Translation Certification for Smart Contracts

Extended Abstract

Jacco Krijnen
j.o.g.krijnen@uu.nl
Utrecht University

Manuel M. T. Chakravarty
manuel.chakravarty@iohk.io
IOHK

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

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

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, that often controls significant amounts of financial assets and can no longer be updated once it has been committed to the blockchain. Bugs in smart contracts are a significant problem in practice [1]. Recent work has also established that smart contract language compilers can exacerbate this problem [8, 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 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.*

Why? Because blockchain users typically engage with contracts that have already been deployed as compiled code by another party. Before committing financial resources to such a contract, they need to check (a) that the compiled code did indeed originate from the provided source code and (b) that the compiler did not introduce any security bugs. While compiler verification helps with Point (b), it does not directly address Point (a). In contrast, a *certifying compiler* [7] squarely addresses both points as it generates a *certificate* of correct compilation together with the compiled code. For a smart contract user, such a certificate serves as a verifiable link between compiled code and its source as well as assurance that the compilation process preserved the source code semantics.

Hence, compiled code certification perfectly meets the requirements of smart contracts; additionally, it is also easier to realise for existing smart contract compilers that have not been developed with verification in mind. They are typically the product of constantly changing open-source projects, where the compiler developers do not have the expertise to keep a correctness proof of the compiler up to date during the continuous evolution of the compiler. It appears more plausible that we can retrofit and maintain a *certifier* for an existing compiler than to verify such a compiler and keep the proof of correctness up-to-date, while the compiler evolves — at least, if we can realise the certification engine with a *grey box approach,* where the certifier matches the general outline, but not all of the implementation details of the compiler.

In this paper, we are reporting on our ongoing effort to develop a certification engine for the on-chain code compiler of the Plutus smart contract system[1] for the Cardano blockchain.[2] The Plutus Tx compiler compiles a subset of Haskell to *Plutus Core*, a variant of System F_{ω}^{μ} [3]. 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 information about the validated transaction, such as the time range during which it will be included in the blockchain. Programs are run in an interpreter with call-by-value semantics, during the transaction validation phase of the blockchain.

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 [6]. We will therefore outline how we develop a certification engine that, using Coq, generates a proof object, a *compilation 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 compilation certificate serves as a verifiable link between source and generated code.

2 Chaining translation relations

We use a transitive chain of *translation relations* R_i as witnesses of the correct translation of a source AST t_1 to a target code AST t_n via intermediate forms t_i. To this end, we dump

TyDe '21, 2021-08-22, online
2021.

[1] https://github.com/input-output-hk/plutus/

[2] http://cardano.org is, at the time of writing, the 4th largest public blockchain by market capitalisation.

all the ASTs $t_1, \dots, t_n$ of the compiler that we certify. The ASTs and translation relations together with the compiler passes f_i are depicted in Figure 1, where the grey section represents the compiler transforming one tree into another. The orange section displays the translation relations R_i, and the blue section represents correctness proofs of the translation relations on the basis of the semantics of the underlying intermediate languages.

We have implemented this approach for a range of concrete intermediate languages of the Plutus Tx compiler, but we illustrate it here using only a simply-typed lambda calculus extended with non-recursive let-bindings.

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}, respectively. In other words, we consider f_i correct if

$$f_i(t_i) = t_{i+1} \text{ implies } R_i(t_i, t_{i+1}).$$

As a concrete example, consider inlining in the simply-typed lambda calculus with non-recursive let-bindings. We

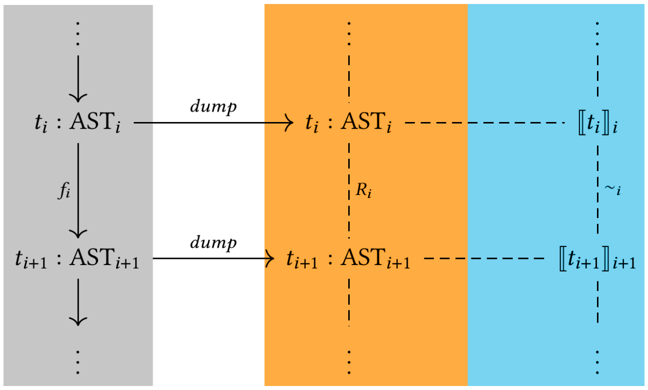

Figure 1. Architecture for a single compiler pass. The grey area represents the compiler, orange and blue represent the certification component in Coq

$$\frac{(x, t') \in \Gamma \quad \Gamma \vdash t' \triangleright t}{\Gamma \vdash x \triangleright t} \ \text{[Inline-Var]}$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' \quad (x, t_1), \Gamma \vdash t_2 \triangleright t_2'}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \triangleright \mathbf{let}\ x = t_1'\ \mathbf{in}\ t_2'} \ \text{[Inline-Let]}$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1' \quad \Gamma \vdash t_2 \triangleright t_2'}{\Gamma \vdash t_1\ t_2 \triangleright t_1'\ t_2'} \ \text{[Cong-App]}$$

$$\frac{}{\Gamma \vdash x \triangleright x} \ \text{[Cong-Var]}$$

$$\frac{\Gamma \vdash t_1 \triangleright t_1'}{\Gamma \vdash \lambda x.t_1 \triangleright \lambda x.t_1'} \ \text{[Cong-Abs]}$$

Figure 2. Characterisation of an inliner

can characterise an inlining relation as in Figure 2. Here, $\Gamma \vdash s \triangleright t$ asserts that s can be translated into t given an environment Γ of let-bound terms. According to Rule [Inline-Var] the variable x may be replaced by t when the pair (x, t') occurs in Γ and t' can be translated to t, accounting for repeated inlining. The rest are congruence rules, where Rule [Inline-Let] also extends the environment Γ. We have omitted details about handling variable capture for the sake of simplifity.

