# Refactoring and Strengthening QWIRE

Amelia Palmer Dusenbury

## I. Introduction

QWIRE is a Coq-embedded language for defining and reasoning about quantum circuits [**?**]. QWIRE demonstrates that standard techniques from programming language semantics and mechanized verification can be applied to quantum programming languages at a nontrivial scale. Rather than treating quantum circuits as informal diagrams or as code with only an intended interpretation, QWIRE embeds circuits directly in Coq, providing a precise syntax, typing discipline, and denotational semantics that are all mechanically checked. This embedding enforces a level of explicitness often avoided in informal presentations of quantum algorithms, but it is essential for principled reasoning about correctness. In practice, informal quantum reasoning is fragile: small specification errors can invalidate entire constructions. QWIRE shows that these issues can be addressed using established semantic tools rather than ad hoc arguments.

A central technical decision in QWIRE is to enforce linearity through the type system rather than by convention. Qubits are treated as linear resources, and their usage is tracked using explicit context splitting based on partial commutative monoids. This design prevents duplication or unintended disposal of wires by construction. As a result, many well-formedness conditions that would otherwise appear as side conditions or proof obligations cannot be expressed at all. From the perspective of mechanization, this substantially reduces the burden of later correctness proofs, since large classes of ill-formed programs are excluded syntactically.

The framework is organized around several representations of circuits, each serving a distinct purpose. At the surface level, circuits are constructed using a higher-order abstract syntax that supports convenient composition and typing. For compilation and structural reasoning, this representation is translated into a first-order form based on de Bruijn indices. At the lowest level, circuits are interpreted denotationally as matrices and superoperators. Although this layering introduces additional complexity, it enables proofs—carried out entirely within Coq—that high-level circuit descriptions correspond to concrete mathematical transformations. Standard examples such as teleportation and Deutsch's algorithm are verified against this denotational semantics rather than justified by informal circuit diagrams.

Developing the framework in full detail also exposes the limits of current semantic techniques. The presence of admitted or aborted lemmas reflects genuine difficulties rather than incomplete implementation. Issues related to higher-order abstract syntax, compositionality, ancilla management, reversibility, and the interaction between safe and unsafe interpretations arise naturally in the formalization. Making these difficulties explicit clarifies which aspects of quantum programming language semantics are well understood and which remain open research problems.

Although QWIRE is primarily a semantic and verification framework, it is not entirely disconnected from execution. The development includes a backend that translates verified circuits into QASM. While this translation is not part of the trusted core, it demonstrates that circuits verified in Coq can be exported to formats used by existing quantum toolchains, showing that the framework supports more than purely abstract reasoning.

QWIRE also influenced subsequent work on quantum programming languages, particularly EWire. It provided an early demonstration that a first-order, linear quantum circuit language could be embedded in a higher-order classical host language while preserving a clear separation between classical control and quantum data. Many design choices in EWire, including its enriched categorical semantics, build directly on ideas first explored in QWIRE, such as treating circuits as values and managing linear contexts explicitly.

Overall, QWIRE is best understood not as a complete semantic theory, but as a concrete demonstration that end-to-end, mechanized reasoning about quantum programs is feasible. By formalizing syntax, typing, compilation, and denotational semantics within a single proof assistant, the project shows that quantum programming need not rely solely on informal arguments or diagrammatic intuition.

Building on this foundation, this project focuses on strengthening and refactoring QWIRE's foundational abstractions, particularly its monadic infrastructure and repository organization. While QWIRE provides a powerful semantic framework, gaps in its law-level infrastructure and build structure complicate extension, reuse, and proof development. My contribution addresses these issues by completing missing algebraic laws and reorganizing the codebase to better reflect its conceptual structure, thereby improving both the robustness and extensibility of the framework.

### A. Contributions

The changes to `Monad.v` are significant because they complete and operationalize the separation between interfaces and laws. In the original version, `Functor`, `Applicative`, and `Monad` operations were defined uniformly, but corresponding law instances were missing or incomplete for several standard monads, most notably lists. As a result, the apparent abstraction suggested by the interfaces could not be relied upon in practice: proofs or tactics parameterized by the monad laws could be instantiated only for those monads with completed law proofs. By completing the law layer, the revised development restores the intended interpretation of these interfaces as algebraic structures rather than mere collections of operations.

A second contribution is that law-based rewriting and automation become predictable and uniform. Tactics such as `simplify_monad` and lemmas like `bind_eq` are designed to reason via the monad laws rather than by unfolding concrete definitions. When law instances are missing, these tools fail silently for certain monads, forcing users to resort to ad hoc, representation-specific rewrites. By proving `Functor_Correct`, `Applicative_Correct`, and `Monad_Correct` for the standard instances, the revised library ensures that the same proof scripts and tactics behave consistently whenever the appropriate law classes are available.

These changes also reduce proof brittleness by decoupling higher-level reasoning from implementation details. In the absence of a complete law layer, proofs tend to depend on particular choices in how operations such as `bind`, `<*>`, or `pure` are defined—for example, the order of list concatenation or the mechanics of state threading. Such details are artifacts of encoding rather than semantic properties. Strengthening the law layer enables proofs to proceed algebraically, relying on associativity and unit laws instead of unfolding definitions, thereby making them more robust to refactoring and easier to maintain.

