A Mechanized Proof in Coq of the Type Soundness of Core L^3

Yawar Raza
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14623
ysr7349@cs.rit.edu

I. Introduction

A. Type Soundness

Statically-typed programming languages have type checkers that detect certain invalid operations, known as type errors, at compile time, reducing the number of errors that would arise at run time. Type checkers are based on a set of rules that determine whether a program contains type errors, which is referred to as a *type system*. A type system is said to accept programs that it considers type-safe and reject programs it considers to have type errors. *Type soundness* is the property that the type system rejects all programs that contain invalid operations, and thus guarantees all type-safe programs will run successfully.

It is not trivial to determine whether a type system is sound. Not only should type systems reject all invalid programs, they should accept as many valid programs as possible. This requires creating more sophisticated type systems with more complex rules. This added complexity creates opportunities for subtle issues to prevent the type system from being sound.

The solution is to use mathematics to prove that a type system does not allow any invalid programs. The language being studied is specified using a formal definition that gives precise meaning to the constructs of the language. This allows type soundness to be represented as a logical formula, which can be proven true by considering all possible expressions and types in the language. If the logical formula for type soundness is successfully proven, it means that a counterexample, no matter how elusive it might be, cannot exist. It is common for research papers that present a new programming language or type system to include a proof of its type soundness.

B. Mechanization of Proofs

Traditionally, proofs of type soundness were developed by hand. Recently, several tools have come into mainstream use that check whether proofs input by the user are correct. *Mechanization* is the act of inputting definitions and proofs into one of these tools. Such tools are often referred to as *proof assistants*. Proof assistants often allow users to input a proof interactively, displaying information about the state of the proof as the user progresses through it. Proof assistants are also sometimes able to carry out some parts of the proof for the user, which is called automation. Coq, Agda, and Twelf are examples of proof assistants, each with different input mechanisms and automation capabilities [1], [2], [3].

Since proof assistants check whether a given proof is correct, they can be used to detect mistakes in proofs for the author to fix. A type soundness proof usually needs to consider numerous cases that are either trivial or similar to other cases. The tedium involved in working through the proof by hand makes it more likely for the author to make small mistakes. Additionally, complex type systems can cause fundamental, yet subtle, problems with completing type safety proofs; Dependent Object Types (DOT) is an example of a language with multiple such problems [4]. Errors in type soundness proofs can also indicate that further research is needed to develop an appropriate proof strategy.

Mechanized proofs have the potential to be more resilient than handwritten proofs. Automation can often solve trivial cases automatically. Automation features can also be used to apply a single fragment of the input proof to multiple similar cases. Sometimes, a researcher wants to study an extension of an existing language. They can add the new features to an existing mechanization; they won't need to worry about any cases already handled by the existing proof, allowing them to focus only on the cases that need additional attention.

C. Contributions

One paper that includes a handwritten type soundness proof, but not a mechanized one, is "L^3: A Linear Language with Locations" [5]. The paper presents L^3, a language that supports type-safe strong updates using linear capability values. L^3 comes in two variants, Core L^3 (often referred to as simply "L^3" in this report) and Extended L^3, each with its own type soundness proof. We aimed to mechanize Core L^3 and its type soundness proof in the Coq proof assistant to verify the paper proof's correctness. While we did not finish mechanizing the proof, we were able to determine suitable ways to represent the metatheoretical constructs used in the paper proof. Some representations of such constructs are eaiser to use in proofs than others, and experimenting with representations to figure out which one worked best was the main activity in this project. This report includes descriptions of some of the representation decisions we had to make.

II. Background

A. Overview of L^3

The goal of L^3 is to design a type system that supports *strong updates*. We will explain strong updates using the following C code:

```
1  typedef struct T { int i; } T;
2  typedef struct U { char* s; } U;
3  T t = {.i = 4};
4  U u = {.s = "hello"};
5  void* p = &t;
6  if (((T*)p)->i < 10) {
7    p = &u;
8    ((U*)p)->s = "world";
9  }
```

The pointer `p`, which initially pointed to a value of type `T`, is reassigned to point to a value of a different type `U`. This is a strong update, and while it is used in a valid way in the above program, it is not supported by mainstream static type systems. Notice how we needed to use `void*` and casts, features that circumvent the type system. This removes our type safety guarantees, like preventing access to `((T*)p)->i` after `p` is assigned to `&u`. L^3 aims to have a type system powerful enough to support strong updates without circumventing the type system, while still satisfying type soundness.

It turns out that simply adding support for strong updates to a standard type system would not satisfy type soundness. L^3 adds an extra restriction that prevents these issues from occurring: all variables must be used exactly once. This is known as a *linear type system*, a long-existing idea that L^3 utilizes to achieve its goal of type-safe strong updates.

Some basic expression forms in L^3 are the unit value $*$, pairs of expressions $\langle e_1, e_2 \rangle$, and functions. Functions are written using lambda calculus notation: $\lambda x.\ e$; this is a function with a single parameter x (a variable) and a body e (an expression). For example, $\lambda x.\ \langle x, x \rangle$ is a function that returns the argument duplicated as a pair. Functions can be applied to an argument: f applied to v looks like $f\ v$ (space-separated juxtaposition). For example, $(\lambda x.\ \langle x, x \rangle)\ *$ applies our function to the unit value, which evaluates to $\langle *, * \rangle$. All three expression forms have types: $*$ has type $\mathbf{I}$, a pair has type $\tau_1 \otimes \tau_2$ if the components have types τ_1 and τ_2 respectively, and a function has type $\tau_1 \multimap \tau_2$ if it accepts arguments of type τ_1 and returns values of type τ_2.

In addition to these basic expressions forms, L^3 has two expression forms fundamental to allowing strong updates. The first form, called pointers, is `ptr` ℓ, which has type Ptr η. The second form, called *capabilities*, is `cap`, which has type Cap $\eta\ \tau$. ℓ and η refer to a separate L^3 construct called *locations*, a simplified notion of memory locations; ℓ refers to *location constants*, while η can refer to both location constants ℓ and *location variables* ρ.

Just like C pointers, L^3 pointers refer to a specific location in memory (the ℓ part of the `ptr` ℓ). Unlike C pointers, the contents of L^3 pointers cannot be accessed without a corresponding capability, where the η in the capability's type matches the η in the pointer's type. The part that's important for keeping strong updates safe, and where the linear type system comes into play, is that capabilities can only be used once. When a pointer containing type τ_1 is assigned a value τ_2, the capability of type Cap $\eta\ \tau_1$ is consumed and a new capability with type Cap $\eta\ \tau_2$ is made available. Without this restriction, the old capability Cap $\eta\ \tau_1$ could be used to try to obtain a value of type τ_1 from the pointer even after it had already been replaced with a value of type τ_2! Thus, only the most up-to-date capability is allowed to be used. As a bonus, because a linear type system requires *exactly* one use rather than *at most* one use, the last capability must be consumed without being replaced, which can only be done by deallocating the pointer, thus preventing memory leaks.

