






Translation Certification for Smart Contracts

Jacco O.G. Krijnen¹(✉) , Manuel M. T. Chakravarty², Gabriele Keller¹ ,
and Wouter Swierstra¹ 

¹ Utrecht University, Utrecht, The Netherlands
{j.o.g.krijnen,g.k.keller,w.s.swierstra}@uu.nl
² IOHK, Singapore, Singapore
manuel.chakravarty@iohk.io

Abstract. Compiler correctness is an old problem, but with the emergence of *smart contracts* on blockchains that problem presents itself in a new light. Smart contracts are self-contained pieces of software that control (valuable) assets in an adversarial environment; once committed to the blockchain, these smart contracts cannot be modified. Smart contracts are typically developed in a high-level contract language and compiled to low-level virtual machine code before being committed to the blockchain. For a smart contract user to trust a given piece of low-level code on the blockchain, they must convince themselves that (a) they are in possession of the matching source code and (b) that the compiler has correctly translated the source code to the given low-level code.

Classic approaches to compiler correctness tackle the second point. We argue that *translation certification* also squarely addresses the first. We describe the proof architecture of a novel translation certification framework, implemented in Coq, for a functional smart contract language. We demonstrate that we can model the compilation pipeline as a sequence of translation relations that facilitate a modular verification methodology and are robust in the face of an evolving compiler implementation.

1 Introduction

Compiler correctness is an old problem that has received renewed interest in the context of *smart contracts*—that is, compiled code on public blockchains, such as Ethereum or Cardano. This code often controls a significant amount of financial assets, must operate under adversarial conditions, and can no longer be updated once it has been committed to the blockchain. Bugs in smart contracts are a significant problem in practice [5]. Recent work has also established that smart contract language compilers can exacerbate this problem [26, Section 3] (in this case, the Vyper compiler). More specifically, the authors report (a) that they did find bugs in the Vyper compiler that compromised smart contract security and (b) that they performed verification on generated low-level code, because they were wary of compiler bugs.

Hence, to support reasoning about smart contract source code, we need to get a handle on the correctness of smart contract compilers. On top of that, we

do also need a *verifiable link* between the source code and its compiled code to prevent *code substitution attacks*, where an adversary presents the user with source code that doesn't match the low-level code committed on-chain.

In this paper, we are reporting on our ongoing effort to develop a certification engine for the open-source on-chain code compiler of the Plutus smart contract system¹ for the Cardano blockchain.² Specifically, we make the following contributions:

- We describe a novel architecture for a translation certifier based on *translation relations*, which enables us to generate *translation certificates*—proof objects that relate the source code to the resulting compiled code and establish the correctness of the translation (Sect. 2).
- We provide formal definitions for the transformation passes that step-by-step translate PIR (Plutus Intermediate Representation) to PLC (Plutus Core) and briefly discuss the challenges associated with the certification of each of these passes (Sect. 3).
- We present a summary of existing approaches to compiler correctness and discuss the importance of generating translation certificates in the domain of smart contracts (Sect. 4).

We also evaluate how our approach to gradual certification copes with changes to the compiler, which is being developed in an independent open source project. Finally, we discuss related work in Sect. 5 and future work in Sect. 6.

2 The Architecture of the Certifier

On-chain code in the Plutus smart contract system is written in a subset of Haskell called *Plutus Tx* [18]. The Plutus Tx compiler is implemented as a plugin for the widely-used, industrial-strength GHC Haskell compiler, combining large parts of the GHC's compilation pipeline with custom translation steps to generate Plutus Core. In this context, it seems infeasible to apply full-scale compiler verification à la CompCert [21]. We will therefore outline the design of a certification engine that, using the Coq proof assistant [6, 9], generates a proof object, a *translation certificate*, asserting the validity of a Plutus Core program with respect to a given Plutus Tx source contract. In addition to asserting the correct translation of *this one program*, the translation certificate serves as a verifiable link between source and generated code.

We model the compiler as a composition of pure functions that transform one abstract syntax tree into another. Figure 1 illustrates the architecture for a single transformation, where the grey area marks the compiler implementation as a function $f_i : \text{AST}_i \rightarrow \text{AST}_{i+1}$. We use a family of types AST_i to illustrate that the representation of the abstract syntax might change after each transformation.

¹ <https://developers.cardano.org/docs/smart-contracts/plutus/>.

² <http://cardano.org> is, at the time of writing, the 5th largest public blockchain by market capitalisation.

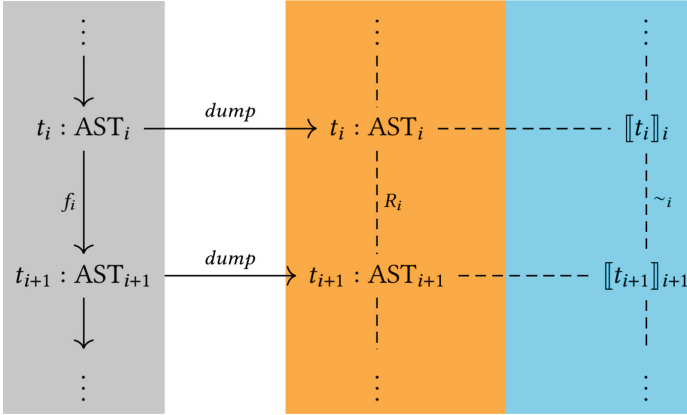


Fig. 1. Architecture for a single compiler pass. The grey area (left) represents the compiler, orange (center) and blue (right) represent the certification component in Coq. (Color figure online)