Finally, these improvements are essential for larger semantic developments in which monads are used pervasively to structure effects, control flow, and state. In such settings, an incomplete law layer leads quickly to cluttered proofs and duplicated reasoning across instances. By fully establishing the laws while deliberately avoiding additional coherence constraints between `Applicative` and `Monad`, the revised `Monad.v` provides a flexible yet sound foundation. Higher-level semantic arguments can rely on clearly stated assumptions, while reusable, law-driven lemmas absorb algebraic complexity, improving modularity and scalability.

The refactoring of QWIRE contributes less by introducing new functionality than by rendering the system's structure legible. In the original layout, conceptual distinctions were flattened: foundational infrastructure, the core language, its semantics, examples, and auxiliary tooling coexisted in a largely undifferentiated space. Understanding the architecture therefore required indirect effort—following `Require Import` chains, reverse-engineering build rules, and inferring which components were foundational versus downstream consumers.

The refactored directory structure instead mirrors the intellectual organization of QWIRE. The separation into `Base`, `Syntax`, `Typing`, `Semantics`, and `CompilationQASM` externalizes what had previously been implicit: a layered system in which later components are intended to rest on earlier ones. The repository becomes self-describing in a useful sense, making it immediately apparent where the calculus is defined, where its meaning is formalized, and which components primarily exercise or export that core. These architectural boundaries also reduce the risk of accidental entanglement—for example, of example code or the QASM backend depending on semantic internals or proof-heavy infrastructure. In a proof-intensive Coq project like QWIRE, where organizational choices have outsized effects on proof effort, aligning conceptual structure with physical organization improves robustness, extensibility, and long-term maintainability.

## II. Background

The semantics of QWIRE are formalized within Coq, enabling machine-checked reasoning about quantum circuits [**?**], [**?**]. Quantum circuits should not be treated merely as suggestive diagrams, nor as fragments of executable code whose meaning is supplied by informal explanation. QWIRE was designed to address this gap. Instead, they should be regarded as programs—objects whose structure, typing discipline, and semantics are precise enough to support machine-checked reasoning. The motivating question is deceptively simple: can circuit-based quantum programs, especially those intertwined with classical control, be given the same semantic rigor expected of classical code? QWIRE answers in the affirmative, but only under a carefully constrained design.

A recurring theme in the framework is separation. Classical computation is prevented from interfering with quantum data; syntax is kept distinct from interpretation; and ergonomic convenience is not allowed to obscure physical or mathematical constraints. These distinctions are enforced by embedding a small quantum circuit language directly into Coq. Circuits are not inert data structures but typed terms equipped with a denotational semantics. Reasoning about circuits therefore occurs inside the language itself rather than at the meta-level.

The circuit language is intentionally austere. It is first-order and linear, with both qubits and classical bits treated as resources that must be used exactly once. This design reflects physical constraints rather than stylistic preference: quantum data cannot be duplicated or discarded without consequence. QWIRE enforces these constraints through its type system rather than by informal discipline. Typing contexts explicitly track wire availability, and circuit composition is defined only when contexts can be split and recombined coherently. Partial commutative monoids provide the underlying algebraic structure for this bookkeeping. As a result, ill-formed or physically meaningless circuits are excluded syntactically, before semantics are considered.

At the same time, the framework avoids collapsing into unnecessary low-level detail. Circuit construction uses a higher-order abstract syntax representation in which Coq functions serve as continuations. This approach eliminates explicit substitution and name management, allowing circuits to be assembled compositionally while keeping typing constraints largely implicit. Circuits can also be packaged as first-class values—boxes—that classical code may generate, parameterize, and compose. This mirrors the structure of many quantum algorithms, where classical computation orchestrates the construction and execution of quantum subroutines rather than merely invoking fixed circuits.

Higher-order abstract syntax, however, is poorly suited to interpretation and compilation. For these purposes, QWIRE introduces a second representation based on first-order syntax with de Bruijn indices. In this core language, wire usage and control flow are fully explicit, and higher-order structure has been eliminated. The translation from HOAS circuits to

de Bruijn circuits marks the boundary between a user-facing surface language and a structurally simpler core that supports recursion, semantic interpretation, and backend development. This separation is familiar from classical language design, but here it is realized within a single formal system.

Semantic interpretation forms the final layer. Circuits are interpreted as superoperators on density matrices, with unitary gates represented as matrices and more general operations as completely positive maps. Typing contexts determine the relevant dimensions, ensuring that well-typed circuits give rise to well-defined semantics without additional side conditions. QWIRE further distinguishes between "safe" interpretations, which enforce physical validity of assertions, and "unsafe" interpretations that model postselection-like behavior. Together, the core translation and denotational semantics assign a precise mathematical meaning to every well-typed circuit.

