\texttt{lab1}(a,1,2))$	subst. 3, 9
11.	$\mathsf{ok}(\texttt{assert}(\cdots);\ \texttt{lab2}(a,1,2))$	fwd. sim. 10
12.	$\forall a\ z.\ z+a=n \to \mathsf{ok}(\texttt{lab2}(a,z))$	fwd. sim. 5–7
13.	$\forall a\ x\ y.\ \mathsf{ok}(\texttt{lab1}(a,x,y))$	induct. on n: 8, 12
14.	$\forall a\ z.\ \mathsf{ok}(\texttt{lab2}(a,z))$	
15.	$\mathsf{ok}(\texttt{lab2}(a,1))$	subst 14
16.	$\forall a.\ \mathsf{ok}(\texttt{iffalse}(a))$	fwd. sim. 15
17.	$\mathsf{ok}(\texttt{if}\ldots)$	fwd. sim. 2, 16
18.	$\mathsf{ok}(\texttt{let a; if}\ldots)$	fwd. sim. 17
19.	$\mathsf{ok}(\texttt{entry}())$	fwd. sim. 18

Figure 4.2: Translation of the MIR code Figure 4.1 into a proof by induction that the `entry()` label is safe to execute. The proof goes in reverse execution order, but the proof is constructed from the bottom up: the steps in this proof would be derived in the order 19, 18, (13+14+3+4, (8+5, 7, 6), (12+9, 11, 10)), 17, (2, 1), (16, 15), where parentheses denote subroutine calls.
The main proof step is "forward simulation," or executing the effect of a line of code and determining the next sufficient condition. When applying functions, substitution is used, and when `label_group` is encountered an induction on n is used.

The challenge is to find an ordering of the blocks such that we can prove that we are safe from the beginning of that block until the end of the program. This is true by assumption for the `return`, so we can prove `iftrue()` is safe, and if we knew that `iffalse()` is safe then we could deduce that `entry()` is safe which is the goal. But to prove `iffalse()` is safe we need to know `lab2()` is safe, which requires `lab1()` to be safe, which requires `lab2()` to be safe again. In short, cycles in control flow translate to circularity in the proof, which makes sense since this is a proof of termination and circularity can certainly lead to nontermination.

This is where the `label_group` statement comes in. This appears in the middle of the `entry()` block, at the location where the original label sequence was declared. A label cannot be called until it is put in scope by the `label_group` command. When we are generating the proof of `entry()` (in program order), and we encounter this statement, we prove `lab1()` and `lab2()` safe as a lemma, by induction on the variant (see Figure 4.2). Then when we reach the branch, we recurse to prove `iftrue()` and `iffalse()`, and we will have the safety of `lab2()` available as required.

While loops are also handled in the same way, because they compile down to label groups of size 1.

Recursive functions are currently not supported but the exact same compilation technique also works for them: each strongly connected component of the call graph is proved to be correct by induction on the variant, and the context holds a collection of function correctness assumptions asserting that calling functions in the group at smaller values of the variant is valid.

4.7 *Tying it all together*

The "entry point" of the correctness proof is the final theorem statement, which is proved by the following rule:

Program correctness $\boxed{\mathsf{okProg}(elf, T)}$

OK-PROG

$$\frac{\mathcal{G} \vdash \mathsf{okStart} \qquad \mathsf{u64Str}(\mathcal{G}.\mathsf{filesz}) \Rightarrow fs \qquad \mathsf{u64Str}(\mathcal{G}.\mathsf{memsz}) \Rightarrow ms}{\mathsf{okProg}(\mathsf{ELF\_lit}(fs, ms, \mathcal{G}.\mathsf{content}),\ \mathcal{G}.\mathsf{result})}$$

Here $\mathsf{okProg}(elf, T)$ is defined as:

$$\mathsf{okProg}(elf, T) := \mathsf{isBasicELF}(elf)\ \wedge\ \forall s \in \mathsf{init}(elf).\ \mathsf{valid}_T(s)$$

which is exactly the statement of theorem 3.3.1. It is applied to $\mathsf{ELF\_lit}(fs, ms, \mathcal{G}.\mathsf{content})$ which constructs the ELF file using fs and ms (which are fields in the header) and $\mathcal{G}.\mathsf{content}$ (the body of the file).

It depends on okStart, which assembles the `_start()` routine, the compiler-provided entry point that calls `main()`. This function is responsible for setting up all global variables and calling the `exit()` syscall after `main()` returns.[17]

[17] Currently, users cannot directly call `exit(0)` to exit early with success, although they can exit with failure at any time using `assert(false)`. It would be possible to expose `exit(0)` provided that the user proves $\mathcal{G}$.result on exit, but this would require forward-declaring the type of `main`.

Start procedure correctness $\boxed{\mathcal{G} \vdash \mathsf{okStart}}$

$$\mathsf{startPCtx}(\mathcal{G}) \in \mathrm{PCtx} := \{\mathcal{G},\ \mathsf{ret} := \bot,\ \mathsf{epi} := \bot,\ \mathsf{se} := \top\}$$
$$\mathsf{mkBCtx}(\mathcal{P}) \in \mathrm{BCtx} := \{\mathcal{P},\ \mathsf{labs} := \emptyset\}$$

OK-START

$$\frac{\mathcal{G} \vdash \mathsf{proc}\ \mathtt{0}\ \{A\}\ \mathsf{asm} \qquad \mathsf{startTCtx}(\mathcal{G}) \Rightarrow \mathcal{T} \qquad \mathsf{mkBCtx}(\mathsf{startPCtx}(\mathcal{G})) \vdash \{\mathcal{T}\}\ A\ \{\bot\}}{\mathcal{G} \vdash \mathsf{okStart}}$$

Regular procedures have a calling convention $\mathcal{F} \in$ CallConv with the following fields:

- $\mathcal{F}.\mathsf{args}$ – the specification of the function arguments (as given by the user).
- $\mathcal{F}.\mathsf{mctx}$ – the calling convention for the function arguments, i.e. which arguments are in which registers (specified by the architecture).
- $\mathcal{F}.\mathsf{ret}$ – the return convention, including both the calling convention and the user type.
- $\mathcal{F}.\mathsf{clob}$ – the function clobbers, i.e. which registers' values will be affected by the call. This is also mostly specified by the architecture.
- $\mathcal{F}.\mathsf{se}$ – true if the function is permitted to perform side-effects, i.e. `proc` vs `func` in the source language.

Regular procedure correctness $\boxed{\mathcal{G} \vdash \mathsf{proc}\ f(\mathcal{F})}$

OK-START

$$\frac{\begin{array}{c} \mathcal{G} \vdash \mathsf{proc}\ f\ \{A_{prol};\ A_{body}\}\ \mathsf{asm} \\ \mathsf{buildArgs}(\mathcal{F}.\mathsf{args}) \Rightarrow \mathcal{V} \qquad \mathsf{buildClob}(\mathcal{F}.\mathsf{clob}, \mathcal{F}.\mathsf{mctx}) \Rightarrow \mathcal{M} \\ \mathsf{buildPrologue}(\mathcal{M}, A_{prol}) \Rightarrow \mathcal{M}', epi \\ \mathcal{P} := \{\mathcal{G},\ \mathsf{ret} := \mathcal{F}.\mathsf{ret},\ \mathsf{epi} := epi,\ \mathsf{se} := \mathcal{F}.\mathsf{se}\} \\ \mathsf{mkBCtx}(\mathcal{P}) \vdash \{(\mathcal{V}, \mathcal{M}')\}\ A_{body}\ \{\bot\} \end{array}}{\mathcal{G} \vdash \mathsf{proc}\ f(\mathcal{F})}$$

This rule handles a bunch of repackaging of the function start sequence and the way it relates to the declared calling convention $\mathcal{F}$, which contains all the information needed for later calls to this function. It loads the function arguments to create the initial variable context $\mathcal{V}$, then loads the argument calling convention and the clobbers into $\mathcal{M}$ and executes the prologue sequence A_{prol} which computes a new machine context $\mathcal{M}'$ as well as an epilogue sequence epi which will undo the effect of the prologue, and finally it creates a procedure context and executes the body of the function A_{body} on the initial state.

This completes the overall structure of the proof. There are undeniably many intricate details in such a proof, because handling a real language on a real instruction set involves a lot of complexity. But hopefully this has at least made it clear how it would be possible to compile a non-toy language down to machine code and produce proofs along the way. In the next section, we will take a step back and consider some of the consequences of the proof method.

4.8 *Meta-analysis of the proof*

4.8.1 *Infinite sets in PA*

One aspect that we have not touched on very much is the fact that the entire proof is being conducted in Peano Arithmetic (PA). Every single definition we have shown so far is defined, and every lemma is proved, in PA. This has one major limitation, which is that there is limited ability to talk about infinite sets. This is especially interesting considering that many of the notions involve infinite sets:

- Values: `nat`
- Valuations: `nat` – finite functions mapping variable indices to values
- Expressions: `set` – these are functions from valuations to values
- Heaps: `nat` – these are finite partial functions from places to values
- Separating propositions: `set` – these are sets of heaps
- Types: `set` – these are functions from valuations to values to separating propositions
- Variable context, Machine context: `set` – function from valuations to separating propositions
- Global context: `set` – contains $\mathcal{G}$.result which is a set of input-output pairs
- Procedure context, Block context: `set` – contain many infinite components

Here we think of `nat` as $\mathbb{N}$ and `set` as $2^{\mathbb{N}}$, so anything involving an infinite set has sort `set` and anything from a countable domain has sort `nat`. Some things are not just finite but also bounded; for example machine states can be bounded by approximately $2^{2^{64}}$ since the machine itself is a bunch of bytes and the address space is limited. Recognizing such things is useful because the powerset of an unbounded set of naturals is a `set` but a bounded set is a `nat`. The powerset of a `set` cannot be constructed.

Even using PA to model `set` at first seems impossible, because it is just first order logic over $\mathbb{N}$, we do not have second order variables. The trick we use to resolve this is the same one used in Metamath for working with proper classes in ZFC. We introduce a sort `set` as a conservative extension of the base language (here PA) where the elements of this sort are interpreted as formulas with one free variable. Given a formula we construct its class as $\{x \mid p(x)\}$: `set`, and given a class A : `set` we can construct $(x \in A)$: `wff`, and these operations are inverses of each other.

The reason this is a conservative extension is because any formula in the new language is equivalent to one where $\{x \mid p(x)\}$ does not appear and $(x \in A)$ only appears when A is a class variable, so in particular if the formula has no class variables then it is equivalent to one that does not mention the class constructs at all; by eliminating classes from every intermediate statement we can show that the formula is derivable in plain PA.