To support certification, the compiler outputs each intermediate tree t_i , so that we can parse these in our Coq implementation of the certifier. Within Coq, we define a high-level specification of each pass. We call this specification a *translation relation*: a binary relation on abstract syntax trees that specifies the intended behaviour of the compiler pass. The orange area in Fig. 1 displays the translation relation R_i of pass i , where the vertical dashed line indicates that $R_i(t_i, t_{i+1})$ holds. To establish this, we define a search procedure that, given two subsequent trees produced by the compiler, can construct a derivation relating the two.

The translation relation is purely syntactic—it does not assert anything about the correctness of the compiler—but rather *specifies* the behaviour of a particular compiler pass. To verify that the compilation preserves language semantics requires an additional proof, the blue area in Fig. 1, that establishes that any two terms related by R_i have the same semantics.

We have implemented this approach for a range of concrete passes of the Plutus Tx compiler. To illustrate our approach in this section, we will use an untyped lambda calculus, extended with non-recursive let-bindings.

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

In the following section, we will extend this to a lambda calculus that is closer to the intermediate language used by the Plutus Tx compiler.

2.1 Characterising a Transformation

To assert the correctness of a single compiler stage f_i , we begin by defining a translation relation R_i on a pair of source and target terms t_i and t_{i+1} ,

$$\begin{array}{c}
\frac{\Gamma(x) = t' \quad \Gamma \vdash t' \triangleright t}{\Gamma \vdash x \triangleright t} \text{ [Inline-Var}_1\text{]} \\
\frac{}{\Gamma \vdash x \triangleright x} \text{ [Inline-Var}_2\text{]} \\
\frac{\Gamma \vdash t_1 \triangleright t'_1 \quad (x, t_1), \Gamma \vdash t_2 \triangleright t'_2}{\Gamma \vdash \mathbf{let } x = t_1 \mathbf{ in } t_2 \triangleright \mathbf{let } x = t'_1 \mathbf{ in } t'_2} \text{ [Inline-Let]} \\
\frac{\Gamma \vdash t_1 \triangleright t'_1 \quad \Gamma \vdash t_2 \triangleright t'_2}{\Gamma \vdash t_1 t_2 \triangleright t'_1 t'_2} \text{ [Inline-App]} \\
\frac{\Gamma \vdash t_1 \triangleright t'_1}{\Gamma \vdash \lambda x. t_1 \triangleright \lambda x. t'_1} \text{ [Inline-Lam]}
\end{array}$$

Fig. 2. Characterisation of an inliner

respectively. This relation characterises the admissible translations of that compiler stage. That is, for all t_i, t_{i+1} , we have $f_i(t_i) = t_{i+1}$ implies $R_i(t_i, t_{i+1})$.

As a concrete example, consider an inlining pass. We have characterised this as an inductively defined relation in Fig. 2. Here, $\Gamma \vdash s \triangleright t$ asserts that program s can be translated into t given an environment Γ of let-bound variables, paired with their definition. According to Rule [Inline-Var₁] the variable x may be replaced by t when the pair (x, t') can be looked up in Γ and t' can be translated to t , accounting for repeated inlining. The remaining rules are congruence rules, where Rule [Inline-Let] also extends the environment Γ . We omitted details about handling variable capture to keep the presentation simple: hence, we assume that variable names are globally unique.

Crucially, these rules do *not* prescribe which variable occurrences should be inlined, since the [Inline-Var₁] and [Inline-Var₂] rules overlap. The choice in the implementation of the pass may rely on a complex set of heuristics internal to the compiler. Instead, we merely define a relation capturing the *possible* ways in which the compiler *may* behave. This allows for a certification engine that is robust with respect to changes in the compiler, such as the particular heuristics used to decide when to replace a variable with its definition or not.

We can then encode the relation $\cdot \vdash \triangleright \cdot$ in Coq as an inductive type `Inline`, which is indexed by an environment and two ASTs, as shown in Fig. 3. This type corresponds closely to the rules of Fig. 2: we define exactly one constructor per rule. However, there are some small differences. Since we cannot omit details about variable capture anymore, we choose a de Bruijn representation for variables and implement the environment Γ as a cons-list. In the `Inline_Let` constructor, we extend the list with the bound term and furthermore shift free variables in the other bound terms. For a let-bound variable n , its corresponding bound term can then be found at the n 'th position in the list using Coq's `nth_error` list-indexing function. For this indexing to work properly, the

```

Inductive binding :=
| LetBound      : term -> binding
| LambdaBound  : binding.

Inductive Inline : list binding -> term -> term -> Type :=
| Inline_Var_1 : forall {env n t},
  nth_error env n = Some (LetBound t) ->
  Inline env (Var n) t

| Inline_Var_2 : forall {env n},
  Inline env (Var n) (Var n)

| Inline_Let : forall {env s t s' t'},
  Inline env s s' ->
  Inline (LetBound s :: shiftEnv env) t t' ->
  Inline env (Let s t) (Let s' t')

| Inline_Lam : forall {env s t},
  Inline (LambdaBound :: shiftEnv env) s t ->
  Inline env (Lam s) (Lam t)

| Inline_App : forall {env s sx t tx},
  Inline env s t -> Inline env sx tx ->
  Inline env (App s sx) (App t tx)

```

Fig. 3. Characterisation of an inliner in Coq

environment also has to be extended at every lambda, as seen in `Inline_Lam`. We distinguish the two types of binding sites with the type `binding`.