Because the entire development resides within Coq, correctness proofs for quantum algorithms can be carried out mechanically. Canonical examples such as teleportation, Deutsch's algorithm, and small arithmetic circuits are proved correct with respect to the denotational semantics rather than justified by informal reasoning. At the same time, the framework is explicit about its limitations. Difficulties involving HOAS compositionality, ancilla management, reversibility, and semantic equivalence arise as concrete proof obligations, some of which remain unresolved. These issues are not hidden but exposed, marking genuine boundaries of current understanding.

In this light, QWIRE is best viewed as a semantic experiment rather than a standalone circuit language. It demonstrates how quantum circuits can be typed, composed, compiled, and interpreted within a single mechanized setting, and how linear quantum data can coexist with classical control without collapsing conceptual distinctions. More broadly, it argues—by construction rather than rhetoric—that quantum programming belongs squarely within programming language theory and formal verification.

### A. Architecture

The system is layered around a typed circuit domain-specific language (DSL) implemented in Coq, with two syntactic representations (higher-order abstract syntax and de Bruijn syntax), a denotational semantics into matrices and superoperators, and a collection of application files that either prove correctness of algorithms or implement compilation and export tooling.

Crucially, the true architecture of the system is determined by `Require Import` dependencies and by the curated build surfaces defined in the `Makefile` (notably the `all` and `qasm` targets). The `_CoqProject` file serves only as a load-path and file list and does not reflect architectural layering. Subsequent work has extended QWIRE to reversible circuits and richer reasoning principles [**?**].

*1) Foundational Infrastructure: Monad and Monoid:* At the base of the stack are `Monad.v` and `Monoid.v`, which provide reusable, non-quantum infrastructure.

*2) Monad.v:* `Monad.v` implements a functional programming effects library in Coq. It formalizes the standard abstraction hierarchy of *Functor*, *Applicative*, and *Monad* using type classes, and separates interfaces from their algebraic laws via corresponding correctness classes. This separation allows effectful constructions to be defined independently from proofs of their laws.

The file defines familiar combinators such as `wbind`, `liftM`, `join`, and `foldM`, together with notation for composition, applicative application, and monadic bind, including a lightweight do-notation. A collection of Ltac tactics (`simplify_monad`, `simpl_m`) automates rewriting using monad laws.

Provides the interfaces for Functor, Applicative, and Monad (and some useful notation/tactics), but leaves key parts of the "laws + automation" story incomplete or uneven across instances. In particular, some effects—most notably lists—have operational definitions but no usable Functor_Correct/Applicative_Correct/Monad_Correct instances because proofs are aborted, which means downstream developments cannot rely on a uniform algebraic theory of effects: generic tactics like simplify_monad only work for the few monads that happen to have completed law proofs (option/state), while list-based code must fall back to ad hoc lemmas and manual rewriting. This undermines the the typeclass architecture (separating interface from properties) because the "properties" layer is patchy, so proofs can't be written modularly against the abstract laws. As a result, reasoning about monadic expressions becomes fragile and non-compositional: the library looks reusable on paper, but in practice its proof power depends on which instances are fully law-certified, forcing later files either to avoid the abstraction or to re-prove missing algebraic facts locally.

*3) Monoid.v:* `Monoid.v` develops a framework for reasoning about commutative and partial commutative monoids. It defines a partial commutative monoid (PCM) interface with unit ($\top$), absorbing element ($\bot$), and a binary operation, together with the expected algebraic laws. A parallel option-valued formulation (PPCM) models genuinely partial combination, which is systematically lifted into a total PCM over `option`.

The file then builds a substantial normalization and automation infrastructure for commutative monoid reasoning. Expressions are reified into a syntactic language, flattened into lists, and further normalized using index lists to factor out commutativity via permutation reasoning. A powerful Ltac tactic, `monoid`, orchestrates these steps and can solve most commutative-monoid equalities automatically, even in the presence of existentials.

Although this development is algebraic rather than quantum, it is architecturally essential: typing contexts and resource splitting in QWIRE are structured as a partial commutative monoid, allowing generic monoid automation to discharge the pervasive reassociation and commutation obligations induced by linear resource usage.

*4) Contexts and Linear Resources:* `Contexts.v` specializes the monoid infrastructure to model typing contexts for quantum and classical wires. Wire types are defined by `WType` (qubit, bit, unit, and tensor), together with size and interpretation functions. Contexts are lists of optional wire

types, wrapped in an `OCtx` validity layer to account for merge failure.

The central operation is context merge, which combines two contexts pointwise and fails on collisions. This operation is shown to form a partial commutative monoid instance, reusing the PCM interface from `Monoid.v`. As a result, the generic monoid automation becomes directly applicable to context equalities.

On top of this algebraic structure, the file defines singleton contexts, validity witnesses, size lemmas, and both computational and inductive characterizations of merging. These support inversion principles and constructive reasoning about context splitting, which are essential for typing rules in the circuit language.

The file also introduces typed patterns indexed by wire types and defines a typing judgment relating patterns to contexts. Quantitative invariants connect pattern structure, wire types, and context sizes. Finally, a gate vocabulary is defined, together with automation tactics (`PCMize`, `validate`, `solve_merge`) that integrate context reasoning with monoid normalization.