We do not actually want to perform such a substitution though, as it would require expanding many definitions and could lead to an explosion in proof size. So instead we consider this as a part of the meta-level argument for the reasonableness of the axiom system.[18]

[18] It is future work, but we would like to prove that anything derivable in MM0-flavored PA can be translated to a statement which is derivable in "textbook PA." In addition to handling the `set` sort mentioned here, this would also handle the sorts `hex`, `char`, `string`, which are all self-evidently encodable as `nat` but are kept separate so that the metatheory can talk about evaluating definitions of type `string`, as discussed in section 4.8.2.

4.8.2 *Syntactic strings*

The `string` sort in `peano_hex.mm0` is defined like this:

```
strict free sort hex;
term x0: hex;
term x1: hex;
-- ...
term xf: hex;

strict free sort char;
term ch: hex > hex > char;

strict free sort string;
term s0: string;
term s1: char > string;
term sadd: string > string > string; infixr sadd: $'+$ prec 51;
def scons (c: char) (s: string): string = $ s1 c '+ s $;
infixr scons: $':$ prec 53;
```

An important property that results from this definition is that there are no terms of type `string` other than those inductively built from `s0`, `s1`, `sadd`, and definitions wrapping these terms (like `scons`). In order to reason about strings, we axiomatize a function `term s2n: string > nat;` which interprets the string as a list of numbers less than 256 in the obvious way.

In the final theorem $\mathsf{okProg}(\mathsf{ELF\_lit}(fs, ms, content), T)$, the term $\mathsf{ELF\_lit}(fs, ms, content)$ is a `string`, which means that it is built essentially out of concatenated string literals: ELF_lit can only use its arguments in a very simple way. In particular, all closed string expressions can be evaluated to an actual string, so this means that the *verifier* can actually evaluate the string expression.

The experimental `output string: my_string_def;` command is a mechanism for doing exactly this. If the verifier supports the `string` output format, then it checks that the string prelude is defined as above, and if so it interprets `s0` as the empty string, `s1 c` as a character, and `sadd` as string concatenation, and evaluates the provided string expression and prints it on standard out.

Since the input-output behavior of the verifier is subject to formal specification, the `output` command functions as a bridge between the MM0 logic and the metalogic or the "real world." The MM0 verifier is made to act like a compiler, and even spits out an executable file, and with suitable specification it will even spit out an executable file *that meets a given specification*, without ever needing to trust the MMC compiler.

5
Looking ahead

5.1 Applications

5.1.1 MM0 as an interchange format

MM0 IS A *logical framework* in the sense that it doesn't prescribe any particular axioms or semantics. This makes it well suited for translations to and from other systems. A downside of this approach is that while correctness with respect to the formal system is well defined, *soundness* becomes unclear in the absence of a fixed foundation. Instead, one gets several soundness theorems depending on what axioms are chosen and what semantics is targeted. General MM0 has a soundness theorem as well, similar to the Metamath soundness theorem[1], but these models are rather unstructured. (There is a simple multi-sorted SOL model for MM0, but it fails completeness.) However, for the short term, proof translation can function as a substitute for a soundness proof, and indeed, a proof translation amounts to building a class model of the source system in the target system.

[1] Mario Carneiro. Models for Metamath. presented at CICM 2016, 2016

Eventually, we hope to use MM0 to prove correctness of other theorem provers, and vice versa, and proof translations play an important role in this. There is a $O(n^2)$ problem with having n mutually supporting bootstraps, as there are n^2 proofs to be done. But the proof of $A \vdash$ 'A is correct' is closely related to the proof of $B \vdash$ 'A is correct'; if we had a method for translating proofs in A to proofs in B, we would obtain the result immediately. Moreover, proof translations compose, so it only requires a spider-web of proof connections before we can achieve such a critical mass. (Of course, this is only enough to get each prover to agree that A is correct. With n verifiers we would need n correctness proofs and an $O(n)$ network of translations to get the full matrix of correctness results.)

Our work in this area is modest, but it has already been quite helpful. We mentioned in section 1.6 some verification times for `set.mm` in MM0. A dataset of this size is not something we would have a hope of creating without a huge investment of time and effort. Instead, we map MM statements to MM0, and then we obtain tens of thousands of MM0 theorems in one fell swoop, a huge data set for testing that we could not have obtained otherwise.

5.2 *Translating MM to MM0*

The Haskell verifier `mm0-hs` contains a `from-mm` subcommand that will convert Metamath proofs to MM0.[2] Because of the similarity of the logics, the transformation is mostly cosmetic; unbundling is the most significant logical change. Whenever Metamath proves a theorem of the form $\vdash T[x, y]$ with no $x \# y$ assumption, we must generate two theorems, $\vdash T[x, x]$ and $\vdash T[x, y]$ (which implicitly assumes $x \# y$ in MM0). In many cases we can avoid this, for example if x and y are not bound by anything, as in $\vdash x = y \to y = z \to x = z$, we can just make them metavariables instead of names, but some theorems require this treatment, like $\vdash \forall x\ x = y \to \forall y\ y = x$.

For definitions, we currently do nothing (we leave them as axioms), but we plan to detect MM style definitional axioms and turn them into MM0 definitions.

[2] The Haskell verifier is deprecated, but this subcommand has not yet been moved to the new system.

5.3 *Translating MM0 to HOL systems*

The `to-hol` subcommand translates MM0 into a subset of HOL in a very natural way. A metavariable $\varphi : s\,\overline{x}$ becomes an n-ary variable $\varphi : s_1 \to \cdots \to s_n \to s$, where $x_i : s_i$, and all occurrences of φ in statements are replaced by $\varphi\,\overline{x}$. All hypotheses and the conclusion, are universally closed over the names, and the entire implication from hypotheses to conclusion is universally quantified over the metavariables.

For example, the axiom of generalization is

$$x : \mathsf{var}, \varphi : \mathsf{wff}\ x;\ \ \varphi \vdash \mathsf{all}\, x\, \varphi,$$

which becomes

$$\forall \varphi : \mathsf{var} \to \mathsf{wff},\ (\forall x : \mathsf{var}, \vdash\ \varphi\, x) \Rightarrow\ \vdash \mathsf{all}\ (\lambda x : \mathsf{var},\ \varphi\, x)$$

after translation.

The actual output of `mm0-hs to-hol` is a bespoke intermediate language (although it has a typechecker), which is used as a stepping-off point to OpenTheory and Lean. One of the nice side effects of this work was that Metamath theorems in `set.mm` finally became available to other theorem provers. We demonstrate the utility of this translation by proving Dirichlet's theorem in Lean[3], using the number theory library in Metamath for the bulk of the proof and post-processing the statement so that it is expressed in idiomatic Lean style. Very little of the Lean library was used for the proof. We only needed to show things like an isomorphism between Lean's $\mathbb{N}$ and Metamath's $\mathbb{N}$, which follows because both systems have proved the universal property of $\mathbb{N}$.

[3] https://github.com/digama0/mm0/blob/master/mm0-lean/mm0/set/post.lean

5.4 *Related work*

The MM0 project draws from ideas in a number of fields, most of which have long histories and many contributors. Here is a sampling of related work in MM0-adjacent fields.

5.4.1 *Bootstrapping theorem provers*

The idea of a bootstrapping theorem prover is not new. There are a number of notable projects in this space, many of which have influenced the design of MM0. However, none of these projects seem to have recognized (in words or actions) the value of parsimony, specifically as it relates to bootstrapping.

At its heart, a theorem prover that proves it is correct is a type of circular proof. While a proof of correctness can significantly amplify our confidence that we haven't missed any bugs, we must eventually turn to other methods to ground the argument, and direct inspection is always the fallback. But the effectiveness of direct inspection is inversely proportional to the size of the artifact, so the only way to make a bootstrap argument more airtight is to make it smaller.

The most closely related projects, in terms of bootstrapping a theorem prover down to machine code, are CakeML and Milawa.

- CakeML[4] appears to be the most active bootstrapping system today. The bootstrap consists of two parts: CakeML is a compiler for ML that is written in the logic of HOL4[5], and HOL4 is a theorem prover written in ML. Since the completion of the bootstrap in 2014, the CakeML team have expanded downward with *verified stacks*[6], formalizing the hardware of an open source processor design they could implement using an FPGA.
  Unfortunately, it seems that the bootstrap is not complete in the sense that the ML that CakeML supports is not sufficient for HOL4, and while a simpler kernel, called Candle, has been implemented in CakeML, it supports a variant of HOL Light, not HOL4, and cannot handle many of the idioms used in the correctness proof of

[4] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, January 2014

[5] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008

[6] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1041–1053. ACM, 2019

the CakeML compiler. Furthermore, the compiler correctness proof takes on the order of 14 hours to run, and while we do not yet have reliable figures, we project that the MM0 toolchain will be able to beat this figure by 3–5 orders of magnitude (or 5–7 counting only the verification time of the completed proof).

- Milawa[7] is a theorem prover based on ACL2 developed for Jared Davis's PhD thesis, which starts with a simple inspectable verifier A which proves the correctness of a more powerful verifier B, which proves verifier C and so on. After another 12 steps or so the verifier becomes practical enough to be able to prove verifier A correct. This project was later extended by Magnus Myreen to *Jitawa*[8], a Lisp runtime that was verified in HOL4 and can run Milawa. Although this isn't exactly a bootstrap, it is an instance of bootstrap cooperation (to the extent that CakeML/HOL4 can be considered a bootstrap), of the sort we described in section 5.1.1.

[7] Jared Curran Davis and J Strother Moore. *A self-verifying theorem prover*. PhD thesis, University of Texas, 2009

[8] Magnus O Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *International Conference on Interactive Theorem Proving*, pages 265–280. Springer, 2011

There are a few other projects that have done bootstraps at the logic level. The original version of Milawa has this characteristic, since it does not go down to machine code but rather starts from a Lisp-like programming language with proof capabilities and uses this language to write a type checker for its own language. This means that issues such as compiling to an architecture at the back end, and verified parsing at the front end, don't come up and have to be trusted.

- "Coq in Coq" by Bruno Barras (1996)[9] is a formalization of the Calculus of Constructions (CC) and a typechecker thereof in Coq, which can be extracted into an OCaml program. Here it is not Coq itself that is being verified but rather an independent kernel; moreover Coq implements not CC but CIC (the calculus of *inductive* constructions), and of course many inductive types are used in the construction of the typechecker, so this fails to "close the loop" of the bootstrap.

[9] Bruno Barras. Coq en coq. 1996

- John Harrison's "Towards self-verification of HOL Light" (2006)[10] writes down a translation of the HOL Light kernel (written in OCaml) in HOL Light, and proves the soundness of the axiom system with respect to a set theoretical model. This is the earliest example we know of a theorem prover verifying its own implementation, but it leaves off verification of OCaml (quite to the contrary, it is explicitly mentioned in the paper that it is possible to violate soundness using string mutability), and the translation from OCaml code to HOL Light definitions is unverified and slightly nontrivial in places.