These inductive types implement the translation relation: its inhabitants are proof derivations which will be a key ingredient of a compilation certificate.

2.2 Proof Search

After defining a translation relation R_i characterising one compiler stage, we now define a decision procedure to construct a proof that for two particular terms t_i and t_{i+1} , produced by a run of the compiler, the relation $R_i(t_i, t_{i+1})$ holds. To find and implement such a search procedure, we generally follow these steps:

1. We write proofs for specific compilations by hand using Coq's *tactics*, a form of metaprogramming. For simple relations, like the inline example sketched above, a proof can often be found with a handful of tactics such as `auto` or `constructor`. This is particularly useful for debugging the design of our relations describing compiler passes. The drawback of this approach is, however, that it is difficult to reason when such proof search may fail. Furthermore, proofs written using such tactics quickly become slow for large terms.

2. Once we are sufficiently confident that a relation accurately captures admissible compiler behaviour, we write a decision procedure of the form `forall (t1 t2 : term), option (R t1 t2)`. These procedures can still produce large proof terms and may not always successfully construct a proof, but they form a useful intermediate step towards full-on proof by reflection.
3. Finally, we write a boolean decision procedure in the style of `ssreflect` [17] of type `term -> term -> bool`, together with a soundness proof stating that it will only return `true` when two terms are related through R_i . Verifying such boolean functions for complex compilation passes is non-trivial; hence, we only invest the effort once we have a reasonable degree of confidence that the relation we have defined accurately describes a given compiler pass.

2.3 Semantics Preservation

Given the relational specification of each individual compiler pass, we can now establish the correctness properties for each pass. In the simplest case, this could be asserting the preservation of a program’s static semantics, i.e., a proof of type preservation. On the other end of the spectrum, we can demonstrate that the translated term is semantically equivalent to the original program. Proving such properties for PIR and Plutus Core passes, however, requires advanced techniques such as step-indexed logical relations [2], which go beyond the scope of the current paper.

In Fig. 1, we denote R_i ’s correctness properties in the blue area by means of an abstract binary relation \sim_i on the semantic objects $\llbracket t_i \rrbracket_i$ of ASTs t_i . In the case of static semantics, we can choose typing derivations as semantic objects, and (for most passes) relate these by simply comparing types syntactically.

We can construct these proofs independently and gradually for each step in the translation. In fact, even without any formal proof about the semantics, inspection of the (relatively concise) definition of a translation relation may already provide some degree of confidence that the translation step was performed correctly. After all, the translation relation asserts the specification of this compiler pass’ admissible behaviour.

2.4 Certificate Generation

A complete translation certificate includes at least the entire set of ASTs t_1, \dots, t_n together with a proof term witnessing the translation relations of type $R_1(t_1, t_2) \wedge \dots \wedge R_{n-1}(t_{n-1}, t_n)$. In addition, any semantic preservation results on translation relations can be instantiated and included as a proof of $\llbracket t_i \rrbracket_i \sim_i \llbracket t_{i+1} \rrbracket_{i+1}$.

Together with the source and compiled program, one can now independently check the certificate using a trusted proof checker, such as the Coq kernel [9]. The definitions of the abstract syntax, translation relations and semantic preservation can be inspected to confirm that the certificate proves the right theorem. One can then be confident that the compiled program is a faithful translation of the source code.

$t ::= x \mid \lambda(x : \tau). t \mid t t$	variable, lambda, function application
$\mid \Lambda(\alpha : \kappa). t \mid t \{ \tau \}$	type abstraction, type application
$\mid \text{let}_r^s x = t \text{ in } t$	term bindings
$\mid \text{data } T \overline{\alpha} = \overline{C_i} \overline{\tau_i} \text{ with } x \text{ in } t$	datatype binding
$r ::= \text{rec} \mid \text{nonrec}$	recursion type of binding
$s ::= \text{strict} \mid \text{nonstrict}$	strictness of binding
$\tau ::= \dots$	types

Fig. 4. Simplified PIR

3 Translation Relations of the Plutus Tx Compiler

The Plutus Tx compiler translates Plutus Tx (a subset of Haskell) to Plutus Core, a variant of System F_ω^μ [13]. The Plutus Core code is committed to the Cardano blockchain, constituting the definitive reference to any deployed smart contract.

Plutus Core programs are pure, self-contained functions (i.e., they do not link to other code) and are passed a representation of the transaction whose validation they contribute to. The programs are run by an interpreter during the transaction validation phase of the blockchain.