*5) HOAS Circuits:* `HOASCircuits.v` defines the core circuit language using higher-order abstract syntax (HOAS). Circuits are parameterized by their output wire type and consist of three constructors: `output`, `gate`, and `lift`. Sequencing is expressed using Coq functions as continuations, so fresh wires and substitution are handled by the meta-language rather than explicitly.

Reusable circuit components are packaged as `Box` values, and circuit composition is defined by a structurally recursive function `compose`. The typing judgment ($\Gamma \vdash c : \mathrm{Circ}$) enforces linear resource usage via explicit context splitting and merging, relying heavily on the machinery from `Contexts.v`. A central theorem shows that composition preserves typing under appropriate merge witnesses.

This HOAS representation forms the user-facing core calculus: it is ergonomic to program in and well-suited to typing proofs, but it relies on Coq functions to represent continuations, which obscures extensionality and parametricity properties needed for global semantic theorems. This representation prioritizes usability and typing guarantees over semantic transparency.

*6) De Bruijn Circuits and Translation:* To recover semantic transparency and support compilation, QWIRE introduces a first-order intermediate representation. `DBCircuits.v` introduces an explicit, first-order circuit representation intended as a compilation-friendly intermediate form. Continuations are no longer functions; instead, circuits are linearized and context evolution is described explicitly by state-transforming functions such as `process_gate` and `remove_pat`.

The file develops extensive infrastructure for fresh-wire allocation, context updates, substitution, and context flattening. Using this machinery, it defines a translation from HOAS circuits to de Bruijn circuits. While the translation itself is implemented, the key typing-preservation theorem is left unfinished (terminated with `Abort`). The de Bruijn language is the intended compilation and semantic intermediate representation, but its typing correspondence to HOAS is not yet fully verified.

*7) Denotational Semantics:* `Denotation.v` defines the semantic backend, interpreting circuits as superoperators on density matrices. A generic `Denote` type class provides uniform notation across syntactic categories. Wire types and contexts are denoted by natural numbers (their sizes), bridging the syntactic resource discipline with matrix dimensions.

Unitary gates are interpreted as matrices, with proofs of well-formedness and unitarity. General gates are interpreted as superoperators, parameterized by a *safe* flag that distinguishes physically valid assertion semantics from unsafe, postselection-like behavior. Substantial infrastructure is devoted to wiring operators (padding, swapping, and indexed application) that mirror the syntactic bookkeeping of contexts and patterns.

Circuit denotation proceeds by first translating HOAS circuits to de Bruijn circuits and then interpreting the latter structurally. Semantic equivalence is defined and registered as a parametric relation, enabling rewriting in later proofs.

*8) Ergonomics and Libraries:* `TypeChecking.v` and `HOASLib.v` provide a surface language and automation layer on top of the core calculus. They introduce notations, derived combinators, and tactics that allow circuits to be written in a DSL-like style while automatically discharging typing and context obligations.

`SemanticLib.v` and `Equations.v` supply a library of semantic specifications and rewrite rules, connecting common circuits to their intended mathematical behavior. These files are consumers of the denotation rather than definers of it.

*9) Higher-Level Semantic Results:* Several files build substantial semantic theories on top of the denotation. `Composition.v` aims to show that denotation respects circuit composition, but its central lemma relies on an admitted parametricity property of HOAS continuations. `Ancilla.v` studies the relationship between safe and unsafe semantics and proves coincidence results for ancilla-free circuits, while carefully aborting more foundational equivalence lemmas that would require stronger semantic invariants.

`Symmetric.v` is a capstone development defining a syntactic class of source-symmetric circuits and proving properties such as ancilla safety and reversibility. Its high-level results depend on several admitted semantic lemmas, reflecting open proof obligations in the semantic algebra.

*10) Examples, Proofs, and Applications:* The remaining files illustrate and apply the framework. `HOASExamples.v` presents a library of example circuits and negative tests for the type system. `HOASProofs.v`, `Deutsch.v`, and `Arithmetic.v` prove correctness of nontrivial algorithms, including teleportation, Deutsch's algorithm, and ripple-carry adders. These developments demonstrate that the core architecture is expressive and that the denotation supports substantial reasoning, even if some general lemmas are still missing.

*11) QASM Backend:* The QASM toolchain (`QASM.v`, `QASMPrinter.v`, and `QASMExamples.v`) forms an optional compilation backend. It translates de Bruijn circuits into a QASM-like abstract syntax and then pretty-prints them as OpenQASM-style text. This backend is intentionally excluded

from the default build and represents an extraction-oriented slice of the project rather than part of the verified semantic core.

*12) Build Structure:* Finally, the `Makefile` encodes the effective architectural entry points. The `all` target builds the core semantic and proof artifacts, while the `qasm` target builds the optional export pipeline. This curated build surface more accurately reflects the project architecture than the flat file list in `_CoqProject`.