[10] John Harrison. Towards self-verification of hol light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag

- "Coq Coq Correct!" (2019)[11] improves on "Coq in Coq" by verifying a typechecker for PCUIC (the polymorphic, cumulative calculus of inductive constructions), which is a much closer approximation

[11] Matthieu Sozeau, Yannick Forster, and Théo Winterhalter. Coq coq correct!

to Coq. It still lacks certain features of the kernel such as the module system and some advanced kinds of inductive types, and some core components like the guard condition are left undefined by the specification. However, the implemented subset of Coq is at least expressive enough to contain the formalization itself. Sadly, the typechecker is not fast enough in practice to be able to typecheck its own formalization.

5.4.2 *Code extraction*

Code extraction is the process of taking a definition in the logic and turning it into executable code, usually by transpilation to a traditional compiled language. Isabelle/HOL[12] can target SML, Scala, OCaml, and Haskell; HOL4 can target OCaml, and Coq[13] can target OCaml and Haskell. This is the most popular way of having simultaneously an object to prove properties about, and a program that is reasonably efficient. However, as argued in [[14]], we believe that this leaves large gaps in the verified part, since the extraction function must be trusted as well as the target language's compiler.

5.4.3 *ISA specification*

ISA specification is becoming more commonplace.[15] is a complete formal specification of the user level Intel x86-64 ISA in the K framework[16], of which we have only touched a small part. Sail is a language specifically for the purpose of specifying ISAs, and it has been used to formalize parts of ARM, RISC-V, MIPS, CHERI-MIPS, IBM Power, and x86[17]. Our `x86.mm0` specification is based on a port of the Sail x86 spec. Centaur[18] is using an x86 specification to build a provably correct chip design.

5.4.4 *Program verification*

Machine code verification does not differ significantly from program verification at other levels, and a number of techniques have developed to deal with it, such as Hoare logic and Separation logic. [[19]] shows how machine code can be verified (in HOL4) by decompiling the machine code into HOL functions.

5.4.5 *Verified compilers*

Verified compilers are programs that produce machine code from a source language with a specified semantics, that have been proven to preserve the semantics of the input program. CompCert[20] is a verified

[12] Florian Haftmann. Code generation from isabelle/hol theories

[13] Pierre Letouzey. Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008

[14] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *International Conference on Interactive Theorem Proving*, pages 362–369. Springer, 2018

[15] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148. ACM, June 2019

[16] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010

[17] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71

[18] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 47–60, 2020

[19] Magnus O Myreen. Formal verification of machine-code programs. Technical report, University of Cambridge, Computer Laboratory, 2009

[20] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012

compiler for a subset of C, and CakeML[21] is a verified compiler for ML. See section 3.1 for a more in depth discussion about verified compiler projects and how they relate to MMC.

[21] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, January 2014

5.4.6 *Verification frameworks*

For functional correctness of a particular program verified compilers are only half the story, as one must now show that the program has the correct behavior in the source language. For a language like C, this is difficult because there is no facility for doing such proofs. VST[22] is a tool for proving correctness of C programs deeply embedded as Coq terms. Iris[23] is a separation logic framework for verifying programs in Coq, from which MMC borrows many concepts.

[22] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. Vst-floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning*, 61(1-4):367–422, 2018

[23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018

5.4.7 *Type soundness theorems*

For languages with good source semantics, general soundness properties are useful for reducing the work of functional correctness. RustBelt[24] is a project to prove soundness of the Rust type system using Iris. Standard ML[25] is well known for being a "real-world" language with a type soundness theorem. CakeML also formally proves a version of ML to be type-safe since this is part of the overall correctness theorem.

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017

[25] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978

5.5 *Conclusion*

Metamath Zero is a theorem prover which is built to solve the problem of bootstrapping trust into a system. Yet at the same time it is general purpose — it does not use a tailor-made program logic, it uses whatever axioms you give it, so it can support all common formal systems (ZFC, HOL, DTT, PA, really anything recursively enumerable). It is extremely fast, at least on hand-written inputs like `set.mm`, and can handle computer-science-sized problems.

The attempt to solve the problem of writing and maintaining a verifiable theorem prover lead to the development of Metamath C, which is a fairly full-featured programming language with a focus on provably correct operation and putting the power of deductive verification in the hands of the user. We expect the language to continue to grow and evolve as we prove more parts of the correctness theorem. It is well placed to fill a niche that has been too long unaddressed.

Although the correctness theorem for MMC is still ongoing, we believe there is value added in clearly delineating the necessary components for a system that pushes the boundaries of formal verification to cover as much as possible, so that we can obtain the highest level of confidence, without compromising when it comes to speed of the overall system, and without putting an upper bound on the performance of verified artifacts other than the target machine itself.

Our hope in providing these tools freely to the public is to popularize and normalize the concept of programming languages with strong type systems where you can fully express and validate invariants in the language itself, and proving functional correctness is just one more tool in the box. Certainly this is not something that everyone will want to do all the time, but we believe strongly in the maxim that "if you build it, they will come:" the applications already exist but the current options are just too hard to use except by experts.[26]

[26] We haven't really made a compelling case that MM0 is actually usable by non-experts, as we haven't attempted to seriously market the program to the wider public. Rather, our focus has been to first get the strongest possible correctness claim and work toward ease of use, instead of the more well-trodden approach of starting from an easy to use tool and working toward correctness.

We also hope to see a future where all the major theorem provers are either proven correct or can export their proofs to systems that are proven correct, so that when we verify our most important software, we bequeath the highest level of confidence we are capable of providing. It's not an impossible dream — the technology is in our hands; we need only define the problem, and solve it.

A

A contradiction in Metamath from grammar ambiguity

This is a metamath file which shows how an ambiguous grammar can lead to the proof of false theorems. This is why Metamath databases generally have to be very careful with parentheses, because unambiguity is soundness-critical.

```
$c wff |- F -> $.  $( Declare constant symbols $)
$v p q $.          $( Declare variable symbols $)
wp $f wff p $.  wq $f wff q $.   $( Declare wffs $)

wF $a wff F $.  $( A symbol for falsity $)
$( Declare implication *without* surrounding parens. This causes grammar ambiguity $)
im $a wff p -> q $.

$( Observe that we can prove syntax theorems in multiple ways $)
ex1 $p wff F -> F -> F $= wF wF wF im im $.  $( F -> (F -> F) $)
ex2 $p wff F -> F -> F $= wF wF im wF im $.  $( (F -> F) -> F $)

$( We only need these two prop calc axioms $)
ax-1 $a |- p -> q -> p $.
${ mp.1 $e |- p -> q $.  mp.2 $e |- p $.  mp $a |- q $. $}

$( Proof of false, taking advantage of "parse confusion" to transform step 1
   (which is true) into step 1' (which is false). Parentheses are shown to
   indicate the intended parse, but they are (critically) not part of the string
   that metamath consumes, leading to the confusion. $)
contradiction $p |- F $=
    wF wF wF im im      $( wff F -> (F -> F) $)
    wF                  $( wff F $)
    wF wF wF im ax-1    $( 1.  |- F -> ((F -> F) -> F) $)
                        $( 1'. |- (F -> (F -> F)) -> F $)
    wF wF ax-1          $( 2.  |- F -> (F -> F) $)
  mp $.                 $( 3.  |- F $)
```

B The type system of MMC

This is a specification of the type system of MMC programs. It is roughly at the level that users interact with the type checker (it is not written in terms of MIR, even though the formally proved version of the type system is at the MIR level.)

Warning: This is intended as a general guide for people who like precision and greek letters. But until the proof of correctness is complete, everything in here is subject to repairs and changes. The language description in section 3.2 is recommended prior reading.

B.1 Syntax

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

$\alpha, x, h, k \in \text{Ident}$ ::=	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
$it \in \text{Item}$ ::=	$\mathsf{type}\ S(\overline{\alpha}, \overline{R}) := \tau$	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

$\tau \in \text{Type} ::= \alpha$	type variable reference
$\mid \lvert\alpha\rvert$	moved type variable
$\mid \mathbf{1} \mid \top \mid \bot \mid \mathsf{bool}$	unit, true, false, booleans
$\mid \mathbb{N}_s \mid \mathbb{Z}_s$	unsigned and signed integers of different sizes
$\mid \tau_1 \land \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 \lor \tau_2$	conjunction (regular, separating), disjunction
$\mid \tau_1 \to \tau_2 \mid \tau_1 \mathrel{-\!\!*} \tau_2 \mid \neg\tau$	implication (regular, separating), negation
$\mid \forall x : \tau_1,\ \tau_2 \mid \sum x : \tau_1,\ \tau_2$	universal, existential quantification
$\mid pe$	assert that a boolean value is true
$\mid pe \mapsto pe'$	points-to assertion
$\mid \boxed{x : \tau}$	typing assertion
$\mid S(\overline{\tau}, \overline{pe})$	user-defined type

$pe \in \text{PureExpr} ::=$ (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
$\mid e_1 \land e_2 \mid e_1 \lor e_2 \mid \neg e$	logical AND, OR, NOT
$\mid e_1 \mathbin{\&} e_2 \mid e_1 \mathbin{\vert} e_2 \mid\ !_s\, e$	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

Missing elements of the grammar include:

- Switch statements, which are desugared to if statements.
- Raw MM0 formulas can be lifted to the 'Type' type as booleans.
- 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 B.2.8.
- 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 let $t := e_1$ 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 $x \to y$" 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 $x^\gamma \leftarrow pe;\ e$ 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 B.2.2.

B.2 Typing

B.2.1 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
- $\Gamma \vdash \tau\ \mathsf{type}$
  determines that a type τ is a valid type in the current context
- $\Gamma \vdash R\ \mathsf{arg}$
  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 $(x := pe : \tau) \in \delta$ then $\Gamma \vdash pe : \tau$ and $x \in \mathrm{Dom}(\Gamma)$, and if $(x \to y) \in \delta$ 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.
- $\Gamma \vdash \cdot \dashv \Gamma'$
  an auxiliary judgment for applying pending mutations to the context.
- $\Gamma \vdash it$ 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 $\Gamma, x \leftarrow pe : \tau$ to denote the insertion of $x \leftarrow pe : \tau$ into the list of mutations, replacing $x \leftarrow pe' : \tau'$ if it is present.

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 $x := pe : \tau$, 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 $x : \tau$, a shorthand for $x := x : \tau$, where $(x : |\tau|) \in \Gamma$.
- A rename map, which is a list of records of the form $x \to y$ 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 $x \to y, y'$ then $y = y'$ and if $x, x' \to y$ then $x = x'$.

B.2.2 Moving types

The last essential element to understand the typing rules is the "moved" modality on types, denoted $|\tau|$. 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.