The Plutus Tx compiler reuses parts of the GHC infrastructure and implements its custom passes by installing a core-to-core pass plugin [15] in the GHC compiler pipeline. On a high level, the compiler comprises three steps:

1. The parsing, type-checking and desugaring phases of GHC are reused to translate a surface-level Haskell program into a GHC Core program.
2. A large subset of GHC Core is directly translated into an intermediate language named Plutus Intermediate Representation (PIR). These languages are similar and both based on System F, with some extensions. Additionally, all referred definitions are included as local definitions so that the program is self-contained.
3. The PIR program is then transformed and compiled down into Plutus Core.

The certification effort reported here focuses on Step 3, which consists of several optimisation passes and translation steps. PIR is a superset of the Plutus Core language: it adds several conveniences, such as user-defined datatypes, strict and non-strict let-bindings that may be (mutually) recursive. The compilation steps translate these constructs into simpler language constructs.

In Fig. 4 we present a simplified version of the PIR syntax, where we omit some constructs for the sake of presentation. The full PIR language specification has been formalised elsewhere [13, 19]. In particular, we ignore the fact that in PIR, let-bindings may contain a group of (mutually recursive) bindings. Similarly, we do not include mutually-recursive datatypes. Furthermore, we omit the syntax of types, and the term-level witnesses of iso-recursive types. We occasionally omit type annotations, when they are not relevant.

We introduce the individual compiler passes that the Plutus Tx compiler performs using the following Haskell program to illustrate their behaviour:

```
-- | Either a specific end date, or "never".
data EndDate = Fixed Integer | Never

pastEnd :: EndDate -> Integer -> Bool
pastEnd end current =
  let inlineMe = False
  in case end of
    Fixed n -> (let floatMe = if current `greaterThanEqInteger` 0
                        then n else 0 in floatMe) `lessThanEqInteger` current
    Never   -> inlineMe
```

This program is a basic implementation of a *timelock*, a contract that states that funds may be moved after a certain date, or not at all. It contains a few contrived bindings (`inlineMe` and `floatMe`) that will be useful to illustrate some transformations. After the program is desugared to GHC Core, it is converted to a term in PIR that corresponds to the following Simplified PIR term:

```
data Bool = True | False with Bool_match in
data Unit = Unit with Unit_match in
let nonrec strict lessThanEqInteger = ... in
  data EndDate = Fixed Integer | Never with EndDate_match in
    \ (end : EndDate) .
    \ (current : Integer) .
      let nonrec nonstrict inlineMe = False in
        EndDate_match end
          (\unit n -> lessThanEqInteger
            (let nonrect nonstrict floatMe =
              Bool_match (greaterThanEqInteger current 0)
                (\unit -> n) (\unit -> 0)
              Unit
            in floatMe)
            current)
          (\unit -> inlineMe)
        Unit
```

Note that case distinction of a type `T` is encoded as the application of a pattern match function `T_match`, which is introduced as part of a data definition. Furthermore, branches of a case distinction are delayed by abstracting over a unit value, since PIR is a strict language.

Next we will discuss the compiler passes, we have included each intermediate form of the above program with some commentary in the appendix which can be found online³.

³ <https://arxiv.org/abs/2201.04919>.

3.1 Variable Renaming

In the renaming pass, the compiler transforms a program into an α -equivalent program, such that all variable names are globally unique, a property also known as the *Barendregt-convention*. The implementation of some subsequent compiler passes depend on it. We can express variable renaming as a translation relation $\Delta \vdash t \triangleright_\alpha t'$, stating that under the renaming environment Δ (consisting of pairs of variables), t is renamed to t' . The environment Δ records all variables that are free in t , paired with their corresponding name in t' .

The case for lambda abstractions is defined as follows:

$$\frac{(x, y), \Delta \vdash t \triangleright_\alpha t' \quad \{z \mid (z, y) \in \Delta\} \cap FV(t) = \emptyset}{\Delta \vdash \lambda x.t \triangleright_\alpha \lambda y.t'} \text{ [Rename-Abs]}$$

The [Rename-Abs] rule states that a lambda-bound variable x may be renamed at its binding-site to y , when t and t' are related under the extended environment. Of course, x may equal y , indicating that no renaming was performed. Additionally, the new binder y should not capture any other free variable z in t that was also renamed to y . Very similar rules can be stated for other binding constructs such as `let`.

Note that this relation does not establish global uniqueness of variables: we consider that an implementation detail internal to the compiler. If this property would be required or convenient in semantic preservation proofs, we will establish it separately, allowing this renaming relation to be as general as possible.

The variable case simply follows from the environment Δ :

$$\frac{(x, y) \in \Delta}{\Delta \vdash x \triangleright_\alpha y} \text{ [Rename-Var]}$$

3.2 Inlining

The rules of the translation relation for inlining in PIR are similar to those in Sect. 2.1. However, the Plutus Tx compiler does more than just inlining let-bound definitions. It also performs dead-code elimination (removing those let-bindings that have been inlined exhaustively) and it renames variables to ensure the global uniqueness of bound variables. This introduces a problem for our certification approach, as we cannot observe and dump the intermediate ASTs, since the transformations are fused into a single pass in the compiler.

We solve this by modeling the individual transformations, composing them using *relational composition*, $\exists t_2. R_1(t_1, t_2) \wedge R_2(t_2, t_3)$. To construct a proof relating two terms, then amounts to also finding the *intermediate term*, t_2 witnessing the composite transformation. To simplify the search of this intermediate AST, we adjust the compiler to emit supporting information about the performed pass; in this case, a list of the eliminated variables. If the compiler emits incorrect information, we may fail to construct a certificate, but we will never produce an incorrect certificate.