## III. Contribution

### A. Strengthened `Monad.v`

The proposition `test` previously relied on implicit typeclass arguments and the tactic `simplify_monad`. Although the goal reduced to associativity of `bind`, the proof was aborted because small changes to typeclass inference caused `simplify_monad` to stop closing the goal. I replaced this with an explicit statement that takes `Functor`, `Applicative`, `Monad`, and `Monad_Correct` as parameters. In the proof, I manually expand the `do`-notation using `change` to rewrite both sides into nested `bind` expressions and then close the goal directly with `bind_associativity`. This avoids proof automation entirely and makes the dependency on associativity explicit.

The more substantial work is completing the previously aborted correctness proofs for the list applicative and monad instances. For the applicative, the original attempt stalled at the composition law, so I introduced a sequence of helper lemmas about `list_liftA`. These include structural lemmas such as `map_map` and `concat_map`, as well as specialized results like `list_liftA_pure_l`, `list_liftA_app_xs`, and a singleton characterization `list_liftA_singleton`. The key step is the lemma `list_applicative_composition_singleton`, which proves the composition law for a singleton argument list. The full composition law `list_applicative_composition` then follows by induction on the argument list, splitting both sides using `list_liftA_cons_xs` and applying the singleton lemma to the head and the induction hypothesis to the tail.

Using these results, I completed `listA_correct` by discharging each applicative law directly. In particular, the composition law now reduces immediately to `list_applicative_composition`, while the identity law is proved by rewriting with `list_liftA_pure_l` and then applying `map_id`, making the underlying list-theoretic dependency explicit. The homomorphism law follows from unfolding `list_liftA` and simple list equalities, and interchange is handled by a dedicated lemma.

For the monad laws, I added auxiliary lemmas about `list_bind`. The right unit law follows from `list_bind_ret`, while associativity is proved by induction on the input list in `list_bind_assoc`, using `list_bind_app` to rewrite the recursive case. With these lemmas in place, the instance `listM_correct` can be completed without further automation.

*1) Impact on `Monoid.v`:* The changes to `Monad.v` have a direct impact on `Monoid.v`, primarily by strengthening the foundational abstractions on which later algebraic developments depend. By replacing aborted proofs and automation-heavy arguments with explicit equational reasoning, the revised monad and applicative correctness instances provide more reliable and transparent laws for downstream files to use. In particular, completing the correctness proofs for the list applicative and monad ensures that any constructions in `Monoid.v` that rely on lists as canonical examples now rest on fully verified algebraic structure rather than assumed properties.

More concretely, `Monoid.v` benefits from the fact that associativity and identity properties are now available as explicit lemmas rather than implicit consequences of automation. Results that depend on monadic sequencing or applicative composition can appeal directly to laws such as `bind_associativity` or the applicative identity and composition laws, without requiring fragile tactic support. This reduces proof brittleness and makes dependencies between algebraic laws more visible, which is especially valuable when reasoning about monoids derived from or interacting with monadic or applicative structure.

Finally, the shift toward explicit, structural proofs in `Monad.v` sets a clearer pattern for algebraic reasoning that carries over into `Monoid.v`. Proofs there can mirror the same style: unfolding definitions, applying well-scoped helper lemmas, and using induction where appropriate, rather than relying on global automation. As a result, the overall development becomes easier to maintain and extend, with `Monoid.v` inheriting both stronger guarantees and a more uniform proof methodology from the revised monad infrastructure.

*2) Impact on `Denotation.v`:* The revisions to `Monad.v` also have important consequences for `Denotation.v`, where monadic structure is used to give meaning to effectful or sequential constructs. By making the monad laws explicit and completing the previously aborted correctness proofs, the denotational definitions in `Denotation.v` can rely on fully verified properties of `bind` and `return_` rather than on assumed behavior or tactic-driven simplifications. In particular, associativity of `bind` is now available in a form that can be applied directly, which is essential for showing that different parenthesizations of sequential composition have the same denotation.

More concretely, proofs in `Denotation.v` that equate denotations of syntactically different but semantically equivalent programs benefit from the strengthened statement of the monad laws. For example, rebracketing of nested `do`-blocks or normalization of sequencing constructs can be justified by direct appeals to `bind_associativity`, rather than by unfolding notation and invoking automation. This makes denotational equivalence proofs clearer, shorter, and more robust to changes in typeclass resolution or rewriting tactics.

Finally, the explicit treatment of the list monad and applicative in `Monad.v` strengthens any denotational models in `Denotation.v` that instantiate semantics using lists, such as nondeterministic or multi-valued interpretations. With `listA_correct` and `listM_correct` fully established,

these denotations are backed by formally proven applicative and monad laws, ensuring that semantic reasoning about sequencing, combination, and identity in `Denotation.v` is sound and independent of proof automation.