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

The moved modality is defined like so:

$\mathbf{1}, \top, \bot, \mathsf{bool}, \mathbb{N}_s, \mathbb{Z}_s, pe$ copy

$$\begin{aligned}
|\tau_1 \wedge \tau_2| &= |\tau_1| \wedge |\tau_2| \\
|\tau_1 \vee \tau_2| &= |\tau_1| \vee |\tau_2| \\
|\tau_1 * \tau_2| &= |\tau_1| * |\tau_2| \\
|\textstyle\sum x : \tau_1,\ \tau_2| &= \textstyle\sum x : |\tau_1|,\ |\tau_2| \\
|S(\overline{\tau}, \overline{pe})| &= |S|\,(\overline{\tau}, \overline{pe}) \qquad \text{(that is, the effect of moving } S \text{ is precalculated)} \\
|pe \mapsto pe'| &= \top \\
\left|\boxed{x : \tau}\right| &= \boxed{x : |\tau|}
\end{aligned}$$

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

$$|\tau \to \tau'| = \begin{cases} \tau \to |\tau'| & \text{if } \tau \text{ copy} \\ \top & o.w. \end{cases}$$

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

$$|\neg\tau| = \begin{cases} \neg\tau & \text{if } \tau \text{ copy} \\ \top & o.w. \end{cases}$$

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

When we get to pointer types in section B.2.8 we will see that $|\&^{\mathbf{own}}\tau| = |\&^{\mathbf{mut}}\tau| = \mathbb{N}_{64}$, 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.)

Note that move commutes with substitution for (expression) variables, $|\tau|\,[e/x] = |\tau[e/x]|$, but it only partially commutes with substitution for type variables: $|\tau[\tau'/\alpha]| \Rightarrow |\tau|\,[\tau'/\alpha]$, because substitution can make a non-copy type copy, so that for example $|\alpha[\mathbb{N}/\alpha]| = |\mathbb{N}| = \mathbb{N}$ but $|\alpha|\,[\mathbb{N}/\alpha] = \top[\mathbb{N}/\alpha] = \top$.

B.2.3 The Typing Rules

We now give the main typing rules for the logic. Note that ghost variable markings are ignored during this phase; they will come back during the ghost propagation phase.

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

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

TP-VAR
$$\Gamma \vdash x^{\gamma} : \tau \Rightarrow x : \tau$$

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

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

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

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

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

$$\textsc{arg-type}\quad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash x : \tau\ \mathsf{arg}}$$

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.

Type validity $\boxed{\Gamma \vdash \tau\ \mathsf{type}}$

$$\textsc{ty-unit}\quad \Gamma \vdash \mathbf{1}\ \mathsf{type} \qquad \textsc{ty-true}\quad \Gamma \vdash \top\ \mathsf{type} \qquad \textsc{ty-false}\quad \Gamma \vdash \bot\ \mathsf{type} \qquad \textsc{ty-bool}\quad \Gamma \vdash \mathsf{bool}\ \mathsf{type}$$

$$\textsc{ty-nat}\quad \Gamma \vdash \mathbb{N}_s\ \mathsf{type} \qquad \textsc{ty-int}\quad \Gamma \vdash \mathbb{Z}_s\ \mathsf{type}$$

$$\textsc{ty-var}\quad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}} \qquad \textsc{ty-core-var}\quad \frac{\alpha \in \Gamma}{\Gamma \vdash |\alpha|\ \mathsf{type}} \qquad \textsc{ty-pure}\quad \frac{\Gamma \vdash pe : \mathsf{bool}}{\Gamma \vdash pe\ \mathsf{type}} \qquad \textsc{ty-not}\quad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \neg\tau\ \mathsf{type}}$$

$$\textsc{ty-and}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \quad \Gamma \vdash \tau'\ \mathsf{type}}{\Gamma \vdash \tau \wedge \tau'\ \mathsf{type}} \qquad \textsc{ty-or}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \quad \Gamma \vdash \tau'\ \mathsf{type}}{\Gamma \vdash \tau \vee \tau'\ \mathsf{type}}$$

$$\textsc{ty-sep}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \quad \Gamma \vdash \tau'\ \mathsf{type}}{\Gamma \vdash \tau * \tau'\ \mathsf{type}} \qquad \textsc{ty-wand}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \quad \Gamma \vdash \tau'\ \mathsf{type}}{\Gamma \vdash \tau \mathrel{-\!\!*} \tau'\ \mathsf{type}}$$

TY-ALL
$$\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash \tau \text{ type}}{\Gamma \vdash \forall x : \tau,\ \tau \text{ type}}$$

TY-SUM
$$\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma, x : |\tau| \vdash \tau \text{ type}}{\Gamma \vdash \sum x : \tau,\ \tau \text{ type}}$$

TY-POINTS-TO
$$\frac{\Gamma \vdash \ell : \mathbb{N}_{64} \quad \Gamma \vdash v : |\emptyset|}{\Gamma \vdash \ell \mapsto v \text{ type}}$$

TY-TYPING
$$\frac{\Gamma \vdash x : |\tau| \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \boxed{x : \tau} \text{ type}}$$

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 will appear later), which can have a natural number size bound, and in hypotheses via TY-PURE.

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.

B.2.4 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
$$\Gamma \vdash \mathsf{true} : \mathsf{bool}$$

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

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

TYE-AND, TYE-OR
$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma_1 \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \land e_2 : \mathsf{bool} \quad \Gamma \vdash e_1 \lor e_2 : \mathsf{bool}}$$

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

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

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

TYE-FUNC-CALL
$$\frac{\mathsf{func}\ f(\bar{R}) : \bar{S} \quad \Gamma \vdash \langle \bar{e} \rangle : \textstyle\sum \bar{R}}{\Gamma \vdash f(\bar{e}) : \textstyle\sum \bar{S}}$$

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).
- Varables can be changed using no-op rules (the TYE-CS-LEFT and TYE-CS-RIGHT rules). We will return to this in section B.2.5.

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

TYE-CS-LEFT
$$\frac{\Gamma; \delta \vdash \cdot \dashv \delta_1 \quad \Gamma; \delta_1 \vdash e \Rightarrow pe^? : \tau \dashv \delta_2}{\Gamma; \delta \vdash e \Rightarrow pe^? : \tau \dashv \delta_2}$$

TYE-CS-RIGHT
$$\frac{\Gamma; \delta \vdash e \Rightarrow pe^? : \tau \dashv \delta_1 \quad \Gamma; \delta_1 \vdash \cdot \dashv \delta_2}{\Gamma; \delta \vdash e \Rightarrow pe^? : \tau \dashv \delta_2}$$

TYE-VAR-MOVE
$$\Gamma; \delta, x := pe : \tau \vdash x \Rightarrow pe : \tau \dashv \delta, x := pe : |\tau|$$

TYE-MUT
$$\frac{\begin{array}{c}\Gamma; \delta \vdash e_1 \Rightarrow pe : \tau \dashv \delta_1 \qquad \forall z,\ (x \to z) \notin \delta_1 \qquad \Gamma \vdash \delta_2 \\ \Gamma; \delta_1, (x \to y), (y := pe : \tau) \vdash e_2 : \tau' \dashv \delta_2 \qquad y \notin \delta_2\end{array}}{\Gamma; \delta \vdash (x^\gamma \leftarrow e_1 \text{ with } y \leftarrow x;\ e_2) : \tau' \dashv \delta_2}$$

TYE-LET-PURE
$$\frac{\begin{array}{c}\Gamma; \delta \vdash e_1 \Rightarrow pe : \tau \dashv \delta_1 \qquad \Gamma \vdash \tau' \qquad \Gamma \vdash \delta_2 \\ \Gamma, x : |\tau|; \delta_1, x := pe : \tau \vdash e_2 : \tau' \dashv \delta_2\end{array}}{\Gamma; \delta \vdash (\text{let } x^\gamma := e_1 \text{ in } e_2) : \tau' \dashv \delta_2}$$

TYE-UNREACHABLE
$$\frac{\Gamma; \delta \vdash e : \bot \dashv \delta_1 \quad \Gamma \vdash \delta_2}{\Gamma; \delta \vdash \text{unreachable } e : \tau \dashv \delta_2}$$

TYE-LET
$$\frac{\begin{array}{c}\Gamma; \delta \vdash e_1 : \tau \dashv \delta_1 \qquad \Gamma \vdash t : \tau \Rightarrow \overline{R} \qquad \Gamma \vdash \tau' \qquad \Gamma \vdash \delta_2 \\ \Gamma, \overline{|R|}; \delta_1, \overline{R} \vdash e_2 : \tau' \dashv \delta_2\end{array}}{\Gamma; \delta \vdash (\text{let } t := e_1 \text{ in } e_2) : \tau' \dashv \delta_2}$$

TYE-PROC-CALL
$$\frac{\text{proc } F(\overline{R}) : \overline{S} \quad \Gamma; \delta \vdash \langle \overline{e} \rangle : \textstyle\sum \overline{R} \dashv \delta'}{\Gamma; \delta \vdash F(\overline{e}) : \textstyle\sum \overline{S} \dashv \delta'}$$

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

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

TYE-GOTO
$$\frac{k(\delta';\bar{R}) \in \Gamma \qquad \Gamma;\delta \vdash \langle \bar{e} \rangle : \sum \bar{R} \dashv \delta'}{\Gamma;\delta \vdash \mathsf{goto}\ k(\bar{e}) : \bot \vdash \delta'}$$

TYE-ASSERT
$$\frac{\Gamma;\delta \vdash e \Rightarrow pe : \mathsf{bool} \dashv \delta'}{\Gamma;\delta \vdash \mathsf{assert}\ e : pe \dashv \delta'}$$

TYE-TYPEOF
$$\frac{\Gamma;\delta \vdash e \Rightarrow pe : \tau \dashv \delta'}{\Gamma;\delta \vdash \mathsf{typeof}\ e : \boxed{pe : \tau} \dashv \delta'}$$

TYE-ENTAIL
$$\frac{\Gamma;\delta \vdash \langle \bar{e} \rangle : \ast\, \overline{A} \dashv \delta' \qquad \vdash p : \ast\, \overline{A} \mathrel{-\!\!\ast} B}{\Gamma;\delta \vdash \mathsf{entail}\ \bar{e}\ p : B \dashv \delta'}$$

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.

B.2.5 No-op steps

In addition to being able to step as a result of executing some expression, we also need the ability to step without anything happening physically. This is primarily needed in order to clean up the context to eliminate a variable, or to merge control flow to a common context, i.e. 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.

The rules given below are not deterministic, but they are used whenever we can't otherwise make progress. Using them too much may end up in a state where a variable is missing, causing later typechecking to fail, so the compiler will try to apply these only as necessary.

