# A Layered Certifying Compiler Architecture

Jacco O.G. Krijnen
Utrecht University
Utrecht, Netherlands
j.o.g.krijnen@uu.nl

Wouter Swierstra
Utrecht University
Utrecht, Netherlands
w.s.swierstra@uu.nl

Manuel Chakravarty
Tweag & IOG
Utrecht, Netherlands
chak@applicative.co

Joris Dral
Well-Typed
Utrecht, Netherlands
joris@well-typed.com

Gabriele Keller
Utrecht University
Utrecht, Netherlands
g.k.keller@uu.nl

## Abstract

The formal verification of an optimising compiler for a realistic programming language is no small task. Most verification efforts develop the compiler and its correctness proof hand in hand. Unfortunately, this approach is less suitable for today's constantly evolving community-developed open-source compilers and languages. This paper discusses an alternative approach to high-assurance compilers, where a separate certifier uses *translation validation* to assess and certify the correctness of each individual compiler run. It also demonstrates that an incremental, layered architecture for the certifier improves assurance step-by-step and may be developed largely independently from the constantly changing main compiler code base. This approach to compiler correctness is practical, as witnessed by the development of a certifier for the deployed, in-production compiler for the Plinth smart contract language. Furthermore, this paper demonstrates that the use of functional languages in the compiler and proof assistant has a clear benefit: it becomes straightforward to integrate the certifier as an additional check in the compiler itself, leveraging the the Rocq prover's *program extraction*.

***Keywords:*** Compiler correctness, Translation validation, Certified compilation, Smart contracts

**ACM Reference Format:**
Jacco O.G. Krijnen, Wouter Swierstra, Manuel Chakravarty, Joris Dral, and Gabriele Keller. 2025. A Layered Certifying Compiler Architecture. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '25), October 12–18, 2025, Singapore, Singapore*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3759163.3760427



*FUNARCH '25, Singapore, Singapore*



ACM ISBN 979-8-4007-2146-5/25/10
https://doi.org/10.1145/3759163.3760427

## 1 Introduction

Compiler verification is an old problem, dating back to the dawn of functional programming [McCarthy and Painter 1967]. More recently, interactive proof assistants have been successfully used to establish the correctness of realistic compilers. These efforts target well defined and stable languages, such as C [Leroy et al. 2016] and Standard ML [Kumar et al. 2014]; a substantial part of the compiler is developed in the proof assistant, thereby closely coupling development and verification. Yet the correctness of compilers for languages that are in flux — the de facto standard in today's landscape of community-developed open source compilers — where verification and development can proceed independently remains a less explored problem.

We propose a more flexible approach to compiler correctness based on translation validation [Pnueli et al. 1998]. Rather than porting a compiler to a proof-assistant and proving its implementation correct, we show how to *certify* the correctnes of a run of the compiler post-hoc, generating a (machine checkable) proof that the source program's semantics have been preserved. Translation validation is not a new idea, but *this paper demonstrates that it is a viable alternative to the verification of industrial strength compilers for functional languages under active development.*

Moreover, by using a *layered* architecture, we show how verification need not be an all-or-nothing endeavour. Each pass is specified, validated and verified separately. By having the compiler emit the intermediate code after each pass, we link the compiler's behaviour to its mechanised formalisation. The (1) specification, (2) decidability of that specification, and (3) verification of each pass happens in a theorem prover, independently of compiler development. Each of these steps, layer by layer, improves the trustworthiness of the overall system. Throughout the paper, we will show that a layered certifier, even when only partially implemented, helps developers to improve the correctness of the compiler and provides machine-checkable certificates to end-users who care about the correct compilation of their programs. This is particularly important in application areas that require a high degree of assurance and a verifiable connection between source and target code.

As an example, consider financial applications, such as blockchains. Here, *smart contracts*—that is, compiled code on public blockchains such as Ethereum or Cardano—control significant amounts of financial assets, yet must operate under adversarial conditions. Bugs in smart contract code are a significant problem in practice [Atzei et al. 2017]. To make matters worse, code is typically hard to update securely, once it has been committed to the blockchain. Recent work has also established that compilers for smart contract languages can exacerbate this problem [Park et al. 2020, Section 3]. Bugs encountered in the Vyper compiler, for example, have been shown to compromise smart contract security. Consequently, translation certificates for smart contracts are valuable tools to improve the security of applications on public blockchains. Moreover, given that the blockchain only carries the compiled code, the irrefutable link to the source code establishes the provenance of the on-chain code.

To demonstrate the feasibility of our approach, we have implemented large parts of a certifier for the Plinth smart contract compiler[1] using the Rocq prover. The code of the certifier is publicly available[2]. The Plinth language and compiler are under active development, independent of our verification effort. Previous work in this domain has focused on *specifying* different passes done by the Plinth compiler [Krijnen et al. 2024]. Yet in itself, this check is too limited: to guarantee that the semantics of a program is preserved, we still need to prove that each pass preserves a program's meaning. The current work builds on these results, presenting an overarching methodology, that also includes the proof of correctness of passes and the development of a certifier. More specifically, this paper makes the following contributions:

- We present our layered methodology for constructing a certifying compiler (Section 2). This approach enables the incremental verification of production compilers under active development, such as the Plinth compiler.
- We present two different architectures for retrofitting the verification on the compiler (Section 3): the *certifier* which constructs machine-checkable certificates in the form of Rocq proof scripts (Section 3.1), and the *formal pedantic mode*, which runs as a check directly in the compiler code base (Section 3.2). Crucially, this tooling is already of value before all layers of all passes have been implemented, gradually providing stronger guarantees as more passes are formalised.
- We formalise the semantics of Plinth's intermediate representation (PIR) and build a framework for proving *validator equivalence* of two programs using Rocq (Section 4).
- To evaluate the viability of this approach, we consider the work necessary for implement the layers of a typical pass in the Plinth compiler and collect initial benchmarks of the certified compilation of an auction contract (Section 5). Moreover, we evaluate the formal pedantic mode by enabling it on a test-suite of the compiler.

It is important to note that translation validation has been succesfully applied in industrial-strength compilers for imperative languages [Lopes et al. 2021; Sewell et al. 2013], but functional compilers have received much less attention. We defer a more detailed comparison between our approach and the existing work on translation validation to Section 6. Our results extend and adapt a great deal of existing work on compiler verification. Yet these technical contributions are connected by a single overarching result: the application of these ideas to an industrial strength compiler, yielding a result far greater than the sum of its parts.

## 2 Layered Certification Methodology

Verification of a production compiler is no small task. The well-known CompCert project contains around 100 000 lines of code and was estimated to have cost six person-years of effort [Kästner et al. 2017], its proof being "among the largest ever performed with a proof assistant" [Leroy et al. 2016]. These formally verified compilers have been developed from scratch with verification in mind. The post-hoc verification of an existing compiler is much harder, especially if it is still under active development. Adding new optimisation passes, language features, or performance improvements all require substantial effort to verify. Yet the continuous evolution of languages and compilers is common practice. How can we address compiler correctness of compilers under active development in a less monolithic fashion?

We propose a *layered certification methodology*, based on translation validation [Pnueli et al. 1998]: instead of proving that the compiler always produces correct code, we check the correctness of code generated by individual compiler runs. Our approach is layered in the sense that we gradually and incrementally work towards the complete certification of the entire compiler. Each layer comes with its own verification artifacts; each artifact improves the overall assurance of the compiler as a whole. Moreover, our methodology can be applied to each compiler pass in isolation. In this way, we manage the complexity of the verification effort, yet the fruit of our labour is of immediate value to compiler developers. The development is split into the following four separate layers, which are illustrated in Figure 1 for a single pass.

The **specification layer** consists of a formal specification of the compiler pass, mechanised in a proof assistant. Such a specification takes the form of an inductively defined relation on pairs of abstract syntax trees ($R_i$ in Figure 1). In our case,

[1] https://github.com/IntersectMBO/plutus
[2] https://github.com/jaccokrijnen/plutus-cert

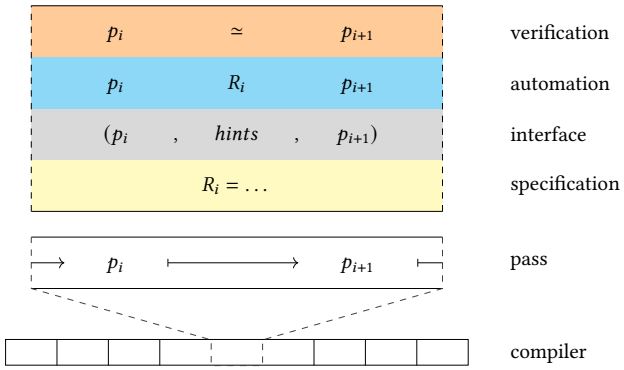



**Figure 1.** The layered architecture for a run of the multi-pass compiler pipeline, highlighting a single pass that transforms program $p_i$ into $p_{i+1}$. The four colored boxes represent the layers implemented in the proof assistant.

we have mechanised them in the Rocq prover (Section 2.1). We will refer to such specifications as *translation relations*.

The **interface layer** bridges the gap between the formal development in Rocq and the compiler (Section 2.2). We modify the compiler to produce a *compilation trace*, that records for each pass its input and output ASTs ($p_i$ and $p_i'$ in Figure 1) together with additional information about the run, such as the pass name and optional *hints* about the transformation that took place. By parsing the compilation trace into Rocq, we can now formulate and prove the theorem stating that a given run of the compiler behaved according to its specification. Given the size of the programs involved however, writing the proofs by hand becomes tedious quite quickly.

The **automation layer** (Section 2.3), defines a *decision procedure* for each translation relation. This decision procedure can establish proof that a particular run of the compiler behaves in accordance with its specification. More precisely, for each pass it decides whether $R_i(p_i, p_{i+1})$ holds.

Finally, the **verification layer** (Section 2.4) consists of a formal proof that each translation relation preserves program semantics (represented using the ≃ operator in Figure 1). The verification layer asserts the correctness of the translation layer. Therefore, any compiler run that is accepted by the automation layer is guaranteed to preserve the semantics of the input program.

In Section 2.5 we elaborate on how each layer increases the assurance of the compiler.