Crucially, these rules do *not* prescribe which variable occurrences should be inlined, since the [Inline-Var] and [Cong-Var] rules overlap. This choice 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.

2.2 Proof search

Given a translation relation R_i characterising one compiler stage, we need a search procedure to prove (automatically) whether any two terms t_i and $t_{i+1} = f_i(t_i)$, produced by a run of the compiler, are related as $R_i(t_i, t_{i+1})$. Automating this process typically proceeds in three steps:

1. We write proofs for specific compilations by hand using Coq's *tactics*. 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 about the proof search from within Coq and it quickly becomes slow for large terms.
2. Once we have some degree of confidence in the defined relations, we invest in writing decision procedures of type `forall (t1 t2 : Term), option (R t1 t2)`. These procedures can still produce large proof terms and need not always succeed in constructing 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 [4], `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 this time and effort once we have a reasonable degree of confidence that the relation we have defined accurately describes a given compiler pass.

Occasionally, the compiler may do more than one transformation in a single pass. For example, the inlining phase of the Plutus Tx compiler additionally performs dead code elimination, removing dead bindings that have been inlined exhaustively. Once we have modeled the individual transformations, we represent such complex passes 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 an *intermediate term*, t_2, that witnesses the composite transformation.

2.3 Semantics preservation

We can verify the correctness properties for each translation relation R_i separately and to varying degrees of fidelity. In the simplest case, this could be asserting the preservation of a program's static semantics, such as a proof of type preservation. On the other end of the spectrum, we might demonstrate that the semantics of the translated term is a refinement of the original term's full dynamic semantics.

In Figure 1, we characterise R_i's correctness properties in the blue area by way of refinement relations $\sim_i$ on the semantic objects $[\![t_i]\!]_i$ of ASTs t_i. We can do that independently and incrementally for each step in the translation. In fact, even without any formal proof about the semantics, manual inspection of a translation relation may already improve confidence into the correctness of a translation step. After all, the translation relation may be considered an implicit specification of this compiler pass' admissible behaviour.

2.4 Certificate generation

A compilation certificate includes the entire set of ASTs $t_1, \ldots, t_n$ together with the proof objects witnessing the translation relations R_i for these ASTs. In addition, it includes the instantiated proofs of the refinement relation to produce a single proof object.

This certificate together with the source code and compiled code can then be independently checked by a trusted proof checker, such as the Coq kernel [2]. The proof itself can be inspected to confirm it proves the right theorem. One can then be confident that the compiled program is a faithful translation of the original source code.

3 Preliminary results

We have begun to apply this approach with initial success to the Plutus Tx compiler, which compiles a Haskell subset into Plutus Core, a flavour of System F_ω^μ with a well-defined semantics [3]. The compiler consists of two 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 compiles GHC Core to Plutus Core. As Plutus Core is strict and doesn't directly support datatypes, both parts are quite complex. Moreover, both consist of a significant number of successive transformation steps.

So far, we have focused our certification effort on the second part, which is implemented as a GHC plugin translating GHC Core to *PIR (Plutus Intermediate Representation)* and then PIR in a number of steps, including inlining, dead code elimination and datatype encodings [5], to Plutus Core. We have now developed translation relations for the entire process from PIR to Plutus Core in Coq.

In our experience, specifying the translation steps with relations is vastly simpler than the logic and analyses required to implement them in the compiler. For example, characterising dead code elimination requires a simple condition on the free variables, whereas the implementation of such a pass needs to maintain a dependency graph.

Moreover, this grey box approach, where we characterise the individual transformation steps by translation relations, without detailing how these steps are implemented in the compiler is proving helpful. After all, the Plutus Tx compiler is being further developed and maintained by a development team at IOHK, largely independently of our certification effort. To pick an 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.

It is worth mentioning that Plutus Tx is a *whole-program compiler*: all code that will be run is known at compile-time and this fact can be used for more precise program analyses than most other compilers. We therefore expect that its optimisation passes will be subject to change and may include more more aggressive optimisation strategies.

In summary, the advantages that we are seeing with this approach so far are thus:

- Incremental development: we can develop the translation relation for each phase together with the corresponding search procedures and its metatheory separately. Furthermore, with just the translation relation and its search procedure (and without the metatheory), we already have got an implicit specification of the admissible outputs of that compilation step. This provides a first, independent assurance of the correctness of one phase. Hence, we could choose to only invest in working out the details of the metatheory for particular critical or difficult transformation steps.
- Robust architecture: as illustrated with the example of the inliner above, the certifier is independent of many specifics of the compiler implementation. Hence, many refactorings and performance improvements of existing passes should not have any effect on the proof infrastructure.

We have implemented the necessary relations and proof searches for some small programs, so that we can recognise their compilation from PIR up until their final form in Plutus Core. These programs implement simple contracts such as a time lock. Not all compiler transformations are covered in their full generality yet. For example, programs defining (mutually) recursive datatypes are not yet recognised during their encoding. We have started the verification of key components of the translation and are investigating different ways to scale up the proof search for larger programs.

References

[1] 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.

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

[3] 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.

[4] Georges Gonthier and Roux Stéphane Le. 2009. *An Ssreflect Tutorial.* Ph.D. Dissertation. INRIA.

[5] 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.

[6] 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.*

[7] George C. Necula and Peter Lee. 2004. The Design and Implementation of a Certifying Compiler. *SIGPLAN Not.* 39, 4 (April 2004), 612–625.

[8] 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.