No-op step $\boxed{\Gamma;\delta \vdash \cdot \dashv \delta'}$

CS-REFL
$$\Gamma;\delta \vdash \cdot \dashv \delta$$

CS-TRANS
$$\frac{\Gamma;\delta_1 \vdash \cdot \dashv \delta_2 \qquad \Gamma;\delta_2 \vdash \cdot \dashv \delta_3}{\Gamma;\delta_1 \vdash \cdot \dashv \delta_3}$$

CS-DROP
$$\frac{\forall x, (x \to y) \notin \delta}{\Gamma;\delta, (y := pe : \tau) \vdash \cdot \dashv \delta}$$

CS-RENAME[1]
$$\Gamma;\delta, (x \to y), (y := pe : \tau) \vdash \cdot \dashv \delta, (x := pe : \tau)$$

CS-FORGET
$$\frac{\Gamma \vdash \Gamma[\overline{x \to pe}]}{\Gamma;\delta, \overline{x := pe : \tau} \vdash \cdot \dashv \delta, \overline{x : \tau}}$$

[1] The CS-RENAME rule should only be used if it is the only way to make progress, i.e. when y is going out of scope. This is needed because it changes the interpretation of expressions containing x.

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 $x \to y$ 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 $x \leftarrow 1$ and the other sets $x \leftarrow 2$, 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

$$x := 0, h : x \geq 0 \vdash \mathsf{if}\ b\ \{\ x \leftarrow 1\ \},$$

which desugars to "$\mathsf{if}\ b\ \mathsf{then}\ x^{\top} \leftarrow 1;\ ()\ \mathsf{else}\ ()$". After the mutation, we have $x \to x', x' := 1$ 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 $x \leftarrow 1 : \mathbb{N}$, but $1 \geq 0$ is not immediately available.) If the compiler cannot find this proof, it can be supplied by:

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

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 $h := (p : 1 \geq 0)$.

B.2.6 Top level typing

The full program consists of a list of top level items, which are type-checked incrementally:

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

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

Individual items are typed as follows:

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

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

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

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

B.2.7 Uninitialized data

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

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

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

That's it. Note that $\tau \le \tau^?$ 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.

B.2.8 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 $(x : \tau) \in \Gamma$ and A for every $h : A$, plus additional "layout" information about the relation of non-ghost variables to the stack frame.

Singleton pointers

The simplest pointer type is $\&^{\mathbf{sn}}\eta$, which is defined as $\mathsf{sn}\ \&\eta$. $x : \&^{\mathbf{sn}}\eta$ simply means that x is a pointer that points to η, which is a "place", a writable location. $\boxed{x : \&^{\mathbf{sn}}\eta} = (x = \&\eta)$, where $\&\eta$ is the location that η is stored in memory; see section B.5. (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:

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

$$\&^{\mathbf{sn}}\eta := \mathsf{sn}\ \&\eta$$

TYE-SNP
$$\frac{\Gamma \vdash \eta\ \mathsf{place}}{\Gamma \vdash \&\eta : \mathbb{N}_{64}}$$

TYE-DEREF
$$\frac{\Gamma; \delta \vdash e : \&^{\mathbf{sn}}\eta \dashv \delta'}{\Gamma; \delta \vdash {}^*e \Rightarrow \eta \dashv \delta'\ \mathsf{place}}$$

TYE-SHR
$$\frac{\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta'\ \mathsf{place}}{\Gamma; \delta \vdash \&e \Rightarrow \&\eta : \&^{\mathbf{sn}}\eta \dashv \delta'}$$

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

TYE-READ
$$\frac{\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta_1\ \mathsf{place} \qquad \Gamma; \delta_1 \vdash \eta \Rightarrow pe : \tau \dashv \delta_2}{\Gamma; \delta \vdash e \Rightarrow pe : \tau \dashv \delta_2}$$

TYE-WRITE
$$\frac{\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta_1\ \mathsf{place} \qquad \Gamma; \delta_1 \vdash (\eta \leftarrow pe;\ e_2) : \tau \dashv \delta_2}{\Gamma \vdash (e \leftarrow pe;\ e_2) : \tau \dashv \delta_2}$$

We needed two new judgments above, $\Gamma \vdash \eta\ \mathsf{place}$, which asserts that η is a place in the context, and $\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta'\ \mathsf{place}$ 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.

Note that writing to a place as in TYE-WRITE changes the type $\&^{\mathbf{sn}}\eta$ to $\&^{\mathbf{sn}}\eta'$ (it rewrites all occurrences of one with the other in the context), if η' is the renamed place after the mutation. This is because the pointer has not changed, but the data being pointed to has been updated, so we should now retrieve the new value, not the (ghost) old value.

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.

$$\text{Type} ::= \cdots \mid \&^{\mathbf{own}}\tau \qquad\qquad |\&^{\mathbf{own}}\tau| = \mathbb{N}_{64}$$

$$\boxed{x : \&^{\mathbf{own}}\tau} = \exists v : \tau, x \mapsto v$$

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

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 $\&^{\mathbf{sn}}a * \&^{\mathbf{sn}}b$.

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 $\mathsf{out}^x\ y : \tau$, if x is a function parameter (which is itself marked as $\mathsf{mut}\ x : \tau$). This has the meaning that the variable x will be mutated so that $\delta \vdash x \to^* y$ when the function reaches the return.

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

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

Here $\delta \vdash x \to^* y$ means that $x \to \cdots \to y \not\to$ 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, $\mathsf{proc}\ F(\overline{R}) : \overline{\mathsf{out}^x\ y : \tau}, \overline{S}$.

The arguments of F must be $e : \tau$ if $R = (x : \tau)$, and must be $\eta : \tau\ \mathsf{place}$ if $R = (\mathsf{mut}\ x : \tau)$. If η is provided for argument x, and $\mathsf{out}^x\ y : \tau'$ is among the out arguments of the function, and y is the corresponding element of the tuple in the $\mathsf{let}\ \langle \overline{y}, t\rangle$ 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.

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 type, a heap reservation type called $\mathsf{ref}^a\ \tau$, the elements of which are called heap variables. The expression $x : \mathsf{ref}^a\ \tau$ 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 $x := pe : \mathsf{ref}^a\ \tau$ 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 $x.\mathsf{field}$ 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.)

$$a \in \mathrm{Lft} ::= \mathsf{extern} \mid x \qquad\qquad \mathrm{Type} ::= \cdots \mid \mathsf{ref}^a\ \tau$$

TP-SUM-REF
$$\frac{\Gamma \vdash t : \mathsf{ref}^a\ \tau \Rightarrow \bar{S} \quad \Gamma, \bar{S} \vdash \langle \overline{t'} \rangle : \mathsf{ref}^a(\tau'[t/x]) \Rightarrow \bar{S}'}{\Gamma \vdash \langle t, \overline{t'} \rangle : \mathsf{ref}^a(\sum x : \tau, \overline{R}) \Rightarrow \bar{S}, \bar{S}'}$$

TY-REF
$$\frac{\mathrm{Var}(a) \subseteq \Gamma \quad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathsf{ref}^a\ \tau\ \mathsf{type}}$$

TYE-REF
$$\frac{\Gamma; \delta \vdash e \Rightarrow (\eta := pe : \tau) \dashv \delta'\ \mathsf{read}^a}{\Gamma; \delta \vdash e : \mathsf{ref}^a\ \tau \dashv \delta'}$$

Here the $\Gamma; \delta \vdash e \Rightarrow (\eta := pe : \tau) \dashv \delta'\ \mathsf{read}^a$ judgment is a conjunction of $\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta_1\ \mathsf{place}$ followed by $\Gamma \vdash \delta_1 \Rightarrow \delta'$, such that $\Gamma; \delta' \vdash \eta := pe : \tau\ \mathsf{read}^a$. 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 $\eta : \tau$ from origin a.

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

CS-LOCK
$$\Gamma \vdash \delta, (x := pe : \tau) \vdash \cdot \dashv \delta, (\mathsf{ref}^x\ x := pe : \tau)$$

CS-UNLOCK
$$\frac{\forall y, (\mathsf{ref}^x\ y := -) \notin \delta}{\Gamma \vdash \delta, (\mathsf{ref}^x\ x := pe : \tau) \vdash \cdot \dashv \delta, (x := pe : \tau)}$$

TYR-VAR
$$\frac{(\mathsf{ref}^a\ x := pe : \tau) \in \delta}{\Gamma; \delta \vdash x := pe : \tau\ \mathsf{read}^a}$$

TYE-READ-REF
$$\frac{\Gamma; \delta \vdash e \Rightarrow \eta \dashv \delta'\ \mathsf{place} \quad \Gamma; \delta' \vdash \eta := pe : \tau\ \mathsf{read}^a}{\Gamma; \delta \vdash e \Rightarrow pe : |\tau| \dashv \delta'}$$

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.

In fact, we can't even really read a ref; the value read is only available as a ghost value, unless it is accessed indirectly via a shared reference. Using heap variables, we can desugar shared references similarly to owned pointers:

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

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

B.2.9 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

B.3 Ghost propagation

Ghost annotations are optional in most cases, because of the ghost propagation pass that automatically makes as many things ghost as possible. The invariant that we uphold is that a ghost variable *must not* have an M-place associated with it, while a regular variable *may* have an M-place. However, it is consistent with this that there are no M-places at all, so we have some inductive conditions on what variables must have M-places, which are roughly analogous to dead-code elimination.

Ghost propagation (dead-store elimination) has to be done in tandem with reachability analysis (dead-code elimination), because if can convert data dependencies into control dependencies, meaning that parts of the code may in fact have the program counter itself being ghost. When this happens, we can't execute anything with side effects or anything whose value is computationally relevant, because the physical machine never reaches these lines.

To express all this, we will use a judgment $\Gamma^\alpha; \delta^\rho \vdash e : \tau^\gamma \dashv \delta'^{\rho'}$ that augments the typing condition with four ghost annotations; in addition we will be modifying the ghost annotations inside δ and δ_1 to make them more strict (i.e. possibly turning $x^\top$ to $x^\perp$).

- α, the variable on Γ, is either $\top$ or $\perp$. If $\alpha = \perp$ then the program counter is ghost, which is to say, we are unable to perform any operation that involves emitting code. This happens when we branch on a ghost variable.
- ρ, the variables associated to the before and after value contexts, are also $\perp$ or $\top$ and indicate whether the beginning or end of e is reachable.
- Because type inference is complete, we can treat the type τ of e as an input to the judgment. Here τ^γ is a type extended with ghost annotations in all subexpressions. The typing rules for such extended types assert that a type is ghost only if all subexpressions are ghost.

B.3.1 Ghost annotated types and tuple patterns

The types that show up in the expression judgment are annotated with γ ghost annotations at all levels, subject to a local coherence condition that states that a ghost type must only have ghost parts. This allows us to only compute some parts of a type as long as we have all the parts we actually need for downstream processing. While the language itself admits ghost annotations on variables in a tuple pattern and variable binders in a struct, these are only upper bounds on the computational relevance, because we are interested in eliminating parts of a type for optimization purposes even if they were not claimed to be ghost.