In addition to its linear type system, another unusual aspect of L^3 is that the paper presents a *semantic* type soundness proof, whereas most papers present "syntactic" type soundness proofs. More about the semantic approach to proving type soundness will be covered in the "Semantic Interpretations" subsection of the "Mechanization Details" section below.

While this project focuses on Core L^3 , the L^3 paper also describes an Extended L^3. The use of linear capabilities in Core L^3 does not allow for certain types of recursive pointer graphs that are allowed in languages like Standard ML. While L^3 cannot support strong updates for these cases, it is desirable to have them in the language and allow only standard, type-preserving updates on them. Extended L^3 does this, and allows such pointers to switch between allowing strong updates but only supporting less flexible recursion, and forbidding strong updates but allowing for more flexible recursion. Extended L^3 has additional forms for expressions and types, in addition to several additional constructs in the rules of the language itself and in its type soundness proof.

B. Introduction to Coq

We give an introduction to Coq by giving simple examples how some concepts from arithmetic can be mechanized. The constructs Coq gives us can allow concepts from any domain to be mechanized, as long as they can be specified precisely. The domain this project focuses on is programming languages; these constructs are used to define the structure and rules of L^3. Concepts from other domains, such as the natural sciences, may also be mechanized using these constructs. This is similar to how concepts from any domain can be represented in software applications.

1) Natural Numbers and Addition: This is how the natural numbers (non-negative integers) are represented in Coq:

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat
.
```

The `Inductive` keyword defines a new data type in Coq. It has two cases: the value `O` and the constructor `S`. `O` simply represents the number 0. `S` wraps another natural number and represents the successor of that natural number (i.e. that number plus 1). For example, the number 4 is represented as

`S (S (S (S O)))`, since $4 = 1+(1+(1+(1+0)))$. Both `O` and `S` are "tags" that are distinguishable from one another, which means that the two cases are disjoint by construction. Furthermore, you can query a **nat** value for which case it belongs to, and if it belongs to the `S` case, you can extract the smaller **nat** contained in it. This is done using pattern-matching:

```
match S (S (S O)) with
| O =>
    (* this case is not evaluated *)
| S n =>
    (* this case is evaluated *)
    (* n is S (S O) inside this case *)
end
```

While pattern-matching is boring in the above example, it is very useful in the definition of functions. Because **nat** is inductively-defined (due to `S` containing another **nat**), we use recursive functions to analyze **nat** values. For example, this is the definition of a function that adds its arguments together:

```
Fixpoint add (m : nat) (n : nat) :=
  match n with
  | O => m
  | S n' => S (add m n')
  end
.
```

The **Fixpoint** keyword defines a function in Coq. This `add` function has two parameters `m` and `n`, each with type **nat**. We pattern-match on `n` so that each case can be implemented differently. If `n` is `O`, then $m+n = m+0 = m$, so we just return `m`. If `n` is an `S` case, then we bind the number contained in `n` to the variable `n'`. We have that $n = n'+1$, so $m+n = m+(n'+1) = (m+n')+1$. This corresponds to the case body `S (add m n')`. Essentially, we are iteratively "peeling off" the `S`'s from the second argument and tacking them onto the final result until there are none left. The inside of the final result is `m`, so the final result has the n `S`'s from `n` and the m `S`'s from `m`, meaning it has exactly $m+n$ `S`'s, so it is equal to $m+n$.

2) Defining $\leq$: While we defined `add` as a function above, predicates such as $\leq$ are defined as relations more often than they are defined as boolean-returning functions. To represent $\leq$ as a relation, we need to give declarative rules that fully characterize its definition. Here is one such set of rules:

$$\forall\ n.\ n \leq n$$

$$\forall\ m, n.\ m \leq n \Rightarrow m \leq n+1$$

These rules can then be directly represented in Coq:

```
Inductive le : nat -> nat -> Prop :=
  | le_eq : forall n,
      le n n
  | le_tr : forall m n,
      le m n ->
      le m (S n)
.
```

Like our definition of **nat**, this definition uses the **Inductive** keyword. However, **nat** has the *sort* **Set**, indicating that it is a normal data type. Here, `le` has the sort **nat** `->` **nat** `->` **Prop**, which indicates that `le` is a relation between two **nat**'s. The **forall** keyword maps to universal quantification, while the `->` is used here to represent implication. Cases of a relation are given names, here `le_eq` and `le_tr`, so that we can specify which rule we would like Coq to apply in a given situation.

Suppose we wanted to prove `le 2 5`. We apply the `le_eq` rule to show that `le 2 2`. Then, we apply the `le_tr` rule three times to get `le 2 3`, then `le 2 4`, then finally `le 2 5`. The preferred way of applying rules like these is by using tactics, which we will discuss further in the "Proofs" subsection of the "Mechanization Details" section. Another way of applying rules is by explicitly using them as constructors; just like `S` wraps a smaller natural number to produce its successor, `le_tr` wraps a proof of `le m n` to produce a proof of `le m (S n)`, and thus the same syntax used to construct values of **nat** can be used to construct proofs of `le`.

3) Defining $\leq$: Another Possibility: We gave one set of declarative rules that can be used to define $\leq$. Here is a different set of rules that also defines $\leq$:

$$\forall\ n.\ 0 \leq n$$

$$\forall\ m, n.\ m \leq n \Rightarrow m+1 \leq n+1$$

These rules can also be represented in Coq:

```
Inductive le' : nat -> nat -> Prop :=
  | le'_O : forall n,
      le' O n
  | le'_S : forall m n,
      le' m n ->
      le' (S m) (S n)
.
```

Like our last definition, we can use this to prove `le' 2 5`. However, the proof proceeds differently. First, we apply `le'_O` to prove `le' 0 3`. Then, we apply `le'_S` twice times to get `le' 1 4` then `le' 2 5`. Thus, while we are able to mechanize $2 \leq 5$ with either `le` and `le'`, the proofs themselves are different for both. Additionally, since both are defined using different rules, more general properties are also proven differently using both definitions. For example, $\forall\ m, n.\ m \leq m+n$ is easier to prove using `le` than `le'`. On the other hand, $\forall\ m, n.\ m \leq n+m$ is easier to prove using `le'` than `le`. This also shows that the implementation of functions is also important; pattern matching on `m` rather than `n` in `add` would switch which property is easier to prove for each `le`. The main takeaway is that, when it comes to mechanization, the way something is represented not only needs to be correct, but it should also be easy to use for the specific properties one is proving. This came up several times in our mechanization of L^3; these challenges are described in the "Mechanization Details" section.

4) Add as a Relation: The relation we defined above corresponded to $\leq$, which is often thought of as "boolean-valued". However, relations can be used for things that aren't often thought of as boolean. For example, this is how addition can be defined as a relation:

```
Inductive add'
  : nat -> nat -> nat -> Prop :=
  | add'_O : forall m,
      add' m O m
  | add'_S : forall m n s,
      add' m n s ->
      add' m (S n) (S s)
.
```

The first two arguments to the relation are the numbers being added, while the third number is the sum. Rather than calculating the sum of two numbers, this relation determines whether an existing number is the sum of the two other numbers. This can be viewed as boolean-valued by it being the answer to the question "Does $m+n=s$?" While it seems less powerful to require the sum to be provided externally, rather than calculated, relations have some properties that make them easier to use than functions in some cases. In our mechanization of L^3, we used a relation to represent the disjoint union of two environments, which is not always defined; relations can work better than functions for partially-defined operations such as this.

III. Mechanization Details

We split our mechanization process into five parts, mirroring the formalization presented in the paper: syntax, operational semantics, static semantics, semantic interpretations, and the soundness proof itself. The mechanization of each part is described in the following sections.

A. Syntax

The syntax of L^3 contains three categories of terms: locations, types, and expressions. As is typical for Coq mechanizations, each of these categories is represented as a non-dependent algebraic data type in Coq. For example, our grammar for expressions maps to a Coq `Inductive` data type with kind `Set` that we call `expr`, and the $\langle e_1, e_2 \rangle$ expression form (representing a pair, or binary tuple) maps to a case in `expr` with the type `expr -> expr -> expr`.

One of the most fundamental decisions when mechanizing a programming language is choosing how to handle variables. Consider an infinite, atomic type called `var` where you can check whether two members are equal or not. This type is typically implemented as an opaque type wrapping `nat`, the type of the natural numbers. One way to represent variables, which we call the *named representation*, is to simply create a case in `expr` with the type `var -> expr`. The problem with this approach is that the expressions $\lambda x.\ x\ z$ and $\lambda y.\ y\ z$ are α-equivalent, but their named representations are not equal in Coq. Research papers assume implicit α-renaming to understand such terms as equal, but Coq does not support such implicit renaming. Using an explicit binary relation to recognize α-equivalent terms would get unwieldy to insert everywhere we need it, so we consider other approaches to representing variables.

One widely-known approach is the use of *de Bruijn indices*. This representation eschews named variables in favor of a *nameless* representation, where variables are not assigned names when they are introduced in lambdas. To refer to a variable in this representation, a *de Bruijn index* (a natural number) is used to indicates the where the variable was introduced relative to where it is used. This natural number counts the number of containing lambdas that are "passed" when "going outwards" from a variable's usage point to its introduction point. For example, $\lambda x.\ \lambda y.\ y$ is represented by $\lambda.\ \lambda.\ 0$ because we skip over 0 lambdas to get from where y is used to where y is introduced. On the other hand, $\lambda x.\ \lambda y.\ x$ is represented by $\lambda.\ \lambda.\ 1$ because we skip over 1 lambda, the one introducing y, to get from where x is used to where x is introduced. The problem with this representation is that it is unclear how to represent free variables, as there is no lambda that introduces them. There is an approach that extends de Bruijn indices to also refer to free variables by indexing into an environment implemented as an ordered list. However, this approach is problematic for us as L^3, with its linear type system, splits environments among subexpressions, so the resultant environments each generally *exclude* some free variables from the original environment. This means that the de Bruijn indices for free variables would need to be reassigned in order to correspond to the smaller environment.

A hybrid between these two approaches, described in detail by Charguéraud, is the *locally nameless representation* [6]. This approach represents bound variables using de Bruijn indices, and thus does not assign names to variables introduced by lambdas. However, free variables are referred to using names as in the named representation. For example, $\lambda y.\ y\ z$ is represented as $\lambda.\ 0\ z$, where the bound variable y is given a de Bruijn index, but free variable z uses an explicit name. Notice that our problem with the named representation involves bound variables and our problem with de Bruijn indices involves free variables, so this hybrid representation solves both of those problems. Thus, this is the approach we use in our mechanization.

Charguéraud also details how to mechanize the locally nameless syntax, which this paragraph will summarize. Since free variables and bound variables are syntactically distinct, we represent them using two separate cases: a bvar case and an fvar case. For example, `expr` has a case `expr_bvar : nat -> expr` for bound variables and a case `expr_fvar : var -> expr` for free variables. The case for lambdas has the type `expr -> expr`, since we do not specify a name upon introducing a variable. There are also several other expression forms and type forms that introduce variables; those are specified similarly. We need to define an operation that turns a bvar into an fvar, which is called *variable opening*. This is used when we analyze the body of a lambda isolated from the lambda itself. As is usual for mechanized programming languages, we also need to define

the variable substitution operation that replaces a particular fvar with another expression. Often, we would like to replace a bvar with another expression directly, without going through an intermediate fvar, which we can do by generalizing variable opening to *opening* with an arbitrary expression.

L^3 also defines a particular class of expressions it calls "values" (which we will call *expression values* in this report). To mechanize this, we define a predicate on expressions as an **Inductive** GADT with kind `expr -> Prop` that we call `expr_val`. The cases of this data type correspond to the different forms that expression values can take. For example, a pair $\langle e_1, e_2 \rangle$ is an expression value iff both e_1 and e_2 are expression values. This is represented with the following case:

```
| expr_val_pair : forall v1 v2,
    expr_val v1 ->
    expr_val v2 ->
    expr_val (expr_pair v1 v2)
```

Another predicate we need on terms is an assertion that that they are *closed*, meaning that they have no free variables. A predicate we need due to the locally nameless representation is an assertion that terms are *locally closed*, meaning that every bvar actually refers to the variable introduced by a lambda. For example, $\lambda.\ 1$ is not locally closed as there aren't enough enclosing lambdas for 1 to actually refer to a variable. These predicates are similarly defined using **Inductive** data types with kinds having a return sort of **Prop**.

B. Operational Semantics and Static Semantics

Operational semantics define how a program can be evaluated. The operational semantics of L^3 is defined as a relation of the form $(\sigma, e) \longmapsto (\sigma', e')$. e and e' are L^3 expressions. σ and σ' are both instances of a construct called a *store*, which maps location constants to the expression values currently stored at those locations. The relation says that, starting from the configuration (σ, e), one can take a single evaluation step to get (σ', e'). This is implemented in Coq as an **Inductive** data type of kind `store -> expr -> store -> expr -> Prop`. The operational semantics consists of a set of rules that apply to different forms of expressions. For example, $(\sigma, (\lambda x.\ e)\ v) \longmapsto (\sigma, e[v/x])$ says that a lambda applied to a expression value substitutes the expression value into the body of the lambda. (Note that this direct substitution differs from most real-world implementations of programming languages, which would instead assign v to the memory location corresponding to x, which is later accessed to resolve the uses of x.)

Static semantics typically define when and how an expression can be assigned a type. The static semantics of L^3 is defined as a relation of the form $\Delta; \Gamma \vdash e : \tau$. e is an L^3 expression and τ is an L^3 type. Δ, a *location context*, is the set of location variables in scope. Γ, a *value context*, maps the set of free variables in scope to the types they are assumed to have. The relation says that, in the combined scope of Δ and Γ, e can be assigned the type τ. This is implemented in Coq as an **Inductive** data type of kind `loc_ctxt -> val_ctxt -> expr -> ty -> Prop`. Like operational semantics, the static semantics consists of a set of rules that apply to different forms of expressions. For example, the rule for typing lambdas says that $\Delta; \Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2$ if $\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2$; that is, if the body of e has type τ_2 given that x has type τ_1, then $\lambda x.\ e$ has type $\tau_1 \multimap \tau_2$, all within the contexts Δ and Γ.

Both stores and value contexts map some sort of atom to some sort of value. They can be generalized into a construct known as an *environment*. L^3, unlike most programming languages, is linear, which means that we need the ability to take the disjoint union of two stores or two value contexts. The disjoint union of two environments E_1 and E_2 is the environment that contains all bindings in E_1 and E_2 and it requires that E_1 and E_2 have disjoint domains. The typing rule for pairs is an example of where the disjoint union (denoted as $\boxplus$ here) is used to split the value context between both subexpressions:

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \tau_1 \qquad \Delta; \Gamma_1 \vdash e_2 : \tau_2}{\Delta; \Gamma_1 \boxplus \Gamma_2 \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2}$$

We needed to determine how to mechanize these environments in a way that also supports the disjoint union operation. There are two common ways that environments can be mechanized:

- An environment can be implemented as a list of pairs `[(x1, v1), (x2, v2), ...]`, where `x1` maps to `v1`, `x2` maps to `v2`, and so on.
- An environment can be implemented as a Coq function, say `f`, where `f x1` is `Some v1` iff `x1` maps to `v1` and `f x2` is `None` if `x2` does not map to a value.

1) Attempt 1: A problem with using lists as environments is that Coq does not recognize different permutations of the same set of pairs as equal, while behaviorally they are equivalent. Thus, we initially decided to use Coq functions to represent both stores and value contexts. We also needed to figure out how to represent the notion of a disjoint union. Because the disjoint union operation is a partial function, we implemented a ternary relation `disjoint_union` with kind `env A -> env A -> env A -> Prop` such that `disjoint_union E1 E2 Eu` means that the disjoint union of `E1` and `E2` is `Eu`. It was implemented in the following way:

```
Inductive xalt :=
  | xalt_left : forall v,
      xalt (Some v) None (Some v)
  | xalt_right : forall v,
      xalt None (Some v) (Some v)
  | xalt_neither :
      xalt None None None
.
Definition disjoint_union E1 E2 Eu :=
  forall x, xalt (E1 x) (E2 x) (Eu x)
.
```

Notice how this says that `Eu x` is defined iff either `E1 x` is defined or `E2 x` is defined, but not both, and that the value of `Eu x` matches that of the environment it came from. The

lack of an `xalt_both` case reflects that `E1` and `E2` must be disjoint for their disjoint union to be defined.

2) Attempt 2: The problem we encountered with the previous definition is that we needed to be able to construct the disjoint union of two environments in one of our proofs. We could implement this using an existence lemma, but this lemma also requires a premise that `E1` and `E2` have disjoint domains. With this, the `disjoint_union` relation is somewhat redundant; combining a `disjoint` predicate with a total `union` function accomplishes the same goal, so we switched our approach to that.

Another problem we encountered is that for parts of our proof, the finiteness of environments is important. While we know our environments must be finite due to how we construct them, this knowledge is lost in their representation as Coq functions, which can be defined on an infinite domain. Thus, this proof of finiteness must be carried around separately. To implement this proof of finiteness, we initially mirrored the ways environments can be constructed:

```
Inductive finite : env A -> Prop :=
  | finite_empty :
      finite empty
  | finite_singleton : forall k v,
      finite (singleton k v)
  | finite_union : forall E1 E2,
      finite E1 ->
      finite E2 ->
      finite (union E1 E2)
.
```

We later replaced the `singleton` and `union` cases with a single `add` case that adds just one binding to the environment, as that was easier to perform induction on. However, an issue we ran into was that performing inversion on `finite (add k v E)` does not result in `finite E`, since the last layer of the proof term can be for *any* binding in the environment, not just the mapping of `k` to `v`. We found that it is generally hard to use an inductive definition like `finite` with the function representation of environments.

3) Attempt 3: We decided to revisit the option of using a list of pairs as our representation of environments. For our operational semantics and static semantics, we knew that it would not be possible to construct proof terms for *all* possible permutations of the stores and value contexts. However, we only need it to be always possible to construct such proof terms for *some* permutation of the stores and value contexts. At this point, we decided to use the TLC library for its implementations of `var` and `env` and its many lemmas about those data types [7]. We define disjoint union using the definition given in a different version of the L^3 paper [8]. To do this, we first define a `merge` relation as follows:

```
Inductive merge :=
  | merge_empty :
      merge empty empty empty
  | merge_left : forall E1 E2 Eu k v,
      merge E1 E2 Eu ->
      merge (E1 & k ~ v) E2 (Eu & k ~ v)
  | merge_right : forall E1 E2 Eu k v,
      merge E1 E2 Eu ->
      merge E1 (E2 & k ~ v) (Eu & k ~ v)
.
```

Then, we note that TLC defines a predicate `ok` with kind `env A -> Prop` where `ok E` means that `E` does not have multiple mappings from the same variable. If we have both `merge E1 E2 Eu` and `ok Eu`, then `E1` and `E2` must be disjoint, since if they both mapped the same variable, `Eu` would then have two mappings from the same variable, which `ok Eu` forbids. This means that `merge E1 E2 Eu` is analogous to our previous `disjoint_union E1 E2 Eu` relation, as long as `ok Eu` is also a hypothesis wherever `merge` is used. This is the approach we settled on for mechanizing environments and the disjoint union operation. Thus, the typing rule for pairs we stated earlier is mechanized as:

```
| typed_pair :
  forall D G1 G2 G12 e1 e2 t1 t2,
    ok G12 ->
    merge G1 G2 G12 ->
    typed D G1 e1 t1 ->
    typed D G2 e2 t2 ->
    typed D G12
      (expr_pair e1 e2)
      (ty_prod t1 t2)
```

While we have been using the terms *disjoint union* and *merge*, it is more appropriate to think about proving the type of a pair by *splitting* the value context between the two components of the pair, to satisfy L^3's linearity, which is done by the `merge` relation. Furthermore, the `merge` relation allows for *any* split that is needed for the typing judgment to be proven.

C. Automating an Example Program Evaluation

While this project focuses on proving metatheoretical properties (properties about *all* programs) such as type soundness, we should be able to use the operational semantics we mechanized to prove that a *particular* program evaluates to the correct final value. Doing this provides an opportunity to catch simple errors in the mechanization, as those will likely prevent the (known correct) evaluation from being able to be proved using it. We did this for our project at an earlier stage of our mechanization; we took an example program from the L^3 paper and were able to prove the evaluation of that program was as expected. However, as we modified how we represented our operational semantics, such as changing our representation of environments, that proof became difficult to maintain. Since it did not directly contribute to the proof of type soundness, we decided to focus our work on the aspects of the mechanization that did.

Unlike metatheoretical proofs, proofs of evaluation are extremely straightforward and lengthy. Thus, these are typically proven by utilizing Coq's automation features to construct the entire proof. In our case, this wasn't completely automatic,

however; there were some places where Coq needed some help to direct its proof search in the right direction. We discuss a noteworthy example of this that we encountered. There is one operational semantics rule that recursively applies a rule to some subexpression of the current expression being evaluated. Such a subexpression is extracted using an *evaluation context*, which has several possible forms; in particular, all forms are recursive except for a "trivial" form that serves as the base case. The automation involved trying out each of these forms in turn; by default, the order they are tried in remained the same at each level. If we chose the "trivial" form to always go first, the subexpression extracted by the trivial context would be the same as the original term, and the recursive operational semantics would loop, infinitely trying the same term over and over. If we chose the "trivial" form to always go last, then the automation would only ever extract "leaf" subexpressions that can never be simplified by the operational semantics, and would not find the bigger subexpression that can be simplified. Thus, we needed to use a "helper tactic" to never search the trivial form at the top level, but search it first at all subsequent levels. This guaranteed that the search always made progress and was exhaustive. This is similar to the problem in Prolog where defining a recursive rule before the base case rules results in an infinite search during execution.

D. Semantic Interpretations

Unlike most proofs of type soundness, L^3's proof of type soundness uses auxiliary definitions called *semantic interpretations*. The primary one is $\mathcal{V}$, which *interprets*, or views, a type τ as the set of possible final configurations (σ, v) where v has type τ ($\mathcal{V}$ stands for "value", since all such v are expression values). For example, $\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]$, the set of final configurations with the product type $\tau_1 \otimes \tau_2$, is

$$\{(\sigma_1 \uplus \sigma_2, \langle v_1, v_2 \rangle) \mid (\sigma_1, v_1) \in \mathcal{V}[\![\tau_1]\!] \wedge (\sigma_2, v_2) \in \mathcal{V}[\![\tau_2]\!]\}$$

where $\uplus$ is the disjoint union of stores and $\langle v_1, v_2 \rangle$ is an L^3 pair expression.

1) Representing $\mathcal{V}$: Representing $\mathcal{V}$ in Coq took some tweaking, because of some restrictions that Coq has. Such restrictions are in place to make sure Coq is logically consistent, which means that Coq doesn't allow a proof for a false proposition. These restrictions, however, can reject certain ways of expressing valid definitions, such as $\mathcal{V}$. We needed to find a way of stating its definition that satisfies all of Coq's restrictions.

a) Attempt 1: We chose to define $\mathcal{V}$ by representing the proposition $(\sigma, v) \in \mathcal{V}[\![\tau]\!]$, since a membership predicate can always be used to unambiguously define a set. We represent this as the ternary relation $V(\tau, \sigma, v)$, which holds whenever the specified τ, σ, and v satisfy the predicate. This was implemented as an `Inductive` data type of kind `ty -> store -> expr -> Prop`. The issue with this is that the membership predicate for the $\mathcal{V}[\![\tau_1 \multimap \tau_2]\!]$ (function types) case has the form $\forall \sigma_1, v_1.\ (\sigma_1, v_1) \in \mathcal{V}[\![\tau_1]\!] \Rightarrow P$ for some proposition P (omitted here). In Coq, this looks like:

```
(* "..." denotes omitted code *)
| V_fun : forall t1 ...,
    (forall s1 v1,
      V t1 s1 v1 -> ...) ->
    V (ty_fun t1 t2) ... ...
```

Note the recursive use of `V`. Recursive uses of `V` are common, like in the rule for $\mathcal{V}[\![\tau_1 \otimes \tau_2]\!]$ we stated earlier. However, this recursive use is to the *left* of an arrow (`->`) within an expression that is itself also to the *left* of an arrow. Coq does not allow expressions to the left of more than one arrow, as this can be used to construct an "infinite" invalid proof.

b) Attempt 2: We still represent the membership predicate $(\sigma, v) \in \mathcal{V}[\![\tau]\!]$, but we instead represent $\mathcal{V}$ a function V with a single parameter τ, where $V(\tau)$ represents $\mathcal{V}[\![\tau]\!]$. Thus, the relations R returned by this function should satisfy $(\sigma, v) \in R$. We represent this type-specific membership predicate as a relation in Coq, meaning that `V (t : ty)` has return kind `store -> expr -> Prop`. (This means that, while defined differently, `V` has the same type as in the previous attempt.)

This solves the problem from our previous attempt; `V t1` and `V (ty_fun t1 t2)` return different relations, so there is no recursive use of relations anymore. However, Coq also needs to forbid infinite loops in functions, such as `V`. It does this by requiring that one of the arguments to the function become "smaller" in any recursive use; specifically, it must be a strictly smaller subterm of the original argument. For example, the case for `V (ty_fun t1 t2)` has a recursive call `V t1`, which Coq allows since `t1` is a subterm of `ty_fun t1 t2`. This prevents infinite loops because the number of recursive calls is now upper-bounded by the size of the original argument (which Coq requires to be finite).

Most cases of $\mathcal{V}$ only require the corresponding definition in `V` to have recursive calls on subterms, but $\mathcal{V}[\![\forall \rho.\ \tau]\!]$ and $\mathcal{V}[\![\exists \rho.\ \tau]\!]$ have a recursive call on $\tau[\ell/\rho]$, which substitutes ℓ for ρ in τ. While this term is the same size as τ, and thus smaller than the original argument, it is not a subterm of the original argument, so Coq doesn't allow the definition of `V` to include these recursive calls.

c) Attempt 3: $\tau[\ell/\rho]$ differs from τ only in the locations that occur in the type. We construct a function f that "erases" all locations from the type (equivalently, mapping them to a unit value). For example, $f(\mathsf{Ptr}\ \ell) = \mathsf{Ptr'}$ and $f(\mathsf{Cap}\ \ell\ \tau) = \mathsf{Cap'}\ f(\tau)$. Note that $f(\tau[\ell/\rho]) = f(\tau)$, since all occurrences of the substituted location were erased. We define a helper function V' with two arguments τ and τ'. We implement the cases of $\mathcal{V}$ in V', and define $V(\tau)$ as $V'(\tau, f(\tau))$. τ' in V' is a "dummy" parameter that we only use for recursive calls. Since recursive calls of V' in $V'(\tau, \tau')$ always use a subterm of τ' as the second argument, Coq can now detect that V' terminates. This method of adding a dummy parameter is a common way to convince Coq that a function terminates.

2) Other Semantic Interpretations: In addition to $\mathcal{V}$, the paper defines other semantic interpretations. $\mathcal{C}[\![\tau]\!]$ is the set of configurations (σ, e) that can execute further, using L^3's

operational semantics, to result in a configuration in $\mathcal{V}[\![\tau]\!]$ ($\mathcal{C}$ stands for "computation"). $\mathcal{V}$ and $\mathcal{C}$ actually reference each other, which would suggest using mutually recursive functions V and C in Coq. However, mutually recursive calls also need to be on strictly smaller subterms, but $\mathcal{C}[\![\tau]\!]$ uses $\mathcal{V}[\![\tau]\!]$ (where both τ's are the same), so this is rejected when implemented in Coq. To solve this, we create a function $C'(R)$ that implements the definition of $\mathcal{C}$, except $V(\tau)$ is replaced with R. Thus, $\mathcal{C}[\![\tau]\!]$ is implemented as $C'(V(\tau))$; when translating the cases of $\mathcal{V}$ to V', uses of $\mathcal{C}$ get translated to $C'(V'(t, t'))$ (for some t and t'). This makes V' a normal recursive function, and C' is not recursive at all, so there is no mutual recursion. Since all uses of $\mathcal{C}$ in the rules of $\mathcal{V}$ were on strictly smaller subterms, these newly introduced recursive calls to V' are on strictly smaller subterms, and Coq accepts this definition of V'.

Before discussing further semantic interpretations, we need to define the notion of a *substitution* used in L^3 paper. A value substitution γ assigns all free variables of a term a closed expression value. When the substitution is applied to the term, it replaces each free variable use with the assigned expression value, resulting in a closed expression. A location substitution δ is similar, except it assigns free location variables to location constants. For both variants of substitutions, we use the implementation of environments we described above that we used for both stores and value contexts.

The $\mathcal{S}[\![\Gamma]\!]\delta$ semantic interpretation is a set of (σ, γ) pairs where γ is a substitution assigning to all variables in the value context Γ. The paper defines $\mathcal{S}[\![\Gamma]\!]\delta$ inductively on Γ, which maps nicely to our list implementation of environments. The paper does not explicitly define an analogous $\mathcal{D}[\![\Delta]\!]$ for the set of δ substitutions assigning to all location variables in Δ; it simply uses the predicate $dom(\delta) = dom(\Delta)$ instead of $\delta \in \mathcal{D}[\![\Delta]\!]$ ($\mathcal{S}$ cannot use this approach as it involves extra conditions related to the associated σ). We, however, found it useful to explicity define such a $\mathcal{D}$ as an explicit Coq relation, using an inductive definition similar to that of $\mathcal{S}$, since we also used our list-based environment implementation for location contexts such as Δ.

The final semantic interpretation is $[\![\Delta; \Gamma \vdash e : \tau]\!]$, which interprets the typing judgment $\Delta; \Gamma \vdash e : \tau$. It gives an *alternate definition* of typing, separate from the static semantics:

$$\begin{aligned}&\forall\ \delta, \sigma, \gamma.\\ &\quad \delta \in \mathcal{D}[\![\Delta]\!] \wedge (\sigma, \gamma) \in \mathcal{S}[\![\Gamma]\!]\delta \Rightarrow\\ &\quad (\sigma, \gamma(\delta(e))) \in \mathcal{C}[\![\delta(\tau)]\!]\end{aligned}$$

The paper the defines type soundness as the property that any typing judgment valid in the static semantics is also valid according to the semantic interpretation definition:

$$\begin{aligned}&\forall\ \Delta, \Gamma, e, \tau.\\ &\quad \Delta; \Gamma \vdash e : \tau \Rightarrow\\ &\quad [\![\Delta; \Gamma \vdash e : \tau]\!]\end{aligned}$$

Why is this a reasonable definition of type soundness? First, note that $\mathcal{C}$ is defined in terms of $\longmapsto^*$ and $[\![\Delta; \Gamma \vdash e : \tau]\!]$ is defined in terms of $\mathcal{C}$. Therefore, $[\![\Delta; \Gamma \vdash e : \tau]\!]$ is essentially a property of the operational semantics of L^3. Recall that type soundness is the property that well-typed programs always completely execute (to an expression value):

$$\begin{aligned}&\forall\ e, \tau.\\ &\quad \bullet; \bullet \vdash e : \tau \Rightarrow\\ &\quad \exists\ \sigma, v.\ (\{\}, e) \longmapsto^* (\sigma, v) \wedge v \text{ is an expression value}\end{aligned}$$

where $\bullet$ is the empty context. This follows directly from the paper's definition of type soundness, when instantiating both Δ and Γ to the empty context; this is approximately "Corollary 3.1." in the paper. It cannot be proven directly, since the inductive hypothesis is not strong enough, thus why $[\![\Delta; \Gamma \vdash e : \tau]\!]$ is defined on general Δ and Γ.

E. Proofs