### 2.1 Specification Layer

The *specification layer* defines the intended behaviour of the compiler. For each compiler pass, the specification layer consists of a translation relation: a binary relation on abstract syntax trees, relating the program before the pass (which we may refer to as "pre-term") and the result of the pass

$$\frac{B(x) = t \qquad B \vdash t \rhd t'}{B \vdash x \rhd t'} \text{ INL\_VAR\_1}$$

$$\frac{}{B \vdash x \rhd x} \text{ INL\_VAR\_2}$$

$$\frac{B \vdash s \rhd s' \qquad (x \mapsto s), B \vdash t \rhd t'}{B \vdash \texttt{let}\ x = s\ \texttt{in}\ t \rhd \texttt{let}\ x = s'\ \texttt{in}\ t'} \text{ INL\_LET}$$

$$\frac{B \vdash s \rhd s' \qquad B \vdash t \rhd t'}{B \vdash s\ t \rhd s'\ t'} \text{ INL\_APP}$$

$$\frac{B \vdash t \rhd t'}{B \vdash \lambda x.\ t \rhd \lambda x.\ t'} \text{ INL\_LAM}$$

**Figure 2.** Specification of an inlining pass for the lambda calculus with `let`

```
Inductive inlining :
  list (string * expr) -> expr -> expr -> Prop :=
  | inl_var_1 {B x t t'} :
      lookup x B = Some t ->
      inlining B t t' ->
      inlining B (Var x) t'
  | inl_var_2 {B x} :
      inlining B (Var x) (Var x)
  | ...
```

**Figure 3.** Same specification as a Rocq inductive datatype

("post-term"). To illustrate these specifications, consider a simple lambda calculus with let bindings:

$$t ::= x \mid \lambda x.\ t \mid t\ t \mid \texttt{let}\ x = t\ \texttt{in}\ t$$

*Let inlining* is a typical compiler pass that may replace some (but not necessarily all) let-bound variables with their definition. Such a transformation is non-local, since we need to keep track of the let-bindings in scope. The specification will take the form of a ternary relation $B \vdash s \rhd t$ which states that pre-term $s$ can be transformed into post-term $t$ under the context $B$, which maps variables to their let-bound definitions. The translation relation is presented in Figure 2 as a set of inference rules, and in Figure 3 as a Rocq inductive.

Specifically, Rule INL_VAR_1 states that we may conclude that a variable $x$ is replaced by a term $t'$ when $x$ was let-bound to term $t$ (first premise) and $t$ itself is translated to $t'$ (second premise). This allows repeated inlining passes. Rule INL_VAR_2 states that a variable occurrence may also be left unchanged, e.g., if it is bound by a lambda or if the compiler decides not to inline the corresponding let binding.

The remaining are congruence rules, where the Rule INL_LET extends the environment $B$ with the definition of $x$.

Note that this relation is not a function, but a proper relation. It may relate one pre-term $t$ to many post-terms, depending on which variable occurrences are inlined. It is therefore more general than an implementation in the compiler, which uses a fixed strategy or heuristic to decide which occurrences to inline. This leaves room for compiler developers to choose *any* strategy, meaning any change to the compiler heuristics or 'magic-numbers' [Peyton Jones and Marlow 2002] is independent of this specification.

The specification layer of the compiler consists of inductively defined translation relations for each compiler pass, implemented as an inductive type in Rocq. We generate documentation in the form of TEX inference rules from the Rocq definitions using the InducTEX tool [Krijnen 2024]. This tool uses MetaCoq's metaprogramming capabilities to produce TEX code. Indeed, the rules in Figure 2 were generated from the definition in Figure 3 using InducTEX. The resulting documentation is more readable for developers unfamiliar with some of the peculiarities of Rocq syntax.

### 2.2 Interface Layer

While having a formal specification is certainly useful, the specification layer is unrelated to the actual compiler. The *interface layer* establishes that connection. Firstly, the compiler dumps the ASTs after each pass, labelled with the name of the pass. We call the collection of dumps a *compilation trace*. Next, the proof assistant parses this trace to the corresponding inductive types.

This does require a compiler modification. Yet it is non-invasive and most compilers already include facilities to dump intermediate programs. In the case of our certifier for the Plinth compiler, we only had to adapt existing pretty printing infrastructure to produce terms that can readily be read into Rocq (see also Section 4).

The interface layer is a trusted component: we do not prove its correctness, yet it is indispensable in our design. Hence, the printer and parser should be as straightforward as possible. For example, we chose our AST definitions in Rocq to closely mirror the internal ones of the Plinth Compiler (Section 4). This significantly reduces the trusted computing base and maintains a close connection between the compiler and proof assistant. Moreover, overall correctness only depends on the correct printing and parsing of the source program and the final compiled program, not on intermediate ASTs. While we cannot formally verify the parser and pretty printer, we can check a 'roundtrip' property – by asserting that after pretty printing the Rocq AST and subsequently parsing the resulting program, reconstructs the original AST. This property gives us a high degree of confidence that the parsers and pretty printers involved do not change the syntactic structure of a program in any meaningful way.

### 2.3 Automation Layer

Given a run of the compiler, the interface and specification layers produce a proof obligation: we need to establish that each compiler pass behaves as expected. We could interactively prove such properties using Rocq's tactics, yet this is inadvisable. The programs and proof goals can become quite large and hard to read. Furthermore, proof automation using tactics, such as `auto`, tend to be slow on such large proof goal. To make matters worse, these tactics may fail or require excessively large search depths to produce the desired proof. Instead, we use *proof by reflection* [Bertot and Castéran 2013, Chapter 16] and write a *decision procedure* for each relation in the specification layer. These decision procedures form the *automation layer*.

A decision procedure for the inlining relation decides if two terms are in the relation:

```
Definition dec_inlining :
  list (string * bool) -> expr -> expr -> bool :=
    fix dec_inlining B s t := match s, t with
    | Var x, t => Var x =? t ||
        (match lookup x B with
          | Some u => dec_inlining B u t
          | None  => false
         end)
    | ...
```

This recursive function accepts an environment of let-bound variables and two terms, and should return `true` if they are in the inlining translation relation. The first case of the `match` checks rules INL_VAR_2 and INL_VAR_1.

To make this connection between the translation relation and decision procedure precise, we prove an equivalence:

```
Lemma inline_equiv B e1 e2 :
  dec_inlining B e1 e2 = true <-> inlining B e1 e2.
```

For many passes, the equivalence can be proven by straightforward induction on the first term. This lemma allows us to immediately construct a proof when two concrete terms `t1` and `t2` (dumped by the interface layer) are in the translation relation:

```
Lemma spec_t1_t2 : inlining [] t1 t2.
Proof. now (apply inline_equiv). Qed.
```

Some compiler passes may rely on the results of a global static analysis, or perhaps they perform multiple transformations at once. In such cases, it can help to extend the interface layer, having the compiler emit further information that aids the decision procedure. For example, The inliner in the Plinth compiler also performs simultaneous variable renaming. In such cases we can simplify the decision procedure by emitting these renamings explicitly alongside the abstract syntax trees in the interface layer.

### 2.4 Verification Layer

The first three layers certify that each compiler run behaves according to a syntactic specification. What remains is to establish that this specification itself is correct. In the *verification layer*, we address the classic compiler verification problem of establishing semantic preservation of each individual pass.

There are many ways of proving preservation of semantics, depending on the style of semantics and the programming languages involved. Here we will assume a big-step operational semantics of the form `t ==> v`, where term `t` evaluates to value `v`. Let us also assume a notion of "top-level" programs that are accepted by the compiler. For our simple lambda calculus these will be closed terms of type $\mathbb{Z} \to \mathbb{Z}$. We then consider two top-level programs $p$ and $q$ observationally equivalent when they have the same input-output (and consequently, termination) behaviour:

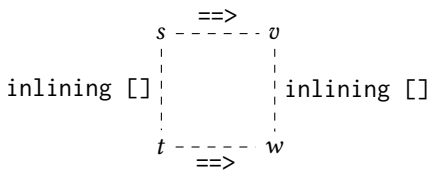
```
Definition eq_obs p q := forall n m,
    Apply p (Const n) ==> Const m <->
    Apply q (Const n) ==> Const m.
```

That is, applying $p$ and $q$ to the same integer constant results in equal output values. Here we have omitted some assumptions about $p$ and $q$ such as their well-typedness. Furthermore both $p$ and $q$ are written in the same language – whereas compilers correctness may also involve target code with *different* semantics.

To complete the verification layer we need to prove a correctness theorem of the following type:

```
forall p q, inlining [] p q -> eq_obs p q.
```

One standard way of proving this is by establishing a simulation, that is, proving that the following diagram commutes for arbitrary closed expressions $s$ and $t$:




When $s$ and $t$ are related through the translation relation for inlining, evaluating either of them results in values that are also related via the translation relation. For the case where $s$ and $t$ are top-level progams, this implies observational equivalence.

Upon completing the correctness proofs of each individual compiler pass, we can establish that a complete run of the compiler has preserved the semantics of its input program.

### 2.5 Gradual assurance

Verification need not be an all-or-nothing endeavour. By developing the verification architecture in layers and pass by pass, it becomes possible to gradually increase the assurance of the compiler. Each layer improves the trustworthiness of the compiler, even if not all layers for each have been fully implemented.

To define the specification layer for a pass, we have to develop a good understanding of the corresponding implementation in the compiler. The informal reasoning and studying of the source code, as well as the task of formulating a translation relation in Rocq's logic, may already uncover bugs or complications in the design of compiler pass.

The interface layer enforces that the formalisation does not happen in a vacuum and works with the *actual* representation of programs, rather than an idealised or simplified version. By testing the round-trip property, we obtain a high degree of assurance that the representation in the proof assistant and compiler are equal.

The automation layer produces a tangible proof that the compiler adheres to its specification. Not only can it be used to check individual runs of the compiler, the resulting proof term can be indepedently validated as it is machine-checkable. Using program extraction, the corresponding decision procedures may be run independently of the proof assistant.

Finally, the verification layer ensures that the translation relation is semantically sound. This guarantees that a particular compiler pass is semantics preserving.

## 3 Certifier

Next, we discuss how the four layers of our methodology can be used to build a *certifier*, working with the compiler, to generate verifiable certificates. The latter can be bundled and distributed together with the compiled software, enabling users to check the certificate before executing the software.

A certificate is a proof script containing a *top-level theorem*. Depending on which compiler passes have been formalised and up to which layer, the certifier assembles all available evidence to support this theorem. As the formalisation of the system is extended, the certifier's claims in the top-level theorem get stronger and stronger.

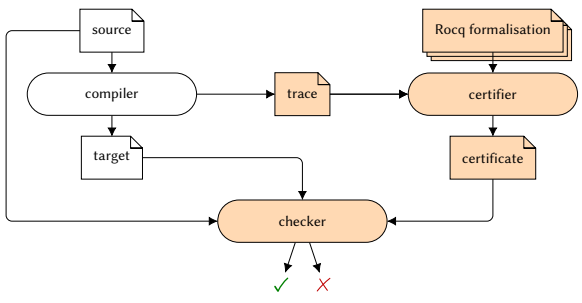



**Figure 4.** Architecture of the certifier (in orange) retrofitted on an existing compiler (white). Square boxes represent files, rounded boxes represent tools.

