Utrecht University
Faculty of Science
Department of Information and Computing Science
Graduate School of Natural Sciences

Automatically Deriving Checkers for Compiler Verification

Computing Science MSc Thesis

Bart Remmers

Supervisors:
Wouter Swierstra
Jacco Krijnen
Gabriele Keller

Acknowledgements

First and foremost, I want to thank my supervisors for their time and guidance throughout working on my thesis project. You made my thesis fun to work on, and your enthusiasm motivated me a lot.

I want to thank my family for being the best roommates I have ever had. To my parents, I appreciate you giving me the opportunity to pursue a master's degree, and supporting me in many ways to make the journey easier. To my sister, thanks for getting a master's degree before me, and raising Mum and Dad's expectations. I also want to thank my two dogs Esmee and Fedde, and my cat Siem. Esmee for giving me a reason to get out of bed, Fedde for adding a lot of energy to my life, and Siem for keeping me on my toes when I needed it the least.

Thanks to all my friends for all the emotional support, and for distracting me from working on my thesis when I needed it. Chiara, thank you for being my personal ChatGPT and helping me write these acknowledgements, you are my best friend, you are amazing, and you are totally not making me write this, smile. Molly, as my token 'enabling' friend, I appreciate you enabling me to have great hang-out sessions, and enabling me to stuff myself with great food. Bas, thanks for the piles of salt you provide to keep my life from being bland, as your name suggests, 'B' is for bestie, 'A' is for awesome, and the 'S' is for supercalifragilisticexpialidocious. Tenjin, thank you for being my personal chef and dungeon master, I love your miso-glazed aubergine, and your homebrew-glazed DnD campaigns. Maikel, since I have been friends with you the longest, I think it would be appropriate to make my acknowledgement to you the longest, and I am not trying to do my other friends a disservice here, I dunno, just seems right you know, but in case any of my other friends might get jealous, don't be, I really don't mean anything by it, besides saying that you, Maikel, are a great friend and I appreciate you a lot.

And to myself, お疲れ様でした, thank you for the hard work.

Abstract

Compiler verification is a hard problem. Verification costs can be reduced by means of translation validation, where individual runs of the compiler are verified, rather than all possible runs. This approach is used for the certification of the Plutus TX compiler. But this currently requires a handwritten decision procedure for each pass of the compiler. This thesis investigates the possibility of automatically deriving these decision procedures. This would reduce the complexity of verification of the compiler, because its derived checkers are less laborious to acquire, easier to maintain, and less prone to human errors.

Contents

Chapter 1

Introduction

Will your compiled program behave as intended? Compiler correctness is a subject of study that addresses such questions. It aims to formalize and verify the preservation of semantics before and after compilation: a program is written to exhibit particular behaviour, and this should be reflected in the execution of its compiled code [18, 3, 1].

Verifying compiler correctness is often non-trivial for modern compilers, considering the intricate symbolic transformations applied for optimization [7, 15], or the added complexity of describing the semantics for parallel programming languages [24]. Especially in large complex pieces of software, the existence of small bugs is not uncommon. For compilers, a small bug could lead to introducing unwanted side effects in a compiled program, such as crashes, silent errors, or arguably the worst, security vulnerabilities that can be leveraged to compromise the host system [10]. So even if executing compiled code behaves as expected on certain test data, with compiler correctness, the aim is instead to use a rigorous system of rules to precisely prove a compiler's correctness.

The CompCert compiler, developed and reported on by Xavier et al. [16], is a compiler for Clight (a large subset of C), which had its correctness verified using the theorem prover Coq. With a theorem prover, it is possible to write mechanized proofs that a program conforms to certain specifications [4, 20]. Such specifications can assure correctness over the CompCert compiler, by proving that it preserves semantics over any compiler program. Since the Clight language covers most of the important C language constructs, CompCert is an example of a correct compiler that can be used for realistic practical applications. The practical use of CompCert made it spark renewed interest in the subject of compiler correctness, and inspired others to assert correctness over several compilers in a similar manner [23, 2].

The main approach used by the authors of CompCert is called 'verified compiler'. Here, the compiler was modelled as a total function in Coq, and verified by defining and proving the desired correctness proofs directly over this function. This approach yields solid evidence of a compiler's correctness, but is unfortunately restricted to compilers that have been implemented in Coq itself.

Another approach described and used for CompCert is translation validation with verified validators. It lends itself better for verifying untrusted compilers, because it only requires a validator function to be implemented in Coq. The validator computes for some input and output of a compilation, whether some property holds and relates the two programs, such as semantic equivalence. Then when the desired correctness proofs are defined for the validator, we can trust it to check whether some property holds over a compilation.

Krijnen et al. investigate verifying the correctness of the Plutus Tx compiler [13], and report that in their case, it is seemingly infeasible to apply the verified compiler approach used by CompCert: The Plutus Tx compiler is already implemented as a plugin in the GHC Haskell compiler, so re-implementing a trusted version in Coq would demand a substantial development effort. They instead define a two-part certification architecture with a similar approach to CompCert's translation validation. The intuition behind this approach is that, instead of proving preservation of semantics for the compiler implementation, a proof can be constructed on a per-compilation basis over the compiler passes of the Plutus Tx compiler.

For the first part of the certification architecture, a translation relation is defined for every compiler pass. These relations form specifications over the passes, and describe in a high level what their admissible behaviour is, so what parts of a program should change and in what ways. If for some compilation the relations holds over all their respective compiler passes, then this implies the entire compilation has behaved according to the specification.

The initial method used for finding a proof that some translation relation holds, was writing a manual proof using Coq's tactics, which is useful for debugging purposes, but quickly becomes a slow method for large terms in the Plutus Tx language. The more preferable method used for finding such proofs was to write manual boolean decision procedures, which decide whether a relation holds over some two terms before and after a compiler pass. For the decision procedures, accompanying soundness proofs were then defined, to verify whether the result of the procedure actually implies that the relation holds. Then in practice, the Plutus Tx compiler dumps all the intermediate results of its compilation passes, for which proofs can then be found using the decision procedure, to determine whether all translation relations hold over the compilation.

The second part of the certification architecture establishes that if a translation relation holds, it implies semantic equivalence over the input and output of the respective compiler pass. This is done by defining the static and dynamic semantics of Plutus Tx programs, and finalizing proofs that these semantics remain the same after applying any of the compiler passes [8]. When such proofs are finalized, it is possible to transitively prove semantic preservation over an entire compilation, by determining that the translation relations hold over all compiler passes.

Even if the manual decision procedures and soundness proofs are an improvement, compared to writing a full proof that a translation relation holds, they still come with some downsides. It is complex and time-consuming to do this implementation by hand, and has to happen for every new translation added to the Plutus Tx compiler. Furthermore, whenever an existing translation is updated, the respective decision procedure and proof have to be updated as well. These downsides create a desire to find alternatives, such as automating the process of acquiring the decision procedures and proofs, such that using them becomes less complex and laborious.

To illustrate the complexity of a manual decision procedure and soundness proof, consider the untyped lambda calculus defined in Coq, as shown in listing 1.1. A translation relation for this calculus could be one that describes let-binding desugaring. Let-bindings can be desugared by wrapping the body of a let-binding with an abstraction, and directly applying the expression bound in the let-binding to this abstraction. This translation does not change what a program evaluates to, it just changes how the program is exactly evaluated.

```
Inductive tm : Type :=
  | tm_var : ref → tm
  | tm_abs : tm → tm
  | tm_app : tm → tm → tm
  | tm_let : tm → tm → tm.
```

Listing 1.1: Simply typed lambda calculus

A translation relation can be defined in Coq as an inductive relation, which for the let-binding desugaring translation can be seen in listing 1.2. Most constructs in the untyped lambda calculus simply desugar to themselves, and for their sub-terms it is checked whether they are a valid desugaring as well. The exception is the `DS_LetApp` constructor, which denotes how a let-binding can be desugared. Note the existence of the `DS_Let` constructor, which allows a let-binding to not be desugared as well. In this relation, let-binding desugaring is modelled as behaviour which is possible, but not required to take place.

```
Inductive desugar : tm → tm → Prop :=
  | DS_Var : forall {n},
      desugar (tm_var n) (tm_var n)
  | DS_Abs : forall {t1 t1'},
      desugar t1 t1' →
      desugar (tm_abs t1) (tm_abs t1')
  | DS_App : forall {t1 t2 t1' t2'},
      desugar t1 t1'→ desugar t2 t2' →
      desugar (tm_app t1 t2) (tm_app t1' t2')
  | DS_Let : forall {t1 t1' t2 t2'},
      desugar t1 t1'→ desugar t2 t2' →
      desugar (tm_let t1 t2) (tm_let t1' t2')
  | DS_LetApp : forall {t1 t1' t2 t2'},
      desugar t1 t1' → desugar t2 t2' →
      desugar (tm_let t1 t2) (tm_app (tm_abs t2') t1').
```

Listing 1.2: Desugar translation relation for the untyped lambda calculus

For the let-binding desugaring translation relation, the implementation of an associated decision procedure can be seen in listing 1.3. Most constructs in the lambda calculus will not be desugared, so the procedure simply recursively checks that its sub-terms are a valid desugaring as well. Valid let-binding desugarings are checked by matching a let-binding in the input program, and an application on an abstraction in the output program, where again it is checked if the sub-terms are valid desugaring.

```
Fixpoint is_desugar (t1 t2 : tm) : bool :=
  match t1, t2 with
  | tm_var v1, tm_var v1' → v1 =? v1'
  | tm_abs t1, tm_abs t1' →
     is_desugar t1 t1'
  | tm_app t1 t2, tm_app t1' t2' →
     is_desugar t1 t1' && is_desugar t2 t2'
  | tm_let t1 t2, tm_let t1' t2' →
     is_desugar t1 t1' && is_desugar t2 t2'
  | tm_let t1 t2, tm_app (tm_abs t2') t1' →
     is_desugar t1 t1' && is_desugar t2 t2'
  | _, _ → false
  end.
```

Listing 1.3: Desugar translation certifier

```
Lemma is_desugar_sound :
  forall t1 t2, is_desugar t1 t2 = true → desugar t1 t2.
Proof with auto using desugar.
  induction t1; induction t2;
  intros H; simpl in H; try discriminate H...
  - apply EqNat.beq_nat_true in H. rewrite H...
  - apply andb_prop in H as [HL HR]...
  - destruct t2_1; try discriminate H.
    apply andb_prop in H as [HL HR]...
  - apply andb_prop in H as [HL HR]...
Qed.
```

Listing 1.4: Desugar translation soundness proof

The translation relation and the decision procedure can be related using the soundness lemma in listing 1.4, which is listed together with the necessary mechanical proof that the lemma holds. This lemma states that, if the decision procedure returns true for some two given programs, then the relation holds over these two programs. That this relation holds can also be proven manually, but the proof would often need to be rewritten when the two programs change over which the relation should hold. Using the decision procedure and soundness lemma is faster at determining the same result as a manual proof, furthermore, it will automatically work for all programs a relation can relate. Nonetheless, manual proofs can still be useful to write for properties that do not hold generically, such as negated completeness over a decision procedure in terms of its respective relation.

The aforementioned decision procedure and soundness proof for let-binding desugaring, are still manageable in complexity and size. But Plutus Tx programs are substantially more complex than those of the untyped lambda calculus: They are typed and include more sophisticated constructs, such as recursive let-bindings. The implications are that the translation relations that Krijnen et al. defined are more complex, and subsequently, the manual decision procedures and proofs they implemented are more complex as well.

A recent paper by Paraskevopoulou et al. presents a framework implemented in the Coq library QuickChick [19], with derivation algorithms for extracting computational contents out of inductive relations, and custom Coq tactics for deriving the desired correctness proofs for said computations. The computations of interest are derived checkers, which are functions that take the arguments of an inductive relation as input, and compute whether a relation holds over those arguments. The purpose of the derived checker matches precisely that of the decision procedures used in the certification engine of the Plutus Tx compiler. Additionally, the soundness proofs that are manually proven over the decision procedures, can be derived automatically for the derived checker using the custom tactics provided by the framework.

The downsides of manually implementing and maintaining decision procedures, and the notion that these decision procedures can possibly be derived automatically, were the motivation behind the following research question of my thesis:

Research question: Can we automatically derive checkers used for compiler verification?

Chapter 2

Background

2.1 Translation certification for Plutus Tx

In Plutus Tx, smart contracts are written using a subset of Haskell, which after compilation are published on the blockchain Cardano. Compiler correctness in the context of smart contracts has seen renewed interest, since their code together with any possible bugs cannot be changed after being committed to a blockchain, which is a critical issue considering that smart contracts often manage important financial resources under adversarial conditions. As such, Krijnen et al. have defined a certification architecture for the Plutus Tx compiler in the theorem prover Coq [13], where the correctness of the Plutus Tx compiler can be ensured on a per-compilation basis.

In a compilation of a Plutus Tx program, it is first type-checked, desugared, and transformed into GHC Core. This GHC Core program is made into a self-contained program, by compiling it into an intermediate language called Plutus Intermediate Representation (PIR), which has included all referenced definitions. Then, several compiler passes are applied to the PIR program to transform it, after which it is compiled down into the low-level lambda calculus Plutus Core.

The certification architecture specifically targets the compiler passes applied to PIR, and consists of two steps to prove semantic preservation over this compilation step. First, a proof is found that a compilation behaves according to some specification. Second, a proof is constructed that the specification implies preservation of semantics, such that this property can also be proven to hold over any compilation run that meets the specification.

In the certification architecture, programs are represented with abstract syntax trees, ASTs for short. The compiler passes which transform PIR to Plutus Core are modelled as applying a composition of pure functions to an AST representing the PIR program. In this compilation step, changes and optimizations are made to the PIR program by using intricate heuristics, but the semantics of the program should stay the same, so a translation relation is defined for each compiler pass, as a specification on its admissible behaviour. If an instance of a relation holds over some input and output ASTs of a compiler pass, then it acts as a proof that the compiler pass behaved according to its specification.

To establish that a translation relation holds over two ASTs, there are several methods in Coq to find the necessary proof. One method is to manually write a mechanical proof using Coq's tactics, however, this is a slow method for proving that a translation relation holds over large programs. The preferred method used in the certification architecture is to write a decision procedure for the translation relations, together with a corresponding soundness proof. The decision procedure has the same arguments as the translation relation, so two ASTs and other possible arguments such as environments, and computes whether the relation holds over said arguments. Coq will not simply trust that a computation works exactly as intended, so to prove that the decision procedure accurately decides whether a relation holds, we need to formalize this by defining and proving a soundness proof. The soundness proof implies that, if the decision procedure returns true for some two ASTs, then the associated translation relation holds over those ASTs. The certification architecture can then apply the decision procedures over the input and output of compiler passes, to compute whether they behaved according to their specification.

Over these specifications, we can then prove that they imply the preservation of semantics. This can be achieved by defining the static and dynamic semantics of a program, and proving that ASTs related by the translation relations have the same semantics [8]. Once these proofs are finalized for the specifications of all the compiler passes, then they can be used to transitively prove semantic preservation over the entire compilation step.

The following are some translations for which the Plutus Tx compiler has a compiler pass, and for which translation relations and semantic preservation can be defined and proven in the certification architecture.

2.1.1 Variable renaming

Any variable in a PIR program can be renamed without changing the semantics of the program, as long as all uses of the variable are renamed correspondingly, and the new name does not coincide with another variable name currently in scope. A program after such a renaming is α-equivalent to the original program: they are semantically the same, they only differ in the variable names they use throughout their terms.

The Plutus Tx compiler includes a renaming pass, which transforms a program into an α-equivalent program with globally unique variable names. Using duplicate variable names in different scopes is allowed in PIR, however, some transformations for PIR might cause such duplicate names to start appearing in the same scope, which introduces variable shadowing and changes the semantics of the program. As such, the Plutus compiler pipeline first applies the renaming transformation to PIR to make its variable names unique, to guarantee that variable shadowing will not become a problem in other future transformations.

2.1.2 Dead-code elimination

An expression in PIR is considered dead code, if it is bound to a variable in a let-binding, but the variable is never used. The Plutus compiler mainly includes a dead-code elimination pass, since other transformations can introduce definitions with dead-code included. Such a dead expression can often, but not always, be removed without changing the behaviour of the program. The exception is for expressions with side effects, for example, expressions that diverge, which could influence the termination behaviour of a program if they are removed, and this termination behaviour should not change. It is also possible for the dead-code property to cascade down expressions, so if some expression is bound to a variable that is only used in dead code, this expression can also be considered dead code.

2.1.3 Inlining

In inlining, variables are replaced with the expression bound to said variables, which in the case of PIR are expressions bound to variables in a let-binding. However, the inlining transformation for PIR does a couple more things besides just inlining expressions. It also performs dead-code elimination when some expression bound in a let-binding is exhaustively inlined, in which case the let-binding is no longer necessary and can be removed. Furthermore, if the same expression is inlined multiple times, some variable names might be duplicated, so the inline transformation also applies renaming to keep variable names globally unique.

The inline transformation for PIR is essentially a composition of individual inline translations, where after each inlining an intermediate program arises. This is problematic in terms of the certification architecture, since to complete a proof that some translation relation holds over the inlining transformation, all these intermediate programs have to be found. To aid the search for these intermediate programs, the compiler outputs additional information on the inlinings that resulted in the programs, which in this case is a list of variables that have been inlined.

2.1.4 Let-floating

In the let-floating translation, let-bindings are moved upwards in a program if possible, to reduce unnecessary repeated evaluation of the let-binding. Consider a strict let binding that appears under a lambda abstraction, where the expression in the let-binding does not reference the variable of the lambda. This let-biding will evaluate to the same result when it appears before or after the lambda, but since the lambda can be passed around, and be evaluated multiple times for different arguments applied to the lambda, we prefer to move the let-binding up so it only has to be evaluated once. Non-strict let binding cannot always be floated, since if they include diverging expressions, this can influence the termination behaviour similarly as in the dead-code elimination translation.

2.1.5 Encoding of non-strict bindings

Plutus Core does not allow non-strict let-bindings, so a translation is included to encode PIR's non-strict let-bindings into strict ones instead. This can be achieved by thunking the expressions in the binding, which means they become abstractions that take a unit as argument, and have the non-strict expression as the body. Anywhere this non-strict expression is then referred to with a variable, a unit can be applied to the variable. This causes the thunk to be unwrapped, and the non-strict expression to be evaluated, exactly there where the results of the expression is necessary, which is the same behaviour as for the initial non-strict let-binding expressions.

2.1.6 Encoding of recursive bindings

A program in PIR can also have recursive binding, which is allowed to be mutually recursive, meaning they can refer to each other regardless of their order in the let-bindings. The Plutus compiler includes a translation, to encode such mutually recursive let-bindings, into non-recursive strict bindings instead. This is done by converting the recursive binding into a lambda, where the variable of the lambda represents the recursive function, and the body is the function that now non-recursively refers to the variable of the lambda. A non-recursive strict fixpoint combinator function can then be defined, to which we can apply the lambda representing the previously recursive functions The fixpoint combinator will cause the lambda to be applied to itself sequentially, and simulates the previous recursive function, but now in a non-recursive strict manner. It depends per let-binding which fixpoint combinator is used, since mutually recursive bindings require more complex fixpoint combinators to correctly simulate their recursive behaviour.

2.1.7 Encoding of non-recursive bindings

After non-strict and recursive let-bindings are encoded into strict non-recursive bindings, the Plutus compiler pipeline applies a translation that β reduces all these let-bindings. In a β reduction, a let-binding is desugared into an abstraction and an application. The body of the binding becomes the body of the abstraction, where the variable bound in the binding now becomes the variable bound by the abstraction, and the expression bound in the let-binding is then applied to the abstraction. Applying β reductions result in semantically equivalent programs, and are simply used to desugar let-bindings.

2.2 QuickChick derivation framework

It has to be said first, that the naming 'QuickChick Derivation Framework' is somewhat misleading. A derivation framework implemented in Coq is indeed intended to be added to a future release of the QuickChick library, but as of the time of writing, this has not yet happened.

In Coq, inductive relations are used to describe a relation, and are frequently used when writing elaborate mechanical proofs. Whether such a property holds can sometimes also be represented using a function computation, however, inductive relations are more expressive in the concepts they can capture, such as undecidability or non-determinism. Unfortunately, inductive relations do compute, which is a significant drawback.

The QuickChick derivation framework was defined by Paraskevopoulou et al.[19], and can be used for extracting computational contents out of inductive relations. It can derive semi-decision procedures, also called checkers, and enumerators and generators, which are grouped under the name producers. A checker can be used to decide whether an inductive relation holds over some set of concrete values for its arguments. If all but one argument of an inductive relation are given, producers can be used to enumerate or generate concrete values for the last remaining argument, such that the inductive relation holds. To generate definitions for these three types of computations in the Coq theorem prover, different instantiations of the same derivation algorithm are used. The framework also includes custom tactics, written in Coq's metaprogramming facility Ltac2, for deriving soundness and completeness proofs over the derived computations with respect to the original inductive relation.