1) A Case of the Type Soundness Proof: In the paper, type soundness is proven by induction on the typing judgment, which is the typical approach of most type soundness proofs. Thus, the proof has cases for each rule of the static semantics. We have completed the mechanization for the simplest case:

$$\frac{}{\Delta; \bullet \vdash * : \mathbf{I}}$$

Here, $\mathbf{I}$ represents the unit type (a canonical singleton type) and $*$ represents its only inhabitant (which is also an expression value). The details of this proof case, which we used as reference when mechanizing it, are included in the technical report version of the paper [9].

In the user interface of Coq, the current proof state is split into two parts. The top half, or the *proof context* lists all variables in scope and hypotheses that are assumed, while the bottom half states the current *goal*, or proposition being proven. Coq proofs are typically "top-down", meaning that given a goal P, the user will tend to apply an implication $P' \Rightarrow P$, which changes the goal to P', to proceed in the proof. Paper proofs, on the other hand, are presented in a more "bottom-up" fashion, starting with hypotheses $\bar{H}$ and applying an implication $H_1 \wedge ... \wedge H_n \Rightarrow H'$ to add a new hypothesis H' to the list. Because of this difference, our proof does not have the same flow as the one presented in the paper.

Our initial goal, after expanding all definitions, is:

$$\begin{aligned}&\forall\ \Delta.\\ &\forall\ \delta, \sigma_s, \gamma.\ \delta \in \mathcal{D}[\![\Delta]\!] \wedge (\sigma_s, \gamma) \in \mathcal{S}[\![\bullet]\!]\delta \Rightarrow\\ &\forall\ \sigma_r, \sigma_{s,r}.\ \sigma_s \uplus \sigma_r = \sigma_{s,r} \Rightarrow\\ &\quad \exists\ \sigma_f, \sigma_{f,r}, v_f.\\ &\qquad \sigma_f \uplus \sigma_r = \sigma_{f,r}\\ &\qquad \wedge (\sigma_{s,r}, \gamma(\delta(*))) \longmapsto^* (\sigma_{f,r}, v_f)\\ &\qquad \wedge (\sigma_f, v_f) \in \mathcal{V}[\![\mathbf{I}]\!]\end{aligned}$$

Note that, unlike the paper proof, we give a name to the result of the disjoint union of two stores. That is because the paper treats disjoint union as a partially-defined operator, while we define it with our ternary `merge` relation.

A Coq proof is made up of commands called *tactics*. We use a tactic to move the first three lines of this goal to the proof context, leaving us with a goal of:

$$\begin{aligned}\exists\ & \sigma_f, \sigma_{f,r}, v_f. \\ & \sigma_f \uplus \sigma_r = \sigma_{f,r} \\ & \wedge\ (\sigma_{s,r}, \gamma(\delta(*))) \longmapsto^* (\sigma_{f,r}, v_f) \\ & \wedge\ (\sigma_f, v_f) \in \mathcal{V}[\![\mathbf{I}]\!]\end{aligned}$$

We then instantiate these existentially quantified variables with $\{\}$, σ_r and $*$, just as the paper does. The paper does this step near the end of this proof case, while we're doing it near the beginning, which shows the difference in proof flow.

We now have:

$$\begin{aligned}& \{\} \uplus \sigma_r = \sigma_r \\ & \wedge\ (\sigma_{s,r}, \gamma(\delta(*))) \longmapsto^* (\sigma_r, *) \\ & \wedge\ (\{\}, *) \in \mathcal{V}[\![\mathbf{I}]\!]\end{aligned}$$

We will prove this conjunction by proving each component separately. In Coq, we replace this goal with *three* goals, one for each component, and we prove these goals one at a time, in turn.

First, we prove $\{\} \uplus \sigma_r = \sigma_r$. Remember that this is actually defined in terms of our `merge` relation. Because we defined the `merge` relation ourselves, we also need to define any of its properties that we use, including that $\{\}$ is an identity element. The proof is through induction on σ_r, which is simple, but not as simple as one might expect of such a basic property. This shows the discrepancy between what is simple on paper and what is simple in Coq.

Next, we prove $(\sigma_{s,r}, \gamma(\delta(*))) \longmapsto^* (\sigma_r, *)$. Coq is able to automatically simplify $\gamma(\delta(*))$ to $*$, turning our goal into $(\sigma_{s,r}, *) \longmapsto^* (\sigma_r, *)$. This is actually $0 \longmapsto$ steps, so we need to prove that $\sigma_{s,r} = \sigma_r$. To do this, we use the hypothesis $(\sigma_s, \gamma) \in \mathcal{S}[\![\bullet]\!]\delta$ that we put in our proof context earlier; because of the definition of $\mathcal{S}$, we have that $\sigma_s = \{\}$. Thus, we have that $\{\} \uplus \sigma_r = \sigma_{s,r}$ and we need to show that $\sigma_{s,r} = \sigma_r$. Note that this identity property is actually the *converse* of the identity property proved in the previous case; this case proved equality assuming `merge`, while the previous case proved `merge` assuming equality. Thus, these two different properties require different proofs.

Finally, we prove $\wedge(\{\}, *) \in \mathcal{V}[\![\mathbf{I}]\!]$. This is simply by the definition of $\mathcal{V}[\![I]\!]$, which is also simple in Coq.

The Coq proof required giving more attention to the stores and their disjoint unions, due to their implementation as our `merge` relation. On the other hand, applying the δ and γ substitutions was automatic in Coq and not mentioned at all in the proof code, while the paper proof explicitly mentions the applications of these substitutions. This difference is because the paper proof is for humans, who understand things differently from mechanization tools like Coq.

2) Lemmas: Lemmas are smaller properties, accompanied with proofs of those properties, that may not be of interest on their own, but are used as part of larger proofs. Lemmas originated in paper proofs and are also supported in Coq. Paper proofs are often deconstructed into lemmas to make the proof flow easier to understand and to highlight the big picture. This is similar to how a programmer may split a large function into several smaller functions that each perform one specific part of the larger function's procedure. Another major purpose of functions in programming is for abstraction, which allows for code reuse. Lemmas can also be used for abstracting properties, and this is heavily utilized in mechanized proofs. Thus, lemmas in mechanized proofs are used analogously to functions in computer programs, which means that full mechanized proofs will include a lot of lemmas that build up to the main proof. The use of lemmas for abstraction is less common in paper proofs, primarily because many uses of lemmas in mechanized proofs are for properties trivial enough to understand that the paper doesn't need to explicitly prove them; they are needed in the mechanized proofs because everything down to the axioms of Coq needs to be explicitly proven for Coq to acknowledge the proof's correctness.

In our mechanization of L^3, lemmas are primarily used to capture properties of relations in isolation of the role they played in the type soundness proof. For example, the two properties of `merge` used in the unit case of the type soundness proof, which was explained above, were expressed and proven as lemmas, which were then referenced in the type soundness proof. We also needed to prove some properties related to the "locally closed" property of terms (which comes from locally closed representation of variables), including how that property is used in the rules for both the operational semantics and the static semantics; these were all proven in lemmas. Since the only property we were inherently interested in mechanizing was type soundness, all other properties we proved in our codebase can be considered lemmas.

IV. Related Work

A. Coq and Mechanized Metatheory

The use of software tools like Coq to mechanize programming languages has developed beyond what can be covered in only a few research papers.