Certificate generation is irrelevant during the development process, but part of the release process of a the compiled software. However, we do not suggest to disable the certifier completely; instead, we provide the option of a lightweight mode, which we call *formal pedantic mode* (Section 3.2). This mode allows to catch compiler bugs early, but it does not produce a certificate, avoiding the overhead of constructing and checking a formal proof.

### 3.1 The architecture of a certifier

Certifying a compiler run is a three-step process, illustrated in Figure 4:

1. The compiler dumps a compilation trace: a text file containing labelled intermediate ASTs of the program that is being compiled, and optionally hints about the transformation that took place.
2. A simple post-processing tool turns the textual dump into a structured Rocq project. It includes the compilation trace as a defined object, and imports the Rocq formalisation of the compiler passes and corresponding layers as a library. Finally, it includes the top-level theorem with proof about the compilation trace.
3. Finally, the checker tool invokes the Rocq compiler to build the project and check the proof of the top-level theorem. Since that theorem mentions a source and target programs, the checker also confirms that they match the provided source and target programs.

We consider each of the steps in detail and turn our attention to PIR again (which we will more formally discuss in Section 4).

#### 3.1.1 Dumping a compilation trace.

The compiler and certifier need to use a common format for the trace. Here, we outline the Rocq definitions of the AST of PIR and an enumeration of the compiler passes. The abstract syntax follows the concrete syntax in Figure 6 (we omit almost all productions of the grammar here, with the exception of terms).

```
Inductive term :=
  | Var     : name -> term
  | LamAbs  : binderName -> ty -> term -> term
  (* ... *).
Inductive pass :=
  Rename | DeadCode | (* ... *) .
Definition comp_trace := term * list (pass * term).
```

A `comp_trace` contains the source parsed by the compiler, together with a list of intermediate ASTs, paired with a label identifying the generating compiler pass.

We opted for a text-based format and used generic programming to implement basic pretty-printing for the types involved in a `comp_trace` (essentially a simplified version of GHC's derived `Show` instances). On the Rocq side, we defined suitable notations to parse a trace directly as source code.

#### 3.1.2 Generating a certificate.

A certificate is a Rocq project that consists of three main components: the formal development of the different passes, the trace, and a top-level theorem. To this end, we developed a small command-line utility that simply includes the first two and creates a proper project structure. To then construct the type of the top-level theorem, the formal development exposes a basic interface:

```
Inductive claim :=
  | AccordingToSpec : rel_decidable -> claim
  | Verified : rel_verified -> claim
  | Unchecked : claim.
```

A `claim` describes what gets checked for a given compiler pass. Depending on the level of formalisation, this could be *according to specification* if the pass has a completed automation layer (as witnessed by a decidable translation relation `rel_decidable`), *verified* (witnessed by a verified, decidable translation relation), or completely *unchecked* otherwise.

The top-level theorem is constructed with some helper functions:

```
Definition claims_prop
  : (pass -> claim) -> comp_trace -> Prop.
Definition trace_dec : forall claims trace,
  option (claims_prop claims trace).
```

The `claims_prop` function computes the type of the top-level theorem, given a claim for each pass and a trace. The resulting type is a conjunction of the form $t_1\ R_1\ t_2 \wedge \cdots \wedge t_{n-1}\ R_{n-1}\ t_n$. Here, $t_i$ is a term and $R_i$ a relation depending on the claim: an `AccordingToSpec` claim will result in the relation being the corresponding translation relation, a `Verified` claim will result in semantic equivalence, and an `Unchecked` claim will be the universal relation, relating any two elements. The `trace_dec` function then allows to decide this top-level theorem, which internally uses the decision procedures and correctness results of the formalisation.

The proof script for this top-level theorem is therefore completely generic, as it is only parametrised by the claims and trace. Our command-line tool uses a simple textual template to construct this final piece of the certificate.

#### 3.1.3 Checking the certificate.

To verify a certificate, the checker uses Rocq to type-check the proof of the top-level theorem and checks that it actually states a property about the provided source and target program. This should be done before distributing a certificate, after which users of the compiled program can independently verify the certificate.

Since the formalisation is included as source code, it is completely transparant which passes up to which layer are

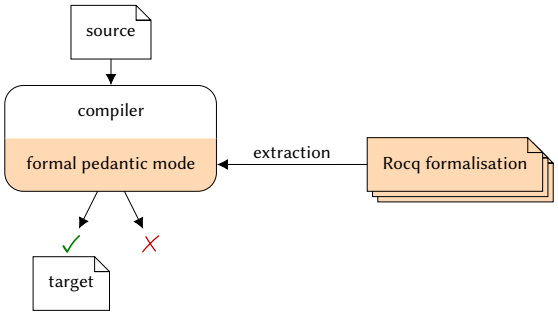



**Figure 5.** Architecture of formal pedantic mode

proven. One can independently inspect the ASTs, translation relations, dynamic semantics and definition of semantic equivalence to understand the top-level theorem.

### 3.2 Formal pedantic mode

*Formal pedantic mode* is a light-weight alternative to the full architecture of Section 3.1, for retrofitting verification on the compiler. The key idea is that after the compiler performs a pass, it also immediately runs the corresponding decision procedures in-process. But since decision procedures are defined in the proof assistant, not in the compiler, this requires some form of crossing language boundaries. In our case, we have leveraged Rocq's *extraction mechanism* [Letouzey 2002] to generate Haskell code, leveraging the functional nature of the formalisation.

Since our interface layer for PIR closely follows the syntax of the compiler (Section 2.2), and both Gallina and Haskell are statically-typed functional languages, it becomes trivial to extract the decision procedures from the automation layer, and integrate them in the Plinth compiler. After all, they have the simple type `term -> term -> bool`.

Figure 5 illustrates how a compiler runs in formal pedantic mode: if any of the decision procedures fail, the compiler fails and does not produce target code. Note that there is no certificate produced to witness the correctness compilation, which is typically not necessary during the development process (only at time of release). In this way, formal pedantic mode offers an efficient alternative that can provide quick feedback for potential miscompilations.

Additionaly, this mode is a valuable addition to a compiler test suite. Compilers typically have a collection of sample programs for measuring performance or testing succesful termination. Since the decision procedures are available as extracted Haskell code, it can easily be integrated and used for testing the correctness of compilation of those test cases. In Section 5 we discuss our initial experience of such an integration for the Plinth compiler and an existing test suite.

## 4 Formalisation

To reason formally about program transformations, we need to fix the semantics of the intermediate languages used by the Plinth compiler. This section outlines the key definitions required for the verification layer and how to establish *validator equivalence* of two programs. We have mechanised these in Rocq.

$$
\begin{array}{lcl}
\kappa & ::= & * \mid \kappa \Rightarrow \kappa \\
\sigma, \tau & ::= & \alpha \mid \sigma \longrightarrow \tau \mid \forall \alpha :: \kappa . \sigma \mid \lambda \alpha :: \kappa . \sigma \mid \sigma\ \tau \mid \\
 & & \texttt{ifix}\ \sigma\ \tau \mid \mathbb{U} \\
t & ::= & x \mid \lambda x : \tau . t \mid t\ t \mid \Lambda \alpha :: \kappa . t \mid t\ \{\tau\} \mid \\
 & & \texttt{iwrap}\ \sigma\ \tau\ t \mid \texttt{unwrap}\ t \mid \texttt{builtin}\ \mathbb{F} \mid \\
 & & \texttt{constant}\ \mathbb{U}\ k \mid \texttt{error}\ \tau \mid \texttt{let}\ [\texttt{rec}]\ \overline{b}\ \texttt{in}\ t \\
b & ::= & [\sim] x : \tau = t \mid \alpha :: \kappa = \tau \mid \\
 & & \texttt{data}\ \alpha\ (\overline{\beta :: \kappa}) = \overline{c}\ \texttt{with}\ x \\
c & ::= & x\ (\tau) \\
\mathbb{U} & ::= & \texttt{Bool} \mid \texttt{Unit} \mid \texttt{Data} \mid \ldots
\end{array}
$$

**Figure 6.** Syntax of PIR and PLC, PIR-specific constructs highlighted

### 4.1 Syntax of PIR and PLC

The Plinth compiler accepts Plinth, a subset of Haskell, and targets Plutus Core (PLC), which can be run on the Cardano blockchain. Most of the optimisation passes are done on the Plutus Intermediate Representation (PIR), a superset of PLC which in turn is a superset of System $F^{\mu}_{\omega}$, a polymorphic lambda calculus with type functions and recursive types. Figure 6 presents the syntax of the kinds, types and terms of both PIR and PLC (PIR-specific constructs have been highlighted in grey). We use square brackets `[]` to indicate optional syntax; a line over syntactic constructs indicates that these may be repeated zero or more times. We sometimes write $\varnothing$ for an empty list of bindings in a `let` group.

PIR and PLC share the same type language which consist of type variables, functions and universal quantification, lambda abstraction, application of type functions, an indexed fixpoint type for iso-recursive types, and built-in types $\mathbb{U}$.

The term language is a Church-style System $F^{\mu}_{\omega}$, where `iwrap` and `unwrap` are the term-level witnesses for iso-recursive types. Additionally, there is a set of built-in functions $\mathbb{F}$ for efficient computation with built-in types, such as arithmetic operations and hashing functions. We leave this set implicit here. Execution may produce an error; each error value is is annotated with its type to aid type checking.

PIR supports let bindings that can be mutually recursive. There are various flavours of let-bindings: term-level binding which can be strict or lazy (indicated by a $\sim$ symbol), type bindings and lastly algebraic datatypes (ADT) which can have type variables and consist of a set of constructors $c$ and an elimination principle $x$. Constructors are declared with a name and type signature.

$$\frac{\Delta \vdash \sigma :: * \qquad \sigma \Downarrow \sigma' \qquad \Delta; (\Gamma, x : \sigma') \vdash t : \tau}{\Delta; \Gamma \vdash (\lambda x : \sigma.\ t) : \sigma' \to \tau}\ \text{T-LamAbs}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau :: \kappa \qquad \tau \Downarrow \tau' \\ F \Downarrow F' \qquad \Delta \vdash F :: (\kappa \Rightarrow *) \Rightarrow (\kappa \Rightarrow *) \\ F'\ (\lambda X :: \kappa.\texttt{ifix}\ F'\ X))\ \tau' \Downarrow \sigma \qquad \Delta; \Gamma \vdash t : \sigma\end{array}}{\Delta; \Gamma \vdash \texttt{iwrap}\ F\ \tau\ t : \texttt{ifix}\ F'\ \tau'}\ \text{T-IWrap}$$

**Figure 7.** Selected rules of the PIR type system

### 4.2 Static semantics

The syntax in Figure 6 defines the types and kinds of PIR. Kinds are either a base kind or function kind, and kinding rules are of the form $\Delta \vdash \tau :: \kappa$. The rules of the kind system are standard (see for example [Pierce 2002]).