3.3 Let-Floating

During let-floating, let-bindings can be moved upwards in the program. This may save unnecessarily repeated computation and makes the generated code more readable. The Plutus Tx compiler constructs a dependency graph to maintain a correct ordering when multiple definitions are floated. For the translation relation, we first consider the interaction of a `let` expression with its parent node in the AST. For example, consider the case of a lambda with a non-strict `let` directly under it:

$$\frac{x \notin FV(t_1) \quad x \neq y \quad t_1 \triangleright_{let} t'_1 \quad t_2 \triangleright_{let} t'_2}{\lambda x. \mathbf{let}_r^{\text{nonstrict}} y = t_1 \mathbf{in} t_2 \triangleright_{let} \mathbf{let}_r^{\text{nonstrict}} y = t'_1 \mathbf{in} \lambda x. t'_2} \text{ [Float-Let-Lam]}$$

This rule states that a non-strict let-binding may float up past a lambda, if the bound term does not reference the lambda-bound variable. Furthermore, we require $x \neq y$, to avoid variable capture in t_2 . This rule does not apply to `strict` let-bindings, as floating them outside a lambda might change termination behaviour of the program. Similar rules express when a `let` may float upwards past the other language constructs. Most of these are much simpler, only binding constructs pose additional constraints on scoping and strictness. Since the compiler pass may float lets more than just one step up, we define the translation relation as the transitive closure of \triangleright_{let} . Note that we do not need to maintain a dependency graph in the certifier, but only need to assert that transformations do not break dependencies.

3.4 Dead-Code Elimination

By means of a live variable analysis, the compiler determines which let-bound definitions are unused. This is mainly useful for definitions that are introduced by other compiler passes. Since PIR is a strict language, however, the compiler can only eliminate those bindings for which it can determine they have no side-effects. For example, a let-bound expression that is unused but diverges cannot be removed, as that could change the termination behaviour of the program.

The analysis in the compiler is not as straightforward as counting occurrences. Even a let-bound variable that does occur in the code, may be dead-code, if it is only used in other dead bindings. This is also known as strongly live variable analysis [16]. We define a translation relation $t \triangleright_{dce} t'$ that captures dead code elimination. The crucial rule is for let-bindings.

$$\frac{t_2 \triangleright_{dce} t'_2 \quad x \notin FV(t'_2)}{\mathbf{let}_r^{\text{nonstrict}} x = t_1 \mathbf{in} t_2 \triangleright_{dce} t'_2} \text{ [DCE-Let-nonstrict]}$$

Note that the condition $x \notin FV(t'_2)$ mentions the *resulting* body of the let t'_2 . This is justified since the rules of \triangleright_{dce} can remove bindings only, but cannot change any other language constructs. This illustrates how succinct we can describe the specification of a complex compiler pass.

In practice, the Plutus Tx compiler also eliminates some strict bindings that obviously do not diverge, such as values.

3.5 Encoding of Non-strict Bindings

The PIR language allows both for strict and non-strict let-bindings, but Plutus Core does not. The *thunking transformation* is used to obtain semantic equivalent definitions which use a strict let-binding. We define the rules as a relation $\Gamma \vdash t \triangleright_{thunk} t'$, where Γ records for every bound variable whether it was bound strictly or non-strictly. The rule for a non-strict binding site is:

$$\frac{\Gamma \vdash t_1 \triangleright_{thunk} t'_1 \quad (x, \mathbf{nonstrict}), \Gamma \vdash t_2 \triangleright_{thunk} t'_2 \quad y \notin FV(t_1)}{\Gamma \vdash \mathbf{let}_{\mathbf{nonrec}}^{\mathbf{nonstrict}} x = t_1 \mathbf{in} t_2 \triangleright_{thunk} \mathbf{let}_{\mathbf{nonrec}}^{\mathbf{strict}} x = \lambda y. t'_1 \mathbf{in} t'_2} \text{ [Thunk-Let-nonstrict]}$$

This rule states that a right hand side is thunked by introducing a lambda abstraction that expects a trivial unit value y as its argument.

The rules for other variable binders extend Γ . The rule for a recursive let-binding also extends the environment under which t_1 is transformed. Finally, we also replace the occurrences of nonstrict variables, adding an application to the unit value, thereby forcing evaluation.

$$\frac{(x, \mathbf{nonstrict}) \in \Gamma}{\Gamma \vdash x \triangleright_{thunk} x ()} \text{ [Thunk-Var]}$$

3.6 Encoding of Recursive Bindings

The Plutus Tx compiler translates (mutually) recursive let-bindings in non-recursive ones using fixpoint combinators. Here we only consider the rule for individual recursive lets in simplified PIR:

$$\frac{t_1 \triangleright_{\mu} t'_1 \quad t_2 \triangleright_{\mu} t'_2 \quad y \notin FV(t_1)}{\mathbf{let}_{\mathbf{rec}}^s x = t_1 \mathbf{in} t_2 \triangleright_{\mu} \mathbf{let}_{\mathbf{nonrec}}^{\mathbf{strict}} fix = \dots \mathbf{in} \mathbf{let}_{\mathbf{nonrec}}^s x = fix (\lambda x. t'_1) \mathbf{in} t'_2} \text{ [EncRec-Let]}$$

This rule relates recursive bindings to non-recursive ones, and expects an explicit definition of the fixpoint operator as well. Since PIR has no primitive construct for term-level fix-points, the compiler generates a definition fix . Note that fix is defined in a non-recursive let, its construction relies on recursive types [19].