There are several textbooks centered around Coq. The textbook nicknamed *Coq'Art* is referred to as the first textbook on Coq, first published in 2004 [10]. It describes Coq in detail, including its logical rules and available tactics. A more gentle introduction to Coq is the *Software Foundations* series of online textbooks, which is presented as interactive Coq scripts, thus opting for a more hands-on approach to introducing mechanization. The first volume teaches the basics of Coq, while the second volume integrates introductory material on programming language metatheory into its presentation of how said metatheory is mechanizes [11], [12]. The third volume introduces a different use case of Coq: verifying the correctness of algorithms [13]. A textbook that focuses on this use of Coq to prove program correctness is *Certified Programming with Dependent Types* [14]. The textbook emphasizes features of Coq that are not as prominent in programming language metatheory but play a big role in certified programming,

including dependently-typed programming and proof automation.

Mechanizing programming language metatheory is now common practice among newer research papers. A paper published in 2005 aimed to make mainstream the practice of including mechanized proofs alongside published research papers [15]. In addition to this call-to-action, the paper presented a set of mechanization exercises, called the POPLMARK challenge, to be used as a benchmark to evaluate the state of mechanization tools at the time. Mechanization is at the stage where there is an increased focus on enhancing the tooling, in terms of both capabilities and user-friendliness. Ott is an example of a "meta-tool" that makes it easier to formally specify a language's definition for mechanization tools like Coq to use [16]. More specifically, the user specifies the syntax and semantics in a form close to how it looks on paper, and Ott generates the corresponding Coq `Inductive` definitions. Ott also works with other mechanization tools, and can output LaTeXmarkup for the language definitions that can be inserted into a research paper.

B. Other Mechanization Tools

Agda is a proof assistant which, like Coq, is based on the idea of propositions-as-types and proofs-as-terms in a dependently-typed language [2]. Unlike Coq, proofs are not created through the use of tactics, but are written directly as terms in the programming language. To aid in this process, Agda provides a user interface that allows the user to specify "holes" in terms and query for the type of the term that needs to written there. This feature, like tactics, allows the user to focus on individual pieces of proofs while writing them.

Twelf is another tool that uses dependent types to represent propositions [3]. Twelf makes use of a totality checker to prove theorems, rather type-checking a proof term built-up by the user as in Coq or Agda. Twelf also allows the user to use higher-order abstract syntax when mechanizing a programming language. That language can thus use Twelf's variable bindings for its own notion of variable bindings.

C. Languages Similar to L^3

Mezzo is a programming language created after L^3 that shares its use of a substructural type system with capabilities (called "permissions" in Mezzo) [17]. A primary motivation behind Mezzo is concurrency, whereas L^3 was created to support strong updates and does not include any concurrency features. Mezzo has a complex type system that includes kinds, dependent types, and subtyping, while L^3 has a simple type system that focuses on linear types and types involving locations. A Coq implementation of Mezzo's type soundness proof is included with the paper. It uses Coq's automation features so that extending Mezzo requires fewer changes to the proof code.

Rust is a programming language that supports a more intricate ownership system than L^3 , including notions like "borrowing" and "lifetimes", while L^3 sticks closer to traditional linear logic [18]. While Rust was created for software development, there have been several efforts to formally study the language. One recent formalization presents a semantic proof of type soundness for its core version of Rust, the same proof strategy used by L^3 [19]. The paper utilizes Iris, a separation logic, to formalize ownership. The type soundness proof is mechanized in Coq and uses a separate library that encodes Iris in Coq, which can be used in other work related to separation logic.

V. CONCLUSION

We completed the mechanization of the syntax, operational semantics, static semantics, and semantic interpretations of L^3. This involved figuring out how to translate various constructs from the paper proof to the mechanized proof in a way that would make the proof work in Coq. This was done by trying to prove certain properties of these constructs that we know to be true; if proving these properties proved difficult, it indicated that the representation we were using was not ideal for proving things in Coq.

We started the proof of type soundness and completed the unit case of the proof. We did not complete the other cases of the proof, which would be similar in structure to the unit case, but would also involve proving more lemmas about `merge` and potentially other relations. We feel confident that, using the representations we have settled on for the paper proof's constructs, the remaining proof cases should be relatively straightforward to prove. The main challenge for this project was determining which representations were suitable for a mechanized proof in Coq.

REFERENCES

[1] The coq proof assistant. [Online]. Available: https://coq.inria.fr

[2] The agda wiki. [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php

[3] The twelf project. [Online]. Available: http://twelf.org/wiki/Main_Page

[4] N. Amin, T. Rompf, and M. Odersky, "Foundations of path-dependent types," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 233–249. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660216

[5] A. Ahmed, M. Fluet, and G. Morrisett, "L3: A linear language with locations," *Fundam. Inf.*, vol. 77, no. 4, pp. 397–449, Dec. 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1365997.1366003

[6] A. Charguéraud, "The locally nameless representation," *Journal of Automated Reasoning*, vol. 49, no. 3, pp. 363–408, Oct 2012. [Online]. Available: https://doi.org/10.1007/s10817-011-9225-2

[7] Charguéraud, Arthur, "Tlc." [Online]. Available: http://www.chargueraud.org/softs/tlc/

[8] G. Morrisett, A. Ahmed, and M. Fluet, "L3: A linear language with locations," in *Typed Lambda Calculi and Applications*, P. Urzyczyn, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 293–307.

[9] A. Ahmed, M. Fluet, and G. Morrisett, "L3: A linear language with locations (technical report)," July 2004. [Online]. Available: http://ftp.deas.harvard.edu/techreports/tr-24-04.pdf

[10] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*. Springer Berlin Heidelberg, 2004. [Online]. Available: https://doi.org/10.1007/978-3-662-07964-5

[11] B. C. Pierce *et al.*, *Logical Foundations*. [Online]. Available: https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html

[12] ——, *Programming Language Foundations*. [Online]. Available: https://softwarefoundations.cis.upenn.edu/current/plf-current/index.html

[13] A. W. Appel, *Verified Functional Algorithms*. [Online]. Available: https://softwarefoundations.cis.upenn.edu/current/vfa-current/index.html

[14] A. Chlipala, *Certified Programming with Dependent Types*. Cambridge, MA: The MIT Press, 2013. [Online]. Available: http://adam.chlipala.net/cpdt/

[15] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic, "Mechanized metatheory for the masses: The poplmark challenge," in *Theorem Proving in Higher Order Logics*, J. Hurd and T. Melham, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 50–65.

[16] Ott. [Online]. Available: http://www.cl.cam.ac.uk/ pes20/ott/

[17] T. Balabonski, F. Pottier, and J. Protzenko, "Type soundness and race freedom for mezzo," in *Functional and Logic Programming*, M. Codish and E. Sumii, Eds. Cham: Springer International Publishing, 2014, pp. 253–269.

[18] The rust programming language. [Online]. Available: https://www.rust-lang.org

[19] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 66:1–66:34, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3158154