Type expressions may contain $\beta$-redexes. To decide type equality during type checking, the Plinth compiler therefore uses a type normalisation algorithm. We define an inductive relation $\sigma \Downarrow \tau$, stating that a type-expression $\sigma$ normalises to $\tau$. Again, such reduction relations are fairly standard for System $F_\omega$.

Typing rules are of the form $\Delta; \Gamma \vdash t : \tau$ (Figure 7), stating that in kinding context $\Delta$ and typing context $\Gamma$, term $t$ has type $\tau$. In contrast to the standard presentation that has a type-conversion rule, we define a syntax-directed system that uses type normalisation, which aligns with the Plinth type-checker. For example, T-LamAbs extends $\Gamma$ after first normalising the type annotation $\sigma$. The PIR type checker ensures that it only checks terms against normalised types. Similarly, our rules ensure that we only assign normalised types to a term.

Type normalisation has another role in the type system: it drives the unfolding of recursive types. In PIR, a recursive type `ifix` $F\ T$ represents a indexed fixpoint [Peyton Jones et al. 2019] of a type-function $F$ of kind $(\kappa \Rightarrow *) \Rightarrow (\kappa \Rightarrow *)$. The corresponding typing rule T-IWrap unfolds fixpoint one step by normalising an application of $F$.

The complete rules of the static semantics of PIR are given in Appendix B.

### 4.3 Dynamic semantics

We define a dynamic semantics of PIR (and thus PLC) using a strict, big-step operational semantics with substitution. We write $t \Downarrow r$, stating that term $t$ evaluates to result $r$. Here, $r$ can be either a value or an `error` . Most rules are entirely standard; we add rules for dealing with errors (which abort the computation) and a set of rules for the various let-constructs of PIR, both recursive and non-recursive ($t \Downarrow_{\texttt{rec}} r$ and $t \Downarrow_{\texttt{nonrec}} r$ respectively).

In our Rocq mechanisation of the dynamic semantics, we have covered the vast majority of PIR language constructs, with the exception of recursive algebraic datatypes. Some of the complex built-in operations (such as hashing primitives) currently have no corresponding formalisation in Rocq. However, by axiomatising their implementation, we can still formally reason about optimisation passes, which almost always leave built-in operations untouched.

The complete rules of the operational semantics of PIR are given in Appendix C.

### 4.4 Validator equivalence

The Plinth compiler is primarily used to compile *validators*, which are programs that can be deployed on the Cardano blockchain to lock digital assets. On Cardano, any proposed transaction needs to be validated before it is executed. For each asset that it tries to transfer, the attached validator program is executed to determine if the transaction may do so. Typical conditions of a validator include public key authentication and enforcing payment deadlines. Validators are the fundamental building blocks for implementing higher-level smart contracts on a UTXO-style blockchain.

A Plinth validator is a function of type `Data` → `Unit`. The argument contains all information the validator may need to perform its checks, such as the proposed transaction and the current time. If the validator succeeds, it terminates with a unit value $\langle\rangle$, otherwise it halts with an error.

**Definition 1** (Validator equivalence $\equiv_{\text{val}}$)**.** *If $p$ and $q$ are well-typed validator scripts, and $i$ is an arbitrary constant of type* `Data`*, we write $p \equiv_{val} q$ to mean*

$$p\ i \Downarrow \langle\rangle \iff q\ i \Downarrow \langle\rangle$$

**Definition 2** (Correct compiler pass)**.** *A translation relation $R$ is correct, when for all well-typed validator scripts $p, q$*

$$p\ R\ q \implies p \equiv_{val} q$$

In other words, any two related programs have the same operational behaviour. To completely verify the compiler, it is required to prove that each translation relation from the specification layer is correct.

Note that this notion of equivalence implies identical termination behaviour of related programs. In some compilers, it may be fine if a non-terminating program is optimised into a terminating one. In our case, this is a problem: a non-terminating validator will never unlock funds, whereas a terminating one might do so.

## 5 Preliminary evaluation

In this section, we evaluate the proposed methodology, documenting our experience with the certification of the Plinth compiler. We briefly go through the formalisation of the dead code elimination pass, and discuss how it impacts the compiler's codebase and what practical challenges we encountered during the proof development.

***Dead code elimination.*** Dead code elimination is an important compiler pass that is run several times in the simplifier pipeline of the Plinth compiler. As the Plinth standard libraries are included as a top-level let binding for every program, dead code elimination is absolutely necessary to keep code size of the generated binaries in check. We have implemented the four layers for this pass and will briefly reflect on each of those. A more detailed treatment of this pass can be found in Appendix A.

The pass is concerned with let bindings. For example, the compiler may analyze the following program

$$\texttt{let } x = 3 \texttt{ in let } y = x + x \texttt{ in } 10$$

and decide that it can be optimised to the term 10. Note that the pass is not always entirely trivial: $x$ is dead code only because $y$ is dead code (and $x$ is not used anywhere else). Furthermore, PIR is strict, meaning that a binding may only be removed when its evaluation has no side-effects (such as throwing an error).

For the specification layer, let us consider the main rule of non-recursive let bindings:

$$\frac{\begin{array}{c}\mathrm{BV}(b) \cap \mathrm{FV}(\texttt{let } bs' \texttt{ in } t') = \varnothing \\ \mathrm{BTV}(b) \cap \mathrm{FTV}(\texttt{let } bs' \texttt{ in } t') = \varnothing \\ b \in \textsc{Pure} \qquad \texttt{let } bs \texttt{ in } t \triangleright \texttt{let } bs' \texttt{ in } t'\end{array}}{\texttt{let } (b; bs) \texttt{ in } t \triangleright \texttt{let } bs' \texttt{ in } t'} \text{ DCE-Elim}$$

Given a let with $b; bs$ as its bindings, $b$ may be removed under a few conditions: (1) the term variables and (2) type variables bound by $b$ may not occur freely in the resulting post-term (expressed as those sets being disjoint), (3) $b$ is a PURE binding (i.e. terminates without side-effects) and (4) the rest of the term can be recursively translated. Since deciding termination is undecidable in general, PURE is a sound subset of terms that have the property (based on the corresponding compiler analysis).

For the automation layer, we have defined the decision procedures for the disjointness checks, the PURE property and the translation relation itself and proven it equivalent to the translation relation. Their implementations straightforwardly follow the inductive structure of the inference rules.

The interface layer has identical functionality for most passes: printing the pre- and post-term in the compiler and then parsing them into Rocq (Section 2.2). For the dead code elimination pass however, we implement some additional behaviour. As a result of removing let bindings, a let expression may end up with zero bindings, so the compiler pass also performs a bit of cleanup, transforming $\texttt{let } \varnothing \texttt{ in } t$ into $t$. Although we could include that transformation in the rules such as DCE-NR-Elim, it clutters the translation relation and obscures the simplicity of two conceptually different transformations.

Instead, we define a separate translation relation (denoted here as $\triangleright'$) that solely captures the cleanup of empty let groups. The overall specification for the pass then becomes $\exists t_1.\ t_0 \triangleright t_1 \wedge t_1 \triangleright' t_2$. Here, $t_0$ and $t_2$ are the terms dumped by the compiler, but AST $t_1$, which may still contain empty let groups, never existed during compilation! Therefore, we construct this *virtual AST* within Rocq in the interface layer, using a straightforward recursive function on $t_0$ and $t_2$.

Although decomposing a pass in this manner incurs some more work in the interface layer, it pays off in the automation and verification layer: another example is the inliner pass, which includes a form of dead code elimination. By analogously decomposing that pass and constructing an intermediate AST in Rocq, translation relations remain simple and can be reused across specifications.

Finally, for the verification layer, we have proven validator equivalence using a simulation-style argument, with induction on the evaluation relation. In the forward direction this requires proving a simple lemma that $[v/x]\ t = t$ when $x \notin \mathrm{FV}(t)$. In the backwards direction, it requires a lemma that any $t \in$ PURE terminates according to the big-step evaluation relation.

While dead code elimination may appear like a relatively simple transformation, its structure is representative of more complex optimisations. Its correctness relies on two program analyses: purity of bound expressions and strong liveness of variables. In general, compiler passes will often rely on analysis results or pre-conditions established by previous transformations. Those properties will have to appear in some form in the translation relation.

Crucially, checking that the property of an analysis holds can be much simpler than performing the analysis itself, as we see with the disjointness check instead of a strongly live variable analysis. Additionally, results of complexer analyses may simply be dumped by the compiler alongside the ASTs of the pass in the interface layer, leaving only the task to check (in the automation layer) that they imply the required property. This is also the reason that CompCert resorts to translation validation for register allocation [Rideau and Leroy 2010].

***Constructing certificates.*** We ran our certifier on a realistic Plinth validator that implements an auction (260 LOC) using a commodity laptop. The compiler pretty-printed and dumped all 362 intermediate ASTs in about 35s, resulting in 160MB of plain text. Generating the Rocq certificate took 2s, and the overall certificate can be compressed into 13MB using standard gzip compression.

We built and type-checked the certificate with verification claims on two passes (dead code elimination and non-recursive let compilation), which took about 14 minutes. The main culprit here is our Rocq parser for AST terms; the actual proof checking is relatively fast. There is plenty of low-hanging fruit to speed up the parser – but as a proof of concept, the current implementation suffices. Note that certificate generation is only run once per software release;

the formal pedantic mode gives similar guarantees more cheaply. Running the certificate checker overnight is perfectly acceptable in this domain, where it is essential to only deploy smart contracts that are correct.

One of the small engineering hurdles we encountered in generating certificates is that a full compilation trace could not fit in a single Rocq script, as `coqc` would run out of memory due to the sheer size of the ASTs involved. Instead, we now produce a single Rocq file per AST and construct the trace by importing the ASTs from the compiled modules. Running Rocq projects at this scale uncovers other issues. For example, the naive Peano encoding of natural numbers obviously does not scale to ASTs that use natural numbers for the representation of variable names. These problems are easy to overcome – we use a more efficient representation now – but often unforeseen.

***Integrating the Rocq formalisation as compiler tests.*** To assess a formal pedantic mode in the Plinth compiler (Section 3.2), we have also integrated our Rocq development in the compiler, measured performance characteristics, and observed the impact on the existing code-base. To this end, we have extracted the Rocq decision procedure of the dead-code elimination pass, checking the pass with several input programs that contain dead code.

We have extended the compiler's build system to run Rocq's extraction mechanism to produce Haskell before the compiler is compiled. To interface with this generated code, there is minimal glue code required, since we have made a point of staying close to the compiler representation in the formalisation.

The Plinth compiler contains a test-suite of *golden tests* for each compiler pass: a set of PIR programs with their expected output. These programs are usually small in size and they test some behaviour relevant to the pass. In the case of dead-code elimination, there are 16 such test cases.