Ghost-annotated type validity $\boxed{\tau^\gamma \text{ ctype}}$

CTY-UNIT, CTY-BOOL
$$\mathbf{1}^\gamma,\ \mathsf{bool}^\gamma \text{ ctype}$$

CTY-NAT, CTY-INT
$$\mathbb{N}_s^\gamma,\ \mathbb{Z}_s^\gamma \text{ ctype}$$

CTY-VAR, CTY-CORE-VAR
$$\frac{\alpha \in \Gamma}{\alpha^\gamma,\ |\alpha|^\gamma \text{ ctype}}$$

CTY-INTER, CTY-UNION, CTY-LIST
$$\frac{\forall i,\ \tau_i^{\gamma_i} \text{ ctype} \quad \forall i,\ \gamma_i \le \gamma'}{(\bigcap \overline{\tau^\gamma})^{\gamma'},\ (\bigcup \overline{\tau^\gamma})^{\gamma'},\ (\ast\, \overline{\tau^\gamma})^{\gamma'} \text{ ctype}}$$

CTY-PROP
$$A^\gamma \text{ ctype}$$

CTY-STRUCT
$$\frac{\gamma_2 \le \gamma_1, \gamma \quad \tau^{\gamma_2}\ \mathsf{ctype} \quad \tau'^{\gamma}\ \mathsf{ctype}}{(\sum x^{\gamma_1} : \tau^{\gamma_2}, \tau')^{\gamma}\ \mathsf{ctype}}$$

Ghost-annotated tuple pattern validity $\boxed{t : \tau^{\gamma} \Rightarrow \overline{R^{\gamma'}}}$

CTP-IGNORE
$$\_ : \tau^{\gamma} \Rightarrow \cdot$$

CTP-VAR
$$\frac{\gamma' \le \gamma}{x^{\gamma} : \tau^{\gamma'} \Rightarrow x^{\gamma'} : \tau}$$

CTP-TYPED
$$\frac{t : \tau^{\gamma} \Rightarrow \overline{R^{\gamma'}}}{(t : \tau) : \tau^{\gamma} \Rightarrow \overline{R^{\gamma'}}}$$

CTP-SUM
$$\frac{\gamma_2 \le \gamma_1, \gamma_3 \quad t : \tau^{\gamma_2} \Rightarrow \overline{R^{\gamma}} \quad \langle \overline{t'} \rangle : (\tau'[t/x])^{\gamma_3} \Rightarrow \overline{R^{\gamma'}}}{\langle t, \overline{t'} \rangle : (\sum x^{\gamma_1} : \tau^{\gamma_2}, \tau')^{\gamma_3} \Rightarrow \overline{R^{\gamma}}, \overline{R^{\gamma'}}}$$

The intuitive meaning of τ^{γ} is that $\tau^{\top}$ is a value that will actually have storage space allocated for it, while $\tau^{\perp}$ is a value that will not need to be calculated (even if it is stored in a non-ghost variable). These rules assume that the full ghost annotation assignment is known and just give constraints on that assignment, but in practice we will start from an assignment that makes everything ghost, and incrementally shift this upward in a coordinated fashion. At the end we may end up with an impossible constraint such as a computationally relevant unbounded integer value, which will cause an error during legalization.

B.3.2 The expression typing judgment

For the expression judgment $\Gamma^{\alpha}; \delta^{\rho} \vdash e : \tau^{\gamma} \dashv \delta'^{\rho'}$, we have Γ, δ, e, τ as inputs and δ' as output, with the annotations $\alpha, \rho, \rho', \gamma$ being solved for by a fixed point algorithm. It is safe to assume that $\gamma \le \rho' \le \rho$ and $\gamma \le \alpha$ in this judgment (that is, the end of a statement is only reachable if the beginning is, and the return value is only needed if the end of the statement is reached), unless τ is a ghost type like $\mathbf{1}$ or A in which case $\gamma \le \rho', \alpha$ need not hold.

We also add an annotation $\sigma \in \{\perp, \top\}$ on functions (including self), which can be seen in rule TYC-PROC-CALL, for example; this is the side effect analysis, see section B.3.3.

Ghost-annotated expression validity $\boxed{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash e : \tau^{\gamma} \dashv \delta_2^{\rho_2}}$

TYC-CS-LEFT
$$\frac{\Gamma; \delta \vdash \cdot \dashv \delta_1 \quad \Gamma^{\alpha}; \delta_1^{\rho} \vdash e : \tau^{\gamma} \dashv \delta_2^{\rho'}}{\Gamma^{\alpha}; \delta^{\rho} \vdash e : \tau^{\gamma} \dashv \delta_2^{\rho'}}$$

TYC-CS-RIGHT
$$\frac{\Gamma^\alpha; \delta^\rho \vdash e : \tau^\gamma \dashv \delta_1^{\rho'} \quad \Gamma; \delta_1 \vdash \cdot \dashv \delta_2}{\Gamma^\alpha; \delta^\rho \vdash e : \tau^\gamma \dashv \delta_2^{\rho'}}$$

TYC-VAR-REF
$$\frac{(x^{\gamma'} := pe : \tau) \in \delta \quad |\tau| = \tau' \quad \gamma \le \alpha, \rho, \gamma'}{\Gamma^\alpha; \delta^\rho \vdash x : \tau'^\gamma \dashv \delta^\rho}$$

TYC-UNIT
$$\Gamma^\alpha; \delta^\rho \vdash () : \mathbf{1}^\gamma \dashv \delta^\rho$$

TYC-TRUE, TYC-FALSE
$$\frac{\gamma \le \alpha, \rho}{\Gamma^\alpha; \delta^\rho \vdash \mathsf{true},\ \mathsf{false} : \mathsf{bool}^\gamma \dashv \delta^\rho}$$

TYC-NAT, TYC-INT
$$\frac{\gamma \le \alpha, \rho}{\Gamma^\alpha; \delta^\rho \vdash n : \mathbb{N}_s^\gamma,\ \mathbb{Z}_s^\gamma \dashv \delta^\rho}$$

TYC-NOT
$$\frac{\Gamma^\alpha; \delta_1^{\rho_1} \vdash e : \mathsf{bool}^\gamma \dashv \delta_2^{\rho_2}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash \neg e : \mathsf{bool}^\gamma \dashv \delta_2^{\rho_2}}$$

TYC-AND, TYC-OR, . . .
$$\frac{\Gamma^\alpha; \delta_1^{\rho_1} \vdash e_1 : \mathsf{bool}^\gamma \dashv \delta_2^{\rho_2} \quad \Gamma^\alpha; \delta_2^{\rho_2} \vdash e_2 : \mathsf{bool}^\gamma \dashv \delta_3^{\rho_3}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash e_1 \wedge e_2,\ e_1 \vee e_2 : \mathsf{bool}^\gamma \dashv \delta_3^{\rho_3}}$$

TYC-IF
$$\frac{\begin{array}{c}\Gamma^\alpha; \delta_1^{\rho_1} \vdash c : \mathsf{bool}^{\gamma'} \dashv \delta_2^{\rho_2} \\ \Gamma^{\alpha \wedge \gamma'}; \delta_2^{\rho_2} \vdash e_1,\ e_2 : \tau^\gamma \dashv \delta_3^{\rho_3}\end{array}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash (\mathsf{if}\ c\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) : \tau^\gamma \dashv \delta_3^{\rho_3}}$$

TYC-STRUCT
$$\frac{\begin{array}{c}\gamma_2 \le \gamma_1, \gamma \quad \Gamma^\alpha; \delta_1^{\rho_1} \vdash e : \tau^{\gamma_2} \dashv \delta_2^{\rho_2} \\ \Gamma^\alpha; \delta_2^{\rho_2} \vdash \langle \bar{e} \rangle : (\tau'[e/x])^\gamma \dashv \delta_3^{\rho_3}\end{array}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash \langle e, \bar{e} \rangle : (\sum x^{\gamma_1} : \tau^{\gamma_2}, \tau')^\gamma \dashv \delta_3^{\rho_3}}$$

TYC-VAR-MOVE
$$\frac{\gamma \le \alpha, \rho, \gamma'}{\Gamma^\alpha; (\delta, x^{\gamma'} := pe : \tau)^\rho \vdash x : \tau^\gamma \dashv (\delta, x^{\gamma'} := pe : |\tau|)^\rho}$$

TYC-MUT
$$\frac{\begin{array}{c}\gamma_1 \le \gamma \quad \Gamma^\alpha; \delta_1^{\rho_1} \vdash e_1 : \tau_1^{\gamma_1} \dashv \delta_2^{\rho_2} \\ \Gamma^\alpha; \delta_2^{\rho_2}, (x \to y), (y^{\gamma_1} := e_1 : \tau_1) \vdash e_2 : \tau_2^{\gamma_2} \dashv \delta_3^{\rho_3}\end{array}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash (x^\gamma \leftarrow e_1\ \mathsf{with}\ y \leftarrow x;\ e_2) : \tau_2^{\gamma_2} \dashv \delta_3^{\rho_3}}$$

TYC-UNREACHABLE
$$\frac{\rho_2 \le \rho \quad \Gamma^\alpha; \delta^\rho \vdash e : \bot^\bot \dashv \delta_1^{\rho_1}}{\Gamma^\alpha; \delta^\rho \vdash \mathsf{unreachable}\ e : \tau^\gamma \dashv \delta_2^{\rho_2}}$$

TYC-LET
$$\frac{\begin{array}{c}\Gamma^\alpha; \delta_1^{\rho_1} \vdash e_1 : \tau_1^{\gamma_1} \dashv \delta_2^{\rho_2} \quad t : \tau_1^{\gamma_1} \Rightarrow \overline{R^\gamma} \\ (\Gamma, \overline{|R|})^\alpha; (\delta_2, \overline{R^\gamma})^{\rho_2} \vdash e_2 : \tau_2^{\gamma_2} \dashv \delta_3^{\rho_3}\end{array}}{\Gamma^\alpha; \delta_1^{\rho_1} \vdash (\mathsf{let}\ t := e_1\ \mathsf{in}\ e_2) : \tau_2^{\gamma_2} \dashv \delta_3^{\rho_3}}$$

TYC-RETURN

$$\frac{\begin{array}{c}\rho_2 \le \rho \quad \rho_1 \le \alpha, \gamma' \quad (\sum \bar{S})^{\gamma'}\ \mathsf{ctype} \\ \mathsf{self}^{\sigma}(\bar{R}) : \bar{S} \quad \Gamma^{\alpha}; \delta^{\rho} \vdash \langle \bar{e} \rangle : (\sum \bar{S})^{\gamma'} \dashv \delta_1^{\rho_1}\end{array}}{\Gamma^{\alpha}; \delta^{\rho} \vdash \mathsf{return}\ \bar{e} : \bot^{\gamma} \dashv \delta_2^{\rho_2}}$$