2.2.1 Derivation algorithm

The intuition behind the derivation algorithm is to inspect the constructors of an inductive relation, derive a sub-computation for every constructor, and combine these into one large computation over the entire relation. Consider the grammar of inductive relations in Coq:

$$\begin{array}{l}\text{Inductive } P : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow \text{Prop} := \\ \mid C_1 : \forall x_1 \ldots,\ (Q_1\ e_{11}\ \ldots) \rightarrow \ldots \rightarrow P\ e_1\ \ldots\ e_n \mid \ldots\end{array}$$

An inductive relation P is defined over some arguments with the types $T_1 \ldots T_n$. The constructor C_1 describes that the instance $Pe_1 \ldots e_n$ holds, if the premise $Q_1 e_1 \ldots$ and other possible premises also hold. It is also allowed to use the relation P recursively in the premises of its own constructors, so Q_1 could be the relation P, or it could use a separate inductive relation. The different e's used in the premises or the result of the constructor can be constants, or they can be values depending on some universal quantifier, for example, the variable x_1.

The goal of a derived checker is to compute whether an inductive relation holds over some concrete values for its arguments. A relation only holds over some arguments, if one of its constructors matches with those arguments, and all of its premises hold. To express this in a computation, the derivation algorithm creates a sub-computation which first matches the result of a constructor with the current goal, and then verifies whether all its premises hold by calling their respective derived checkers. Pseudocode on how such a sub-computation for the constructor C_1 would look, can be seen in listing 2.1, where the variables $in_1 \dots in_n$ are the concrete values for which the checker will decide if the relation holds.

```
match in₁,...,inₙ with
| e₁,...,eₙ → check_Q₁ e₁₁...
end.
```

Listing 2.1: Derive sub-computation for a checker of an inductive relation

In this sub-computation, if the premise Q_1 would be the relation P, the premise can simply be checked by making a recursive call to the checker that is currently being derived. If Q_1 is another inductive relation, the derivation algorithm can be applied to said relation as well, to derive the desired checker for the premise.

All the sub-computations for the individual constructors of a relation can then be combined into one larger final derived computation, however, this comes with some additional considerations. Functions in Coq need to be total, so the derived checker is required to terminate and should have a decidable result. As stated before, in one of the sub-computations the derived checker can recursively call itself, and termination is not necessarily a guarantee for a recursive function.

Thus, the derivation algorithm includes a fuel argument in any computations it derives, which decreases for every recursive call, and causes the computation to terminate once the fuel runs out. This means the checker might not be able to decide whether an inductive relation holds over some arguments, hence it is called a semi-decision procedure.

A second fuel argument is included as well, which does not decrease for every recursion, and is used as a fresh fuel value when calling a derived computation of another inductive relation. This could be for example another derived checker used to check whether a premise holds, which by Coq's rules is required to terminate already, so the fuel argument used for this checker does not have to decrease to further guarantee termination.

The high-level structure of a derived checker for an inductive relation can be seen in listing 2.2. First, the checker will match on the fuel argument, to see if there is any fuel left. If fuel has run out, only the derived sub-computations of non-recursive constructors will be considered, so those that do not recursively call the checker. Such non-recursive sub-computations are guaranteed to terminate, which means that if fuel runs out, the checker is guaranteed to terminate as well by only using those non-recursive computations. When the checker still has some fuel left, it will simply consider the sub-computations derived for all constructors, and for every recursive call to the checker, the fuel variable `size`' is used, which is the fuel decreased by one.

```
Fixpoint checker size top_size in₁ ... inₙ : option bool :=
  match size with
  | O → (** check non-recursive constructors *)
    backtracking [
      fun tt → match in₁, ..., inₙ with,
                 | e₁,...,eₙ → check_Q₁₁ e₁₁...
                 end;
      fun tt → None ]
  | S size' → (** check all constructors *)
    backtracking [
      fun tt → ...
    ]
  end.
```

Listing 2.2: High-level structure of derived checker

To combine all the sub-computations, they are first thunked by enclosing them in '`fun tt →`', which postpones their evaluation until it is necessary for the checker to decide its result. All these thunks are then wrapped using the `backtracking` function. This function will evaluate the thunks one by one and returns `True` if one of the thunks evaluates to true, `False` if they all evaluate to false, and `None` if one of the thunks evaluates to `None`, and none of the thunks returned `True`. If one of the sub-computations returns true, it means that one of the constructors of a relation could have been used to construct the goal, so the checker has enough information to decide that its result should be true. For the checker to decide that a relation does not hold, all the different sub-computation should have returned false, since none of the constructors can be used to construct the goal. Lastly, in the case where fuel is zero a thunk is added that simply returns `None`, since if fuel has run out, and none of the sub-computations evaluated to true, the checker should return `None`. When a checker returns `None`, this implies the checker needs to be given more fuel to be able to decide whether some relation holds, since it needs to do more recursion to decide a result.

The approach and structure of deriving enumerators and generators is very similar to that of a derived checker. Sub-enumerators or -generators are derived from the constructors, which try to come up with a concrete value for some last argument of a relation, such that the respective constructor can be used to create an instance of the relation that holds. The derived sub-enumerators or -generators are wrapped in the `enumarating` or the `backtrack` function, which simply combines them into one larger enumerator or generator respectively. Producers can be defined over recursive types that can inhabit infinitely large values, which means that without the necessary adjustments, a derived producer might not terminate. Similarly to the derived checker, a derived producer is thus also given a fuel argument to guarantee termination, however, in this case, the fuel represents an upper bound on the size of the values which are produced. There might exist some value for the last remaining argument of a relation such that it holds, but if a producer is not given enough fuel, it will not be able to produce said value.

2.2.2 Non-trivial constructors

The derivation algorithm covers three non-trivial scenarios that can occur in the constructors of inductive relations, namely, non-linear patterns, function calls, and existentially quantified variables. If sub-computations for such constructors were derived in the same manner as for the aforementioned constructor C_1, it would result in computations inconsistent with Coq's rules.

A constructor with a non-linear pattern uses the same variable multiple times in its result. An example of a sub-computation derived for a non-linear pattern could be the following:

```
match in₁, in₂, ..., inₙ with
| e₁, e₁, ..., eₙ → ...
end.
```

Here, the variable e_1 is matched on twice, which is not allowed in a match statement in Coq, so this computation would not compile. As a solution, the derivation algorithm uses a pre-processing step, where such duplicate occurrences of a variable are replaced with a fresh variable, and a premise is added that checks if the original variable and the fresh variable are equal. After applying the pre-processing step to the constructor of the previous example, the valid derived sub-computation would look like the following instead:

```
match in₁, in₂, ..., inₙ with
| e₁, e_fresh, ..., eₙ → e₁ = e_fresh && ...
end.
```

Function calls in the result of a constructor pose a similar problem as for non-linear patterns, since matching against function calls in the match-statement is not allowed in Coq. The derivation algorithm applies essentially the same pre-processing step as for non-linear patterns, where the function application is replaced by a fresh variable, and an equality check is added on the fresh variable and the result of the function call. The following is an example of a valid sub-computation derived, after applying the pre-processing step over a constructor with the function call $f\ e_1$ in its result:

```
match in₁, ..., inₙ with
| e_fresh, ..., eₙ → f e₁ = e_fresh && ...
end.
```

The last non-trivial case covered is for constructors where the derived sub-computation needs to include existentially quantified variables. Consider the following constructor, which states that an inductive relation holds transitively:

$$\begin{array}{l}\text{Inductive } P : T_1 \rightarrow T_2 \rightarrow \text{Prop} := \\ \mid C : P\ e_1\ e_2\ \rightarrow\ P\ e_2\ e_3\ \rightarrow P\ e_1\ e_3 \mid \ldots\end{array}$$

Here, e_2 is not bound in the result of the constructor, so in $P\ e_1\ e_3$. The derivation algorithm, without the necessary adjustments, will derive a sub-computation which matches the arguments in_1 and in_2 with e_1 and e_3 respectively. Thus, in the computation, the values of the variables e_1 and e_3 are established using the match statement, but the value for the variable e_2 is yet unknown. This poses the problem that the computation needs some value for the variable e_2 to check the premises of the constructor, but using e_2 as an unbound variable is illegal in Coq.

The solution lies in the intuition that there might exist some value for e_2, which can be used in the derived sub-computation. This is where a derived producer comes into play, which can be used to solve two problems in the derived sub-computation. For the constructor C in the above example, values for e_2 can be generated with a derived producer of the inductive relation P, such that the premise Pe_1e_2 holds. For any other parts of the sub-computation where e_2 is needed, the value given by the producer can simply be used. Using this solution, the derivation algorithm derives the following sub-computation for the constructor C, which essentially uses an existentially quantified variable generated by a derived producer (in this case a derive enumerator):

```
match in₁, in₂ with
| e₁, e₃ →
    bind (enumST top_size (fun e' → P e₁ e'))
    $ fun e₂ → bind (checker size' top_size e₂ e₃)
    $ fun _ → return true.
end.
```

The function enumST is the derived enumerator for P, which is given a function with the argument e', and will produce values for e' until $P\ e_1\ e'$ holds. The function bind is used to forward the produced values of e', to the next function with the argument e_2. This function then checks whether the premise $P\ e_2\ e_3$ holds, using a recursive call to the currently derived checker. The result of this checker is then once again bound to a function that simply returns true. The reasoning behind this binding structure, is to sequence multiple goals, until they collectively can be achieved. Consider that even though the producer might generate a value for e_2 such that $P\ e_1\ e_2$ holds, this does not mean that the premise $P\ e_2\ e_3$ also holds. So, values for e_2 are generated continuously by the derive producer, until the checkers in any next functions decide that all other necessary premises hold as well.

It is also possible that the producer does not manage to generate any values for e_2, such that all the necessary premises hold. This could either mean that the producer has run out of values to inhabit e_2 with, or the producer requires more fuel to find such values. For a derived producer used in a checker, the former would lead to the checker returning False, since for the premises it can be concluded they do not hold, whereas for the latter the checker would return None, since the producer needs to be given more fuel for a decisive result.

2.2.3 Deriving correctness proofs

The derivation algorithm is defined in QuickChick together with Ltac2 tactics, which can derive correctness proofs over the computations extracted from some inductive relation. The types of correctness proofs for which an Ltac2 tactic is included are soundness, completeness, and monotonicity proofs, for which the definitions can be seen in listing 2.3.

```
Lemma sound : forall e₁, ..., eₙ, s,
  checker s e₁ ... eₙ = Some true → P e₁, ..., eₙ.

Lemma complete : forall e₁, ..., eₙ,
  P e₁, ..., eₙ → ∃ s, checker s e₁ ... eₙ = Some true.

Lemma monotonic : forall e₁, ..., eₙ, s1, s2, b, s1 ≤ s2 →
  checker s1 e₁ ... eₙ = Some b → checker s2 e₁ ... eₙ = Some b.
```

Listing 2.3: Proof types which can be derived using an Ltac2 tactic

The soundness proof proves that if a derived checker returns true of some arguments, then the inductive relation P holds over those arguments. Completeness states the opposite, so if P holds over some arguments, the checker should return true over those arguments for a large enough size, hence size is an existentially quantified variable in this proof. Monotonicity implies that if a checker came to a conclusion for some amount of fuel, then increasing that amount of fuel will never lead to another conclusion.

The way the Ltac2 tactics derive these proofs, is by matching all the different sub-computations or constructors of P on the right side of the implication, with the right ones on the left side of the computation. Consider a soundness proof for a derived checker. If a checker returned true, then at least on of the sub-computation inside the checker must have returned true. If a sub-computation returns true, then one of the constructors of P could have been used to construct it for the current goal. The Ltac2 tactic matches the right sub-computations with the right constructors, and proves for all of them that the above holds. In a completeness proof the idea is instead that if P holds, it should have been constructed with one of its constructors. In this case, one of the sub-computations should return true, which means the checker, for a large enough size, should return true as well. Again, all the different cases for each constructor of P are proved separately, and the Ltac2 tactic can derive the entire proof. For monotonicity, the same intuition is used again, but this time sub-computations are matched with other sub-computations.

Soundness and completeness proofs also have a negated version, which can be seen in listing 2.4.

```
Lemma neg_sound : forall e₁, ..., eₙ, s,
  checker s e₁ ... eₙ = Some false → ¬(P e₁, ..., eₙ).

Lemma neg_complete : forall e₁, ..., eₙ,
  ¬(P e₁, ..., eₙ) → ∃ s, checker s e₁ ... eₙ = Some false.
```

Listing 2.4: Negative soundness and completeness proofs

Negated soundness can simply be proven using the completeness and monotonicity proofs. If a checker returns `Some false`, monotonicity implies it is impossible for the checker to also return `Some true` for another fuel size. The completeness proof states that for $P\ e_1,\ \ldots,\ e_n$ to hold, the checker should return `Some true` for some fuel size. Since the checker can never return `Some true` when it already returns `Some false`, it proves that $P\ e_1,\ \ldots,\ e_n$ never holds, so the negated soundness proof holds.

Unfortunately, the negated completeness proof cannot be derived, since it does not generally hold. An inductive relation can describe non-terminating computations, which can cause its derived checker to return `None` for infinitely large fuel amounts. So even though we might know that $P\ e_1,\ \ldots,\ e_n$ should not hold, it is impossible to determine that the checker will return `Some false` for some fuel amount.

2.2.4 Considerations and limitations

A derived computation will include the sub-computations in the same order, as the constructors, from which they were derived, appear in the inductive relation. This means that the order of the constructors can have considerable influence on the performance of derived checkers and producers. If it is known that one of the constructors is used substantially more often for practical instances of some relation, then it is more efficient to define that constructor first in the relation, since the derived computations will then consider the right sub-computation first as well.

Another consideration when deriving computation from a relation is whether the derived computations will need to use a producer to find a value for some existentially quantified variables. For such cases, a sub-computation does not just compute a result for one instance of the relation, but it might compute the result for substantially more instances, by generating a lot of values for the existential variable. It might not always be possible, but in some contexts, an inductive relations can be rewritten, or given more information by means of an additional argument, such that using a producer is no longer necessary in a derived computation.

There are also some hard limitations for the derivation algorithm. It is unable to derive computations for mutually inductive types, or use multiple producer outputs in a derived computation. This is not because these scenarios are practically impossible, they are simply the results of design decisions made by the implementers of the algorithm. If having a decision procedure for these cases is desired, the implementers make suggestions on how checkers and producers can manually be implemented using a minor variation of their derivation algorithm.

2.3 MetaCoq

The MetaCoq project [22] provides an alternative way of implementing Coq plugins, and aims to be easier to use than the traditional method of implementing plugins in OCaml, such as done for the QuickChick plugin. Developing a Coq plugin in OCaml is non-trivial since, as stated by Liesnikov et al. [17], ”for non-experts, the OCaml code of both Coq and Coq plugins is hard to access and almost impossible to adapt without the help of experts”. Furthermore, a Coq plugin written in OCaml needs to be compiled to a binary first, and then wired up into the Coq ecosystem before it can be tested or used in any Coq code. This all adds substantial complexity to a project in which a Coq plugin is developed in OCaml, and needs to be integrated into the existing Coq code base. In MetaCoq, plugins are implemented as a syntax transformation over an AST, for which the functionality needed is exposed through pure Coq functions, so any plugin made with MetaCoq is entirely contained within the Coq ecosystem. This makes the integration of MetaCoq in any project or development process easier, since MetaCoq can simply be imported as a conventional Coq library, and any development and testing can be done interactively in the same Coq file.

The pipeline of most applications of a MetaCoq plugin is made up of the following three steps, quoting an existing Coq definition, applying a syntax transformation, and unquoting the result into a new Coq definition.

Quoting an existing Coq definition means parsing the definition into an AST representation over which the syntax transformation can be applied. The syntax transformation then describes the logic of the plugin, for example, deriving induction principles over inductive types [17]. The AST resulting from the transformation can be 'unquoted', which type-checks the new definition that the AST represents, and if it is type-correct, the definition is injected into the current Coq context, so that the definition can be used as if it was explicitly defined.

To elaborate more on MetaCoq and how it can be used to make Coq plugins, first consider the addition function over Peano numerals defined in listing 2.5, which when 'quoted' give us the AST as can be seen in listing 2.6.

```
Fixpoint add (a b : nat) : nat :=
  match a with
  | 0 → b
  | S a → S (add a b)
  end.
```

Listing 2.5: An addition function on Peano natural numbers

```
tFix [{|
  dname := nNamed "add";
  dtype := tProd (nNamed "a") (tInd inat [ ])
             (tProd (nNamed "b") (tInd inat [ ]) (tInd inat [ ]));
  dbody := tLambda (nNamed "a") (tInd inat [ ])
             (tLambda (nNamed "b") (tInd inat [ ])
               (tCase (inat, 0)
                 (tLambda (nNamed "a") (tInd inat [ ]) (tInd inat [ ]))
                 (tRel 1)
                 [(0, tRel 0);
                  (1, tLambda (nNamed "a") (tInd inat [ ])
                             (tApp (tConstruct inat 1 [ ])
                             [tApp (tRel 3) [tRel 0; tRel 1]]))]));
  rarg := 0 |}] 0
```

Listing 2.6: The MetaCoq AST of the addition function

The AST includes two main terms, one to represent the type of the function, and the other for the body of the function, which is assigned to the attributes `dtype` and `dbody` respectively. The terms include several constructors to denote the different concepts that occur throughout Coq definitions, such as `tRel` to denote a DeBruijn index pointing to some bound variable, `tLambda` denoting an abstraction of some variable over a body, and `tProd` representing the product type of such a lambda. Please note, the terms represent definitions in their desugared form, so for example, a match statement is desugared to a case statement with several cases.

The core logic of a MetaCoq plugin is implemented as a syntax transformation over ASTs, such as the one for the add function in listing 2.6. Match statements can be applied to the AST to assert which constructs make up the AST, and after gathering all the necessary information this way, some common goal can be achieved for several different Coq definitions. Consider the simple algorithm shown in listing 2.7, which adds an additional constructor to an already existing Coq relation.

The algorithm first quotes the identifier `ind`, to get its hand on the AST of the definition bound by said identifier. The AST is not guaranteed to represent a relation to which a constructor can be added, so first a match is done on the AST with `tInd` to verify if it is actually an inductive relation. The first quote on `ind` merely returns the AST of a lazy reference to the relation that is targeted, so the function `tmQuoteInductive` is used to acquire the AST of the entire definition of the inductive relation. The algorithm also quotes the identifier `ctor`, which returns the AST of the body of the constructor that will be added to the relation. All the necessary variables are then passed to the `add_ctor` function, which is a pure Coq function that injects the constructor into the AST of the already existing relation. The resulting definition is evaluated using `tmEval`, and then unquoted into a Coq definition using the function `tmMkInductive`.

Another important concept that MetaCoq makes use of is the TemplateMonad, as also can be seen in the definition of `add_constr` in listing 2.7. This TemplateMonad is returned by most functions exposed by MetaCoq, and to be able to chain several such functions together, MetaCoq also includes a binding function

```
Definition add_constr {A} (ind : A) (idc : ident) {B} (ctor : B) : TemplateMonad unit :=
  tm ← tmQuote ind ;;
  match tm with
  | tInd ind0 _ →
    decl ← tmQuoteInductive (inductive_mind ind0) ;;
    ctor ← tmQuote ctor ;;
    d’ ← tmEval lazy (add_ctor decl ind0 idc ctor) ;;
    tmMkInductive d’
  | _ → tmFail "The provided term is not an inductive"
  end.
```

Listing 2.7: A function using the TemplateMonad and MetaCoq’s functionality to add a new constructor to an already existing relation

that is applied using the notation `expr1` ;; `expr2`. In the aforementioned notation, the expression `expr1` is evaluated first, the result is then unwrapped from the TemplateMonad, and then used in the evaluation of expression `expr2`, without the programmer explicitly having to deal with the TemplateMonad.

The use of the TemplateMonad and binding function enforces the statements are evaluated top-down, and allows for a sense of linear execution of any algorithm or plugin implemented using MetaCoq. This is important since stateful changes are made to the Coq context, such as adding a new definition with an extra constructor, as is done in the algorithm `add_constr`. Consider a scenario where a new definition is added to the context, and then later on in the algorithm, we try to quote this definition into an AST. In such cases, if there is no sense of linear execution of the algorithm, there are no guarantees that the definition we try to quote actually exists yet.