To get a sense of the performance characteristics, we measured the execution time of the dead-code elimination decision procedures for each of the golden tests. They typically run in a handful of milliseconds on a commodity laptop, adding roughly 1–5% to the overall pass execution time.

We have also tested the decision procedure on a significantly larger program: the interpreter for the Marlowe smart contract language [Lamela Seijas and Thompson 2018]. This interpreter evaluates contracts written in the Marlowe DSL, used to implement a class of financial contracts. We noticed a much longer execution time of 2–3s, dominating the overall pass execution time. This can be explained by the fact that our naive decision procedure is quadratic, as it repeatedly computes the set of free variables in recursive calls. There are numerous ways to resolve this issue. A tupling transformation would make the check linear again. Alternatively, we can adapt the compiler to provide information about the set of dead bindings that have been eliminated by a given pass, removing the need for proof search in the decision procedure altogether. Even if formal pedantic mode is slow for a certain pass, most smart contracts are relatively small; developers may always choose to run the formal pedantic mode less frequently.

The impact on the Plinth compiler codebase has been very modest. The extracted code from Rocq is around 1K lines of code, whereas the glue code is only around 250 LOC. Compared to the overall size of the Plinth project, this is negligible.

Interestingly, we discovered a mismatch between the implementation and the specification, as one of the tests failed. It turned out that it was not a compiler bug, but that the implementation was recently extended to include an exceptional case where parts of a datatype binding can be eliminated (but not the full binding). This example indicates the importance of such a test-suite to keep the compiler implementation and its specification in sync.

***Proof engineering considerations.*** After working on the formalisation of several compiler passes, we have noticed that it is relatively easy to implement the specification and automation layer, which suffice to set up the a specification checker for certificates or formal pedantic mode of a pass. For example, common duplication across translation relation definitions such as compatibility rules (Section A.1) can be factored out and reused. Moreover, dumping and parsing of the interface layer can be reused, as most passes do not need to dump more information than the pre- and post-terms.

Writing decision procedures for inductive definitions can still be quite some work. To address this, we have adapted work of the QuickChick project [Paraskevopoulou et al. 2022]. That work uses meta-programming in Rocq to generate decision procedures and soundness proofs, but for a limited subset of inductive definitions. As a result, we can sometimes save time by generating (parts of) the decision procedures and soundness lemmas.

The verification layer however requires most of the work. In part, this is expected because of the larger proofs involving the formalised metatheory. Writing out the required lemmas about substitutions, variable binding, and other common patterns in working with our validator equivalence requires substantial effort.

Another challenge mechanising proofs about PIR is that it is a *not* a toy language: it requires reasoning about language constructs that can be awkward or inconvenient to manipulate, that are often omitted in more idealised language formalisations. For example, mutually recursive let-groups cause the AST types to have nested recursion, resulting in more complicated induction schemes and a programming style aimed at passing Rocq's termination checker. Other constructs such as built-in functions (which require a notion of partial application) and errors (with atypical control flow)

all require extra rules in the big-step semantics, resulting in many more cases in each proof.

Although most of our Rocq development is specific to the PIR pipeline, the ideas of the methodology are generally applicable for compilers that use a nano-pass architecture. Furthermore tooling such as InducTEX can readily be used for generating TEX from other inductive specifications.

At the same time, the layered approach is not limited to the Rocq prover. Developers at IOG have started adopting the layered methodology for the PLC backend of the compiler, but using the Agda language [Bove et al. 2009].

## 6 Discussion

### 6.1 Related work

***Correct compilation.*** There is a rich line of work on compiler verification, we will not attempt to give an exhaustive overview of the area. We have previously mentioned other verification projects using interactive proof assistants, notably CakeML [Kumar et al. 2014] and CompCert [Leroy et al. 2016]. Those projects focus on compilers implemented in a proof assistant for well defined and stable languages. Despite the higher cost of such approach, verified compilers (when total) have a completeness guarantee that translation validation cannot offer: each run of the compiler is correct.

The idea of translation validation goes back to work on compiling synchronous programming languages [Cimatti et al. 1997; Pnueli et al. 1998]. Over the years, it has been succesfully applied to optimising compilers on a larger scale, such as GCC [Necula 2000; Sewell et al. 2013] and LLVM [Lopes et al. 2021], catching numerous compiler bugs. These works have a similar motivation to ours (retrofitting verification on existing compilers) but differ in the type of languages, which have a low-level imperative style and are first-order. The semantics are based on control flow graphs, often combined with symbolic execution and SMT solvers for finding proofs. While this allows for a high degree of automation, the verification is restricted to intra-procedural optimisations.

Cogent [O'Connor et al. 2021] is a certifying compiler based on translation validation, also built for a purely functional language. It is aimed at systems programming, and features a uniqueness type system to avoid manual memory management, as well as garbage collection. In contrast to our approach, it only works for a white-box approach in the sense that it assumes full access and control over the compiler, as the compiler generates the target code (a well-defined subset of C), and embedding of the source program as well as the correctness proofs. Furthermore, the correctness proofs are concerned with the translation steps between vastly different representations and semantics, whereas optimisations are deferred to the (trusted) C compiler.

The term *certifying compiler* was originally introduced in the context of proof-carrying code [Necula 1997], which focuses on distributing compiled code together with proof of a safety policy, such as type-safety or memory-safety. Such untrusted code can then safely interact with a host system. Our certificates are similarly distributed with compiled code, but do not interact with a host system: Plutus validators are stand-alone programs and our certificate is meant to support reasoning about functional properties of the source code.

There are several other compilers for smart contract languages. ConCert is a smart contract verification framework in Rocq [Annenkov et al. 2020]; there is ongoing work on verifying a compiler for the Albert language on the Tezos blockchain; the K framework has been used to verify Ethereum contracts using translation validation techniques similar to those described here [Park et al. 2020].

***Existing Formalisations.*** The type system for PLC has been formalised before [Chapman et al. 2019], using an intrinsically-typed syntax in Agda. Typing rules for PIR have been described previously [Peyton Jones et al. 2019], but do not align completely with the compiler implementation of the type-checker. In the current work, we give a semantics that stays true to the current implementation of the Plinth compiler.

Krijnen et al. [2024] have previously described specifications of various passes of Plinth compiler. That work, however, does not discuss the verification of those specifications, which we address in the *verification layer*—a key contribution of the current paper. Without the validator equivalence, as defined in Section 4 and employed in Section 5, there may still be bugs that change the meaning of a program in both the compiler and the specification.

### 6.2 Further work

***End-to-end verification.*** The surface language of the Plinth compiler is a subset of Haskell, which is directly desugared via GHC Core into PIR, we have not yet included those languages in the formalisation. Can we extend this approach to create an end-to-end certifying compiler? This would require a formalisation of Haskell's rich surface language as well as GHC's Core. To make matters more complicated, PIR is a strict language and the subset of Haskell is compiled as if it were a strict language. Any correctness preservation proof would have to work under the (unusual) assumption that GHC Core is evaluated strictly.

As an alternative, we have started building on the work done in the context of CertiCoq [Anand et al. 2017], a project that aims to implement a verified compiler for Rocq's Gallina. Other work [Annenkov et al. 2022] has recently shown how to extend CertiCoq's pipeline to target a typed intermediate language $\lambda^T_\square$. To achieve end-to-end certification, we envision a verified translation from this lambda calculus to PIR. This approach does have its own drawbacks. Firstly, Plinth was designed specifically to have a single language for on-chain and off-chain code. Writing smart contracts in Gallina would break this property, even if the resulting contracts

can be extracted to Haskell. More practically, any realistic smart contract requires numerous functions from the Plinth standard libraries, all written in Haskell. We have an initial experiment using hs2coq [Spector-Zabusky et al. 2018] to port these libraries to Gallina that appears quite promising.

***Semantic verification.*** So far, we have verified the translation relations for dead code elimination and non-recursive let compilation. The existing work by Krijnen et al. [2024] gives an overview of the various passes of Plinth. We expect that most of these, such as variable renaming or let desugaring, will be relatively easy to verify with the architecture presented here. Other passes, such as the translation from algebraic datatypes to their Scott encodings [Peyton Jones et al. 2019] will be harder to verify, since the correctness relies on parametricity properties. To that end, we have ongoing work on a logical relation for PIR, to prove a much more powerful *contextual equivalence*, which implies validator equivalence.

***Language support.*** In the semantics described so far, we have not yet formalised the entire Plinth language, as we do not yet cover recursive ADTs. Plinth's treatment of (locally bound) ADTs is somewhat non-standard. We do not believe this to be a fundamental limitation of our approach, but have simply chosen to focus our engineering effort elsewhere.

## References


Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. https://doi.org/10.1017/S0956796822000077

Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. 2020. ConCert: a smart contract certification framework in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 215–228.

N. Atzei, M. Bartoletti, and T. Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust (POST 2017) (LNCS, Vol. 10204)*.

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 73–78.

J. Chapman, R. Kireev, C. Nester, and P. Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction (MPC 2019) (LNCS, Vol. 11825)*.

Alessandro Cimatti, Fausto Giunchiglia, Paolo Pecchiari, Bruno Pietra, Joe Profeta, Dario Romano, Paolo Traverso, and Bing Yu. 1997. A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings 9*. Springer, 202–213.

Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. 2017. Closing the gap–the formally verified optimizing compiler CompCert. In *SSS'17: Safety-critical Systems Symposium 2017*. CreateSpace, 163–180.

Jacco Krijnen. 2024. InducTeX: A MetaCoq plugin for typesetting inductive definitions. (2024). Extended abstract presented at the Tenth International Workshop on Coq for Programming Languages.

Jacco O.G. Krijnen, Manuel M.T. Chakravarty, Gabriele Keller, and Wouter Swierstra. 2024. Translation certification for smart contracts. *Science of Computer Programming* 233 (2024), 103051. https://doi.org/10.1016/j.scico.2023.103051

Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.

Pablo Lamela Seijas and Simon Thompson. 2018. Marlowe: Financial contracts on blockchain. In *International symposium on leveraging applications of formal methods*. Springer, 356–375.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert—a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

Pierre Letouzey. 2002. A new extraction for Coq. In *International Workshop on Types for Proofs and Programs*. Springer, 200–219.

Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.

John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science* 1 (1967).

George C Necula. 1997. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 106–119.

George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021).

Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. 2022. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 966–980.

D. Park, Y. Zhang, and G. Rosu. 2020. End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract. In *Computer Aided Verification (CAV 2020) (LNCS, Vol. 12224)*.

Michael Peyton Jones, Vasilis Gkoumas, Roman Kireev, Kenneth MacKenzie, Chad Nester, and Philip Wadler. 2019. Unraveling recursion: compiling an IR with recursion to System F. In *International Conference on Mathematics of Program Construction*. Springer, 414–443.

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (July 2002), 393–434.