The actual transformation for PIR is much more involved, since mutually recursive binding groups require a more involved fixpoint combinator of which the definition depends on the size of the group.

3.7 Encoding of Datatypes

Datatype definitions are encoded using lambda and type abstractions according to the Scott encoding [1]. To show the idea of the rather general \triangleright_{data} translation relation, we show a rule specialised to the *Maybe* datatype.

$$\frac{t \triangleright_{data} t'}{\text{data } Maybe \alpha = Just \alpha \mid Nothing \text{ with } maybe \text{ in } t} \text{ [Scott-Maybe]}$$

$$(\wedge Maybe. \lambda Just. \lambda Nothing. \lambda maybe. t') \tau_{Maybe} t_{Just} t_{Nothing} t_{maybe}$$

The [Scott-Maybe] rule relates the datatype definition to a term that abstracts over the type *Maybe*, its constructors *Just* and *Nothing* and the matching function *maybe*, which are each lambda encoded. For the exact definitions of τ_{Maybe} , t_{Just} , $t_{Nothing}$ and t_{maybe} we refer to the general formalisation of PIR [19].

3.8 Encoding of Non-recursive Bindings

A non-recursive let-binding is simply compiled into a β redex:

$$\frac{t_1 \triangleright_{\beta} t'_1 \quad t_2 \triangleright_{\beta} t'_2}{\text{let } \overset{\text{strict}}{\text{nonrec}} x = t_1 \text{ in } t_2 \triangleright_{\beta} (\lambda x. t'_2) t'_1} \text{ [Redex-Let]}$$

Note that at this point in the compiler pipeline, $\text{let}_{\text{nonrec}}^{\text{strict}}$ is the only type of let-binding that can still occur.

4 Evaluation

In this section, we evaluate our approach to proof engineering for an independently developed, constantly evolving compiler under the application constraints imposed by smart contracts.

4.1 Compilers and Correctness

The standard approach to compiler correctness is *full compiler verification*: a proof that asserts that the compiler is correct as it demonstrates that, for any valid source program, the translation produces a semantically equivalent target program. Examples of this approach include the CompCert [21] and CakeML [20] projects, showing that (with significant effort) it is possible to verify a compiler end-to-end. To do so, the compiler is typically implemented in a language suitable for verification, such as the Coq proof assistant or the HOL theorem prover.

In contrast, the technique that we propose for the Plutus Tx compiler is based on *translation validation* [27]. Instead of asserting an entire compiler correct, translation validation establishes the correctness of individual compiler runs.

A statement of full compiler correctness is, of course, the stronger of the two statements. Translation validation may fail to assert the correctness of some compiler runs; either because the compiler did not produce correct code or because the translation certifier is incomplete. In exchange for being the weaker property, translation validation is potentially (1) less costly to realise, (2) easier to retrofit to an existing compiler, and (3) more robust in the face of changes to the compiler.

The idea of *proof-carrying code* [23] is closely related to translation validation, shifting the focus to compiled programs, rather than the compiler itself. A program is distributed together with a proof of a property such as memory or type safety. Such a proof excludes certain classes of bugs and gives direct evidence to the users of such a program, who may independently check the proof before running it. Our certification effort, while related, differs in that we keep proof and program separate and in that we are interested in full semantic correctness and not just certain properties like memory and type safety.

4.2 Certificates and Smart Contracts

Smart contracts often manage significant amounts of financial and other assets. Before a user engages with such a contract, which has been committed to the blockchain as compiled code, they may want to inspect the source code to assert that it behaves as they expect. In order to be able to rely on that inspection, they need to know without doubt that (1) they are looking at the correct source code and (2) that the source code has been compiled correctly.

While a verified smart contract compiler addresses the second point, it doesn't help with the first. An infrastructure of *reproducible builds*, on the other hand, solves only the first point. The latter is the approach taken by Etherscan⁴: to verify that a deployed Ethereum smart contract was the result of a compiler run, one provides the source code and build information such as the compiler version and optimisation settings.

In contrast, a *certifying compiler* [24] that generates an independently verifiable certificate of correct translation, squarely addresses both points. By verifying a smart contract's translation certificate, a smart contract user can convince themselves that they are in possession of the matching source code and that this was correctly compiled to the code committed to the blockchain.

4.3 Engineering Considerations

Gradual Verification. The certifier architecture outlined in this paper allows for a gradual approach to verification: during the development of the certification engine, each individual step in the process increases our overall confidence in the compiler's correctness, even if we have not yet completed the end-to-end semantic verification of the compiler pipeline.

⁴ <https://etherscan.io/verifyContract>.

By defining only the translation relations, we have an independent formal specification of the compiler’s behaviour. This makes it easier to reason informally and to spot potential mistakes or problems with the implementation.