Chapter 3

Preliminary Study

To get acquainted with the derivation framework of Paraskevopoulou et al. I first applied it to derive a checker for deciding equality over natural numbers. Then, to try and bridge the gap to the Plutus Tx compiler, I focused on a toy compiler for a simply typed lambda calculus, and derived checkers for its translation relations for let-binding desugaring and inlining. This gave me some insights into what problems can be expected when applying the framework to the more complex translation relations used for Plutus Tx. Finally, I benchmarked the derived checkers against manually written checkers, manual Coq proofs, or Coq proofs using automatic tactics, which were all used for proving correctness of the translation relations. The results were promising, and showed that derived checkers have comparable performance to the manual or automatic proofs. Furthermore, the results also indicated that a derived checker is a feasible alternative to a manually implemented checker.

3.1 Derived checker for equality over natural numbers

First, I defined an inductive relation for equality over natural numbers, which can be seen in listing 3.1. The relation acts as a piece of proof, and in this case, can only be created for exactly those pairs of natural numbers that are equal. For the trivial case that zero and zero are equal, we can use the first constructor of the relation to create the respective proof. The second constructor can be given the proof that n and m are equal, and will then return the proof that the successors of n and m are equal, and thus it inductively defines equality over all other non-zero natural numbers.

```
Inductive NatEq: nat → nat → Prop :=
  | Eq_Zero : NatEq 0 0
  | Eq_Succ (n m : nat) : NatEq n m → NatEq (S n) (S m).

Derive DecOpt for (NatEq n m).
```

Listing 3.1: Inductive relation for equality over natural numbers, for which a checker is derived using QuickChick

The listing 3.1 also shows at the bottom, how to use the derivation framework implemented in the QuickChick library, to derive a checker for this inductive relation. This checker is supposed to decide over two arbitrary natural numbers whether they are equal, or rather, whether my inductive relation for equality holds over those numbers. To guarantee that this claim actually holds, the QuickChick library includes a definition for a soundness proof that we can define over the derived checker. The definition of the soundness proof can be seen in listing 3.2, where P is the inductive relation the soundness proof is defined for, which in my case is `NatEq n m`. It also includes a variable s to denote the amount of fuel given to the checker, which is included to guarantee that the checker terminates when it is computing a decision.

```
Class DecOptSoundPos (P : Prop) {H : DecOpt P} :=
  sound : forall s, decOpt s = Some true → P.
```

Listing 3.2: QuickChick soundness proof definition

When we define such a soundness proof, we do not have to manually prove the theorem, since the derivation framework includes custom tactics for deriving such proofs. The soundness theorem for `NatEq n m`, and how to derive its proof using the custom tactic derive_sound, can be seen in listing 3.3.

```
Instance NatEqSound n m : DecOptSoundPos (NatEq n m).
Proof. derive_sound. Qed.
```

Listing 3.3: Soundness proof for derived checker of NatEq

Now it is possible to compute whether the inductive relation holds over two natural numbers, instead of having to prove it manually. Consider the example in listing 3.4, where I can simply apply the soundness proof, let Coq compute the result of the derived checker, and finish the proof with the reflexivity tactic. The number 100 was chosen arbitrarily for the fuel, and will have to be increased when deciding equality over larger natural numbers. Increasing the fuel is always an option, but comes with the cost of longer execution times.

```
Example ten_eq : NatEq 10 10.
Proof. apply sound with (s := 100). reflexivity. Qed.
```

Listing 3.4: Example of how to use derived checker for NatEq

3.2 Toy compiler

After focusing on the relatively simple relation `NatEq n m`, I applied the QuickChick derivation framework to the more complicated translation relations of a toy compiler written in Coq [11]. I was provided with this toy compiler to practice deriving checkers on more complex inductive relations, and to gain some preliminary results on the performance of the derivation framework, before moving on to using the framework for the Plutus Tx compiler.

Programs in the toy compiler are a simply typed lambda calculus, which is represented using abstract syntax trees, or ASTs for short, for which the definition is shown in listing 3.5. As can be seen, a program ranges over unit, boolean, or arrow types, and includes constructs for variable referencing, abstractions, applications, if-else statements, and let-bindings.

```
Inductive ty : Type :=
  | ty_bool : ty
  | ty_unit : ty
  | ty_arrow : ty → ty → ty.

Inductive tm : Type :=
  | tm_unit : tm
  | tm_true : tm
  | tm_false : tm
  | tm_var : ref → tm
  | tm_abs : ty → tm → tm
  | tm_app : tm → tm → tm
  | tm_if : tm → tm → tm → tm
  | tm_let : tm → ty → tm → tm.
```

Listing 3.5: Definition AST of simply typed lambda calculus

ASTs are a common way of representing programs in compilers when implementing compiler passes for analysis, verification, or optimization. We can define translation relations over ASTs, such that the relation describes the correct behaviour expected to happen in a compiler pass. I focused on two such translation relations defined in the toy compiler, one for desugaring let-bindings, and another for inlining let-bindings.

3.2.1 Let-binding desugaring

First, let us consider let-binding desugaring. We already showed in listing 1.2 of the introduction, what a let-binding desugaring relation looked like for an untyped lambda calculus. For terms of the toy compiler, the same principle holds: A let binding can be desugared into an application and abstraction. In either case, you end up with a body where a variable references the same bound expression. Furthermore, the terms before and after let-binding desugaring will evaluate to the same result, so semantics are preserved over this translation.

```
Inductive desugar : tm → tm → Prop :=
  | DS_Unit :
      desugar tm_unit tm_unit
  | DS_True :
      desugar tm_true tm_true
  | DS_False :
      desugar tm_false tm_false
  | DS_Var : forall {n},
      desugar (tm_var n) (tm_var n)
  | DS_Abs : forall {t1 T1 t1'},
      desugar t1 t1' →
      desugar (tm_abs T1 t1) (tm_abs T1 t1')
  | DS_App : forall {t1 t2 t1' t2'},
      desugar t1 t1'→ desugar t2 t2' →
      desugar (tm_app t1 t2) (tm_app t1' t2')
  | DS_If : forall {t1 t1' t2 t2' t3 t3'},
      desugar t1 t1' → desugar t2 t2' → desugar t3 t3' →
      desugar (tm_if t1 t2 t3) (tm_if t1' t2' t3')
  | DS_Let : forall {t1 t1' T1 t2 t2'},
      desugar t1 t1'→ desugar t2 t2' →
      desugar (tm_let t1 T1 t2) (tm_let t1' T1 t2')
  | DS_LetApp : forall {t1 t1' T1 t2 t2'},
      desugar t1 t1' → desugar t2 t2' →
      desugar (tm_let t1 T1 t2) (tm_app (tm_abs T1 t2') t1').
```

Listing 3.6: Translation relation for let binding desugaring

In the toy compiler, the translation relation in listing 3.6 was already defined to describe the correct behaviour of a let-binding desugaring. The constructor of interest is `DS_LetApp`, which describes exactly how a let-binding can be desugared to an application and abstraction. Since it is also valid not to desugar a let-binding, the constructor `DS_Let` is also defined, which simply checks whether all the sub-terms of a let-bindings are a valid desugaring.

Similarly as in the earlier example for `NatEq n m`, I used QuickChick to derive a checker over the `desugar` translation relation, together with a soundness proof for said checker, as can be seen in listing 3.7.

```
Derive DecOpt for (desugar t1 t2).

Instance DesugarSound t1 t2 : DecOptSoundPos (desugar t1 t2).
Proof. derive_sound. Qed.
```

Listing 3.7: Deriving checker and soundness proof for the desugar relation

Fortunately, for this inductive relation, QuickChick was also able to derive a checker and proof straight away. It was not needed to alter the definition of the desugar translation relation, or derive other sub-checkers for any constraints, which will be the case for the let-binding inlining relation later on.

The derived checker and soundness proof can now be used in the pipeline of the toy compiler, to verify that any desugaring that takes place is correct.

3.2.2 Let-binding inlining

In let-binding inlining, the expression bound in the let-binding can be inlined wherever a variable refers to that expression. This is a more specific version of variable inlining, which inlines expressions for all possible variables, not just those defined in a let-binding. The main benefit of variable inlining is that a variable no longer needs to be dereferenced to the expression during the evaluation of a program. Unfortunately, it also has drawbacks and can cause the same expression to need to be evaluated multiple times, whilst always having the same result. Furthermore, inlining a large expression many times will increase the size of an AST substantially, which can worsen performance in future translations. Luckily, where and why an expression was inlined is of lesser concern when proving the correctness of the translation, since let-binding inlining preserves the semantics of a program regardless.

For let-binding inlining, the toy compiler also included a translation relation. Before QuickChick was able to derive a checker over this relation, I had to make some necessary alterations, which resulted in the translation relation in listing 3.8.

```
Inductive inline : benv → tm → tm → Prop :=
  | Inl_Unit : forall benv,
      inline benv <{ tm_unit }> <{ tm_unit }>
  | Inl_True : forall {benv},
      inline benv <{ true }> <{ true }>
  | Inl_False : forall {benv},
      inline benv <{ false }> <{ false }>
  | Inl_Var : forall {benv n},
      inline benv (tm_var n) (tm_var n)
  | Inl_VarInl : forall {benv n t},
      benv_let_binding benv n t →
      inline benv (tm_var n) t
  | Inl_Abs : forall {benv t1 T t1'},
      inline (shift_benv (LambdaBound :: benv)) t1 t1' →
      inline benv <{ \T, t1 }> <{ \T, t1' }>
  | Inl_App : forall {benv t1 t2 t1' t2' },
      inline benv t1 t1' →
      inline benv t2 t2' →
      inline benv <{ t1 t2 }> <{ t1' t2' }>
  | Inl_If : forall {benv t1 t1' t2 t2' t3 t3'},
      inline benv t1 t1' →
      inline benv t2 t2' →
      inline benv t3 t3' →
      inline benv
        <{ if t1 then t2 else t3 }>
        <{ if t1' then t2' else t3' }>
  | Inl_Let : forall {benv t1 t1' T1 t2 t2'},
      inline benv t1 t1' →
      inline (shift_benv (LetBound t1' :: benv)) t2 t2' →
      inline benv
        <{ let t1 : T1 in t2 }>
        <{ let t1' : T1 in t2' }>.
```

Listing 3.8: Translation relation for let-binding inlining

This relation is not only defined over two programs, but also over an environment that keeps track of the bindings that variables can refer to. In the toy compiler, a variable is defined as a De Bruijn index `n`, which denotes that the variable refers to the value bound `n` bindings back (zero-indexed). The environment with bindings is populated in backwards order, so when a variable occurs, we can find the respective binding by using the reference of the variable as index in our environment.

The constructors `Inl_Abs` and `Inl_Let` illustrate how bindings, for abstractions or let-bindings respectively, are added to the environment. Here, the indices for all other bindings will increase by one, so to re-align all the references of variables to any bindings, the function `shift_benv` is applied to the environment. The constructor `Inl_VarInl` then shows how the translation relation models let-binding inlining. The premise of this constructor holds when the environment stores some expression `t` bound by a let-binding at index `n`. Then, for variables with the reference `n`, if the premise holds, it means the expression `t` can be inlined in place of the variable.

This latter constructor `Inl_VarInl`, was the part of the translation relation that had to be rewritten. In the original definition in listing 3.9, the premise uses a function to retrieve the n-th binding out of the environment, and checks whether it returns some expression `t` bound in a let-binding. Using function applications, or equality checks in the constraints of a translation relation is not necessarily problematic when using QuickChick to derive checkers. However, in this case, the function returns an option type, which is a type defined in the standard library of Coq. As of right now, I have not yet figured out if, and how it is possible to get QuickChick to derive checkers over relations that used definitions of the standard library in their constraints. So instead, I opted to write a custom inductive relation `benv_let_binding`, to capture the same behaviour as of the function, and the equality check on its result. With the necessary changes, I was able to derive a checker and soundness proof using QuickChick in exactly the same manner as for the aforementioned inductive relations `NatEq n m` and `desugar t1 t2`.

```
Inductive inline : benv → tm → tm → Prop :=
  ...
  | Inl_VarInl : forall {benv n t},
    nth_error benv n = Some (LetBound t) →
    inline benv (tm_var n) t
  ...
```

Listing 3.9: Original definition of `Inl_VarInl`

3.3 Benchmark

To get some rudimentary results on the performance of QuickChick's derived checkers, I did a simple benchmark within the context of proving correctness over the translation relations of the toy compiler. First, I implemented a Haskell program to easily generate different problem instances. The program generates ASTs of the toy compiler, applies let-binding desugaring and inlining over them, and then puts the resulting ASTs inside a Coq file with different types of proofs attached. One of these proofs uses the derived checker, and the performance of the checker is measured by how long this proof takes to finish, so how long the Coq file takes to compile. I made a script which uses the Haskell program to generate instances of several given AST sizes, times how long the different proofs types take to finish, and then exports the resulting times to a csv file. Finally, I created a Python script to plot the results on a bar graph, to better visualize the results of the benchmark.

3.3.1 Haskell generator

The toy compiler already included a Haskell program for generating simple lambda calculus terms, applying transformations over them, and adding them in a Coq file with some proofs attached for verification of correctness. Unfortunately, this Haskell program was based on a much older iteration of the toy compiler, for a lambda calculus without any types included. Needlessly to say, this Haskell program was outdated and did not work any more anyway, so I updated, and repurposed the program for my benchmark.

I opted to remove the original generator in favour of a generator written in the Haskell testing library QuickCheck [6, 5]. Using QuickCheck gave me better control over how randomness manifests when an AST is generated. The first problem of total randomness is that in the case of the AST of the toy compiler, it is unlikely to generate large ASTs. Instead, I implemented a less random QuickCheck generator, which generates ASTs with exactly n nodes, for some given size n. This way, whenever we give a sufficiently large value for n, the generator is guaranteed to generate large ASTs. The second problem of a fully random AST generator is that it disregards types, and thus will generate AST which are not necessarily well-typed. The logic of a generator can be changed to be made aware of what a valid AST should look like in terms of its types, however, this is a rather complex endeavour. For the translation relations for the toy compiler, it is not relevant whether an AST is well-typed or not, so I refrained from making the generator return well-typed ASTs.

The original Haskell program also already included functionality for non-deterministically applying let-binding desugaring and inlining. In practice, compilers do not apply transformations everywhere where it is possible, instead, it is only done in a handful of places where it is considered meaningful based on heuristics. Thus, to simulate this, desugaring and inlining of let-bindings is applied non-deterministically according to randomly generated boolean values. I largely kept the functions for applying transformations intact, I only changed how random boolean values are distributed to the functions, to improve efficiency.

Finally, the Haskell program writes the generated ASTs before and after inlining and/or desugaring, to a handful of Coq files. The structure of the files is to write the AST definitions to a single file, and write the to-be-timed proofs to separate files importing the required AST definitions. This structure was used such that after having done a single instance of the benchmark, only the definitions of the AST have to be archived away, not the definition of the proofs. Archiving the AST definitions is done for reproduction purposes, but archiving the proofs definitions would be redundant, since these are always the same.

3.3.2 Proof types

The different proof types I selected for comparison in my benchmark, I will call Repeat Constructor, EAuto, Manual Checker, and Derived Checker. The Repeat Constructor proof simply repeatedly applies the `constructor` tactic, as can be seen in listing 3.10, which repeatedly tries to find the right constructor to further progress the goal. This method is sufficient to finalize the proof that a translation relation for let-binding desugaring or inlining holds. However, the `constructor` tactic will have to search for the right constructor, so Repeat Constructor is not necessarily guaranteed to be fast.

```
Lemma is_desugar_manual : desugar generatedTerm desugaredTerm.
Proof. repeat (constructor). Qed.
```

Listing 3.10: The Repeat Constructor proof type for the let-binding desugaring translation relation

In the EAuto proof in listing 3.11, I apply Coq's `eauto` tactic for a depth of 100000, which semi-automatically tries to prove the current goal, by searching through several tactics it can apply, until a depth of 100000 is reached. Similarly to Repeat Constructor, one of the strategies the `eauto` tactic applies is Coq's `constructor` tactic, which for this case is enough to finalize a proof. However, the `eauto` tactic also considers other options, and uses backtracking to go through these options and see which one might lead to a complete proof. This means its search space is much larger than of Repeat Constructor, so the EAuto proof type is expected to be slower. The depth of 100000 was chosen somewhat arbitrarily, the important part is that the `eauto` tactic is given enough depth to complete the goal.

```
Lemma is_desugar_auto : desugar generatedTerm desugaredTerm.
Proof. eauto 100000 using desugar. Qed.
```

Listing 3.11: The EAuto proof type for the let-binding desugaring translation relation

The last two proof types are Manual Checker, and Derived Checker, shown in listing 3.12 and 3.13 respectively. They both apply a soundness proof, let Coq compute the result of the checker, and use the reflexivity tactic to resolve to goal. For Manual Checker, I used a checker and respective soundness proof which I defined in Coq manually. In Derive Checker, I made use of the checker and soundness proof derived by the derivation framework in the QuickChick library. Like Coq's `eauto` tactic, the derived checker takes some depth as argument, and will try to compute a goal up until a given depth. This depth is passed along when applying the derive soundness proof, and was chosen arbitrarily to be 10000 such that the derived checker had enough depth to resolve the goal.

```
Lemma is_desugar_dec_manual : desugar generatedTerm desugaredTerm.
Proof. apply is_desugar_sound. reflexivity. Qed.
```

Listing 3.12: The Manual checker proof type for the let-binding desugaring translation relation

3.3.3 Results

I gathered my results using a simple benchmarking suite script. The script takes as arguments a list of desired AST sizes, and the number of iterations every AST size should be benchmarked. For every AST size, and iteration, instances are generated using the Haskell program, and the compilation time of the resulting Coq proof files is stored, to collect measurements on the efficiency of the different proof types. A single instance generated by the Haskell program creates a file for the Repeat Constructor, EAuto, Manual Checker, and Derive Checker proof types for both the let-binding desugaring and inlining translation relations. So a single generated instance covers 8 scenarios, which all will be timed individually.

The parameters I choose for the benchmark suite are the AST sizes 100, 1000, and 10000, and for all sizes, the benchmark was done 5 times. I used a simple Python script to plot the results of the suite as two bar graphs, one for desugaring and another for inlining, which can be seen in figures 3.1 and 3.2 respectively. Note: the y-axis in figure 3.2 uses a logarithmic scale to better visualize the difference in proof times.

We can see that for both the desugar and inline translation relations, all proof types have similar performance for ASTs of 100 nodes. These results are not that meaningful, they simply imply that ASTs of 100 nodes are too small for any differences in performance to manifest. If we look at the results for ASTs of 1000, and 10000

```
Lemma is_desugar_dec_derive : desugar generatedTerm desugaredTerm.
Proof. apply (sound 10000). reflexivity. Qed.
```

Listing 3.13: The Derive Checker proof type for the let-binding desugaring translation relation

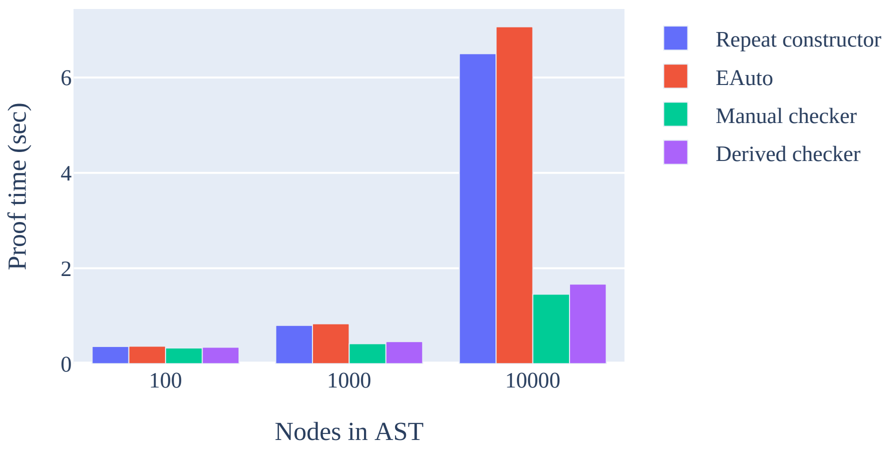

Figure 3.1: Benchmark results for proving desugaring

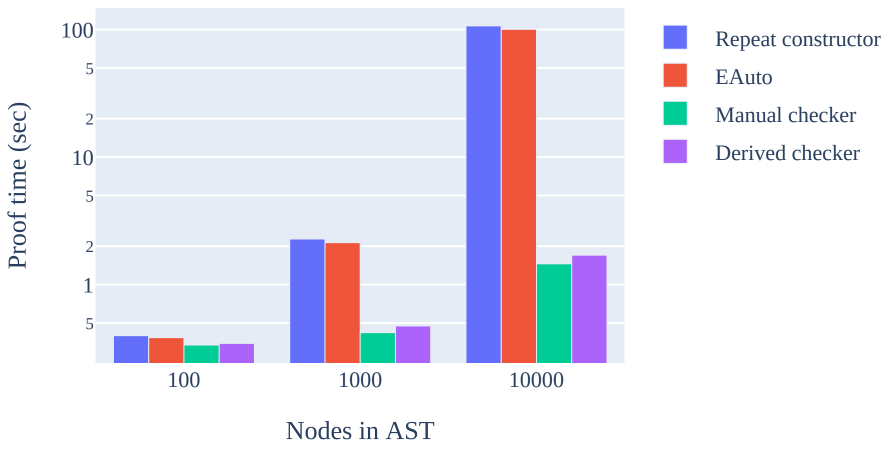

Figure 3.2: Benchmark results for proving inlining

nodes, instead, we can see that the proof times for Repeat Constructor and EAuto become increasingly slower compared to Manual Checker and Derive Checker.