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166.

Silvain Rideau and Xavier Leroy. 2010. Validating register allocation and spilling. In *International Conference on Compiler Construction*. Springer, 224–243.

Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 471–482.

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 14–27.

# A Case Study: Dead Code Elimination

Dead code elimination is an important compiler pass that is run several times in the simplifier pipeline of the Plutus compiler. As the Plutus standard libraries are included in a top-level let binding for every program, dead code elimination is absolutely necessary to keep code size of the generated binaries in check. In this section, we illustrate the verification process for this compiler pass.

The complete verification of each compiler pass is beyond the scope of the current paper. Instead, we describe the key definitions and lemmas required to verify a typical compiler pass completely. The work presented here has been mechanised in the Rocq prover.

## A.1 Specification layer

Informally, code is dead when it is not used in the evaluation of the rest of the program. The Plutus compiler concerns itself with eliminating dead `let` bindings by way of a live variable analysis. Since the `let` construct is strict by default, a binding can only be removed if it is known to terminate. Otherwise, the compiler might transform a non-terminating program into a terminating one, changing the program's semantics.

In Figure 8 we define a predicate on bindings, Pure, that characterises the subset of bindings that the compiler considers "pure", i.e., their evaluation is guaranteed to terminate. Strict bindings are pure if they are values. Non-strict bindings are always considered pure, since no evaluation happens at its definition. Strict bindings, on the other hand, can only be removed if they bind a value (such as a lambda abstraction or constant), this is the same criterion used in the compiler. Datatype bindings and type bindings do not require evaluation, hence they are trivially pure.

A detailed account of the specification for the dead code elimination pass has previously been given by Krijnen et al. [2024]. It defines a translation relation that is composed of several syntactic properties, such as well-scopedness and uniqueness of variable names. In this section, we rephrase this specification, giving a single binary inductive relation that immediately captures dead code elimination. Compared to their work, this definition is more general as it does not require globally unique variables and the corresponding semantic argument is more direct.

In Figure 8, we sketch the translation relation for dead code elimination $\triangleright$ for terms and $\triangleright_B$ for let-bindings. We can divide the rules in two categories:

- **Compatibility rules** Dead code elimination need not remove any bindings at all. Hence, the identity relation should be included in our translation relation. For each language construct, we add a compatibility rule, stating the translation relation is closed under that construct. DCE-Apply is a compatibility rule.

$\boxed{b \in \textsc{Pure}}$

$$\frac{}{\sim x : \tau = t \in \textsc{Pure}} \qquad \frac{t \in \text{value}}{x : \tau = t \in \textsc{Pure}}$$

$$\frac{}{\texttt{data}\ \alpha = \ldots \in \textsc{Pure}} \qquad \frac{}{\alpha :: \kappa = T \in \textsc{Pure}}$$

$\boxed{s \triangleright t}$

$$\frac{s \triangleright t \qquad t \triangleright t'}{s\ t \triangleright s'\ t'}\ \text{DCE-Apply}$$

$$\frac{\begin{array}{c}\mathrm{BV}(b) \cap \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ t') = \varnothing \\ \mathrm{BTV}(b) \cap \mathrm{FTV}(\texttt{let}\ bs'\ \texttt{in}\ t') = \varnothing \\ b \in \textsc{Pure} \qquad \texttt{let}\ bs\ \texttt{in}\ t \triangleright \texttt{let}\ bs'\ \texttt{in}\ t'\end{array}}{\texttt{let}\ b; bs\ \texttt{in}\ t \triangleright \texttt{let}\ bs'\ \texttt{in}\ t'}\ \text{DCE-Elim}$$

$$\frac{b \triangleright_B b' \qquad \texttt{let}\ bs\ \texttt{in}\ t \triangleright \texttt{let}\ bs'\ \texttt{in}\ t'}{\texttt{let}\ b; bs\ \texttt{in}\ t \triangleright \texttt{let}\ b'; bs'\ \texttt{in}\ t'}\ \text{DCE-Keep}$$

**Figure 8.** Translation relation for dead code elimination (selected rules)

- **Binding-related rules** When relating a let group of the form $\texttt{let}\ b; bs\ \texttt{in}\ t$, there are two cases: either the binding $b$ was removed in the post-term or it was kept. If the binding has been removed (DCE-Elim), the post-term should be of the form $\texttt{let}\ bs'\ \texttt{in}\ t$. This transformation is only correct under certain conditions: (1) $b$ is a pure binding, to preserve termination behaviour; (2) the remainder of the let group is related; (3) the bound term variables (BV(b)) do not occur freely in the post-term ($\mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ t')$), and (4) the same requirement on type variables. If the binding is not eliminated (DCE-Keep), we only require that (1) there is a related binding $b'$ in the post-term and (2) the rest of the let group is related.

Note that the last two conditions of the DCE-Elim rule in Figure 8 ensure that any variables bound by the let binding are unused. It is important to check for the free variables in the *post-term* rather than those in the *pre-term*. Consider a let-bound variable $x$ that occurs in other dead code:

$$\texttt{let}\ x = 3\ \texttt{in}\ \texttt{let}\ y = x + x\ \texttt{in}\ 10 \triangleright 10$$

Here, $x$'s definition is dead code while it occurs freely in the pre-term, but not in the post-term. This distinction ensures that our relation describes *strong* live variables.

```
1  Fixpoint dec_dce (t t' : term) : bool :=
2    match t t' with
3    | Apply s t, Apply s' t' =>
4       dec_dce s s' && dec_dce t t'
5    | Let NonRec (b::bs) tb, Let NonRec (b'::bs') tb' =>
6       if dec_dce_B b b'
7       then
8         dec_dce
9           (Let NonRec bs tb)
10          (Let NonRec bs' tb')
11      else
12        dec_pure b &&
13        dec_disj (bv b) (fv (Let NonRec bs' tb')) &&
14        dec_disj (btv b) (ftv (Let NonRec bs' tb')) &&
15        dec_dce (Let NonRec bs tb) t'
16   | ...
17   end
18 with dec_dce_B : binding -> binding -> bool
```

**Figure 9.** Decision procedure for dead code elimination (selected cases)

### A.2 Automation layer

In the automation layer, we define the decision procedures for each of the relations used to specify the dead code elimination pass. Implementating these procedures is usually quite straightforward: many cases can be read off from the inference rules, as they are mostly syntax-directed.

In Figure 9 we show a code fragment of the decision procedures. Those for Disjoint and Pure straightforwardly implement their relations, so we omit their implementations here. The function `dec_dce` is defined mutually recursive with `dec_dce_B`, which decide $\triangleright$ and $\triangleright_B$ correspondingly. Here we show only the cases for function application (line 3, corresponding to rule DCE-Apply) and non-recursive lets with a non-empty list of bindings. In the latter case, both DCE-Elim and DCE-Keep are applicable, so we first try to apply the DCE-Keep rule (line 6); if that fails, we try the DCE-Elim rule (lines 9–12). The resulting decision procedure is sound and complete with respect to the translation relation, which we prove by induction on the pre-term:

**Lemma 3** (Soundness of decision procedure)**.** `dec_Term t t' = true` $\iff t \triangleright t'$.

### A.3 Interface Layer

For most passes, the interface layer has the same functionality: printing the pre- and post-term in the compiler and then parsing them into Rocq (Section 2.2). For the dead code elimination pass however, we extend the interface layer with further pass-specific behaviour: constructing a *virtual intermediate AST*.

In the dead code elimination pass, the compiler not only removes unused bindings, but it will also clean up let groups that as a result have no bindings. Although this is a very local transformation, it clutters the translation relation and the decision procedure when formulated simultaneously with dead code elimination. Therefore we decompose the two transformations and implement another translation relation (denoted here as $\triangleright'$) that solely captures the removal of empty let groups. The overall specification for this pass then becomes $\exists t_1.\ t_0 \triangleright t_1 \wedge t_1 \triangleright' t_2$. Here, $t_0$ and $t_2$ are the terms dumped by the compiler. Within the compiler however, the AST $t_1$, which may still contain empty let groups, never exists. Therefore, we construct this tree within Rocq, using a straightforward recursive function on $t_0$ and $t_2$.

Although decomposing a pass in this manner incurs some more work in the interface layer, it pays off in the automation and verification layer: another example is the inliner pass, which includes a form of dead code elimination. By analogously decomposing that pass and constructing an intermediate AST in Rocq, translation relations remain simple and can be reused across specifications.

### A.4 Verification Layer

Finally, we establish that the specification of dead code elimination preserves program semantics. In this section, we sketch the key lemmas that are necessary to complete the proof. Before formally proving validator equivalence, we first prove that the translation relation preserves typing.

**Lemma 4** (Dead code preserves typing)**.** *If $t \triangleright t'$ and $\Delta; \Gamma \vdash t : \tau$, then $\Delta; \Gamma \vdash t' : \tau$.*

*Proof.* By induction on the derivation of the translation relation. The case of DCE-Elim relies on a weakening property, since $\Gamma$ and $\Delta$ contains variables that are eliminated in the post-term, but do not occur freely. □

For proving validator equivalence, we first prove an obvious fact about substitution:

**Lemma 5** (Substitution of eliminated bindings)**.** *If $x \notin fv(t)$, then for all terms $s$, $[s/x]t = t$*

In other words, substitution behaves as the identity on unused variables. A similar lemma is needed for type substitution.

Similarly, we prove that pure bindings can be safely added to a let expression:

**Lemma 6** (Purity of Pure bindings)**.** *If $b \in$ Pure, and* `let` $bs$ `in` $t \Downarrow v$, *then also* `let` $b; bs$ `in` $t \Downarrow v$

Finally, we prove a lemma that dead-code elimination includes the identitiy relation:

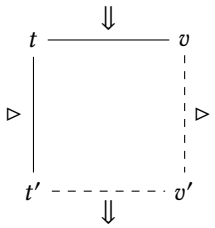
**Lemma 7** ($\triangleright$ is reflexive)**.** *For all terms $t$, $t \triangleright t$.*

Proof: using the inference rules without DCE-Elim.

With these three lemmas in hand (and with many smaller facts about substitution), we can prove validator equivalence.