Implementing the decision procedures for translation relations ties the implementation to the specification: we can show on a per-compilation basis that a pass is sound with respect to its specification as a translation relation. Furthermore, we can test and debug translation relations by automatically constructing evidence for various input programs.

Finally, by proving semantics preservation of a translation relation, we gain full confidence in the corresponding pass for compiler runs that abide by that translation relation.

Agility. The Plutus Tx compiler is developed independently of our certification effort. Moreover, it depends on large parts of a large code base—namely, that of the Glasgow Haskell Compiler (GHC). In addition, both GHC and the Plutus Tx-specific parts evolve on a constant basis; for example, to improve code optimisation or to fix bugs.

In that context, full verification appears an insurmountable task and a proof on the basis of the compiler source code would constantly have to adapt to the evolving compiler source. Hence, the architecture of our certification engine is based on a *grey box approach*, where the certifier matches the general outline (such as the phases of the compiler pipeline), but not all of the implementation details of the compiler. For example, our translation relation for the inliner admits any valid inlining. Improvements of the compiler heuristics to produce more efficient programs by being selective about what precisely to inline don’t affect the inliner’s translation relation, and hence, don’t affect the certifier.

Trusted Computing Base (TCB). The fact that the Plutus Tx compiler is not implemented in a proof assistant, but in Haskell complicates direct compiler verification. It might be possible to use a tool like `hs-to-coq` [29], which translates a subset of Haskell into Coq’s Gallina and has been used for proving various properties about Haskell code [11]. However, given that those tools often only cover language subsets, it is not clear that they are applicable. More importantly, such an approach would increase the size of the trusted computing base (TCB), as the translation from Haskell into Coq’s Gallina is not verified. Similarly, extraction-based approaches suffer from the same problem if the extraction itself is not verified, although there are projects like `CertiCoq` [3] that try to address that issue.

In any case, our architecture has a small TCB. We directly relate the source and target programs, taking the compiler implementation out of the equation. Trusting a translation certificate comes down to trusting the Coq kernel that checks the proof, the theorem with its supporting definitions and soundness of the Plutus Core interpreter with respect to the formalised semantics. Of course, these components are part of the TCB of a verified compiler too. This aspect

also motivated our choice of Coq over other languages such as Agda, due to its relatively small and mature kernel.

5 Related Work

Ethereum was the first blockchain to popularise use of smart contracts, written in the Solidity programming language. Solidity is an imperative programming language that is compiled to EVM bytecode, which runs on a stack machine operating on persistent mutable state. The DAO vulnerability [12] has underlined the importance of formal verification of smart contracts. Notably, a verification framework has been presented [10] for reasoning about embedded Solidity programs in F^* . The work includes a decompiler to convert EVM bytecode, generated by a compiler, into Solidity programs in F^* . The authors propose that correctness of compilation can be shown by proving equivalence of the embedded source and (decompiled) target program using relational reasoning [7]. However, this would involve a manual proof effort on a per-program basis, and relies on the F^* semantics since the embeddings are shallow. Furthermore, components such as the decompiler are not formally verified, adding to the size of the TCB.

The translation validation technique has been used for the verification of a particular critical Ethereum smart contract [26] using the K framework. The work demonstrates how translation validation can successfully be applied to construct proofs about the low-level EVM bytecode by mostly reasoning on the (much more understandable) source code. The actual refinement proof is still constructed manually, however.

The Tezos blockchain also uses a stack-like language, called Michelson. The Mi-Cho-Coq framework [8] formalises the language and supports reasoning with a weakest precondition logic. There is ongoing work for developing a certified compiler in Coq for the Albert intermediate language, intended as a target language for certified compilers of higher-level languages. This differs from our approach as it requires the compiler to be implemented in the proof assistant.

ConCert is a smart contract verification framework in Coq [4]. It enables formal reasoning about the source code of a smart contracts, defined in a different (functional) language. The programs are translated and shallowly embedded in Coq’s Gallina. Interestingly, the translation is proven sound, in contrast with approaches such as *hs-to-coq* [29], since it is implemented using Coq’s metaprogramming and reasoning facility *MetaCoq* [28].

The Cogent certifying compiler [25] has shown that it is possible to use translation validation for lowering the cost of functional verification of low-level code: a program can be written and reasoned about in a high-level functional language, which is compiled down to C. The generated certificate then proves a refinement relation, capable of transporting the verification results to the corresponding C code. The situation is different from ours: the Cogent compiler goes through a range of languages with different semantic models and uses the forward-simulation technique as a consequence. In contrast, we are working with variations of lambda calculi that have similar semantics, allowing us to use logical relations and translation relations.

In their Coq framework [22], Li and Appel use a similar technique for specifying compiler passes as inductive relations in Coq. Their tool reduces the effort of implementing program transformations and corresponding correctness proofs. The tool is able to generate large parts of an implementation together with a partial soundness proof with respect to those relations. The approach is used to implement parts of the CertiCoq backend.

6 Conclusions and Further Work

The Plutus Tx compiler translates a Haskell subset into Plutus Core. The compiler consists of three main parts: the first one reuses various stages of GHC to compile the Haskell subset to GHC Core—GHC’s principal intermediate language. The second part translates GHC Core to PIR and the final part compiles PIR to Plutus Core. As Plutus Core is strict and doesn’t directly support datatypes, these parts are quite complex. Moreover, they consist of a significant number of successive transformation steps.