Repeat Constructor and EAuto have very similar performance, where interestingly enough, for desugar EAuto seems to perform better, whereas for inline it is Repeat Constructor that has the better performance. Their proof times do not increase consistently for desugar and inline, for desugar the proof times only grow up to about 6 seconds, whereas for inline they end up taking as long as 100 seconds to finalize a proof for AST of size 10000.

The Manual Checker, and Derived Checker, however, perform very consistently over the different AST sizes for both desugar and inline. Their proof times increase ever so slightly for larger AST sizes, where for the size 10000 they both complete a proof in just under 2 seconds.

I am most interested in the performance of the derived checkers, since I want to know if it is a viable option to use derived checkers and soundness proofs in the Plutus Tx compiler. It is clear to see from the result that Derive checker performs substantially better than both Repeat Constructor, and EAuto, so over these proof types a derived checker has the upper hand. Furthermore, the derived checker only ends up being a fraction of a second slower than Manual Checker, which is a promising result for using the derived checker.

A translation relation might need to be rewritten, before a checker and soundness proof can be derived. A manual checker and soundness proof has to be implemented manually, and might require rewriting every time a translation relation changes slightly. The derived checker being a little slower than a manual checker could be a perfectly fine compromise to make, when the downsides of a derived checker are outweighed by the downsides of maintaining a manual checker.

3.3.4 Expected difficulties

There are some other problems and phenomena that are expected to occur, as we scale this case study to the Plutus Tx compiler. First, the paper states that the performance of the derived checker heavily depends on the order of the constructors of an inductive relation [19]. It will be interesting to see what the impact is of the order of constructors, and if there are particular orders that are favourable or orders that cause the execution time of a checker to become really slow. Second, the paper also describes producers that have to be derived for particular constructors. For those constructors, some variable is not bound in the output for the constructor, so a producer is used to derive instances of the existentially bound variable. Such a scenario has not occurred in the toy compiler translation relations, but for more complex translations it is expected that relations may be more complex, so it is more likely to include such an existentially bound variable. Deriving a producer for such cases is not expected to be difficult, it is the effect a producer has on the performance of a derived checker that is interesting. Essentially, a producer churns out instances of a variable that fit some criteria, and the checker then computes its goal for all instances. Usually, the checker only computes the goal for one set of instances for variables in a constructor, but when using a producer, this increases to possibly many more cases that need to be computed.

How often these scenarios arise, where deriving and using a producer is necessary, will have to be seen. In those cases, it will be interesting to see the impact on performance for the derived checker. If the performance using a producer is underwhelming, we can investigate if there are options available to address this. One solution, if possible, would be to rewrite the constructor such that using a producer is no longer necessary. This could be done be including more information that is passed around in an inductive relation, such as the environment in the inline relation, so that the value of the existentially bound variable is now provided by the compiler.

Chapter 4

Scaling up to Plutus

The Plutus compiler is substantially more complex than the toy compiler we previously targeted, predominantly caused by the Plutus Tx language being more expressive than the language of the toy compiler. So, scaling up the approach used with QuickChick in the toy compiler to the Plutus compiler was not expected to be without issues.

Most of the issues encountered could be fixed by simple rewrites or restructures in the constructors of a relation's definition. However, several translation relations in the Plutus compiler are using mutual recursion, which posed a new problem entirely, since QuickChick's derivation algorithm does not support mutually inductive relations. Fixing QuickChick's incompatibility issues with mutually inductive translation relations required a rather involved approach of introducing a non-mutually inductive proxy, and unifying the proxy with the original translation relation with the necessary Coq proofs. Then, QuickChick's derivation algorithm could be applied to the proxy instead, which results in computations that can be used for the mutually inductive relation.

4.1 Core Plutus definitions

First, consider some core definitions for the AST used in the Plutus compiler, which is the root of most of the issues that occur when using QuickChick to derive computation over Plutus' translation relations. The two definitions `kind` and `ty`, seen in listing 4.1, make up the types of a Plutus Tx program. The inductive `kind` denotes the arity of a type, and is needed to be able to support higher-order types. The definition `ty` represents the different constructs that can be used in a Plutus type, such as universal quantification, fix points, or abstraction and applications over some type variable.

```
Inductive kind :=
  | Kind_Base : kind
  | Kind_Arrow : kind → kind → kind.

Inductive ty :=
  | Ty_Var : tyname → ty
  | Ty_Fun : ty → ty → ty
  | Ty_IFix : ty → ty → ty
  | Ty_Forall : binderTyname → kind → ty → ty
  | Ty_Builtin : ·some typeIn → ty
  | Ty_Lam : binderTyname → kind → ty → ty
  | Ty_App : ty → ty → ty.
```

Listing 4.1: The Definitions of Types for Plutus Tx

Completing the Plutus AST are the definitions of terms and bindings in listing 4.2. The definitions `term` and `binding` are defined mutually inductively so they can refer to one another, since a term can have bindings and a binding can include a term. Most lambda concepts in a Plutus term will be familiar to those who have used a functional programming language, such as abstraction and application over a variable, or non-recursive and recursive let-bindings. A note on the latter constructs, whether a let-binding is non-recursive or recursive is encoded using the flag `Recursivity`, which is a boolean type denoting the two cases. Also included in Plutus types and terms are `Ty_Builtin`, `Constant`, and `BuiltIn`, which make use of `some typeIn`, `some valueOf`, and `DefaultFun` respectively, and can be used as a bridge to use native Coq types, values, or function definitions.

```
Inductive term :=
  | Let      : Recursivity → list binding → term → term
  | Var      : name → term
  | TyAbs    : binderTyname → kind → term → term
  | LamAbs   : binderName → ty → term → term
  | Apply    : term → term → term
  | Constant : ·some valueOf → term
  | Builtin  : DefaultFun → term
  | TyInst   : term → ty → term
  | Error    : ty → term
  | IWrap    : ty → ty → term → term
  | Unwrap   : term → term
with binding :=
  | TermBind : Strictness → vdecl → term → binding
  | TypeBind : tvdecl → ty → binding
  | DatatypeBind : dtdecl → binding.
```

Listing 4.2: The Definitions of Terms for Plutus Tx

As stated before, QuickChick does not support mutual induction, however, the mutual inductive definition of terms and bindings in a Plutus AST does not directly cause issues for QuickChick. The issues arise when defining propositions or relations over mutually inductive types, since it is often easiest to define such propositions mutually inductively as well. For example, if two Plutus terms are equal, any underlying bindings need to be equal, and when two bindings are equal, any underlying terms need to be equal as well. Thus it is natural, but not strictly necessary, to define equality over two Plutus terms or bindings in a mutually inductive manner, so that they can make use of each other's definitions.

The translation relations in the Plutus project are defined over the mutually inductive Plutus terms, so similarly to defining equality, these translation relations are often defined mutually inductively as well. Such translation relations of the Plutus compiler are what we are interested in deriving computation over with QuickChick, so now it is problematic that QuickChick does not support mutual induction.

4.2 Mutually inductive relations

The authors of the QuickChick framework made implementation decisions that make it unable to derive anything for mutually inductive relations. In their paper, they state that they relied on Coq's typeclasses to aid in proof automation, but since typeclasses cannot be mutually inductive, neither can any computation that QuickChick derives over a relation [19]. Whilst I made some small adjustments to the QuickChick framework to support unaccounted-for cases, it was far beyond the scope of my research to refactor the framework to get rid of the uses of typeclasses.

Since altering the QuickChick framework to make it work for mutually inductive relations was infeasible, the other logical approach was to instead target the mutually inductive relations. Rewriting a mutual inductive relation in place was given some thought, since if a relation was not mutually inductive in the first place, then QuickChick could simply derive the desired computation over the relation. However, the translation relations are used throughout many other parts of the Plutus project, so it is much more preferable to preserve the original mutual inductive relation, to prevent breaking proofs defined elsewhere that would then need to be fixed.

Instead of rewriting a relation and overwriting the original definition, it is also possible to reformulate the relation to avoid mutual recursion in the same manner, and use it alongside the original definition. With the necessary proofs to unify the original and derived relations, it is then possible to use the new relation for our purposes. Consider having some proposition described by two relations, one mutually inductive, and another non-mutually inductive. If the two relations are equivalent, it should be possible to finalize a semantic equivalence proof in Coq over the two relations. Then, if QuickChick is used to derive a decision procedure and its soundness proof over the non-mutually inductive relation, this computation and proof can then indirectly be used for the mutual inductive relation as well. When the aforementioned decision procedure returns `Some true`, the soundness proof implies that the non-mutually inductive relation holds, and the semantic equivalence proof can be used to determine that the mutually inductive relation then holds as well. Thus, we would have our hands on a decision procedure derived by QuickChick over some 'proxy' relation, that when composed with the necessary soundness and semantic equivalence proofs, can act as a decision procedure for the original mutually inductive relation.

For smaller and simpler relations in the Plutus project, it is easy enough to derive some non-mutually inductive proxy manually. Unfortunately, doing so for the larger translation relations in the Plutus compiler becomes a cumbersome and error-prone task. This raises the question of whether we can first come up with some generic algorithm for deriving non-mutually inductive proxies over mutually inductive relations, and secondly, whether this algorithm can then be performed programmatically in Coq, so that it is easy to apply over the translation relations of the Plutus compiler.

To find a working method for deriving non-mutually inductive proxies, consider what it means for a relation to be mutually inductive, to understand how such a relation might be altered to 'remove' its mutual induction. The following two properties describe what it means for a definition of relations to be mutually inductive:

1. A set of inductive relations describing several properties, such as one for equality over terms, and another for equality over bindings.

2. The premises in the relations (possibly) reference all the other relations in the mutually inductive definition.

Using this intuition, I considered two different approaches for deriving non-mutually inductive proxy relations, which target the two aforementioned properties of such relations.

The first method I call 'disjoint proxies', which uncouples the relations in a mutually inductive definition, by removing any premises that reference any of the other relations. To preserve the semantics of the now non-mutually inductive relations, they still need to depend on each other's definitions by using some additional dependency, which can be supplied to the relation as additional arguments. The definitions of the disjoined proxies stay very similar to the original mutually recursive definition, which was expected to make programmatically deriving these proxies an easier task. However, I never got to programmatically derive the disjoined proxies, since the additional dependencies introduced in the proxies posed a problem which I did not manage to solve.

The second approach I call 'collapsed tagged proxy', which instead collapses all the constructors of the relations into one bigger non-mutually inductive relation. With the use of tags, the necessary parts of the constructors can be tagged to preserve the semantics of the original mutually inductive relations. Over collapsed tagged proxies I was able to derive decision procedures and soundness proofs, such that these could be used over the original mutually inductive relations. Thus, this approach was implemented in a derivation algorithm, such that collapsed tagged proxies can be derived programmatically over mutually inductive relations.

4.2.1 Disjoint proxy

My first attempt to derive a non-mutually inductive proxy was to uncouple the several relations in a mutually inductive definition, by removing their references of each other. However, these cross-uses refer to the other relations which represent other propositions that need to hold, so simply removing the cross-uses would alter the meanings of all the relations. To preserve this meaning, the mutual dependencies can be abstracted out by introducing additional dependency arguments, where the dependencies entail the same propositions as the another relations, so that any references to such relations can be replaced by the respective dependency. The intuition of this approach was that it would keep the definitions of the relations as close as their original shape, which later on should make it easier to derive the disjoint proxies programmatically.

```
Inductive Even : nat → Prop :=
  | E0 : Even 0
  | ES : forall n, Odd n → Even (S n)
  | ESS : forall n, Even n → Even (S (S n))
with Odd : nat → Prop :=
  | OS : forall n, Even n → Odd (S n)
  | OSS : forall n, Odd n → Odd (S (S n)).
```

Listing 4.3: The mutually inductive definition of Even and Odd for natural numbers

To illustrate what a disjoint proxy would look like, first consider the simple mutually inductive relations `Even` and `Odd` in listing 4.3, which describes when a natural numeral, or rather Peano numeral, is even or odd. The constructor `E0` denotes that the natural number 0 is trivially even. The constructors `ES` and `ESS` respectively state that if the number `n` is odd, then its successor is even, and if `n` is even, then the successor of the successor of `n` is even. Analogously the constructors `OS` and `OSS` describe the same for oddness.

The definitions of Even and Odd are mutually recursive, since in the premises of the constructors ES and OS they use each other's definitions. Using the aforementioned approach, we want to replace these references to each other with some dependencies that entail the same proposition, but are supplied as an additional argument to the relation instead. I considered two types of dependencies that can be supplied through an argument, a proposition over the same arguments as the references that it will be replacing, or a decision procedure of such a proposition, which thus will also have the same arguments.

Propositions as dependencies

```
Inductive EvenProp (odd_P : nat → Prop) : nat → Prop :=
  | E0 : EvenProp odd_P 0
  | ES : forall n, odd_P n → EvenProp odd_P (S n)
  | ESS : forall n, EvenProp odd_P n → EvenProp odd_P (S (S n)).

Inductive OddProp (even_P : nat → Prop) : nat → Prop :=
  | OS : forall n, even_P n → OddProp even_P (S n)
  | OSS : forall n, OddProp even_P n → OddProp even_P (S (S n)).
```

Listing 4.4: The disjoint definition of Even and Odd using a proposition as a dependency

Using the first type of dependencies for the relations Even and Odd resulted in the definitions in listing 4.4, thus, what the disjoint proxies would look like if you use propositions as dependencies. Focussing on the relation EvenProp, you can see it is given the additional argument odd_P of the type nat → Prop. The type of the dependency means that odd_P is either a function or relation that, when applied to some natural number, results in a proposition that either holds or does not. Assuming that odd_P factually describes the same proposition as the relation Odd, we can replace the uses of Odd with the dependency odd_P, which will give us the relation EvenProp with the same semantics as the relation Even. If you do the same for the dependency even_P in the relation Odd, this results in the definition of the relation OddProp.

Unfortunately, QuickChick's derivation algorithm does not work over disjoint proxies that use propositions as dependencies. Remember, we wanted to derive computations with QuickChick over the non-mutually inductive proxy, together with the necessary proofs, so the computation can be used for the original mutually inductive relation. When QuickChick encounters a proposition in the premise of a constructor, it usually attempts to find the respective derived computation for that premise, and use that in the computation it is currently deriving. However, QuickChick does not currently support the use of proposition in premises, which are supplied to the relation through an argument, and will simply throw an error that such cases are unhandled.

Decision procedures as dependencies

Since using propositions as dependencies is not supported by QuickChick's implementation, this leads us to use decision procedures as dependencies instead. These decision procedures can be the respective procedures of the propositions we would have used as dependencies before, and hopefully, we can use QuickChick to derive these decision procedures.

```
Inductive EvenDec (odd_dec : nat → option bool) : nat → Prop :=
  | E0 : EvenDec odd_dec 0
  | ES : forall n, odd_dec n = Some true → EvenDec odd_dec (S n)
  | ESS : forall n, EvenDec odd_dec n → EvenDec odd_dec (S (S n)).

Inductive OddDec (even_dec : nat → option bool) : nat → Prop :=
  | OS : forall n, even_dec n = Some true → OddDec even_dec (S n)
  | OSS : forall n, OddDec even_dec n → OddDec even_dec (S (S n)).
```

Listing 4.5: The disjoint definition of Even and Odd using a decision procedure as a dependency

If we apply this second approach to the relations Even and Odd we get the relations in listing 4.5. The decision procedure odd_dec is given as an argument to the relation EvenDec, which should be a valid decision procedure for the original Odd relation. A decision procedure returns Some true, when the proposition it describes holds, thus we replaced any previous references to Odd with an equality check that odd_dec should return `Some` `true`. Again, analogously we replaced references to `Even` with equality checks on `even_dec` to get the `OddDec` relation.

The rewritten propositions in the constructors of `EvenDec` and `OddDec` have now become equality checks over optional boolean values. For checking if equality checks hold, QuickChick makes use of the typeclass `DecEq` instead of the `DecOpt` typeclass mentioned before. If we now want to derive a decision procedure over `EvenDec` and `OddDec`, QuickChick will require a `DecEq` over the optional booleans in the equality, instead of a `DecOpt` instance over some proposition, as in the other approach. QuickChick includes several `DecEq` instances already for Coq's basic types, such as a `DecEq` instance for optionals and booleans. Thus, QuickChick can successfully derive the decision procedures and respective soundness proofs over the disjoint proxies for the relations `EvenDec` and `OddDec`.

Resolving the dependencies

The dependency added to the disjoint proxies solved the issue of not being able to use QuickChick's derivation algorithm, but it now presents us with a new problem. We want to define and prove that if the proxy holds, then so does the original mutually inductive relation. Which value should be used for the dependencies in the proxies? Similarly, if we want to know if the original mutually inductive relation holds for some arguments, and we want to use the decision procedures derived from the proxies to determine this, then what value do we use for the additional dependency argument that is now included in the decision procedures? In conclusion, we need to resolve the dependencies in the disjoint proxy or the decision procedures of those proxies, before we can use either of them in terms of the original mutually inductive relation.

It is possible to resolve the dependencies in the decision procedures, by defining additional mutually recursive fixpoint functions, where these functions are recursively used as the missing dependencies. To elaborate, the decision procedure for the `EvenDec` relation includes the argument `odd_dec`, and since this should be a valid decision procedure for the `Odd` relation, we want to use the decision procedure for the `OddDec` relation. However, the decision procedure derived over `OddDec` has the type (`nat` → `option` `bool`) → `nat` → `option` `bool`, so we need to resolve the first argument representing the dependency, before we can use this decision procedure in the other decision procedure for the `EvenDec` relation. If we want to resolve the dependency in the decision procedure for `OddDec`, we want to use the decision procedure derived over the `EvenDec` relation, but then we first need to resolve the dependency in that decision procedure, and thus we have gone full circle.

```
Fixpoint even_dec (s : nat) (n : nat) : option bool :=
  let odd := noneIf0 odd_dec s
  in @decOpt (EvenDec odd n) (DecOptEvenDec odd n) s
with odd_dec (s : nat) (n : nat) : option bool :=
  let even := noneIf0 even_dec s
  in @decOpt (OddDec even n) (DecOptOddDec even n) s.
```

Listing 4.6: Using a fixpoint function to tie up the knot on the dependencies of the decision procedures of the `Even` and `Odd` relations

By wrapping both decision procedures in an additional mutually recursive fixpoint function, we tie the knot on this circular dependency, and acquire two new functions that no longer require an additional decision procedure as an argument. Listing 4.6 shows the two mutually recursive fixpoint functions `even_dec` and `odd_dec`, which have the same arguments as the decision procedures of `EvenDec` and `OddDec` minus the dependencies, but with an additional size parameter. The `even_dec` function is defined to be `decOpt` (`EvenDec odd n`) (`DecOptEvenDec odd n`), which is a notation that refers to the decision procedure derived over `EvenDec` with the arguments `odd` and `n` applied. The argument `odd` is defined using a let-binding, and refers mutually recursively to the fixpoint function `odd_dec`, wrapped by the function `noneIf0`. The function `noneIf0` returns a partially applied decision procedure of the type `nat` → `option` `bool`, which will return `None` if the size argument `s` is 0, otherwise it returns `odd_dec` with the size value `s` − 1 applied. Thus, the type of the function `odd` is `nat` → `option` `bool`, and it can be used as a decision procedure for the dependency in the decision procedure of the `EvenDec` relation. Using the `noneIf0` is necessary to allow the recursive use of the decision procedures of `EvenDec` and `OddDec` in a circular way, whilst still guaranteeing that the mutually recursive function is total and will return a decidable result, as is required by Coq for any function. The two functions `even_dec` and `odd_dec` can be used to define two `DecOpt` instances for the original mutually inductive relations `Even` and `Odd`. These `DecOpt` instances simply require an underlying function implementation of the type `nat` → `nat` → `option` `bool`, where the first `nat` is the size parameter, thus we can simply delegate to the functions `even_dec` and `odd_dec`.