As we saw previously in Section 4, this requires first proving a simulation diagram, in which evaluation commutes with the translation relation. We do this separately in both direction, here we demonstrate the forward direction:

**Lemma 8** ($\Downarrow$ respects $\triangleright$ (forward)). *If $t \triangleright t'$ and $t \Downarrow v$, then there exists $v'$, such that $t' \Downarrow v'$ and $v \triangleright v'$*




The proof is done by induction on the evaluation relation, using Lemma 5 for the case where a let binding is eliminated. Note that because of the mutual *family* of evaluation relations (see Appendix C), this proof requires a more elaborate mutual induction scheme. The backwards direction proceeds similarly, and uses lemma 6

**Theorem 9** (Correctness of dead code elimination). *If $\varnothing; \varnothing \vdash t : \texttt{Data} \to \texttt{Unit}$ and $t \triangleright t'$, then $t \equiv_{val} t'$*

Proof: Let i be a constant of the built-in type 'Data', by Lemma 7, $i \triangleright i$. By rule DCE-Apply, $ti \triangleright t'i$. Forward direction: assume $ti \Downarrow v$, then by Lemma 8, there exists a $v'$ such that $t'i \Downarrow v'$ and $v \triangleright v'$. By type preservation, either $v = \langle\rangle = v'$ or $v = \texttt{error} \ = v'$, so we can conclude that $v = v'$. The backwards direction is symmetric.

## B Static semantics of PIR

In the following figures, we have formalised the type and kind system of PIR. These rules are rendered based on the Rocq inductives that we use in the formalisation.

- Figure 10 gives the kinding rules
- Figure 11 describes how types can be normalised in a normal form
- Figure 12 give the typing rules for term constructs in the PIR language
- Figure 13 gives typing rules for constructor signatures, and the different type of bindings in a let group.

## C Operational semantics of PIR

We have defined the operational semantics of PIR using a substitution-based reduction. Once again, the rules are simply based on the Rocq formalisation. First, we define values as a predicate on terms in Figure 14. This includes neutral terms for partially-applied built-in functions.

In Figure 15, the rules for evaluation of most term constructs are given. For example, the case for function application E-Apply uses substitution after evaluation the argument to a value (and checking it is not an error). The semantics also include rules for dealing with errors that halt the program immediately: in Figure 16, most language constructs have a rule explaining how to propagate a thrown error. For partial applications of built-in functions, we require a set of rules that deal with neutral values (Figure 17).

We define a separate evaluation relation for both these non-recursive let bindings ($t \Downarrow_{\texttt{nonrec}} v$) and recursive let bindings ($t \Downarrow_{\texttt{rec}} v$) in Figure 18; these are embedded with trivial rules in the main evaluation relation (see bottom of Figure 15). Both these rules process the individual let bindings in the binding group one by one. The evaluations rules for both non-recursive (E-Let) and recursive let bindings (E-LetRec) appeal to these relations.

To evaluate non-recursive let-binding groups, we require several rules to distinguish strict and non-strict recursive let bindings, term bindings and type bindings. An example rule, E-Let-NonRec-TermBind, shows how to define the semantics of strict term-binding by evaluating the right-hand-side and substituting the resulting value.

For recursive binding groups, we have a similar evaluation relation, extended with some additional context, keeping track of the entire group of bindings $b_0$. This is needed as the let bindings may be mutually recursive. We write $\overline{b_0} \vdash \texttt{let rec } \overline{b} \texttt{ in } t \Downarrow_{\texttt{nonrec}}$, where $b_0$ stores to the complete sequence of let-bindings being processed. The E-LetRec rule instantiates $b_0$ to the complete binding group. The rule E-LetRec-TermBind defines the evaluation of a single (non-strict) recursive binding, by substituting a single unfolding of the bound variable's right hand side, appropriately wrapped in the binding group.

$\boxed{\Delta \vdash \tau :: \kappa}$

$$\frac{\Delta(X) = K}{\Delta \vdash X :: K}\ \textsc{K-Var} \qquad \frac{\Delta \vdash T_1 :: * \qquad \Delta \vdash T_2 :: *}{\Delta \vdash T_1 \to T_2 :: *}\ \textsc{K-Fun}$$

$$\frac{\Delta \vdash T :: K \qquad \Delta \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *)}{\Delta \vdash \texttt{ifix}\ F\ T :: *}\ \textsc{K-IFix} \qquad \frac{\Delta, X :: K \vdash T :: *}{\Delta \vdash (\forall X :: K.T) :: *}\ \textsc{K-Forall}$$

$$\frac{\text{built-in type } \mathbb{U} \text{ has kind } K}{\Delta \vdash \mathbb{U} :: K}\ \textsc{K-Builtin} \qquad \frac{\Delta, X :: K_1 \vdash T :: K_2}{\Delta \vdash (\lambda X :: K_1.T) :: K_1 \Rightarrow K_2}\ \textsc{K-Lam}$$

$$\frac{\Delta \vdash T_1 :: K_1 \Rightarrow K_2 \qquad \Delta \vdash T_2 :: K_1}{\Delta \vdash T_1\ T_2 :: K_2}\ \textsc{K-App}$$

**Figure 10.** Kinding of types

$\boxed{\sigma \Downarrow \tau}$

$$\frac{T_1 \Downarrow \lambda X :: K.T_1^n \qquad T_2 \Downarrow T_2^n \qquad \left[T_2^n/X\right] T_1^n \Downarrow T^n}{T_1\ T_2 \Downarrow T^n}\ \textsc{N-Beta} \qquad \frac{T_1 \Downarrow T_1^n \qquad T_1^n \in \mathit{neutral} \qquad T_2 \Downarrow T_2^n}{T_1\ T_2 \Downarrow T_1^n\ T_2^n}\ \textsc{N-App}$$

$$\frac{T_1 \Downarrow T_1^n \qquad T_2 \Downarrow T_2^n}{T_1 \to T_2 \Downarrow T_1^n \to T_2^n}\ \textsc{N-Fun} \qquad \frac{T \Downarrow T^n}{\forall X :: K.T \Downarrow \forall X :: K.T^n}\ \textsc{N-Forall}$$

$$\frac{T \Downarrow T^n}{\lambda X :: K.T \Downarrow \lambda X :: K.T^n}\ \textsc{N-Lam} \qquad \frac{}{X \Downarrow X}\ \textsc{N-Var}$$

$$\frac{F \Downarrow F^n \qquad T \Downarrow T^n}{\texttt{ifix}\ F\ T \Downarrow \texttt{ifix}\ F^n\ T^n}\ \textsc{N-IFix} \qquad \frac{}{\mathbb{U} \Downarrow \mathbb{U}}\ \textsc{N-Builtin}$$

**Figure 11.** Type normalisation

Received 2025-06-13; accepted 2025-07-21

$\boxed{\Delta; \Gamma \vdash t : \tau}$

$$\frac{\Gamma(x) = T \qquad T \Downarrow T^n}{\Delta; \Gamma \vdash x : T^n}\ \text{T-Var} \qquad \frac{\Delta \vdash T_1 :: * \qquad T_1 \Downarrow T_1^n \qquad \Delta; (\Gamma, x : T_1^n) \vdash t : T_2^n}{\Delta; \Gamma \vdash (\lambda x : T_1.\ t) : T_1^n \to T_2^n}\ \text{T-LamAbs}$$

$$\frac{\Delta; \Gamma \vdash t_1 : T_1^n \to T_2^n \qquad \Delta; \Gamma \vdash t_2 : T_1^n}{\Delta; \Gamma \vdash t_1\ t_2 : T_2^n}\ \text{T-Apply} \qquad \frac{(\Delta, X :: K); \Gamma \vdash t : T^n}{\Delta; \Gamma \vdash (\Lambda X :: K.t) : (\forall X :: K.T^n)}\ \text{T-TyAbs}$$

$$\frac{\Delta; \Gamma \vdash t_1 : (\forall X :: K_2.T_1^n) \qquad \Delta \vdash T_2 :: K_2 \qquad T_2 \Downarrow T_2^n \qquad \left[T_2^n/X\right] T_1^n \Downarrow T^n}{\Delta; \Gamma \vdash t_1\ \{T_2\} : T^n}\ \text{T-TyInst}$$

$$\frac{\begin{array}{ccc}\Delta \vdash T :: K & \Delta \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *) & (F^n\ (\lambda X :: K.\texttt{ifix}\ F^n\ X))\ T^n \Downarrow T_0^n \\ T \Downarrow T^n & F \Downarrow F^n & \Delta; \Gamma \vdash M : T_0^n\end{array}}{\Delta; \Gamma \vdash \texttt{iwrap}\ F\ T\ M : \texttt{ifix}\ F^n\ T^n}\ \text{T-IWrap}$$

$$\frac{\Delta; \Gamma \vdash M : \texttt{ifix}\ F^n\ T^n \qquad \Delta \vdash T^n :: K \qquad (F^n\ (\lambda X :: K.\texttt{ifix}\ F^n\ X))\ T^n \Downarrow T_0^n}{\Delta; \Gamma \vdash \texttt{unwrap}\ M : T_0^n}\ \text{T-Unwrap}$$

$$\frac{c \text{ has type } \mathbb{U}}{\Delta; \Gamma \vdash \texttt{constant}\ \mathbb{U}\ c : \mathbb{U}}\ \text{T-Constant} \qquad \frac{\begin{array}{c}\texttt{builtin}\ \mathbb{F} \text{ has type } T \\ T \Downarrow T^n\end{array}}{\Delta; \Gamma \vdash \texttt{builtin}\ \mathbb{F} : T^n}\ \text{T-Builtin} \qquad \frac{\begin{array}{c}\Delta \vdash T :: * \\ T \Downarrow T^n\end{array}}{\Delta; \Gamma \vdash \texttt{error}\ S : T^n}\ \text{T-Error}$$

$$\frac{\begin{array}{cccc}\Delta' = \Delta, \overline{\text{binds}_\Delta(b)} & \overline{\text{binds}_\Gamma(b) \Downarrow \text{binds}_\Gamma(b)^n} & \Delta; \Gamma \vdash_{\texttt{nonrec}} \overline{b} & \\ & \Gamma' = \Gamma, \overline{\text{binds}_\Gamma(b)^n} & \Delta'; \Gamma' \vdash t : T^n & \Delta \vdash T^n :: *\end{array}}{\Delta; \Gamma \vdash \texttt{let}\ \overline{b}\ \texttt{in}\ t : T^n}\ \text{T-Let}$$

$$\frac{\begin{array}{cccc}\Delta' = \Delta, \overline{\text{binds}_\Delta(b)} & \overline{\text{binds}_\Gamma(b) \Downarrow \text{binds}_\Gamma(b)^n} & \Delta'; \Gamma' \vdash_{\texttt{rec}} \overline{b} & \\ & \Gamma' = \Gamma, \overline{\text{binds}_\Gamma(b)^n} & \Delta'; \Gamma' \vdash t : T^n & \Delta \vdash T^n :: *\end{array}}{\Delta; \Gamma \vdash \texttt{let rec}\ \overline{b}\ \texttt{in}\ t : T^n}\ \text{T-LetRec}$$

**Figure 12.** Typing of terms

$\boxed{\Delta \vdash_{\mathsf{c}} c : \tau}$

$$\frac{c = x\ (\overline{S \to U}) \qquad U \text{ is result type} \qquad \overline{\underline{\Delta} \vdash S :: *}}{\Delta \vdash_{\mathsf{c}} c : U}\ \text{W-Con}$$