In this paper, we focused on the certification effort covering the third part of that pipeline; specifically, the translation steps from PIR to Plutus Core. We developed translation relations for all passes described in Sect. 3, such that we can, for example, produce a proof relating the previously described timelock example in PIR to its final form in Plutus Core. For some of these passes, such as inlining, we have implemented a verified decision procedure, but most of the evidence is generated semi-automatically by using Coq tactics. We have not yet covered all transformations in their full generality; for example, we do not cover (mutually) recursive datatypes yet. We have also started the semantic verification of key passes of the translation [14] and are investigating different ways to improve the efficiency of proof search for larger programs.

Our next steps comprise the following: (1) filling in the remaining gaps in translation relations (such as covering mutually recursive datatypes); (2) complete all decision procedures; (3) drive the semantic verification forward; and (4) develop techniques to further automate our approach and improve the efficiency of the certifier.

The first three steps pose a significant amount of work, but we do not expect major new conceptual questions or obstacles. This is different for Step (4), where we anticipate the need for further research work. This includes more compositional definitions of the translation relations, such that we can generate at least part of the decision procedures (semi-)automatically. Moreover, we already perceive efficiency to be a bottleneck and we plan to work on optimising the proof search. Finally, we plan to apply our approach to the first part of the Plutus Tx compiler (Haskell subset to GHC Core).

References

1. Abadi, M., Cardelli, L., Plotkin, G.: Types for the Scott numerals (1993)
2. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 69–83. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_6
3. Anand, A., et al.: CertiCoq: a verified compiler for Coq. In: The Third International Workshop on Coq for Programming Languages (CoqPL) (2017)
4. Annenkov, D., Nielsen, J.B., Spitters, B.: ConCert: a smart contract certification framework in Coq. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 215–228 (2020)
5. Maffei, M., Ryan, M. (eds.): POST 2017. LNCS, vol. 10204. Springer, Heidelberg (2017). <https://doi.org/10.1007/978-3-662-54455-6>
6. Barras, B., et al.: The Coq proof assistant reference manual: Version 6.1. Ph.D. thesis, Inria (1997)
7. Barthe, G., Fournet, C., Grégoire, B., Strub, P.Y., Swamy, N., Zanella-Béguelin, S.: Probabilistic relational verification for cryptographic implementations. ACM SIGPLAN Not. **49**(1), 193–205 (2014)
8. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying tezos smart contracts. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 368–379. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_28
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-662-07964-5>
10. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96 (2016)
11. Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Weirich, S.: Ready, set, verify! applying hs-to-coq to real-world Haskell code (experience report). In: Proceedings of the ACM on Programming Languages 2(ICFP), pp. 1–16 (2018)
12. Buterin, V.: CRITICAL UPDATE Re: DAO Vulnerability (2016). <https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/>, Accessed 10 Dec 2021
13. Hutton, G. (ed.): MPC 2019. LNCS, vol. 11825. Springer, Cham (2019). <https://doi.org/10.1007/978-3-030-33636-3>
14. Dral, J.: Verified Compiler Optimisations. Master’s thesis, Utrecht University (2022)
15. GHC Team: GHC 9.0 User Manual. https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/extending_ghc.html
16. Giegerich, R., Möncke, U.: Invariance of approximative semantics with respect to program transformations. In: GI-11. Jahrestagung, pp. 1–10. Springer, Heidelberg (1981). https://doi.org/10.1007/978-3-662-01089-1_1
17. Gonthier, G., Le, R.S.: An Ssreflect Tutorial. Ph.D. thesis, INRIA (2009)
18. IOHK: The Plutus Platform and Marlowe 1.0.0 documentation. <https://plutus.readthedocs.io/en/latest/plutus/tutorials/plutus-tx.html>
19. Jones, M.P., Gkoumas, V., Kireev, R., MacKenzie, K., Nester, C., Wadler, P.: Unraveling recursion: compiling an IR with recursion to system F. In: Hutton, G. (ed.) MPC 2019. LNCS, vol. 11825, pp. 414–443. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33636-3_15

20. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. *ACM SIGPLAN Not.* **49**(1), 179–191 (2014)
21. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert—a formally verified optimizing compiler. In: *ERTS 2016: Embedded Real Time Software and Systems*, 8th European Congress (2016)
22. Li, J.M., Appel, A.W.: Deriving efficient program transformations from rewrite rules. *Proc. ACM Program. Lang.* **5**(ICFP), 1–29 (2021)
23. Necula, G.C.: Proof-carrying code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119 (1997)
24. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. *SIGPLAN Not.* **39**(4), 612–625 (2004)
25. O’Connor, L., et al.: Cogent: uniqueness types and certifying compilation. *J. Funct. Program.* **31**, e25 (2021)
26. Lahiri, S.K., Wang, C. (eds.): *CAV 2020*. LNCS, vol. 12224. Springer, Cham (2020). <https://doi.org/10.1007/978-3-030-53288-8>
27. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) *TACAS 1998*. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054170>
28. Sozeau, M., et al.: The MetaCoq project. *J. Autom. Reas.* **64**, 947–999 (2020)
29. Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total Haskell is reasonable Coq. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 14–27 (2018)