Whenever we derive any computation directly with QuickChick, it is also possible, and frankly necessary to derive a soundness proof over the computation before the derived computation can be useful. Whether a derived decision procedure returns true or false is meaningless, if we do not also have a valid Coq proof that such results imply whether the respective relation holds or not. Unfortunately, when we use the mutually recursive definitions `even_dec` and `odd_dec` as decision procedures for `Even` and `Odd`, the computation does not follow the structure expected by the QuickChick Ltac tactic used to derive soundness proofs, so we cannot use it to derive a soundness proof, thus we have to define and prove the soundness proofs manually. There are several approaches to relating the mutually recursive functions `even_dec` and `odd_dec` with the relations `Even` and `Odd` using a soundness proof, which all again require a dependency to be resolved, but this time in soundness proofs instead of the decision procedures.

We already have the following definitions: a decision procedure over `EvenDec` and its respective soundness proof, a decision procedure over `OddDec` and its respective soundness proof, and the fixpoint functions `even_dec` and `odd_dec` that resolve the dependencies over the aforementioned two decision procedures. The functions `even_dec` and `odd_dec` under the hood use decision procedures derived over the relations `EvenDec` and `OddDec`, thus we want to use the soundness proofs of the decision procedures, to prove that `even_dec` and `odd_dec` are sound in terms of the relations `EvenDec` and `OddDec`. Furthermore, we want to extend these soundness proofs so that they proof soundness between `even_dec` and `Even`, and between `odd_dec` and `Odd`.

```
Lemma EvenDec_Even_sound odd_dec n s :
  (forall n' , odd_dec n' = Some true → Odd n') →
  @decOpt (EvenDec odd_dec n) (DecOptEvenDec odd_dec n) s = Some true → Even n.

Lemma OddDec_Odd_sound even_dec n s :
  (forall n' , even_dec n' = Some true → Even n') →
  @decOpt (OddDec even_dec n) (DecOptOddDec even_dec n) s = Some true → Odd n.
```

Listing 4.7: The soundness proof between `EvenDec` and `Even`, and between `OddDec` and `Odd`, both with an additional soundness proof as a dependency

Let us consider a soundness proof between the decision procedure derived over `EvenDec` and `Even` relations first, which is the first definition seen in listing 4.7, of which the proof is left out for brevity. The decision procedure, referred to with `decOpt`, can only be sound with the `Even` relation if its dependency has been resolved, and this dependency is sound in terms of the `Odd` relation. Similarly to how dependencies were resolved in the decision procedures, we can resolve the dependency of the relation `EvenDec` by using the decision procedure of `OddDec` with its dependency already resolved. This means that to prove soundness between the decision procedure of `EvenDec` and the `Even` relation, we need a soundness proof between the decision procedure of `OddDec` and `Odd`.

The soundness between the decision procedure of `OddDec` and `Odd` is the second definition that can be seen in listing 4.7. Analogously to before, to prove that the decision procedure of `OddDec` and `Odd` are sound, you need a soundness proof between the decision procedure of `EvenDec` and `Even`. This leaves you with two new soundness proofs, and both have another soundness proof as a dependency. Thus, before these soundness proofs can be used the way they were intended, the soundness proof dependencies have to be resolved first.

Unable to resolve soundness-proof dependencies

I did not manage to come up with another way of defining the necessary soundness proofs for the decision procedures derived over the disjoint proxies of the `Even` and `Odd`. So I did not see another way then to resolve the soundness proof dependencies, to acquire the necessary soundness proofs over the disjoint proxies. I tried to define mutually inductive soundness proofs, where I simultaneously define and prove soundness between `even_dec` and `Even`, and between `odd_dec` and `Odd`. This either lead to situations where I had a handful of cases left to prove, but ended up getting stuck, or where I supposedly had a full proof, but Coq did not accept the proof since it was not a valid decreasing proof. The latter happens for similar reasons as why a fixpoint function needs a decreasing argument, if recursive uses of a proof are not strictly decreasing, Coq can not guarantee that it will be a decidable proof.

I exhausted all the options that I could come up with for finalizing the correct soundness proofs over decision procedures over the disjoint proxies. However, I still had other options to consider of rewriting the mutually inductive relations into non-mutually inductive relations. So, I opted to leave the disjoint proxies for what they were, and focus on other rewrite approaches in the hopes that they would have more favourable results instead.

4.2.2 Collapsed tagged proxy

Instead of splitting a mutually inductive relation up into multiple non-mutually inductive relations, such as done in the disjoint proxy approach, the intuition behind the collapsed tagged proxy is that these relations can be merged into one single relation that is non-mutually inductive. As the name suggests, the several relations in a mutually inductive definition are collapsed into one single relation, and their arguments are tagged such that the semantics of the relations are preserved in the now non-mutually inductive collapsed tagged proxy. Any former mutually inductive references to other relations will have changed into recursive references to the collapsed tagged proxy itself, which means the proxy is no longer mutually inductive, and we should be able to derive computation over the collapsed tagged proxy with QuickChick.

We use the mutually inductive relations `Even` and `Odd` defined in listing 4.3, as an example of what a derivation of a collapsed tagged proxy would look like, hereafter just referred to as the proxy. The definition of the proxy depends on another definition, namely, the definition of the tag it uses, thus, the tag of the proxy is the first thing that is derived over a mutually inductive relation.

```
Inductive Even_proxy_tag : Set :=
  | Odd_tag (n : nat)
  | Even_tag (n : nat).
```

Listing 4.8: The tag derived for the `Even` and `Odd` relations

Listing 4.8 shows the tag type that is derived for the mutually inductive `Even` and `Odd` relations. The tag is a type that itself does not have any further arguments, and includes a constructor for each relation in a mutually inductive definition, where the constructor has the same arguments as the relation that it represents. Both the relations `Even` and `Odd` have a single natural number as an argument, so the constructors `Even_tag` and `Odd_tag` have the same arguments as these relations respectively.

The purpose of this tag type is that it can encode mutually inductive relations into one single non-mutually inductive relation. To elaborate on how this encoding works, consider the following constructor in the `Even` relation.

```
  ES : forall n, Odd n → Even (S n)
```

Since the constructors of the tag have matching arguments with the relations they represent, any uses of the mutually inductive relations can be replaced with their respective constructor. Thus, any uses of `Even` and `Odd` in the constructor above can be replaced with `Even_tag` and `Odd_tag` respectively. This will not result in a valid constructor, since the constructors `Even_tag` and `Odd_tag` cannot be used as premises or results of a constructor by themselves. However, let us say we also have some relation `R` that represents a proposition with one single tag as an argument. Then, if we replace the uses of `Even` and `Odd` with the respective constructor of the tag, and wrap the tag with the relation `R`, we get the following constructor:

```
  ES_proxy : forall n, R (Odd_tag n) → R (Even_tag (S n))
```

This constructor creates the instance `R` (`Even_tag` (`S n`)), so an instance of the relation `R`, which means this constructor could be valid a constructor for relation `R`. Furthermore, if we apply the same manner of rewriting over all the constructors of the relations `Even` and `Odd`, they would all result in valid constructors for the relation `R`. If we rewrite all the constructors of the relation `Even` and `Odd`, and add them all together under some relation `Even_proxy`, instead of `R`, this will result in the definition in listing 4.9, which is the definition of the collapsed tagged proxy of `Even` and `Odd`.

It is important to understand for the definition of this collapsed tagged proxy, that a constructor with `Even_tag` in its result can only have originated from a constructor of the `Even` relation. Furthermore, all constructors of the `Even` relation will always include `Even_tag` in its results, when rewritten to a constructor of the proxy. So all the constructors of the `Even` relation are encoded in the proxy by exactly those relations with `Even_tag` in their result, and vice versa for the `Odd` relation and `Odd_tag`.

Now consider some premise `Odd n`, such as in the `ES` constructor shown above. This premise will only hold if the current instance in question of `Odd n` can be constructed using the constructors of the `Odd` relation. When deriving the proxy, any occurrence of `Odd n` will be rewritten into `Even_proxy` (`Odd_tag n`). Using the same intuition as before, any instance of the relation `Even_proxy` (`Odd_tag n`), can only be constructed using a constructor originating from the `Odd` relation. Presumably, this means the same semantics described by the constructors of the `Even` and `Odd` relations, should be captured by the proxy in listing 4.9, and thus this should be a valid collapsed tagged proxy for `Even` and `Odd`.

```
Inductive Even_proxy : Even_proxy_tag → Prop :=
  | E0_proxy :
      Even_proxy (Even_tag 0)
  | ES_proxy : forall n : nat,
      Even_proxy (Odd_tag n) →  Even_proxy (Even_tag (S n))
  | ESS_proxy : forall n : nat,
      Even_proxy (Even_tag n) → Even_proxy (Even_tag (S (S n)))
  | OS_proxy : forall n : nat,
      Even_proxy (Even_tag n) → Even_proxy (Odd_tag (S n))
  | OSS_proxy : forall n : nat,
      Even_proxy (Odd_tag n) → Even_proxy (Odd_tag (S (S n))).
```

Listing 4.9: The collapsed tagged proxy for the `Even` and `Odd` relations

Proxy soundness

Instead of assuming that the proxy indeed describes the same propositions as the relations it was derived from, we can define and prove the necessary proofs in Coq for soundness and completeness to establish this formally.

```
Lemma Even_proxy_sound : forall tag : Even_proxy_tag,
  Even_proxy tag → match tag with
                    | Odd_tag n → Odd n
                    | Even_tag n → Even n
                    end.
Proof. intros tag H. induction H; constructor; assumption. Qed.
```

Listing 4.10: A manual Coq soundness proof for the collapsed tagged proxy derived for the `Even` and `Odd` relations

The definition and proof of the soundness of the proxy in terms of the relation `Even` and `Odd` is shown in listing 4.10. This definition states that if the proxy holds over a tag where the tag is `Even_tag n` then `Even n` should hold, and if the tag is `Odd_proxy n` then `Odd n` should hold. The definition also shows the sequence of Coq tactics that can be used to come to a full proof of this soundness definition.

Conveniently, soundness over any collapsed tagged proxy has a generic solution. We have the assumption that the proxy holds over some tag, and depending on which constructor was used for the tag, we need to prove that either `Even` or `Odd` holds. By applying induction over the proxy, with the `induction` tactic, Coq will create a case for us to prove for each possible constructor of the proxy, and it will update the assumptions and goal we have according to what can be concluded from the respective constructor. Furthermore, for every constructor with a recursive use of the proxy relation, Coq will supply an induction hypothesis in the assumption of the case of the respective constructors.

Let us consider the case of the constructor `ES_proxy` of the proxy. Firstly, this constructor could only have been used if all its premises hold, so the premise `Even_proxy` (`Odd_tag n`) is added to the assumptions. Secondly, this constructor has as result `Even_proxy` (`Even_tag` (`S n`)), so we can now assume that the tag in the assumption was constructed with `Even_tag` (`S n`). Knowing that the tag was constructed using `Even_tag` (`S n`), the match statement in the soundness proof can be evaluated, which will now imply that `Even` (`S n`) should hold, so this is now the new goal we need to prove for the case of the constructor `ES_proxy`. Lastly, since the premise of the constructor is a recursive use of the proxy relation, Coq will inject an induction hypothesis into the assumptions, which implies soundness of the proxy in terms of the `Even` and `Odd` relations, so essentially the soundness we are currently proving.

For the case of the `ES_proxy` relation, we now have the information that `Even_proxy` (`Odd_tag n`) holds, the proxy is sound in terms of the `Even` and `Odd` relations, and the goal we need to prove is `Even` (`S n`). Using Coq's tactic `constructor`, Coq will try and find a constructor with which the current goal can be constructed, and apply this constructor. Since our goal is `Even` (`S n`), Coq will find the constructor `ES`, apply this constructor, and our goal will become `Odd n`. Note, that the constructor applied here, `ES`, is the constructor from which `ES_proxy` is derived, for which we are currently proving the case. Then, the Coq tactic `assumption` tries to finalize the current goal with the information that is currently available in the assumption. We have the assumption that `Even_proxy` (`Odd_tag n`) holds, and if we apply the induction hypothesis to this assumption, it will give us the assumption that `Odd n` holds. Having `Odd n` in both the assumptions and as the goal, we can simply apply the assumption to finalize the proof for the current case

For all cases that Coq requires us to prove after applying induction over the proxy, we can finalize the proof for the different cases by using the tactics `constructor` and then `assumption`. Thus, the soundness proof in listing 4.10 can be proven by only using the tactics `induction`, `constructor`, and `assumption`.

Proxy completeness

We can also define the completeness of the proxy in terms of the relations `Even` and `Odd`, however, it is substantially more complex to prove completeness than soundness. This extra complexity is due to that Coq does not play nice with a match statement over a tag in the premises of the completeness proof. When we use the induction tactic in the soundness proof, Coq supplies us with an induction hypothesis where necessary, but in the completeness proof it fails to do so, and without this induction hypothesis we cannot finalize the completeness proof.

To get around this quirk, we can define completeness using auxiliary mutually recursive functions, which instead of using induction hypotheses, can make use of recursive calls to each other. Then, we can define our completeness proof, and prove it by simply delegating to the auxiliary function we defined. Using this approach for completeness of the proxy in terms of the `Even` and `Odd` relation, results in the definitions in listing 4.11.

```
Fixpoint proxy_complete_even n (H : Even n) : Even_proxy (Even_tag n)
with proxy_complete_odd n (H : Odd n) : Even_proxy (Odd_tag n).
Proof.
  + inversion H; subst.
    - constructor.
    - constructor. apply proxy_complete_odd. apply H0.
    - apply ESS_proxy. apply proxy_complete_even. apply H0.
  + inversion H; subst.
    - constructor. apply proxy_complete_even. apply H0.
    - apply OSS_proxy. apply proxy_complete_odd. apply H0.
Qed.

Lemma Even_proxy_complete : forall tag : Even_proxy_tag,
  match tag with
  | Odd_tag n → Odd n
  | Even_tag n → Even n
  end → Even_proxy tag.
Proof.
  destruct tag; [apply proxy_complete_odd | apply proxy_complete_even]; assumption.
Qed.
```

Listing 4.11: The type of a Coq completeness proof for the collapsed tagged proxy derived for the `Even` and `Odd` relations

This completeness proof still uses the same intuition as the soundness, where we do induction over `Even` or `Odd`, apply some constructor over our goal, and finalize the proof using the information in the assumptions. However, instead of this being done in the proof directly, this is now captured by the auxiliary function that we defined.

Unfortunately, the several cases we have to prove after induction cannot be proven using the same set of tactics, such as in the soundness proof. This is due to that when we use the tactic `constructor`, multiple constructors match the goal we need to prove, and Coq incorrectly applies the wrong constructor. This can simply be solved by applying the right constructor manually, for example using `apply ESS_proxy`, however, this does require us to write out the proof in full, which is more work.

Another method that was considered here to reduce the complexity of the proof, was to make use of Coq's auto tactic. This tactic applies backtracking, so even if it applies the wrong constructor, it can simply backtrack and use the right constructor instead. The problem with the `auto` tactic is that it uses the recursive calls to the other auxiliary functions too eagerly.

The definitions `proxy_complete_even` and `proxy_complete_odd` are immediately added to the assumptions when defining a proof for these auxiliary functions. However, we cannot immediately use a recursive call, since then we do not call the function with reducing arguments, as is required in a fixpoint function of Coq. This is something the `auto` tactic does not consider, thus it derives an invalid proof that is not strictly decreasing. In our manual proof, the use of the recursive calls works correctly, since we first apply a constructor, which means the arguments in the recursive call have been reduced.

By defining and proving soundness and completeness for the collapsed tagged proxy, we establish that it is semantically equivalent to the mutually inductive relations it was derived from. Thus, we can apply QuickChick's derivation algorithm to the proxy to derive its decision procedures and the relevant soundness proofs, and use these computations for the mutually inductive relations instead by making use of the soundness and completeness proofs.

4.2.3 MetaCoq proxy derivation automation

The translation relations for the Plutus compiler are the mutually inductive relations we want to derive decision procedures over with QuickChick, and thus have to derive collapsed tagged proxies for them as well. However, there are a substantial amount of these translation relations, so manually deriving the proxies for these relations is a lot of work, and the extra definitions and proofs will clutter up the files to which they have to be added. Preferably, we want to programmatically derive the collapsed tagged proxy of some mutually inductive relation using some derivation algorithm. This would reduce the amount of manual labour, reduces complexity and clutter in the workspace of the project, and an algorithm is unlikely to introduce any errors in the definition it derives if the algorithm is implemented correctly.

So, to make working with the collapsed tagged proxies easier, a derivation algorithm was implemented with MetaCoq for deriving these proxies. MetaCoq is a meta-programming library that can be used to apply syntax transformations over ASTs of already existing Coq definitions. Thus, we can derive the tag type, collapsed tag proxy, and even parts of the soundness or correctness proofs automatically, if we implement this as syntax transformations in MetaCoq.

The MetaCoq derivation algorithm of the tag type, and the collapsed tagged proxy follow the same approach as was explained before. We first quote the definition of the mutually recursive relation we are deriving the proxy for, so that we acquire the AST representation of this relation. From this AST we can retrieve the type signature of the underlying relations, which is sufficient to then be able to derive the types of the arguments of the constructors of the tag type. The definition of the tag type can then be unquoted, which will cause the definition to be injected into the current Coq context, so that it can be used as if it was explicitly defined there.

To derive the collapsed tagged proxy, we again iterate over the relations in the AST of the mutually inductive relation. We consider all the different constructors of all the different relations, and we rewrite any previous references to the mutually inductive relations. These mutually inductive references can be in-place replaced with a reference to the proxy with the right constructor of the tag as an argument, such that the constructor still entails the same semantics as the constructor it was derived from. All the rewritten constructors can then be added to a new relation definition for a proxy, which can then be unquoted to add the proxy definition to the Coq context.

It is not as straightforward to derive full soundness or completeness proofs with MetaCoq, in comparison with the derivation of an inductive type or relation. The definitions of types and relations only need to have a valid syntax: We do not have to convince Coq further that such definitions are necessarily correct. For proof definitions, we also have to convince Coq that what the proof implies is correct within Coq's system of rules, which is done by making use of the tactics that Coq includes. Thus, if we want to derive a full soundness or completeness proof, we are also required to derive the proof that is usually finalized using tactics, which is not trivial to achieve with MetaCoq.

We can instead solely derive the type of such a proof, and unquote it to add the type as a definition to the current context. Such a type definition does not describe a proof that should hold, yet, thus Coq does not require a full proof using tactics, which means the type can be derived similarly as the tag a proxy derived before. Then the proof that the type describes can be defined and proven more conveniently in future, by simply using the already derived proof type definition, and then manually finalizing the proof with the necessary tactics.

```
Definition Even_proxy_sound_type : Prop :=
  forall tag : Even_proxy_tag,
    Even_proxy tag → match tag with
                     | Odd_tag x → Odd x
                     | Even_tag x → Even x
                     end.
```

Listing 4.12: The type of the soundness proof derived by the MetaCoq derivation algorithm for decision procedures derived over the `Even` and `Odd` relations

Listing 4.12 shows the type of the soundness proof that the MetaCoq derivation algorithm derives over the mutually inductive `Even` and `Odd` relations. If you look back at the manual soundness proof for the `Even` and `Odd` relations in listing 4.10, you can see that the types in both of these definitions exactly match. We only have to supply Coq with a proof, if the type is used in a proof definition, such as a Lemma, Theorem, or Corollary etc.. Thus, we can derive the type of the soundness proof for some proxy with MetaCoq, and finalize the soundness proof afterwards by using it in a proof definition.

```
MetaCoq Run (deriveCTProxy Even).

Lemma Even_proxy_sound : Even_proxy_sound_type.
Proof. deriveCTProxy_sound even_hint_db. Qed.
```

Listing 4.13: The soundness proof for the collapsed tagged proxy derived from the `Even` and `Odd` relations

In listing 4.13 first can be seen, how the MetaCoq derivation algorithm `deriveCTProxy` can be applied to derive the tag, collapsed tagged proxy, and soundness type over the `Even` and `Odd` relations. After running the algorithm using `MetaCoq Run`, the definitions of the tag, proxy, and soundness proof type will have been added to the Coq context, and hereafter can be used in other definitions.

The listing then shows, how the soundness type can then be used in a full soundness proof for the derived proxy. The proof is defined by simply giving the proof a name, and then using the derived soundness type as a type. Since we defined a `lemma` here, Coq now does require us to give a full proof for soundness. Luckily, as explained before, soundness for all collapsed tagged proxies can be proven using a generic approach.