$\boxed{\Delta; \Gamma \vdash_{\mathsf{b}} b}$

$$\frac{\Delta \vdash T :: * \qquad T \Downarrow T^n \qquad \Delta; \Gamma \vdash t : T^n}{\Delta; \Gamma \vdash_{\mathsf{b}} [\sim]x : T = t}\ \text{W-Term}$$

$$\frac{\Delta \vdash T :: K}{\Delta; \Gamma \vdash_{\mathsf{b}} X :: K = T}\ \text{W-Type}$$

$$\frac{\Delta' = \Delta, \overline{Y :: J} \qquad \overline{\underline{\Delta'} \vdash_{\mathsf{c}} c : \mathrm{constrLastTy}(\underline{d})}}{\Delta; \Gamma \vdash_{\mathsf{b}} \mathtt{data}\ (X :: K)\ (\overline{Y :: J}) = \overline{c}\ \mathtt{with}\ x}\ \text{W-Data}$$

$\boxed{\Delta; \Gamma \vdash_{\mathsf{nonrec}} \overline{b}}$

$$\frac{\overline{b} = b, \overline{b'} \qquad \mathrm{binds}_\Gamma(b) \Downarrow \mathrm{binds}_\Gamma(b)^n \qquad \Delta; \Gamma \vdash_{\mathsf{b}} b \qquad (\Delta, \mathrm{binds}_\Delta(b)); (\Gamma, \mathrm{binds}_\Gamma(b)^n) \vdash_{\mathsf{nonrec}} \overline{b'}}{\Delta; \Gamma \vdash_{\mathsf{nonrec}} \overline{b}}\ \text{W-BindingsNonRec}$$

$\boxed{\Delta; \Gamma \vdash_{\mathsf{rec}} \overline{b}}$

$$\frac{\overline{b} = b, \overline{b'} \qquad \Delta; \Gamma \vdash_{\mathsf{b}} b \qquad \Delta; \Gamma \vdash_{\mathsf{rec}} \overline{b'}}{\Delta; \Gamma \vdash_{\mathsf{rec}} \overline{b}}\ \text{W-BindingsRec}$$

**Figure 13.** Well-formedness of constructors and bindings

$\boxed{t \in value}$

$$\frac{}{\lambda x : T.\ t \in value}\ \text{V-LamAbs} \qquad \frac{}{\Lambda X :: K.t \in value}\ \text{V-TyAbs}$$

$$\frac{v \in value}{\mathtt{iwrap}\ F\ T\ v \in value}\ \text{V-IWrap} \qquad \frac{}{\mathtt{constant}\ \mathbb{U}\ c \in value}\ \text{V-Constant}$$

$$\frac{\langle 0, nv\rangle \in neutral}{nv \in value}\ \text{V-Neutral}$$

$\boxed{\langle n, t\rangle \in neutral}$

$$\frac{n < arity(\mathtt{builtin}\ \mathbb{F})}{\langle n, \mathtt{builtin}\ \mathbb{F}\rangle \in neutral}\ \text{NV-Builtin} \qquad \frac{v \in value \quad \neg isError(v) \quad \langle \mathtt{succ}\ n, nv\rangle \in neutral}{\langle n, nv\ v\rangle \in neutral}\ \text{NV-Apply}$$

$$\frac{\langle \mathtt{succ}\ n, nv\rangle \in neutral}{\langle n, nv\ \{T\}\rangle \in neutral}\ \text{NV-TyInst}$$

**Figure 14.** Values and neutral terms

$\boxed{t \Downarrow v}$

$$\frac{}{\lambda x : T.\ t \Downarrow \lambda x : T.\ t}\ \text{E-LamAbs} \qquad \frac{t_1 \Downarrow \lambda x : T.\ t_0 \quad t_2 \Downarrow v_2 \quad \neg isError(v_2) \quad [v_2/x]\ t_0 \Downarrow v_0}{t_1\ t_2 \Downarrow v_0}\ \text{E-Apply}$$

$$\frac{}{\Lambda X :: K.t \Downarrow \Lambda X :: K.t}\ \text{E-TyAbs} \qquad \frac{t_1 \Downarrow \Lambda X :: K.t_0 \quad [T_2/X]\ t_0 \Downarrow v_0}{t_1\ \{T_2\} \Downarrow v_0}\ \text{E-TyInst}$$

$$\frac{t_0 \Downarrow v_0 \quad \neg isError(v_0)}{\mathtt{iwrap}\ F\ T\ t_0 \Downarrow \mathtt{iwrap}\ F\ T\ v_0}\ \text{E-IWrap} \qquad \frac{t_0 \Downarrow \mathtt{iwrap}\ F\ T\ v_0}{\mathtt{unwrap}\ t_0 \Downarrow v_0}\ \text{E-Unwrap}$$

$$\frac{}{\mathtt{constant}\ \mathbb{U}\ c \Downarrow \mathtt{constant}\ \mathbb{U}\ c}\ \text{E-Constant}$$

$$\frac{\mathtt{let}\ \overline{b}\ \mathtt{in}\ t \Downarrow_{\mathtt{nonrec}} v}{\mathtt{let}\ \overline{b}\ \mathtt{in}\ t \Downarrow v}\ \text{E-Let} \qquad \frac{\overline{b} \vdash \mathtt{let\ rec}\ \overline{b}\ \mathtt{in}\ t \Downarrow_{\mathtt{rec}} v}{\mathtt{let\ rec}\ \overline{b}\ \mathtt{in}\ t \Downarrow v}\ \text{E-LetRec}$$

**Figure 15.** Big-step operational semantics (basic terms)

$$\frac{}{\mathtt{error}\ T \Downarrow \mathtt{error}\ T}\ \text{E-Error} \qquad \frac{t_1 \Downarrow \mathtt{error}\ T}{t_1\ t_2 \Downarrow \mathtt{error}\ T}\ \text{E-Error-Apply1}$$

$$\frac{t_2 \Downarrow \mathtt{error}\ T}{t_1\ t_2 \Downarrow \mathtt{error}\ T}\ \text{E-Error-Apply2} \qquad \frac{t_1 \Downarrow \mathtt{error}\ T}{t_1\ \{T_2\} \Downarrow \mathtt{error}\ T}\ \text{E-Error-TyInst}$$

$$\frac{t_0 \Downarrow \mathtt{error}\ T'}{\mathtt{iwrap}\ F\ T\ t_0 \Downarrow \mathtt{error}\ T'}\ \text{E-Error-IWrap} \qquad \frac{t_0 \Downarrow \mathtt{error}\ T}{\mathtt{unwrap}\ t_0 \Downarrow \mathtt{error}\ T}\ \text{E-Error-Unwrap}$$

$$\frac{t_1 \Downarrow \mathtt{error}\ T'}{\mathtt{let}\ ((x : T = t_1), \overline{b})\ \mathtt{in}\ t_0 \Downarrow \mathtt{error}\ T'}\ \text{E-Error-Let-TermBind}$$

**Figure 16.** Big-step operational semantics (errors)

$$\frac{}{\texttt{builtin}\ \mathbb{F} \Downarrow \texttt{builtin}\ \mathbb{F}}\ \textsc{E-NeutralBuiltin} \qquad \frac{\langle 0, nv\ v\rangle \in neutral}{nv\ v \Downarrow nv\ v}\ \textsc{E-NeutralApply} \qquad \frac{\langle 0, nv\ \{T\}\rangle \in neutral}{nv\ \{T\} \Downarrow nv\ \{T\}}\ \textsc{E-NeutralTyInst}$$

$$\frac{\neg(\langle 0, t_1\ t_2\rangle \in neutral) \qquad t_1 \Downarrow nv_1 \qquad \langle 0, nv_1\rangle \in neutral \qquad t_2 \Downarrow v_2 \qquad \neg isError(v_2) \qquad nv_1\ v_2 \Downarrow v_0}{t_1\ t_2 \Downarrow v_0}\ \textsc{E-NeutralApplyPartial}$$

$$\frac{\neg(\langle 0, t_1\ \{T\}\rangle \in neutral) \qquad t_1 \Downarrow nv_1 \qquad \langle 0, nv_1\rangle \in neutral \qquad nv_1\ \{T\} \Downarrow v_0}{t_1\ \{T\} \Downarrow v_0}\ \textsc{E-NeutralTyInstPartial}$$

$$\frac{isFullyApplied(nv_1\ v_2) \qquad compute(nv_1\ v_2) = v}{nv_1\ v_2 \Downarrow v}\ \textsc{E-NeutralApplyFull}$$

$$\frac{isFullyApplied(nv_1\ \{T\}) \qquad compute(nv_1\ \{T\}) = v}{nv_1\ \{T\} \Downarrow v}\ \textsc{E-NeutralTyInstFull}$$

**Figure 17.** Big-step operational semantics (built-in functions)

$\boxed{t \Downarrow_{\texttt{nonrec}} v}$

$$\frac{t_1 \Downarrow v_1 \qquad \neg isError(v_1) \qquad [v_1/x]\ (\texttt{let}\ \overline{b}\ \texttt{in}\ t_0) \Downarrow v_2}{\texttt{let}\ ((x : T = t_1), \overline{b})\ \texttt{in}\ t_0 \Downarrow v_2}\ \textsc{E-Let-NonRec-TermBind}$$

$$\frac{[T/X]\ (\texttt{let}\ \overline{b}\ \texttt{in}\ t_0) \Downarrow_{\texttt{nonrec}} v_1}{\texttt{let}\ ((X :: K = T), \overline{b})\ \texttt{in}\ t_0 \Downarrow_{\texttt{nonrec}} v_1}\ \textsc{E-Let-TypeBind}$$

$$\frac{t_0 \Downarrow v_0}{\texttt{let}\ \varnothing\ \texttt{in}\ t_0 \Downarrow v_0}\ \textsc{E-Let-Nil}$$

$\boxed{t \Downarrow_{\texttt{rec}} v}$

$$\frac{\overline{b_0} \vdash (\texttt{let rec}\ \overline{b}\ \texttt{in}\ \left[(\texttt{let rec}\ \overline{b_0}\ \texttt{in}\ t_1)/x\right] t_0) \Downarrow_{\texttt{rec}} v_1}{\overline{b_0} \vdash \texttt{let rec}\ ((\sim x : T = t_1), \overline{b})\ \texttt{in}\ t_0 \Downarrow_{\texttt{rec}} v_1}\ \textsc{E-LetRec-TermBind}$$

$$\frac{t_0 \Downarrow v_0}{\overline{b_0} \vdash \texttt{let rec}\ \varnothing\ \texttt{in}\ t_0 \Downarrow_{\texttt{rec}} v_0}\ \textsc{E-LetRec-Nil}$$

**Figure 18.** Big-step operational semantics (`let`-bindings)