### B. Refactoring

Refactoring QWIRE into an explicit directory hierarchy makes the project readable in the same way its architecture is already readable to Coq: by layers and dependency direction. The old flat layout forced users to infer "what depends on what" by chasing `Require Import` chains and reverse-engineering build targets; the new layout surfaces those conceptual boundaries immediately. Putting `Contexts.v`, `Monad.v`, and `Monoid.v` in `src/Base` marks them as infrastructure rather than "just more files," while separating `src/Syntax` (HOAS/DB), `src/Typing`, and `src/Semantics` clarifies the intended pipeline: define circuit syntax, establish typing/resource discipline, and only then interpret programs denotationally. This reduces accidental architectural entanglement (examples or backends reaching into semantic internals), and it makes it far easier for a new contributor to locate the right abstraction boundary when extending the system. The reorganization also improves maintenance and reuse. The QASM toolchain is isolated in `src/CompilationQASM`, communicating that it is an optional export slice rather than part of the trusted semantic core, while `examples/` and `libraries/` cleanly separate "client code" (proofs and demonstrations) from the framework itself. That separation pays off directly in proof engineering: edits to the core calculus or semantics have clearer blast radii, and rebuild/debug cycles become more predictable because the physical layout aligns with the logical build surfaces (core vs. examples vs. QASM). Finally, the structure is friendlier for downstream use: someone who wants QWIRE as a library can depend primarily on `src/` and optionally pull in `libraries/` and `examples/`, instead of vendoring an undifferentiated directory and hoping they guessed the right subset.

Placing `Contexts.v`, `Monad.v`, and `Monoid.v` in `src/Base` clearly marks them as foundational infrastructure rather than incidental utilities. Likewise, separating `src/Syntax` (HOAS and De Bruijn), `src/Typing`, and `src/Semantics` makes the intended pipeline explicit: first define circuit syntax, then impose typing and resource discipline, and only afterwards give a denotational interpretation. This organization reduces accidental architectural entanglement—such as examples or backends reaching into semantic internals—and makes it easier for contributors to identify the correct abstraction boundary when extending the system.

The reorganization also improves maintainability and reuse. Isolating the QASM toolchain in `src/CompilationQASM` communicates that it is an optional export layer rather than part of the trusted semantic core. Similarly, separating `examples/` and `libraries/` cleanly distinguishes *client code* (demonstrations and reusable verified algorithms) from the framework itself. This separation pays off directly in proof engineering: changes to the core calculus or semantics have well-defined blast radii, and rebuild/debug cycles become more predictable because the physical layout mirrors the logical build surfaces (core versus libraries versus examples).

Finally, the new structure is friendlier for downstream use. Projects that want QWIRE as a library can depend primarily on `src/`, optionally pulling in `libraries/` and `examples/` as needed, rather than vendoring an undifferentiated directory and guessing which files are essential. In this sense, the directory hierarchy is not merely cosmetic: it encodes architectural intent, enforces dependency discipline, and turns the filesystem itself into a guide for correct extension and reuse.

*1) _CoqProject:* The change from the original to the updated `_CoqProject` represents a substantial refactoring of the project structure and namespace discipline. The original file used a single recursive mapping, `-R . Top`, which placed every `.v` file in the repository under the logical namespace `Top`. All source files lived in the project root, resulting in a flat physical layout and a flat logical namespace with no separation between core definitions, libraries, or examples.

In the updated version, the single global mapping is replaced by three explicit mappings that mirror the directory structure of the project. The core language is placed under `src` and mapped recursively using `-R src QWIRE`, so that a file such as `src/Base/Monad.v` is now known to Coq as `QWIRE.Base.Monad`. Supporting libraries are mapped with `-R libraries QWIRE.Libraries`, and example files are mapped with `-Q examples QWIRE.Examples`. This introduces a clear logical separation between core code, libraries, and examples. The use of `-Q` for examples also means that only the top-level example files are mapped, rather than all subdirectories recursively.

Most files from the original project have not been removed but systematically relocated into subdirectories under `src`. For example, `Contexts.v`, `Monad.v`, and `Monoid.v` are now located in `src/Base/`. Files related to syntax have been split between `src/Syntax/HOAS/` (for `HOASCircuits.v` and `HOASLib.v`) and `src/Syntax/DB/` (for `DBCircuits.v`). Typing is isolated in `src/Typing/TypeChecking.v`, while semantic components such as `Denotation.v`, `SemanticLib.v`, `Composition.v`, `Oracles.v`, `Ancilla.v`, `Symmetric.v`, and `UnitarySemantics.v` are grouped under `src/Semantics/`. Each of these moves changes the logical module names from their original flat form to structured names under the `QWIRE` namespace.

Some files have been reclassified rather than merely moved. The original `GHZ.v`, which previously appeared alongside core definitions, is now located at `libraries/Algorithms/GHZ.v`, making explicit that it is an algorithmic library component rather than part of the core language. Similarly, `HOASExamples.v` has been moved from the project root to `examples/HOASExamples.v`, clearly separating example code from core and library code.

The updated `_CoqProject` introduces several files that were not present in the original configuration. Notably, the `src/CompilationQASM/` directory adds `QASM.v` and `QASMPrinter.v`, indicating a new compilation or backend layer. The `libraries` directory adds `Equations.v`

and algorithmic examples such as `Arithmetic.v` and `Deutsch.v`. The `examples` directory is also expanded with `HOASProofs.v` and `QASMExamples.v`. These additions broaden the scope of the project beyond its original focus.