The Ltac `deriveCTProxy_sound` is defined alongside the MetaCoq derivation algorithm, and applies the generic solution for proving the soundness of collapsed tagged proxies. This Ltac makes use of Coq's `auto` tactic for automatic proof search, and for this tactic to succeed, it requires a hint database that has hints for the constructors of the mutually inductive relations that the collapsed tagged proxy was derived over. Thus, when we apply the Ltac in the soundness proof for the proxy of `Even` and `Odd`, as shown in listing 4.13, we give the Ltac the hint database `even_hint_db` as an argument, which contains hints to the constructors of the relations `Even` and `Odd`.

Presumably, it is also possible to derive parts of a completeness proof for the proxy using MetaCoq, however, I decided against doing this for several reasons. The completeness proof for a proxy is substantially more complex, since it uses additional auxiliary fixpoint functions. Similarly, as for soundness, we could just derive the type of the completeness lemma, but then we would still have to manually define the fixpoint functions, so this only half solves our goal of no longer needing to manually define the type of the completeness proof. We could also derive the type of the fixpoint function, but we cannot repurpose any parts of the derivation algorithm already defined for the soundness type derivation. Furthermore, the function has a mutually recursive definition, which is more complex to derive with MetaCoq.

Another problem with completeness proofs over collapsed tagged proxies is that they do not have a simple generic approach for proving them, as is the case for the soundness proofs. A generic approach might still exist, but defining this approach in an Ltac would be considerably more complex than the Ltac we defined for proving soundness, and doing so would take time.

Finally, and most importantly, completeness proofs are not as important in the context in which we will be using the MetaCoq derivation algorithm, which is for the translation relations of the Plutus compiler. We are interested in deriving a decision procedure over the proxy of some translation relations, and finalizing a soundness proof for the decision procedures in terms of the translation relations. In most cases, a completeness proof over the proxy will not be necessary to finalize such soundness proofs. The only exception is when a sub-relation is used in negated form in the premise of another relation, in which case we need to prove the completeness of the decision procedure of the sub-relation, to proof the soundness of the decision procedure of the other relation. I have only encountered one translation relation of the Plutus compiler so far, for which defining and proving completeness was necessary.

In summary, the work required to automate completeness proofs has relatively few advantages. Furthermore, adding more derivations to the MetaCoq algorithm, and defining an Ltac that can finalize the completeness proofs, requires time and is complex. Thus, instead, I opted to just manually define and proof completeness over the proxy wherever necessary.

4.2.4 Integration with QuickChick

Over a non-mutually inductive collapsed tagged proxy we can now apply QuickChick's derivation algorithm, to acquire a decision procedure of the proxy, together with the respective soundness, completeness, and monotonicity proofs. However, a decision procedure of the proxy, cannot immediately be used for the mutually inductive relation. Before we can do this, the necessary integration between QuickChick's derivations and the original mutually inductive relations needs to happen.

```
Instance DecOptEven n : DecOpt (Even n) :=
{| decOpt s := @decOpt (Even_proxy (Even_tag n)) (DecOptEven_proxy (Even_tag n)) s |}.

Instance DecOptEven_sound n : DecOptSoundPos (Even n).
Proof. derive__sound (Even_proxy_sound (Even_tag n)). Qed.

Instance DecOptEven_complete n : DecOptCompletePos (Even n).
Proof. derive__complete Even_proxy_complete. Qed.

Instance DecOptEven_monotonic n : DecOptSizeMonotonic (Even n).
Proof. apply DecOptEven_monotonic. Qed.
```

Listing 4.14: The soundness proof for the collapsed tagged proxy in terms of the mutually inductive `Even` and `Odd`

Listing 4.14 shows all the definitions possible for this integration. I say possible here, since the proofs for completeness and monotonicity are often not necessary. Note, these definitions specifically relate the decision procedure of proxy to the mutually inductive relation `Even`. If we want to achieve the same for the relation `Odd`, the same definitions would have to be defined for `Odd` as well.

First and foremost, we define a `DecOpt` instance for the relation `Even n`, so that we can implement the decision procedure `decOpt` for this relation. We simply delegate to the decision procedure of the collapsed tagged proxy derived over `Even`, where we use `Even_tag` as an argument to specify we are using the decision procedure for the relation `Even`. Now that we have defined a `DecOpt` instance for `Even`, whenever QuickChick wants to make use of a decision procedure over `Even`, it will end up using the decision procedure of the collapsed tagged proxy instead.

When we use QuickChick's derivation framework, such as we did for the proxy, this `DecOpt` instance would be derived for us automatically. For this derived `DecOpt` instance we can then define the soundness, completeness, and monotonic proofs, and finalize the proofs automatically as well using Ltac tactics exposed by QuickChick. However, since we defined the `DecOpt` instance for `Even` manually, we will have to define and finalize any necessary proofs manually as well.

The first proof defined in listing 4.14 is an instance of `DecOptSoundPos` for the relation `Even`, which describes a soundness proof of the `DecOpt` instance implemented for `Even` in terms of the relation `Even` itself. Since the `DecOpt` for `Even` uses the decision procedure derived over the collapsed tagged proxy, this proof will entail that this decision procedure needs to be sound in terms of the relation `Even`. We already have the two soundness proofs necessary to finalize this proof: A soundness proof of the decision procedure of the proxy in terms of the proxy, which we can derive with QuickChick, and a soundness proof of the proxy in terms of mutually inductive relation `Even`, as shown in listing 4.13.

We can compose these two soundness proofs, and use it as a transitive proof of soundness for the decision procedure for the proxy in terms of the `Even` relation. Furthermore, since this intuition holds for every such soundness proof, we defined the Ltac `derive__sound` so that the soundness proof can be finalized easily. Since the soundness of the proxy is defined in terms of multiple mutually inductive relations, such as `Even` and `Odd`, we supply the Ltac with the soundness proof of the proxy over a specific constructor of the tag type. For soundness in terms of the `Even` relation, this means we use the soundness proof `Even_proxy_sound` specifically over the arguments `Even_tag n`. Now, the Ltac will first prove that the decision procedure is sound in terms of the proxy over some argument `Even_tag n`, that the proxy over this argument is sound in terms of the relation `Even n`, and thus proof that the soundness proof described by `DecOptSoundPos` (`Even n`) holds.

The listing 4.14 also shows completeness and monotonicity proofs for the `DecOpt` instance defined for the `Even` relation. For completeness, similarly to soundness, we have one completeness proof derived by QuickChick for the decision procedure and the proxy, and another completeness proof we can define manually for the proxy and the mutually inductive relations `Even` and `Odd`. Again, we can make use of an Ltac, `derive__complete` in this case, to prove completeness transitively. For completeness, there is no ambiguity about which mutually inductive relation is targeted, since they are all complete in terms of one proxy, so the Ltac does not require an additional argument.

To finalize a monotonicity proof, we can simply directly apply the monotonicity proof derived by QuickChick for the decision procedure of the proxy. A monotonicity proof states that if for some amount of fuel the decision procedure has a result, this result will never change for a larger amount of fuel. Whether or not monotonicity holds is only affected by the implementation of the decision procedure, and not the relation for which the decision procedure is used. Thus, if we have proven monotonicity for the decision procedure of the proxy, and we use this decision procedure for the `Even` relation, it is still monotonic and we can just reuse the proof that has already been finalized.

We now have presented a way of deriving a decision procedure for a mutually inductive relation, using a MetaCoq derivation algorithm, and the QuickChick derivation framework. We can first derive a non-mutually inductive collapsed tagged proxy over relations, such as `Even` and `Odd`, with a MetaCoq derivation algorithm. We can then easily define soundness for the proxy using an already derived soundness type, and then proof soundness with an Ltac. If necessary, a completeness proof for the proxy can be defined manually as well. Then, we can derive a decision procedure with its respective proofs over the proxy using QuickChick. And finally, with the necessary definitions of a `DecOpt` instance and its proofs, the derived decision procedure of the proxy can be use for the mutually inductive relations the proxy was derived over.

4.3 Other problems

Besides mutual inductive definitions, which now can be solved by deriving a proxy, several other problems manifest when applying QuickChick to the translation relations of the Plutus compiler. The `well_scoped` translation relation of Plutus is a practical example of such a relation, for which QuickChick cannot derive any computation due to many other issues alongside its mutual inductive definition. Thus, the `well_scoped` relation will be used as one of two running examples, to explain when and why certain problems occur, and how these problems can be resolved, so computation can be derived over the relation with QuickChick.

Listing 4.15 shows the parts of the definition of `well_scoped` that cause these remaining issues for QuickChick, the full definition of the relation `well_scoped` can be seen in listing A.1 in the appendix. A Plutus term is well-scoped when any variables, terms, or data-types occurring in a term are bound in the two contexts Δ or Γ. Some definitions might be bound inside the Plutus term itself, for example when abstracting a variable over some underlying sub-term, or when binding an entire sub-term in a let-binding. The former case is handled in the constructor `WS_LamAbs`, which adds the bound variable to the context Γ, and proceeds to check if the term under the abstraction is well-scoped under the new context.

The latter case is handled using the relations `well_formed_nonrec` and `well_formed_rec`, for non-recursive or recursive let-bindings respectively, since well-scopedness for these two cases is described differently. In non-recursive let-bindings, every subsequent binding can only include uses of the bindings defined before it, whereas, in recursive let-bindings, all bindings can include uses of each other. The two relations for recursive and non-recursive let-bindings, thus, can describe well-scopedness accordingly, by adding the bindings to the context in different manners.

Then, both these relations use the relation `well_formed`, which describes well-scopedness over the individual bindings, which can bind either a type, a data-type, or a sub-term. Well-scopedness for the sub-term is already described by the relation `well_scoped`, so this relation is used in the `well_formed` relation, which is the reason why well-scopedness is a mutually inductive definition.

Another definition that illustrates some problems that can arise when using the QuickChick derivation framework, is the definition of `In` in listing 4.16, as defined in the core module `Coq.Lists.List`. `In` is a function that checks if an element is contained in some list, and returns a proposition that either holds or not, depending on whether the element is in the list or not. The function `In` is frequently used as an auxiliary definition in the translation relations of the Plutus compiler, such as in the `WS_Var` constructor of the `well_scoped` relation in listing 4.15. Just as for the `well_scoped` relation, any issues that the definition of `In` causes for QuickChick's derivation framework had to be resolved. Thus, `In` will be used as the second running example in the explanations of the problems that have occurred, when applying QuickChick to the translation relations of the Plutus compiler.

4.3.1 Functions used as propositions

First, we consider the problem of using `Coq.Lists.List.In`, as seen in listing 4.16, in the premises of translation relations. The definition of `In` is a computation which, instead of returning a boolean value, returns the propositional alternatives `True` and `False`. The propositions `True` and `False` are respectively always or never inhabited, as such they can be returned by a function as the proposition versions of a boolean. Then, because the function `In` returns a proposition, it is possible to use the function in the premises of constructors of relations.

This manner of describing a proposition using a function under the hood is problematic for QuickChick, when it derives a computation over a relation. All premises in the constructor of a relation are of the type `Prop`, and QuickChick expects these to either be other relations, or recursive uses of the current relation. When QuickChick encounters uses of `In` in the premises of a constructor, it finds not a relation but a function, which is unexpected, thus QuickChick throws an error and fails to derive anything.

Hypothetically, if QuickChick would support a function such as `In` as a premise, it would then require some sub-computation over `In`, which can be used in the computation that QuickChick is currently deriving. For

```
Inductive well_scoped (Δ Γ: ctx) : Term → Prop :=
  | WS_Var : forall x,
      In x Γ →
      well_scoped Δ Γ (Var x)
  | WS_LamAbs : forall x T t,
      well_scoped_ty Δ T →
      well_scoped Δ (x :: Γ) t →
      well_scoped Δ Γ (LamAbs x T t)
  | WS_Apply : forall t1 t2,
      well_scoped Δ Γ t1 →
      well_scoped Δ Γ t2 →
      well_scoped Δ Γ (Apply t1 t2)
...
  | WS_Constant : forall u a,
      well_scoped Δ Γ (Constant (Some (ValueOf u a)))
...
  | WS_Let : forall bs t Δ' Γ',
      Δ' = rev (btvbs bs) ++ Δ →
      Γ' = rev (bvbs bs) ++ Γ →
      well_formed_nonrec Δ Γ bs →
      well_scoped Δ' Γ' t →
      well_scoped Δ Γ (Let NonRec bs t)
  | WS_LetRec : forall bs t Δ' Γ',
      Δ' = rev (btvbs bs) ++ Δ →
      Γ' = rev (bvbs bs) ++ Γ →
      well_formed_rec Δ' Γ' bs →
      well_scoped Δ' Γ' t →
      well_scoped Δ Γ (Let Rec bs t)

with well_formed_nonrec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_NonRec :
      well_formed_nonrec Δ Γ nil
  | W_ConsB_NonRec : forall b bs,
      well_formed Δ Γ b →
      well_formed_nonrec (btvb b ++ Δ) (bvb b ++ Γ) bs →
      well_formed_nonrec Δ Γ (b :: bs)

with well_formed_rec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_Rec :
      well_formed_rec Δ Γ nil
  | W_ConsB_Rec : forall b bs,
      well_formed Δ Γ b →
      well_formed_rec Δ Γ bs →
      well_formed_rec Δ Γ (b :: bs)

with well_formed (Δ Γ : ctx) : Binding → Prop :=
  | W_Term : forall s x T t,
      well_scoped_ty Δ T →
      well_scoped Δ Γ t →
      well_formed Δ Γ (TermBind s (VarDecl x T) t)
  | W_Type : forall X K T,
      well_scoped_ty Δ T →
      well_formed Δ Γ (TypeBind (TyVarDecl X K) T)
  | W_Data : forall X YKs cs matchFunc Δ',
      Δ' = rev (map tvd_name YKs) ++ Δ  →
      (forall c, In c cs → Δ' |-ok_c c) →
      well_formed Δ Γ (DatatypeBind (Datatype X YKs matchFunc cs)).
```

Listing 4.15: Definition of the mutually inductive well-scoped Plutus translation relation

```
Fixpoint In (a:A) (l:list) {struct l} : Prop :=
  match l with
  | nil → False
  | b :: m → b = a ∨ In a m
  end.
```

Listing 4.16: Definition of the `In` function in Coq.Lists.List

example, if we derive a decision procedure, so a `DecOpt` instance, over a relation with `In` as premise, the derived procedure would need to use the `DecOpt` instance for `In` as a sub-computation. When an inductive relation is used in a premise, we can use QuickChick to derive a `DecOpt` instance over the relation for us. However, for the definition of `In` this is not possible, since QuickChick only derives computations over inductive relations, and not over other computations.

A second option that was explored was to define the instance of `DecOpt` for `In` manually, such that QuickChick could find and use that instance instead. Unfortunately, QuickChick already fails its derivation before it even considers whether such an instance is available. I refrained from further investigating whether QuickChick's implementation can be changed to make use of such manually defined `DecOpt` instances, since this was outside of the scope of my research.

The solution to QuickChick's problems with the definition of `In` was to define another relation that describes the same proposition as the function `In`, and use this new relation in `In`'s stead. Since the function `In` is polymorphic itself, it was first attempted to define a polymorphic relation alternative to `In`, but as will be explained later in sub-section 4.3.2, polymorphic relation definitions come with their own issues for QuickChick. Fortunately, in the translation relations of the Plutus compiler, the function `In` is only used over three different types for its polymorphic type argument, namely `name`, `type`, and `Constr`. Thus, it suffices to define the non-polymorphic relations `NameIn`, `TypeIn`, and `ConstrIn` respectively, as alternatives to the function `In`.

```
Inductive NameIn (x : name) : list name → Prop :=
  | NI_Here  : forall {xs},
      NameIn x (x :: xs)
  | NI_There : forall {x' xs},
      x <> x' →
      NameIn x xs →
      NameIn x (x' :: xs).
```

Listing 4.17: Definition of the `NameIn` relation used as an alternative to Coq's `In` function

Listing 4.17 shows the relation `NameIn` as an alternative for the `In` function specifically over the type `name`. To guarantee that `NameIn` and `In` describe the same proposition, a semantic equality proof over them was defined and proven as well. The relations `TypeIn` and `ConstrIn`, and their semantic equivalence proof, were acquired by simply copying the definition of `NameIn`, and changing the necessary types in the type signatures.

Then, since the relations `NameIn`, TypeIn, and `ConstrIn` have the same type signature as `In`, any uses of `In` can easily be in-place replaced with the respective new relation. When doing so for the translation relations of Plutus, this can lead to breaking parts of proofs defined over the relations, which use auxiliary proofs defined over the `In` function. These broken parts of proofs can be fixed by applying the equivalence proofs of `NameIn` and `In` in the right manner, such that any auxiliary proofs over `In` can still be used.

After replacing any use of `In` in a translation relation of Plutus with `NameIn`, `TypeIn`, or `ConstrIn`, the relation no longer uses a function as a proposition in its premises. Thus, we can now apply QuickChick's derivation algorithm over the translation relation, without it failing to derive a computation due to finding a function in the premises of a constructor. Since `NameIn`, `TypeIn`, and `ConstrIn` are now used in the premises of the relation, Quickchick will require certain sub-computation to be derived over these relations as well. However, since `NameIn`, `TypeIn`, and `ConstrIn` are defined as rather simple inductive relations, we can apply QuickChick to these relations to derive such required sub-computation without issue.

4.3.2 Polymorphic relations

QuickChick's derivation algorithm is applied to inductive relations, and it is not uncommon for relations in Coq to be defined polymorphically over some type argument. As mentioned in the previous sub-section, it was first attempted to use a polymorphic relation as an alternative to the `In` function, to solve QuickChick's incompatibility with the use of the In function as a premise. However, when doing so, several issues were uncovered on using QuickChick on polymorphic relations, which lead us to define non-polymorphic alternative relations to the In function instead.

```
Inductive InPoly {A : Type} (x : A) : list A → Prop :=
  | NI_Here  : forall {xs},
      InPoly x (x :: xs)
  | NI_There : forall {x' xs},
      x <> x' →
      InPoly x xs →
      InPoly x (x' :: xs).
```

Listing 4.18: Polymorphic relation alternative to Coq's In function

First, consider the polymorphic relation shown in listing 4.18, which was defined as an alternative for Coq's In function. This definition is the same as the non-polymorphic NameIn relation in listing 4.17, besides that it uses the type variable A to range over all types, instead of just the type name. We wanted to use InPoly as an alternative in the premises of translation relations of Plutus, and since we wanted to derive computations over these relations with QuickChick, we needed to derive the necessary computations over InPoly using QuickChick as well.

```
QCDerive DecOpt for (InPoly x xs).

Instance DecOptInPoly_sound (x : A) (xs : list A) : DecOptSoundPos (InPoly x xs).
Proof. derive_sound. Qed.
```

Listing 4.19: A use of QuickChick's derivation framework for the InPoly relation, which derives the incorrect non-polymorphic soundness proof

Listing 4.19 shows how QuickChick can be applied to the InPoly relation in the same way as we have done for other relations so far. QuickChick seemingly has no issues deriving a DecOpt instance, and the respective soundness proof can be derived as well using the derive_sound Ltac tactic. Furthermore, I was able to use InPoly as an alternative to the In function, and derive the necessary computations using QuickChick for several translation relations of Plutus, before any issues with the polymorphic definition of InPoly manifested.

Incorrect non-polymorphic soundness proofs

The first issue with the polymorphic definition of InPoly, became apparent after trying to derive computation over a translation relation of Plutus, where the type argument of In was not of the type name. I was still able to derive the necessary DecOpt instances over the translation relation, but when I applied the Ltac tactic derive_sound to finalize the respective soundness proof, it would get stuck halfway into the proof. Former experiences lead me to believe there was an issue with the soundness proofs of one of the sub-computations, so I investigated the soundness proof finalized for the InPoly relation.

It turned out that the soundness proof, as seen in listing 4.19, silently used the type string as a type argument for InPoly relation. This meant that the soundness proof was a valid, yet non-polymorphic soundness proof, specifically for the InPoly relation with the type string applied to its type argument. Coincidentally, the translation relations that I had considered so far, all used In with the type name for its type argument, where name is a type synonym for the type string. Then, when I focused on a translation relation which used In over another type than name, and attempted to derive a soundness proof over the DecOpt instance for that relation, the necessary sub-proof for soundness was missing, and thus the derive_sound tactic failed to finalize the desired soundness proof.

The reason that the type string was chosen as the type for the InPoly relation, when defining the soundness proof for the DecOpt instance over InPoly, is due to how Coq handles requirements on typeclass instances. Whenever QuickChick derives computation over a polymorphic relation, it requires an instance of Dec_Eq and Enum to be available over the type arguments of the relation. This is something I was unaware of before, and thus it did not occur to me to investigate how such requirements are handled when defining a soundness proof.