TYC-PROC-CALL

$$\frac{\begin{array}{c}\gamma' \le \gamma \quad \sigma \wedge \rho_2 \le \alpha, \gamma, \sigma' \quad (\sum \bar{R})^{\gamma},\ (\sum \bar{S})^{\gamma'}\ \mathsf{ctype} \\ \mathsf{self}^{\sigma'}(-) : - \quad \mathsf{proc}^{\sigma}\ F(\bar{R}) : \bar{S} \quad \Gamma^{\alpha}; \delta_1^{\rho_1} \vdash \langle \bar{e} \rangle : (\sum \bar{R})^{\gamma} \dashv \delta_2^{\rho_2}\end{array}}{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash F(\bar{e}) : (\sum \bar{S})^{\gamma'} \dashv \delta_2^{\rho_2}}$$

TYC-LABEL

$$\frac{\begin{array}{c}\forall i,\ (\Gamma, \overline{k^{\alpha}(\delta; \bar{R})}, (\bar{R})_i)^{\alpha_i}; (\delta_i, (\bar{R})_i)^{\rho_i} \vdash e_i : \bot^{\bot} \dashv \delta_i'^{\rho_i'} \\ (\Gamma, \overline{k^{\alpha}(\delta; \bar{R})})^{\alpha}; \delta_1^{\rho_1} \vdash e' : \tau^{\gamma} \dashv \delta_3^{\rho_3}\end{array}}{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash (\mathsf{label}\ \overline{k(\bar{R}) := e}\ \mathsf{in}\ e') : \tau^{\gamma} \dashv \delta_3^{\rho_3}}$$

TYC-GOTO

$$\frac{\begin{array}{c}\alpha' \le \alpha \quad \rho_2 \le \rho \quad \rho_1 \le \gamma \quad k^{\alpha'}(\delta_1^{\rho_1}; \bar{R}) \in \Gamma \\ \Gamma^{\alpha}; \delta^{\rho} \vdash \langle \bar{e} \rangle : (\sum \bar{R})^{\gamma} \dashv \delta_1^{\rho_1}\end{array}}{\Gamma^{\alpha}; \delta^{\rho} \vdash \mathsf{goto}\ k(\bar{e}) : \bot^{\gamma'} \vdash \delta_2^{\rho_2}}$$

TYC-ASSERT

$$\frac{\begin{array}{c}\rho_2 \le \sigma, \alpha, \gamma \quad \mathsf{self}^{\sigma}(-) : - \\ \Gamma^{\alpha}; \delta_1^{\rho_1} \vdash e : \mathsf{bool}^{\gamma} \dashv \delta_2^{\rho_2}\end{array}}{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash \mathsf{assert}\ e : e^{\gamma'} \dashv \delta_2^{\rho_2}}$$

TYC-TYPEOF

$$\frac{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash e : \tau^{\bot} \dashv \delta_2^{\rho_2}}{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash \mathsf{typeof}\ e : \boxed{e : \tau}^{\gamma} \dashv \delta_2^{\rho_2}}$$

TYC-ENTAIL

$$\frac{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash \langle \bar{e} \rangle : (\ast\, \overline{A})^{\bot} \dashv \delta_2^{\rho_2} \quad \vdash p : \ast\, \overline{A} \mathrel{-\!\!\ast} B}{\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash \mathsf{entail}\ \bar{e}\ p : B^{\gamma} \dashv \delta_2^{\rho_2}}$$

These rules have the same form as the TYE-* rules (slightly simplified to focus on the new part, the α, ρ, γ annotations). The key new thing to notice is the inequality side conditions in most of the rules. For example:

- TYC-VAR-MOVE requires $\gamma \le \alpha, \rho, \gamma'$ because if the result of the move is actually needed (γ) then we must execute code (α) that is reachable (ρ) and the data to move must actually be available (γ').
- TYC-IF is the main rule that changes α. Inside the branches, α becomes $\alpha \wedge \gamma'$, because if we did not evaluate the condition we can't enter the branches.
- TYC-RETURN requires $\rho_2 \le \rho$, just to ensure the lemma $(\Gamma^{\alpha}; \delta_1^{\rho_1} \vdash e : \tau^{\gamma} \dashv \delta_2^{\rho_2}) \to \rho_2 \le \rho_1$, but in practice we can always set ρ_2 to $\bot$ because it is unreachable. Similar conditions appear in TYC-UNREACHABLE and TYC-GOTO, since these expressions do not terminate normally. The other condition, $\rho_1 \le \alpha, \gamma$ says that if we reach the return (ρ_1), then we must be able to execute the return statement (α), and we need the value to return (γ').

- TYC-PROC-CALL makes use of the σ annotations on functions. If a function has $\sigma = \top$, then it may perform a side effect, so we cannot omit it. We require $\gamma' \leq \gamma$ because if we need the result (γ') then we need the arguments (γ), and we require $\sigma \wedge \rho_2 \leq \alpha, \gamma, \sigma'$ because if the function F is side effecting (σ) and the call is reachable (ρ_2), then we must execute the call (α), we need the arguments (γ), and this function is itself side-effecting (σ').
- TYC-ASSERT requires that $\rho_2 \leq \sigma, \alpha, \gamma$ because if we reach the assert (ρ_2), then because failure is a side effect (σ) we have to execute it (α) and we need the condition (γ).
- In TYC-GOTO, we add α and ρ annotations to $k(\delta, \bar{R})$ to coordinate the entry to this block. Here $\alpha' = \bot$ is a bit unusual, because it means that the block we are jumping to does not physically exist. In this case, we don't need to jump to it, so no code is needed ($\alpha' \leq \alpha$), we have an arbitrary postcondition ρ_2 that may as well be $\bot$, and we need $\rho_1 \leq \gamma$ because if the goto is reachable then we need the value.

We also need an annotated version of the no-op step judgment, in order to weaken variables when they are no longer live. This is exactly the same as CS-*, but with the new rule CCS-GHOST that allows us to make a variable ghost. In particular, since the TYC-MUT rule does not remove the old value of the variable, it is in general a copy and not actually a mutation, so we will want to use CCS-GHOST just after constructing the expression e_1 to kill the old value so that we can safely replace it in-place with the new value.

Ghost annotated no-op step $\boxed{\Gamma; \delta \vdash \cdot \dashv \delta'}$

CCS-REFL
$$\Gamma; \delta \vdash \cdot \dashv \delta$$

CCS-TRANS
$$\frac{\Gamma; \delta_1 \vdash \cdot \dashv \delta_2 \quad \Gamma; \delta_2 \vdash \cdot \dashv \delta_3}{\Gamma; \delta_1 \vdash \cdot \dashv \delta_3}$$

CCS-DROP
$$\frac{\forall x, (x \to y) \notin \delta}{\Gamma; \delta, (y^{\gamma} := pe : \tau) \vdash \cdot \dashv \delta}$$

CCS-RENAME
$$\Gamma; \delta, (x \to y), (y^{\gamma} := pe : \tau) \vdash \cdot \dashv \delta, (x^{\gamma} := pe : \tau)$$

CCS-FORGET
$$\frac{\Gamma \vdash \Gamma[\overline{x \to pe}]}{\Gamma; \delta, \overline{x^{\gamma} := pe : \tau} \vdash \cdot \dashv \delta, \overline{x^{\gamma} : \tau}}$$

CCS-GHOST
$$\frac{\gamma' \leq \gamma}{\Gamma; \delta, (x^{\gamma} := pe : \tau) \vdash \cdot \dashv \delta, (x^{\gamma'} := pe : \tau)}$$

B.3.3 Side effects

While the ghost analysis pass is primarily intraprocedural, it contains one interprocedural part, namely the assignment of σ annotations to the procedures. When $\sigma = \top$, the procedure may perform a side effect, which is defined as anything which performs IO (i.e. compiler intrinsics and syscalls), plus $\mathsf{assert\ false}$ which causes early termination (which is also observable).

If a side effectful operation is reachable from a procedure, then we mark the procedure itself as side effectful (note that func functions cannot have side effects). Note in particular that mutation is not considered a side effect, because the compiler has full visibility into what is going on and can track the values appropriately.

B.4 Optimization and legalization

At this point, we are mostly done with user level errors; if type checking and the ghost analysis pass succeed then we should be able to complete compilation. The only exception to this is types that are too large to exist (which will be caught in this pass) and operations that cannot be compiled, such as unbounded integer operations.

Because the source language makes use of unbounded integer operations even in computationally relevant positions, it is not sufficient to simply require that any variable or expression of type $\mathbb{N}_\infty$ is ghost; for example a reasonable operation might be $x, y : \mathbb{N}_{64} \vdash \mathsf{let}\ z : \mathbb{N}_{64} := (x + y)\ \%\ 2^{64}$, which we expect to be compiled to an ADD instruction, despite the fact that $x + y : \mathbb{N}_\infty$ is an intermediate in this computation.

We call this phase legalization because it performs general rewriting in order to replace source level operations with operations which exist on the target architecture. So for example we can replace the subexpression $(x + y)\ \%\ 2^{64}$ by $x +_{64} y$, where $x +_{64} y$ is addition modulo 2^{64} that we expect to exist on the target machine. Once we have done so, there are no longer any unsized intermediates in the operation, and we can proceed with compilation.

B.5 Semantics

Semantics plays a rather more important role in Metamath C compared to other languages because the target architecture for the compiler is literally separation logic. So we need a way to interpret every judgment just described into a separating proposition or theorem.

B.5.1 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 δ with this information at each variable site.

- For each procedure in the global environment of declared items, we have a (persistent) proposition $\mathsf{proc\text{-}ok}(\ell : \overline{R} \to \overline{S})$ 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 $\mathsf{proc\text{-}ok}(\ell : (\boxed{v : \mathbb{N}}, h : v < n, \overline{R}) \to \overline{S})$ 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 $\mathsf{jump\text{-}ok}(\ell : (\delta, \bar{R}) \to \bot)$ 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 $\mathtt{return}(\bar{R})$ continuation is also a jump target of this form (where the calling convention uses `ret` instead of `jump`).
- Each variable $x : |\tau|$ becomes a (regular) proposition $\boxed{x : |\tau|}$.

The value context δ is extended to Δ by extending some of the variable records with @ μ annotations. They are interpreted like so:

- We store no additional information regarding the rename map.