As a consequence of these changes, existing `Require Import` statements must be updated. Imports that previously relied on the flat namespace (for example, `Require Import Contexts.` or `Require Import Top.Contexts.`) must now use fully qualified paths such as `From QWIRE.Base Require Import Contexts.` or `From QWIRE.Semantics Require Import Denotation.`. The refined namespace structure improves modularity and clarity, but it requires systematic updates throughout the codebase to reflect the new directory and module layout.

*2) Makefile:* The original Makefile is a relatively small and flat build script designed for a single-directory Coq project. Its default goal is `invoke-coqmakefile`, so invoking `make` mainly delegates control to the generated `CoqMakefile`. Only a small number of targets are handled explicitly, and most compilation steps are expressed as hand-written rules for individual `.vo` files. This design results in a partial and somewhat ad hoc build process, where some files are compiled directly and others are not included in any standard target.

In the original Makefile, all Coq files are compiled under a single logical root using the option `-R . Top`. This places every module in a flat namespace, which simplifies small projects but scales poorly as the codebase grows. In contrast, the updated Makefile reflects a deliberate project reorganization into `src/`, `libraries/`, and `examples/` directories. The Coq load path is correspondingly refined using `-R src QWIRE`, `-R libraries QWIRE.Libraries`, and `-R examples QWIRE.Examples`. As a result, module names become hierarchical and stable, and imports in Coq source files must explicitly reflect this structure.

The original Makefile defines a small number of custom targets such as `all` and `qasm`, each of which builds a fixed list of top-level `.vo` files. Several other compilation rules are present but are marked as "Not built at all," indicating that they are not part of the standard build. The updated Makefile replaces this approach with higher-level semantic targets. The `src` target builds the core QWIRE framework, the `libraries` target builds reusable theory and algorithm modules, and the `examples` target builds demonstration and proof files. The `all` target composes these three phases, yielding a complete and structured build of the entire project.

In the original Makefile, many `.vo` targets invoke `coqc` directly, which partially bypasses the dependency management provided by `coq_makefile`. The updated Makefile instead delegates almost all compilation to the generated `CoqMakefile` by invoking `make -f CoqMakefile` with explicit `.vo` targets. This ensures that Coq's automatically computed dependencies are respected across directories. Direct calls to `coqc` are retained only for a generic pattern rule that allows users to compile individual `.v` files in the repository root.

The original Makefile filters out only a minimal set of known targets before forwarding remaining goals to `CoqMakefile`. The updated version substantially expands this set to include structural targets such as `src`, `libraries`, and `examples`, as well as maintenance and informational targets. This change reduces accidental forwarding and makes the control flow of the build system clearer and more predictable.

The original Makefile does not provide dedicated cleaning targets and implicitly relies on forwarded rules. The updated Makefile introduces a clear hierarchy of cleaning commands: `clean` removes all compiled Coq artifacts, `cleanall` additionally removes generated documentation and build files, and `cleanuser` removes only root-level user artifacts while preserving the main project build. A `help` target is also added, documenting common build commands and significantly improving the usability and maintainability of the build system.

## IV. Impact

Although QWIRE is primarily a research and verification framework rather than an end-user programming language, its primary users are developers and researchers who extend the system, formalize new semantic results, or build verified quantum algorithms on top of its core infrastructure. From this perspective, the contributions of this project address concrete and recurring obstacles to correct and effective use of the system as a software artifact.

Completing the law-level infrastructure in `Monad.v` directly addresses a real need for reliable and modular proof development. In the original codebase, the abstraction hierarchy suggested by the `Functor`, `Applicative`, and `Monad` interfaces could not be uniformly relied upon, because several standard instances—most notably lists—lacked completed correctness proofs. As a result, developers were forced either to avoid generic, law-based reasoning or to re-establish basic algebraic properties locally using ad hoc lemmas. This undermined the intended separation between interfaces and laws and made proofs brittle with respect to refactoring or changes in automation. By completing the missing law instances and replacing automation-dependent arguments with explicit, structural proofs, the revised infrastructure restores the validity of these abstractions and enables predictable, sound reuse of generic lemmas and tactics across the development.

The refactoring contribution similarly addresses a practical need faced by anyone attempting to extend QWIRE correctly. The original flat directory layout obscured the conceptual structure of the system, forcing developers to reconstruct architectural boundaries indirectly by following `Require Import` dependencies or inspecting build rules. By reorganizing the repository into directories that mirror the logical layers of the framework—foundational infrastructure, syntax, typing, semantics, libraries, and examples—the refactoring makes architectural intent explicit and reduces the risk of incorrect extension or unintended coupling between components. This strengthens maintainability by making dependency boundaries visible and enforceable rather than implicit.

Together, these changes do not introduce new language features or semantic results, but they materially improve the robustness and extensibility of the framework. For a proof-intensive system like QWIRE, where small structural

weaknesses can invalidate large proof developments, these improvements directly affect whether the system can be used reliably as a foundation for further work.