When a soundness proof is defined over a DecOpt instance of InPoly, the two instances Dec_Eq A and Enum A need to be satisfied, where A is the same as the type variable seen in listing 4.18. Coq will automatically attempt to satisfy such requirements on typeclass instances, and since an instance for Dec_Eq and Enum is already available for the type string, the type variable A is silently chosen to be the type string.

```
Instance DecOptInPoly_sound
  {A : Type} {D : Dec_Eq A} {E : Enum A}
  (x : A) (xs : list A) : DecOptSoundPos (InPoly x xs).
Proof. derive_sound. Qed.
```

Listing 4.20: A use of QuickChick's derivation framework for the `InPoly` relation, which derives a correct polymorphic soundness proof

Listing 4.20 shows how to define the correct polymorphic soundness proof for the relation `InPoly`, by explicitly adding any requirements on typeclasses to the definition of the proof. By supplying the instances of typeclasses as arguments, these instances have already been satisfied, and Coq will not attempt to find such instances for you automatically. Now the correct polymorphic soundness proof can be used, as long as the requirements of the typeclasses is satisfied wherever necessary.

Unnecessary typeclass requirements

The second problem of using QuickChick over polymorphic relations is also related to the typeclass requirements, but now rather because such requirements cannot always be satisfied, but also because they do not always have to be satisfied. As mentioned before, an instance for `Dec_Eq` and `Enum` is already available for the type `string`, so there are no issues for the translation relations of Plutus that use the relation `In` with the type `string` for its type argument. However, we run into an issue when a translation relation uses `In` over some type for which these typeclass instances are not available.

Consider relations that use `In` over `type`, which is the definition that represents types in Plutus. As will be explained in section 4.3.4, it is undesirable to define an `Enum` instance for the definitions `type` and `term`. But without an `Enum` instance for `type`, we cannot derive computations over translation relation that use `In` with `type` as its type argument.

We fixed our problems with the `In` function, by defining the non-polymorphic relations `NameIn`, `TypeIn`, and `ConstrIn`, where `TypeIn` is the alternative of `In` with `type` for its type argument. We were able to derive the necessary `DecOpt` instance, and respective soundness proof over `TypeIn` just fine. This raises the question, why do these instances and proofs over `TypeIn` not also require the instance `Enum type`?.

As mentioned before, when deriving a `DecOpt` instance over a polymorphic relation with some type argument `A`, QuickChick adds an argument for the instances `Dec_Eq A` and `Enum A` to the decision procedure. So, when we derive a `DecOpt` instance over the relation `InPoly` in listing 4.18, the derived computation has as an argument the instance `Enum A`. However, when we inspect the derived computation, the argument for the instance `Enum A` looks as follows: `(_ : Enum A)`.

In Coq the underscore symbol '`_`' is used to denote a value that is unused, as such, the 'required' instance `Enum A` in the decision procedure for `InPoly`, is never used and thus also not required. So, when we derive computation over a translation relation that uses `In` with `type` for its type argument, if this requirement for `Enum A` was never imposed in the first place, we would not need to provide an instance for `Enum type` either. Unfortunately, changing the implementation of QuickChick, to no longer require any instance of `Enum` there where these instances are not necessary, is not a trivial task, and lies beyond the scope of my research.

4.3.3 Negative completeness proofs

The QuickChick derivation framework allows you to derive soundness, completeness, and monotonicity over any computation that is derived from a relation. Furthermore, having a completeness and monotonicity proof is sufficient to also finalize a negated soundness proof over a relation. However, negated completeness proofs were also necessary for one particular translation relation of Plutus, but as the authors of the QuickChick derivation framework state: "completeness for negation cannot be derived and it does not hold in general"[19]. Thus, whenever negated completeness proofs are necessary they need to be defined and finalized manually.

Consider the `rename` translation relation of Plutus, for which we wanted to derive a `DecOpt` instance and its soundness proof using QuickChick. This relation uses the `appears_free_in` relation in its premises, but it uses the relation in negated form. This means that if we want to derive soundness over the decision procedure of `rename`, we need a negated soundness proof over a decision procedure of the relation `appears_free_in`. Luckily, as mentioned before, a negated soundness proof can be finalized with just a soundness and monotonicity proof, and these proofs we can simply derive over the decision procedure of `appears_free_in` using Ltac tactics exposed by QuickChick.

However, the relation `appears_free_in` uses the relation `NameIn`, `TypeIn`, and `ConstrIn` in its premises in negated form. Thus, for analogues reasons as with soundness, if we want to derive a completeness proof over the relation `appears_free_in`, we need a negated completeness proof over `NameIn`, `TypeIn`, and `ConstrIn`. Since no derivation is possible for these negated soundness proofs, we needed to define and finalize these proofs for `NameIn`, `TypeIn`, and `ConstrIn` manually instead.

```
Instance DecOptNameIn_complete_neg x xs : DecOptCompleteNeg (NameIn x xs).
Proof. revert x xs. in_complete_neg_proof. Qed.
```

Listing 4.21: The definition and proof for negated completeness for the decision procedure of the relation `NameIn`

Listing 4.21 shows how a negated completeness proof can be defined for the relation `NameIn` using QuickChick's type class `DecOptCompleteNeg`. Even though QuickChick does not include any means of deriving a negated soundness proof, it does expose a typeclass with which to define such proofs. QuickChick will require a negated completeness proof for some other derived computation, in which cases it can require an instance for the typeclass `DecOptCompleteNeg`, to verify whether the required negated completeness proof exists.

The proof in listing 4.21 is finalized using the Ltac tactic `in_complete_neg_proof`, which is simply an Ltac tactic that includes the manually defined proof for negated completeness. The Ltac was defined to be able to apply the proof several times. We also need to define a negated completeness proof for the relations `TypeIn`, and `ConstrIn`, and since the negated proof necessary for these relations is the same as for the `NameIn` relation, we use the same definition and Ltac tactic as use in listing 4.21.

4.3.4 Unbound quantified variables

Universally or existentially quantified variables which are not bound in the result type of a constructor, pose issues for the computation which QuickChick wants to derive over relations with such constructors. These computations make use of match statements over the variables used in the result type of a constructor, as a means to instantiate values for those variables. This leaves any variables uninstantiated if that variable is not used in the result, but only in the premises of the constructor. Without the necessary handling of such variables, they would stay uninstantiated, and if the variable is then used in the computation nonetheless, the computation refers to an unbound variable.

Unbound universally quantified variables

Unbound universally quantified variables occur in certain sub-propositions in the premises of a constructor, consider for example some premise ... → (`forall x, P x →  Q x`) → .... The variable `x` is only bound within this sub-proposition, consequentially, it cannot be used in the result type of a constructor that includes this premise, and thus the variable `x` would be left uninstantiated in a derived computation of QuickChick.

Unfortunately, QuickChick does not support the use of such sub-propositions with universally quantified variables. Consider, if QuickChick were to derive a checker over a relation that contains the aforementioned sub-proposition, it would require a sub-checker over the sub-proposition to compute whether or not it holds. This checker would first have to search for all possible instantiations of the variable `x`, for which `P x` holds, such that it can then check whether `Q x` holds. For the latter part, it would technically be possible to derive a decision procedure for the relation `Q` with QuickChick, such that it can be checked for instances for `x` if `Q x` holds. However, the values of `x` for which `P x` holds are possibly infinite, which can make finding all instances of `x` an undecidable problem, and thus non-trivial to represent in a computation. Presumably, the additional complexity of supporting sub-propositions with universally quantified variables is why the authors of QuickChick did not add support for such variables.

Unbound existentially quantified variables

Existentially quantified variables are the remaining variables in a constructor, which are not bound in the result type of the constructor, and are not bound by some `forall` in a sub-proposition used as a premise in the constructor. Such variables are one of the problems that the authors of QuickChick were able to tackle, and the necessary handling was added to the way QuickChick derives its computation [19]. For existentially quantified variables, the notion holds that there exists some instance for the type of the variable, such that this instance can be used by the derived computation to achieve its goal. The solution the authors of QuickChick implemented was adding an algorithm to QuickChick for deriving enumerators, such that values for existentially quantified variables can enumerated, and thus instantiated. These enumerators are derived in terms of a premise of a constructor, such that the enumerator only enumerates those values for which the premise hold, which is a goal that coincides with other computations that QuickChick derives.

However, in the context of the Plutus translation relations, there are still two main issues with how QuickChick handles existentially quantified variables. First, the computations that QuickChick derives can enumerate over at most one existentially quantified variable, and there are translation relations where multiple such variables occur. The second problem holds in general for the enumerators that QuickChick derives. For some types the amount of values you can enumerate over is very extensive, which makes an enumerator expensive to compute, and sometimes unlikely to return any useful values for a variable.

This latter problem is especially important to consider, if we do decide we want to use enumerators for existentially quantified variables that occur in translation relations of Plutus. We want to derive decision procedures of translation relations, such that we can verify the translations that take place in the compilation of a Plutus program. For a realistic compilation over a large program, this means that some variables that need to be instantiated with an enumerator are substantial in size as well.

In the translation relations of Plutus, many such existentially quantified variables are either of the type `type` or `term`, or are some context that includes the type `type` or `term`. Both the definitions of `type` and `term` have quite a few constructors, and for every additional constructor, an enumerator has an additional branch to consider enumerating over. The necessary values for the existentially quantified variables of the type `type` and `term` are expected to be rather large, when we want to verify the translations of Plutus over a realistic compilation of a Plutus program. Thus, for the translation relations of the Plutus compiler, deriving enumerators over the types `type` and `terms` is considered undesirable, since they are expected to be too expensive to achieve any useful results.

Eliminating unbound quantified variables

A solution that can be used to fix the issues surrounding both the unbound universally and existentially quantified variables, is to change the translation relations such that we get rid of those variables altogether. If the Plutus compiler dumps more information on some translations alongside the ASTs before and after the translation, then this information can possibly be included in the translation relations, and replace the role of any universally or existentially quantified variables in the relation. However, using this approach requires knowledge of the internals of the Plutus compiler, so that you know exactly which information to dump together with the ASTs of some translation. Since I am not familiar at all with the internals of the Plutus compiler, this approach was not applied in practice.

Fortunately, for all translation relations of Plutus that I considered, all the necessary information was already available to rewrite the relation to no longer include either universally or existentially quantified variables. One such case is the constructor `W_Data` in listing 4.22, which is a constructor of the `well_scoped` relation as defined in listing 4.15.

```
| W_Data : forall X YKs cs matchFunc Δ',
    Δ' = rev (map tvd_name YKs) ++ Δ  →
    (forall c, In c cs → constructor_well_formed Δ' c) →
    binding_well_formed Δ Γ (DatatypeBind (Datatype X YKs matchFunc cs))
```

Listing 4.22: The constructor `W_Data` of the `well_scoped` relation

The second premise of this constructor includes a sub-proposition, where the variable `c` is the universally quantified variable, `cs` is a list of constructors, and Δ′ is a context (an existentially quantified variable, but this will be dealt with later). The premise states that for all constructors `c` in the list `cs`, `constructor_well_formed` Δ′ `c` should hold. Since `cs` is used in the result of the constructor, it will be instantiated in a computation that QuickChick derives, and thus all constructors in `cs` will be known to us. Thus, we can get rid of the universally quantified variable `c`, if we simply use the already known constructors in the list `cs` instead.

```
Inductive constructor_well_formed (Δ : ctx) : constructor → Prop :=
  | W_Con : forall x T ar,
      well_scoped_ty Δ T →
      constructor_well_formed Δ (Constructor (VarDecl x T) ar)
```

Listing 4.23: The definition of the relation `constructor_well_formed`, which is defined over a single constructor

We first consider the relation `constructor_well_formed` in listing 4.23, which should hold over all constructors that appear in `cs`. Conveniently, the relation `constructor_well_formed` is solely used in the constructor `W_Data` defined above, so if we are changing this premise anyway to remove its universally quantified variables, we have the freedom to also change the definition and use of the relation `constructor_well_formed`. This relation is currently defined to hold over one individual constructor, but it is also possible to define an alternative to this relation that holds over a list of constructors. Listing 4.24 shows the 'plural' alternative to the `constructor_well_formed` relation, and holds over a list of constructors instead.

```
Inductive constructors_well_formed (Δ : ctx) : list constructor → Prop :=
  | W_Nil  :
      constructors_well_formed Δ []
  | W_Cons : forall x T ar cs,
      well_scoped_ty Δ T →
      constructors_well_formed Δ cs →
      constructors_well_formed Δ ((Constructor (VarDecl x T) ar) :: cs)
```

Listing 4.24: The definition of the relation `constructors_well_formed`, which is defined over a list of constructors

The relation `constructors_well_formed` holding over some list of constructors `cs`, is semantically equivalent to `constructor_well_formed` holding over all individual constructors contained in `cs`. Thus, we can rewrite the second premise of the constructor `W_Data` to use the relation `constructors_well_formed` instead, which would result in the following premise:

```
  constructors_well_formed Δ′ cs
```

The premise now no longer includes the variable `c`, and the semantics of the premise have stayed the same. However, this premise does still include the context Δ', which is an existentially quantified variable, so we want to remove this variable as well.

Luckily, removing existentially quantified variables is easy, when it is defined in an equality premise what the value of the variable should be. The first premise of `W_Data` is $\Delta' = \texttt{rev}\ (\texttt{map}\ \texttt{tv\_name}\ \texttt{YKs}) + + \Delta$, where the right-hand side of the equality is a pure function, that only uses variables which are not existentially quantified. In a computation that QuickChick derives over the constructor `W_Data`, the right-hand side of the equality can simply be computed to determine the value of Δ'. Thus, we can get rid of existentially quantified variables such as Δ', by in-place replacing any occurrence of Δ' with the term defined on the right-hand side of the equality.

```
| W_Data : forall X YKs cs matchFunc,
    constructors_well_formed (rev (map tvd_name YKs) ++ Δ) cs →
    binding_well_formed Δ Γ (DatatypeBind (Datatype X YKs matchFunc cs))
```

Listing 4.25: The rewritten constructor `W_Data` of the `well_scoped` relation, which no longer includes existentially quantified variables

Listing 4.25 shows the rewritten version of the constructor `W_Data`, which is semantically equivalent to the constructor we had before, but now no longer includes the universally and existentially quantified variables `c` and Δ'. If all such variables are eliminated from the definition of an inductive relation, QuickChick can derive computations over the relation without any further issues.

For most translation relations of the Plutus compiler, I was able to rewrite the relations such that the universally and existentially quantified variables were eliminated. This meant that I was now able to apply the QuickChick derivation framework to the translation relations of the Plutus compiler, and derive the desired decision procedures, without QuickChick failing due to unsupported sub-proposition, or needing enumerators over the types `type` or `term`.

4.3.5 External definitions in function applications

The QuickChick derivation framework includes a rather annoying bug, for which an issue has been opened in GitHub[1], that makes it impossible to derive computation over relations with certain uses of definitions from outside the relation. The use of external definitions in relations is perfectly valid in Coq, but QuickChick's implementation incorrectly determines a definition is unbound, when the definition is applied as an argument to some function. Interestingly enough, QuickChick has no issues with a function if it is defined outside of the function, only if this holds for one of its arguments.

Unfortunately, external definitions are frequently used as arguments to functions in the translation relations of the Plutus compiler. It is likely possible to fix the implementation of the framework, to correctly handle

[1]`https://github.com/QuickChick/QuickChick/issues/311`

such external definitions used as arguments to a function, however, this is outside of the scope of my research. Luckily, an easy fix to this problem is to hoist the application of the external definitions out of the relation, by defining an additional auxiliary function.

As an example we use the constructor `W_Data` of the `well_scoped` relation again, of which the definition can be seen in listing 4.25 in the previous subsection. This constructor includes the function application `map tvd_name YKs`, where `tvd_name` is defined outside of the relation and applied to `map` as an argument. Thus, when we apply the QuickChick derivation framework to the relation `well_scoped`, it incorrectly assumes that `tvd_name` is an unbound function. Even if we would fix any issues with `map` and `tvd_name`, we would end up running into the same problems again for the function application of `rev` and ++, since these functions also use external definitions in their arguments. So, to tackle all issues with `map`, `tvd_name`, `rev`, and ++ functions at once, we hoist them all out of the relation using the following function definition:

```
Definition map_tvd_name_rev_app x c := rev (map tvd_name x) ++ c.
```

This function copies the same function applications as used in the first premise of `W_Data`, and the arguments of this function are exactly those variables which are bound within the constructor `W_Data`. If we replace the problematic function application in the first premise of `W_Data` with our auxiliary function, we get the following premise:

```
constructors_well_formed (map_tvd_name_rev_app YKs Δ) cs
```

We now have a constructor with one function that is defined outside of the relation, the function still describes the same function applications as before, and all its arguments are now bound within the constructor. We can now apply QuickChick's derivation algorithm to the `well_scoped` relation, and the function will no longer cause the algorithm to fail its derivation.

The solution to this problem is not very elegant, since for some translation relations it leads to several very ad-hoc auxiliary functions needing to be defined before the relation. Needlessly to say, fixing the bug in QuickChick would be more desirable, since then it would not be necessary to clutter up the Plutus project with such silly ad-hoc function definitions. Nonetheless, it is a working solution, thus it was used to fix several problematic function applications throughout the translation relations of the Plutus compiler.

4.3.6 Nested datatypes in constructor patterns

Another bug in the QuickChick derivation framework, causes it to generate invalid match-statements over constructors that include nested patterns of several data types, which are types with one single constructor. Match-statements are required to be exhaustive in Coq, meaning that if not all cases have been covered yet, some catch-all case needs to be appended at the end of the match statement. On the contrary, if all cases have been covered, this catch-all case is not necessary, and if it is added anyway, Coq will end up throwing errors. Currently, when a constructor of a relation includes embedded data types, the implementation of QuickChick's derivation algorithm incorrectly adds a catch-all case to a match-statement, even though it should not.

```
Inductive vdecl := VarDecl : binderName → ty → vdecl.
Inductive constr := Constructor : vdecl → nat → constr.
```

Listing 4.26: The definitions in the Plutus project for the data types `vdecl` and `constr`

To understand when and why the problem arises, consider the two datatype definitions of Plutus for `vdecl` and `constr` in listing 4.26. There are two things important to note here. First, both these types have only one single constructor. Second, the first argument of the data type `constr` is the type `vdecl`, thus `vdecl` is nested in `constr`. Now consider the following constructor `W_Con` of the `constructor_well_formed` relation, as seen in listing 4.23:

```
| W_Con : forall x T ar,
    well_scoped_ty Δ T →
    constructor_well_formed Δ (Constructor (VarDecl x T) ar)
```

When we apply QuickChick to derive a computation over the relation `constructor_well_formed`, QuickChick creates a match-statement for all the constructors of the relation, which in this case is only the constructor `W_Con`. The match-statement matches the patterns in the result of a constructor with some values for the arguments of the relation, to compute whether that constructor could have been used to inhabit the relation for those arguments. For the `W_Con` relation, this means QuickChick creates a match statement over some value of the type `constr`, and adds a case that matches with the pattern `Constructor (VarDecl x T) ar`. Since

Constructor and VarDecl are the only constructors of constr and vdecl respectively, the match case that QuickChick adds will match any possible instance of the type constr. Thus, it is no longer necessary to add a catch-all case to the match-statement, however, QuickChick does this anyway.

The implementation of QuickChick includes a function that can be applied to the abstract syntax of the pattern used in a match-statement, to check if that pattern will match all possible cases. Again, for the W_Con constructor, this pattern will be Constructor (VarDecl x T) ar. If the pattern includes one constructor of a data type, so for example Constructor, and only uses variables as arguments to that constructor, the function will correctly conclude that all cases are covered by the pattern. However, if one of the arguments of the constructor of a data type is another constructor, the function immediately determines the pattern does not cover all cases. Since the constructor VarDecl is used as an argument to the Constructor constructor, the function concludes that the pattern of the W_Con constructor does not match all cases.

As long as all constructors used in a pattern are all constructors of a data type, so a type with one constructor, then the pattern matches all possible cases. So, the solution to the problem with QuickChick's implementation is to use this function that checks if a pattern matches all cases, and applies it recursively to underlying sub-patterns as well. Now, the function first determines that Constructor is the constructor of a data type, recursively applies itself to the arguments of Constructor, determines that VarDecl is the constructor of a data type, and correctly concludes that the pattern of W_Con matches all cases. After implementing this fix in my fork of the QuickChick derivation framework[2], we can apply QuickChick to the relation constructor_well_formed, and it will correctly derive a match-statement over W_Con that does not include an unnecessary all-matching case.

4.3.7 Relations over mutually recursive types

The authors of the QuickChick derivation framework explicitly state that the framework does not support mutually inductive definitions [19], but nothing is said about non-mutually inductive relations over mutually recursive types. Unfortunately, QuickChick does not support relations over such mutually recursive types either, or at least not fully. I do not know whether not supporting these types was intentional, or whether it was possibly forgotten to be implemented. Either way, I ended up making the necessary adjustments to QuickChick's implementation to support relations over mutually recursive types, and for all my purposes, the fix has worked as intended. However, I lack the knowledge to make an informed decision on whether my fix works generically for all scenarios that can arise when using mutually recursive types.