- Each $x^\gamma := pe : \tau$ may either be left as is or extended to $x^\top$ @ $\mu := pe : \tau$ 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. The former corresponds to the separating proposition $\boxed{pe : \tau}$, and the latter to $\mu \mapsto pe * \boxed{pe : \tau}$.
- For the shared variables extension, we store a list of active locks $x := pe : \tau$ corresponding to uses of the cs-lock rule. We say $x := pe : \tau$ is an active lock if $(\mathsf{ref}^x\ x := pe : \tau) \in \delta$. For each active lock, we also store $\boxed{pe : \tau}$.
- For each heap variable $\mathsf{ref}^x\ y^\gamma := pe' : \tau'$ such that $x := pe : \tau$ is an active lock, we store the pure proposition $(\mu \mapsto pe * \boxed{pe : \tau} \Rightarrow \mu' \mapsto pe' * \boxed{pe' : \tau'})$ if x @ μ and y @ μ', with the μ conjuncts omitted if one or both of x and y is ghost.
- For each heap variable $\mathsf{ref}^{\mathsf{extern}}\ y^\gamma := pe' : \tau'$, we store the pure proposition $(P \Rightarrow \mu' \mapsto pe' * \boxed{pe' : \tau'})$ where P is the frame proposition (that is, P is an implicit additional precise separating proposition passed in and out of the function).

Bibliography

[1] The QED manifesto. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, pages 238–251, 1994.

[2] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formalized Reasoning*, 11(1):43–76, 2018.

[3] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] Jesús Aransay and Jose Divasón. Formalization and execution of linear algebra: From theorems to algorithms. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation*, pages 1–18, Cham, 2014. Springer International Publishing.

[5] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[6] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 84–98, 2009.

[7] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Carnegie Mellon University, 2014.

[8] Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn. *Logic and Proof*. Carnegie Mellon University, 2017.

[9] Seulkee Baek. Reflected decision procedures in lean. Master's thesis, Carnegie Mellon University, 2019.

[10] Clemens Ballarin. Locales and locale expressions in isabelle/isar. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, pages 34–50, 2003.

[11] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pak. The role of the mizar mathematical library for interactive proof development in mizar. *J. Autom. Reasoning*, 61(1-4):9–32, 2018.

[12] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. Mizar: State-of-the-art and beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 261–279, Cham, 2015. Springer International Publishing.

[13] Bruno Barras. Coq en coq. 1996.

[14] Bruno Barras. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.

[15] Bruno Barras and Benjamin Grégoire. On the Role of Type Decorations in the Calculus of Inductive Constructions. In *International Workshop on Computer Science Logic*, pages 151–166. Springer, 2005.

[16] Bruno Barras and Benjamin Werner. Coq in Coq. *Available on the WWW*, 1997.

[17] Stefan Berghofer and Tobias Nipkow. Proof terms for simply typed higher order logic. In *International Conference on Theorem Proving in Higher Order Logics*, pages 38–52. Springer, 2000.

[18] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[19] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media, 2013.

[20] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

[21] Jasmin Christian Blanchette, Max W. Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pages 3–17, 2015.

[22] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[23] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, Mar 2015.

[24] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. Vst-floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning*, 61(1-4):367–422, 2018.

[25] Mario Carneiro. Grammar ambiguity in `set.mm`. 2013.

[26] Mario Carneiro. Models for Metamath. presented at CICM 2016, 2016.

[27] Mario Carneiro. Metamath Zero: The Cartesian Theorem Prover. preprint, 2019.

[28] Mario Carneiro. Specifying verified x86 software from scratch. In *Workshop on Instruction Set Architecture Specification*, 2019.

[29] Alonzo Church. A Formulation of the Simple Theory of Types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[30] Thierry Coquand and Gérard Huet. *The Calculus of Constructions*. PhD thesis, INRIA, 1986.

[31] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148. ACM, June 2019.

[32] Jared Curran Davis and J Strother Moore. *A self-verifying theorem prover*. PhD thesis, University of Texas, 2009.

[33] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in Dependent Type Theory, 2015.

[34] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[35] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[36] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description), 2015.

[37] Peter Dybjer. Inductive Families. *Formal aspects of computing*, 6(4):440–465, 1994.

[38] R. Kent Dybvig. *The SCHEME programming language*. Mit Press, 2009.

[39] Maxime Dénès. Propositional extensionality is inconsistent in Coq, Dec 2013.

[40] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017.

[41] Jordan S. Ellenberg and Dion Gijswijt. On large subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression. *Ann. of Math. (2)*, 185(1):339–343, 2017.

[42] Jean-Christophe Filliâtre and Andrei Paskevich. Why3—where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.

[43] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. *From Frege to Gödel: A source book in mathematical logic*, 1931:1–82, 1879.

[44] M. Ganesalingam and W. T. Gowers. A fully automatic theorem prover with human-style output. *Journal of Automated Reasoning*, 58(2):253–291, Feb 2017.

[45] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference,*

TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, pages 327–342, 2009.

[46] Michèle Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis (Ottawa, Ont., 1980)*, volume 915 of *Lecture Notes in Math.*, pages 68–85. Springer, Berlin-New York, 1982.

[47] Shilpi Goel, Anna Slobodova, Rob Sumners, and Sol Swords. Verifying x86 instruction implementations. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 47–60, 2020.

[48] Georges Gonthier. Point-free, set-free concrete linear algebra. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving*, pages 103–118, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[49] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013.

[50] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013.

[51] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

[52] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[53] Adam Grabowski, Artur Kornilowicz, and Christoph Schwarzweller. On algebraic hierarchies in mathematical repository of mizar. In *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016.*, pages 363–371, 2016.

[54] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 98–113, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[55] Florian Haftmann. Code generation from isabelle/hol theories.

[56] Florian Haftmann and Makarius Wenzel. Constructive type classes in isabelle. In *International Workshop on Types for Proofs and Programs*, pages 160–174. Springer, 2006.

[57] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[58] Jesse Michael Han and Floris van Doorn. A Formalization of Forcing and the Unprovability of the Continuum Hypothesis. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[59] John Harrison. Towards self-verification of hol light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191, Seattle, WA, 2006. Springer-Verlag.

[60] John Harrison. Hol light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[61] John Harrison. The HOL light theory of euclidean space. *J. Autom. Reasoning*, 50(2):173–190, 2013.

[62] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct 1969.

[63] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[64] Hao Huang. Induced subgraphs of hypercubes and a proof of the sensitivity conjecture. *arXiv preprint arXiv:1907.00847*, 2019.

[65] Fabian Immler and Bohua Zhan. Smooth manifolds and types to sets for linear algebra in isabelle/hol. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 65–77, 2019.

[66] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[67] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[68] C. Kaliszyk and K. Pąk. Progress in the independent certification of mizar mathematical library in isabelle. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 227–236, Sep. 2017.

[69] Cezary Kaliszyk, Karol Pąk, and Josef Urban. Towards a mizar environment for isabelle: Foundations and language. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 58–65, New York, NY, USA, 2016. ACM.

[70] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales - A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, pages 149–166, 1999.

[71] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Advances in Formal Methods. Springer US, 2013.

[72] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *International Conference on Interactive Theorem Proving*, pages 362–369. Springer, 2018.

[73] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. *SIGPLAN Not.*, 49(1):179–191, January 2014.

[74] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of cc with predicative induction and judgmental equality. *arXiv preprint arXiv:1111.0123*, 2011.

[75] Holden Lee. Vector spaces. *Archive of Formal Proofs*, 2014. http://isa-afp.org/entries/VectorSpace.html, Formal proof development.

[76] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[77] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012.

[78] Pierre Letouzey. Extraction in Coq: An overview. In *Conference on Computability in Europe*, pages 359–369. Springer, 2008.

[79] Robert Y. Lewis. An extensible ad hoc interface between lean and mathematica. In *Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017.*, pages 23–37, 2017.

[80] Robert Y. Lewis. A formal proof of Hensel's lemma over the p-adic integers. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 15–26, 2019.

[81] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1041–1053. ACM, 2019.

[82] Zhaohui Luo. Ecc, an extended calculus of constructions. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 386–395. IEEE, 1989.

[83] Assia Mahboubi. The rooster and the butterflies. In *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, pages 1–18, 2013.

[84] Assia Mahboubi and Enrico Tassi. Mathematical Components, 2017.

[85] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[86] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982.

[87] Simone Martini. Several types of types in programming languages. In *International Conference on History and Philosophy of Computing*, pages 216–227. Springer, 2015.

[88] Norman Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press, 2019.

[89] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[90] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of cc. In *International Workshop on Types for Proofs and Programs*, pages 240–258. Springer, 2002.

[91] Magnus O Myreen. Formal verification of machine-code programs. Technical report, University of Cambridge, Computer Laboratory, 2009.

[92] Magnus O Myreen and Jared Davis. A verified runtime for a verified theorem prover. In *International Conference on Interactive Theorem Proving*, pages 265–280. Springer, 2011.

[93] Tobias Nipkow. Archive of formal proofs.

[94] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[95] Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[96] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, pages 748–752, 1992.

[97] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. *ACM SIGPLAN Notices*, 51(1):217–231, 2016.

[98] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions, 2015.

[99] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, pages 209–228, 1989.

[100] Robert Pollack. Polishing up the tait-martin-löf proof of the church-rosser theorem. 1995.

[101] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[102] Willard V Quine. New Foundations for Mathematical Logic. *The American mathematical monthly*, 44(2):70–80, 1937.

[103] John C Reynolds. Intuitionistic reasoning about shared mutable data structure. *Millennial perspectives in computer science*, 2(1):303–321, 2000.

[104] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[105] Claudio Sacerdoti Coen. A plugin to export coq libraries to xml. In Cezary Kaliszyk, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 243–257, Cham, 2019. Springer International Publishing.

[106] Daniel Selsam, Percy Liang, and David L. Dill. Developing bug-free machine learning systems with formal mathematics. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 3047–3056, 2017.

[107] Daniel Selsam and Leonardo Moura. Congruence closure in intensional type theory. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 99–115, Berlin, Heidelberg, 2016. Springer-Verlag.

[108] Konrad Slind and Michael Norrish. A brief overview of hol4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[109] Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[110] Matthieu Sozeau, Yannick Forster, and Théo Winterhalter. Coq coq correct!

[111] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[112] Ken Thompson et al. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.

[113] Sebastian Ullrich and Leonardo de Moura. Counting immutable beans: Reference counting optimized for purely functional programming. *IFL 19 (update this when proceedings are published)*, 2019.

[114] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[115] Floris van Doorn. Constructing the propositional truncation using non-recursive hits. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 122–129, 2016.

[116] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. Homotopy type theory in lean. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, pages 479–495, 2017.

[117] Jakob von Raumer. Formalizing double groupoids and cross modules in the lean theorem prover. In *Mathematical Software - ICMS 2016 - 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, pages 28–33, 2016.

[118] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989.

[119] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 33–38, 2008.

[120] Markus Wenzel. Type classes and overloading in higher-order logic. In *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97, Murray Hill, NJ, USA, August 19-22, 1997, Proceedings*, pages 307–322, 1997.

[121] Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, pages 530–546. Springer, 1997.

[122] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, volume 2. University Press, 1912.

[123] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285:85–100, 2012.

[124] H. P. Williams. Fourier's method of linear programming and its dual. *The American Mathematical Monthly*, 93(9):681–695, 1986.

[125] Minchao Wu and Rajeev Goré. Verified Decision Procedures for Modal Logics. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.