A concrete aspect of this improved functional usability is the revised build structure encoded in the updated `Makefile`. In the original project, users were effectively forced to rebuild large portions of the codebase repeatedly, even when working on independent extensions or client files. The updated build separates the core framework (`src` and `libraries`) from downstream or user-authored code. As a result, users can build the core project once and subsequently compile individual files incrementally (`make MyFile.vo`) without triggering a full rebuild.

This change materially affects whether QWIRE can be used as a software component rather than only as a monolithic artifact. It enables a realistic development workflow in which users treat QWIRE as a stable dependency and iteratively develop their own proofs or extensions on top of it. From a software engineering perspective, this improves usability in the strict sense of fitness for use: it supports incremental compilation, enforces clearer dependency boundaries, and makes the system viable for sustained extension and experimentation.

### A. Pull Request Status

No pull request was submitted upstream as part of this project. The changes developed here are foundational and cross-cutting, affecting core proof infrastructure, directory layout, namespace organization, and build assumptions. Such changes are difficult to review or merge incrementally and would require prior coordination with maintainers to align on architectural direction and downstream compatibility.

In addition, the QWIRE repository is not currently under active development, with few files most recently updated months prior to this work. In this context, submitting a large structural pull request without evidence of active maintainer engagement would risk imposing unreviewed architectural decisions or leaving the contribution unintegrated.

The work was therefore developed and evaluated as a self-contained extension and reorganization. However, the changes are designed to preserve existing semantics and interfaces, and they are structured to be upstreamable in principle. In particular, the completed law-level proofs and refactored build structure could form the basis of a future pull request if reviewed in coordination with maintainers or with experienced Coq developers/researcher familiar with the codebase and its proof discipline.

## V. Conclusion

This project focused on strengthening and extending the QWIRE framework by addressing foundational weaknesses in its proof infrastructure and software organization. Specifically, it completed missing law-level proofs in `Monad.v`, restoring the intended separation between abstract interfaces and their algebraic properties, and refactored the repository structure and build system to reflect QWIRE's conceptual architecture. These changes improve the reliability of law-based reasoning, reduce proof brittleness, and enable QWIRE to be used as a stable dependency rather than only as a monolithic artifact. While no new language features or semantic results were introduced, the project materially improves the soundness, extensibility, and engineering viability of the existing system.

Working on QWIRE highlighted both the power and the difficulty of maintaining large, proof-intensive software systems. A central challenge was distinguishing genuine semantic limitations from accidental weaknesses in the supporting infrastructure. Several difficulties encountered during the project—such as aborted proofs, fragile automation, and subtle dependencies introduced by higher-order abstract syntax—were not bugs to be fixed locally, but structural issues requiring careful analysis and explicit design choices. The project reinforced the importance of making assumptions and invariants explicit, especially in a setting where mechanized verification enforces global consistency and small changes can have wide-ranging effects.

The experience also emphasized the role of software engineering practices in formal methods research. Refactoring directory structure, clarifying dependency boundaries, and supporting incremental builds were not merely organizational improvements, but essential steps toward making the system usable in practice. In a proof assistant setting, architectural clarity directly affects proof effort, reviewability, and the feasibility of extension. Treating the build system and repository layout as part of the trusted engineering surface proved as important as completing individual proofs.

With additional time, future work would focus on further upstreaming and validation of these changes, including review by experienced Coq developers and potential coordination with maintainers. On the technical side, several admitted semantic lemmas—particularly those related to higher-order abstract syntax parametricity and compositionality—remain open and would be natural targets for deeper investigation. More broadly, the project suggests that continued progress on QWIRE will depend as much on strengthening its foundational infrastructure and engineering discipline as on extending its semantic expressiveness.

Overall, this project demonstrates that careful engineering work—completing abstractions, stabilizing foundations, and making architecture explicit—can have a substantial impact on the practical usability of a formal verification framework. It reinforces the view that advanced software engineering principles apply as strongly to mechanized semantics and proof assistants as they do to conventional software systems.

## References


[1] J. Paykin, R. Rand, and S. Zdancewic, "QWIRE: A core language for quantum circuits," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 2017, pp. 846–858, doi: 10.1145/3009837.3009894.

[2] R. Rand, J. Paykin, and S. Zdancewic, "QWIRE practice: Formal verification of quantum circuits in Coq," in *Proceedings of the 14th International Conference on Quantum Physics and Logic (QPL 2017)*, Nijmegen, The Netherlands, Jul. 2017, vol. 266, pp. 119–132, *Electronic Proceedings in Theoretical Computer Science*, doi: 10.4204/EPTCS.266.8.

[3] R. Rand, J. Paykin, D.-H. Lee, and S. Zdancewic, "ReQWIRE: Reasoning about reversible quantum circuits," *Electronic Proceedings in Theoretical Computer Science*, vol. 287, pp. 299–312, Jan. 2019, doi: 10.4204/EPTCS.287.17.

[4] M. Rennela and S. Staton, "Classical control, quantum circuits and linear logic in enriched category theory," *Logical Methods in Computer Science*, vol. 16, no. 1, Mar. 2020, doi: 10.23638/LMCS-16(1:30)2020.