To elaborate on my fix, we first need to understand how mutually recursive types are represented in the AST used by the QuickChick framework. The types term and binding of the Plutus compiler, for example, are both referred to as the type term, and then an index 0 or 1 is used to differentiate between the two types respectively.

The reason QuickChick does not work for mutually recursive types is because it ignores this index, and simply always defaults to 0. This means that when QuickChick is applied to a relation defined over the type binding, QuickChick will do its derivation as if the relation was defined over the type term. This will result in QuickChick deriving a computation that is not well-typed, when it is applied to a relation defined over the type binding. Note here, that if a relation is defined over the type term, the use of the index 0 is warranted, since this is the correct index of term, so for such cases QuickChick works as intended.

I fixed the issues QuickChick had with relations over mutually recursive types in my fork of QuickChick[2], by simply no longer ignoring the aforementioned index, and using it throughout the implementation of QuickChick there where necessary. Then, when QuickChick is applied to a relation over a mutually recursive type, it reasons over the correct type in the mutually recursive definition, and the resulting computation is the well-typed computation we expected.

4.3.8 TypeIn and ValueOf

The two definitions TypeIn and ValueOf defined in the Plutus project, cause issues for QuickChick's derivation framework when they are used a certain way in the constructors of a relation. I do not know exactly why this problem manifests, and since I found a solution very quickly, I did not spend much time investigating the root of this issue further. I will focus on ValueOf, but for TypeIn the same solution can be used to resolve any issues.

First consider the definition of valueOf in listing 4.27, which shows that valueOf has one constructor ValueOf that takes one argument of the type uniType u. The function uniType maps a set of default universes, defined under the type DefaultUni, to concrete types of Coq. Thus, those concrete types of Coq are the types that the constructor ValueOf can take as arguments.

[2]https://github.com/Xiving/QuickChick

```
Inductive valueOf (u : DefaultUni) :=
  ValueOf : uniType u → valueOf u.
```

Listing 4.27: The definition for `valueOf` in the Plutus project

Now consider the following constructor `WS_constant`, which is a constructor of the `well_scoped` relation as defined in 4.15:

```
| WS_Constant : forall u a,
      well_scoped Δ Γ (Constant (Some (ValueOf u a)))
```

QuickChick fails to derive computation over the `well_scoped` relation, due to the use of the constructor `ValueOf` here. My best guess is that the underlying `uniType` function, makes it hard, or even impossible, for QuickChick to reason over what the type of `a` should be in this relation. Luckily, we can rewrite this constructor to the following:

```
| WS_Constant : forall u (x : valueOf u),
      well_scoped Δ Γ (Constant (Some x))
```

Since `valueOf` is a type with only one constructor, there is no difference in whether we require some variable of the type valueOf u and apply it to `Some`, or whether we explicitly apply the constructor `ValueOf u a` to `Some`. The value `a` used in the former `WS_Constant` constructor disappears, however, in the latter `WS_Constant` it is still contained within the value `x`.

After rewriting any relations that use `TypeIn` or `ValueOf`, such that they make use of `typeIn` or `valueOf` instead, we can successfully derive computation over the relations using QuickChick.

4.3.9 Decidable equality and enumerators

As mentioned before, sometimes QuickChick needs a `DecEq` or `Enum` instance for certain types. An `Enum` instance is only necessary over a type, when a relation has an existentially quantified variable over that type, and you want to derive a computation over relation without getting rid of the existential variable. In such cases, an instance of `Enum` can simply be derived using QuickChick. The `DecEq` instances, however, are needed for every type over which an equivalence is used in the premise of a relation, which is a frequent occurrence in the translation relations of Plutus.

For Coq's most basic type, the QuickChick library already includes `DecOpt` instances, but the relations of Plutus oftentimes include equalities over types defined in Plutus as well. Luckily, for the most critical definitions of Plutus, such as `type` and `term`, decidable equality is already proven using a definition included in the standard Coq library. To create an instance for the QuickChick typeclass `DecEq`, for the necessary Plutus types, we can simply delegate to the decidable equality instances which have already been finalized.

4.3.10 Sub-modules and qualifiers

The last problem I wanted to mention, is that the implementation of the QuickChick derivation framework does not correctly handle certain qualifiers that can be used in Coq. Some translation relations of the Plutus compiler are defined with the same name several times in the different sub-modules `Type`, `Term`, or `Annotation`. Then, if we want to used the relation `well_scoped` over a term for example, we can target that relation using the respective qualifier `Term.well_scoped`.

Unfortunately, QuickChick does not behave well, when using it to derive computation over relations that use such qualifiers to refer to themselves, or to refer to other relations in their premises. QuickChick will either conclude that the definition targeted with the qualifier does not exist, and it will fail its derivation. Or arguably worse, QuickChick will interpret the qualifier `Term.well_scoped` as `Type.well_scoped` by ignoring the prefix of the qualifier, in which case it sometimes silently derives an incorrect computation.

The solution to this problem is to phase out such qualifiers, which for the relations `Type.well_scoped` and `Term.well_scoped` can be done by moving them out of their sub-modules, and giving them distinct names such as `well_scoped_ty` and `well_scoped_tm`. So, for all the translation relations in the Plutus compiler that I considered, I moved them out of their sub-modules if necessary, and removed any qualifiers that were used for such relations. After removing all uses of the qualifiers, QuickChick can no longer interpret them incorrectly, and thus QuickChick now works as intended on the translation relations of Plutus.

Chapter 5

Results

The QuickChick derivation framework was applied to several translation relations and auxiliary relations of the Plutus compiler, to derive decision procedures for the relations, together with the necessary soundness, completeness, and monotonicity proofs. Certain relation definitions were problematic for QuickChick to derive computation over, which either required a fix in the implementation of QuickChick, or the definition of the relation needed to be adjusted to mitigate the issues. Furthermore, since QuickChick is incompatible with mutually inductive relations, for such relations a non-mutually inductive proxy had to be derived, such that QuickChick could be applied to the proxy instead.

As a result of my efforts, I was able to derive decision procedures for most translation relations of Plutus that I have considered. For the `inline` translation relation, I was able to apply its decision procedure over two ASTs of a realistic inline translation of the Plutus compiler, thus, I was able to successfully verify that the `inline` relation defined in Coq, holds over the input and output ASTs of the inline translation that took place in a compilation of the Plutus Compiler.

These results were acquired using my own fork of the QuickChick derivation algorithm [14], and using a MetaCoq algorithm for deriving non-mutually inductive proxies [21]. Both these derivation algorithms were applied in the `dec−deriving` branch of the `plutus−cert` repository [12], which also includes any of the necessary changes made to the translation relations.

5.1 Relations that work

The translation relations of Plutus for which QuickChick was able to derive a decision procedure, are listed in table 5.1. This table shows in the first three columns which proofs were finalized for the derived decision procedure of some relation, and the remaining columns show the problems that had to be addressed before this decision procedure could be derived, which are ordered roughly by increasing complexity. For clarity, these problems are also listed in table 5.2 together with a brief description, as a reminder of what the different problems entail. When a relation has several distinct definitions over types, terms, and annotations, such as for the relation `well_scoped`, these have been grouped under the same relation name.

For all decision procedures that were derived, a soundness proof was finalized as well, since this is the most important proof for our use case. We want to compute using a decision procedure whether a translation relation is inhabited, thus if the decision procedure returns that the relation is inhabited, we want to use a soundness proof that implies that the relation holds. For some decision procedures, a completeness and monotonicity proof was written as well, not because we need these proofs directly, but because those proofs together are sufficient for QuickChick to be able to finalize a negated soundness proof. Such negated soundness proofs are necessary for embedded relations which are used in negated form in other relations, over which we want to derive a decision procedure with its soundness proof. If this embedded relation in turn also includes a relation which is used in negated form, a negated completeness proof over that relation is necessary, which is the last proof type listed in the table.

The only relations over which a negated completeness proof was finalized are the `NameIn`, `TypeIn`, and `ConstrIn` relation, which are the relations we defined as alternatives to the `In` function. These relations are used in negated form in the `appears_free_in` relation, and this relation is used in negated form for the `rename` relation. Thus for soundness over the decision procedure of the `rename` relation, we needed the negated completeness proofs of `NameIn`, `TypeIn`, and `ConstrIn`. Unfortunately, I was not yet able to derive a decision procedure over the `rename` relation, but in anticipation that we will in the future, the required negated completeness proofs for `NameIn`, `TypeIn`, and `Constr` are already finalized.

After the proofs, the columns in table 5.1 show the problems that had to be resolved, before we could derive a decision procedure over the listed relations. The bottom half of the table includes the auxiliary relations

Relation Name	Soundness	Complete & Monotonic	Negated Completeness	Mutually Inductive	Completeness for Proxy	Existential Variables	Function as Proposition	Function Applications	TypeIn or ValueOf	Sub-modules/Quantifiers
well_scoped	x			x		x	x	x	x	
inline	x			x		x		x		
appears_free_in	x	x		x	x	x	x	x		x
appears_bound_in	x	x					x			x
unique	x									x
value	x			x			x			
is_pure	x									
NameIn	x	x	x							
TypeIn	x	x	x							
ConstrIn	x	x	x							
LookUp	x	x								
lt_nat	x	x								

Table 5.1: Summary of the relations for which a decision procedure was derived, the proofs finalized for those decision procedures, and which problems had to be fixed before deriving the decision procedure.

Problem name	Description
Mutually Inductive	For mutually inductive relations, a non-mutually inductive proxy can be derived with MetaCoq, so QuickChick can be applied to the proxy instead
Completeness for Proxy	When a completeness proof is necessary over the decision procedure of a mutually inductive relation, a manual completeness proof needs to be written over the proxy derived for the relation
Existential Variables	Relations with existential variables need to be rewritten to no longer contain such variables, when we do not want to define an enumerator for the type of the variable, such as `type` and `term` of the Plutus compiler
Functions as propositions	Functions that return a proposition, such as the function `In`, need to be replaced with a relation describing the same proposition
Function applications	When a relation includes a function that uses an external definition as argument, these need to be hoisted out of the definition using an additional auxiliary function
TypeIn or ValueOf	Uses of the constructors `TypeIn` and `ValueOf` need to be replaced with universally quantified variables of the type `typeIn` and `valueOf`
Sub-modules and Quantifiers	Relations defined in sub-modules have to be hoisted out of those sub-modules, and the use of any qualifiers to such sub-modules needs to be removed

Table 5.2: A brief description of the problems listed in table 5.1.

used for the more complex translation relations of Plutus, which have simpler definitions, thus QuickChick can derive decision procedures over them without any issues. Then the more interesting relations are shown at the top of the table, which have more complex definitions, and several problems occurred when trying to apply QuickChick's derivation algorithm over them.

Most problematic are mutually inductive relations, since QuickChick will outright fail any derivations for those relations, since it does not support mutual induction. For such relations a collapsed tagged proxy needed to be derived using the MetaCoq derivation algorithm, so that QuickChick can be applied to the non-mutually inductive proxy instead. Then, the necessary proofs need to be defined to relate any computation derived over the proxy to the original mutually inductive relation, which in certain scenarios such as for the `appears_free_in` relation also includes proving completeness over the proxy.

Existentially quantified variables are also a difficult problem to solve, since their solution differs case-by-case. QuickChick requires an enumerator to instantiate the existentially quantified variables, however, implementing such enumerators is undesirable for certain types, such as `type` and `term` defined for Plutus. Furthermore, QuickChick is limited to using an enumerator for at most one such variable at the same time, so any relation with several existentially quantified variables is problematic.

Most translation relations had to be rewritten to no longer include any existentially quantified variables. This can be achieved by using a non-existential variable that appears in the result of a constructor instead, without changing the semantics of the relation. However, the purpose of an existentially quantified variable depends heavily on its context, so how a variable for the result of a constructor can be used in its stead, also depends on the context. In table 5.1 can be seen for which relations existential variables had to be removed, but the methods used to remove the variables differ per relation.

The remaining problems shown in the table are all trivial problems, which can be solved using simple rewrite rules. When a function is used a proposition, such as `In`, it can be replaced by alternative relations such as `NameIn`, `TypeIn`, or `ConstrIn`. Function applications that use a definition defined outside of the relation, as an argument, can be hoisted out of the relation by defining an additional auxiliary function. Any occurrence of the constructor `TypeIn u` or `ValueOf u`, can be replaced with a universally quantified variable of the type `typeIn u` and `valueOf u` respectively. And lastly, any relations defined in a sub-module need to be hoisted out of their sub-module, and any qualifiers to those sub-modules need to be removed.

5.2 Relations that do not work

Out of all the translation relations that I considered, the `rename` relation is the only relation for which I was unable to derive a decision procedure. The `rename` relation makes use of several existentially quantified variables, and for each one, I have to come up with a way to rewrite the relation to get rid of such variables, before I can apply QuickChick to the relation successfully. One constructor describes a valid renaming of a recursive let-binding, for example, where the constructor extends the two contexts of the relation using two existentially quantified variables. This constructor then states that if for some extension of the two contexts, the premises of the constructor hold, then the renaming of the recursive let-binding is valid. Thus, we need to come up with a way to replace these extensions of the context, with known values bound in the result of the constructor somehow, such that they are no longer existentially quantified variables.

It is speculated that the necessary values for these extensions of the contexts can either be computed with a function, or the compiler can be extended to produce this extra info. Unfortunately, I was unable to realize such a function, or extend the Plutus Tx compiler, so as of writing this solution remains but a suspicion.

5.3 Example derivation of relation

Following the mantra 'show, don't tell', I will focus on the `well_scoped` relation of Plutus, and show what needs to be done to derive a decision procedure over this relation using MetaCoq and QuickChick. This example will summarize the results of implementing a derivation algorithm in MetaCoq to derive a non-mutually inductive proxy over `well_scoped`, and how this can be integrated with the QuickChick derivation framework. Furthermore, with this example, I hope to show the relative ease of deriving a decision procedure using MetaCoq and QuickChick, in comparison to manually defining and proving such decision procedures and their necessary proofs.

Before we can derive anything over the `well_scoped` relation, we need to rewrite the relation to no longer include any problematic definitions, that cause QuickChick's derivations to fail. Such definitions do not pose any problems for the MetaCoq derivation algorithm, but since these definitions will persist in the proxy derived with MetaCoq, they will continue to be an issue for QuickChick when applied to the proxy. Any issues that occur in the `well_scoped` relation have been covered in section 4.3, which also elaborates on the necessary

```
(* Proxy derivation *)

MetaCoq Run (deriveCTProxy well_scoped).

Local Hint Constructors well_scoped well_formed_nonrec well_formed_rec well_formed :
    well_scoped_hints.

Theorem well_scoped_proxy_sound : well_scoped_proxy_sound_type.
Proof. deriveCTProxy_sound well_scoped_hints. Qed.


(* QuickChick derivation over proxy *)

QCDerive DecOpt for (well_scoped_proxy tag).

Instance DecOptwell_scoped_proxy_sound tag : DecOptSoundPos (well_scoped_proxy tag).
Proof. derive_sound. Qed.


(* Using DecOpt of proxy for original mutually inductive relation *)

Instance DecOptwell_scoped c1 c2 tm : DecOpt (well_scoped c1 c2 tm) :=
{| decOpt s := ·decOpt (well_scoped_proxy (well_scoped_tag c1 c2 tm)) _ s |}.

Instance DecOptwell_scoped_sound c1 c2 tm : DecOptSoundPos (well_scoped c1 c2 tm).
Proof. derive__sound (well_scoped_proxy_sound (well_scoped_tag c1 c2 tm)). Qed.


(* Optionally *)

Instance DecOptwell_formed_nonrec c1 c2 bs : DecOpt (well_formed_nonrec c1 c2 bs) :=
{| decOpt s := ·decOpt (well_scoped_proxy (well_formed_nonrec_tag c1 c2 bs)) _ s |}.

Instance DecOptwell_formed_nonrec c1 c2 bs : DecOptSoundPos (well_formed_nonrec c1 c2 bs).
Proof. derive__sound (well_scoped_proxy_sound (well_formed_nonrec_tag c1 c2 bs)). Qed.


Instance DecOptwell_formed_rec c1 c2 bs : DecOpt (well_formed_rec c1 c2 bs) :=
{| decOpt s := ·decOpt (well_scoped_proxy (well_formed_rec_tag c1 c2 bs)) _ s |}.

Instance DecOptwell_formed_rec c1 c2 bs : DecOptSoundPos (well_formed_rec c1 c2 bs).
Proof. derive__sound (well_scoped_proxy_sound (well_formed_rec_tag c1 c2 bs)). Qed.


Instance DecOptwell_formed c1 c2 bs : DecOpt (well_formed c1 c2 bs) :=
{| decOpt s := ·decOpt (well_scoped_proxy (well_formed_tag c1 c2 bs)) _ s |}.

Instance DecOptwell_formed c1 c2 b : DecOptSoundPos (well_formed c1 c2 b).
Proof. derive__sound (well_scoped_proxy_sound (well_formed_tag c1 c2 b)). Qed.
```

Listing 5.1: Full derivation of a decision procedure for the `well_scoped` relation, using a derived collapsed tagged proxy

changes that need to be made to resolve the issues. Then, the definitions of the `well_scoped` relation before and after all the necessary rewrites is shown in listing A.1 and A.2 in the appendix respectively.

Now that the `well_scoped` relation no longer includes any problematic definitions, we can apply the MetaCoq and QuickChick derivation algorithms over the relation, to acquire the desired decision procedure over the relation. Listing 5.1 shows the entire derivation for a decision procedure of `well_scoped`, which applies the following steps:

- Use MetaCoq to derive the tag, proxy, and proxy soundness proof type
- Finalize soundness over the proxy using the derived soundness proof type and an Ltac
- Apply QuickChick to derive a decision procedure over the proxy
- Integrate the proxy decision procedure with the original mutually inductive relation

First, the MetaCoq derivation algorithm `deriveCTProxy` is applied on the `well_scoped` relation, which will derive the definitions `well_scoped_proxy_tag`, `well_scoped_proxy`, and `well_scopep_proxy_sound_type` over the proxy. To be able to derive soundness over the proxy with an Ltac tactic, we first need to create a Coq hint database `well_scoped_hints`, which contains hints for the constructors of all the relations in the mutually inductive definition of `well_scoped`. Then, we can define the soundness proof easily with the already defined proof type `well_scoped_proxy_sound_type`, and finalize it by applying the Ltac tactic `deriveCTProxy_sound` which uses the hints database. This gives us a non-mutually inductive proxy over `well_scoped`, for which has been proven with a soundness proof that if the proxy holds, then so does one of the relations in the mutually inductive definition of `well_scoped`.

Over the proxy we can derive a decision procedure and its soundness proof as intended for QuickChick. We derive a `DecOpt` instance for the proxy, defined a `DecOptSoundPos` instance over soundness over the proxy, and finalize the soundness proof using the `derive_sound` Ltac.

Now that we have a decision procedure for the proxy, we can manually define the decision procedure for `well_scoped` using the `DecOpt` typeclass, and simply have this instance delegate to the decision procedure of the proxy. We again define a soundness proof using `DecOptSoundPos`, but this time over the `well_scoped` relation, and finalize the soundness proof by applying the `derive__sound` Ltac tactic. Then, the `derive__sound` Ltac tactic creates the necessary transitive soundness proof, by using both the derived soundness proof of the decision over the proxy, and the soundness proof of the proxy in terms of the relation `well_scoped`.

We are most interested in a decision procedure and the respective soundness proof over the `well_scoped` relation, since this is the relation that represents a property which needs to hold for programs resulting from certain translations of Plutus, such as programs after the dead-code pass. Optionally, it is also possible to acquire a decision procedure and proof over the other relations in the mutually inductive definition of `well_scoped`. This can be achieved using the same definition as for `well_scoped`, but over another constructor of the tag, so instead of the `well_scoped_tag` for the `well_scoped` relation, the `well_formed_tag` can be used for the `well_formed` relation.

5.4 Derivation as verification

The final goal for the derived decision procedures of the translation relations of Plutus is to use them in the certification framework, to verify the translations that take place in a compilation of the Plutus Tx compiler. Unfortunately, I was unable to apply the derived decision procedures to this extent, so that they could verify a translation for every compilation. However, I did manage to use a derived decision procedure of the `inline` translation relation of Plutus, and verify one single realistic inlining translation of the Plutus compiler inside Coq. Furthermore, the decision procedure of the `unique` relation could be extracted to Haskell, and successfully established uniqueness over a dozen programs that resulted from the dead-code pass of Plutus.

For the former, I was provided with two AST dumps from before and after the inline translation of the Plutus compiler, from a compilation over a realistic program, namely, the timelock example used by Krijnen et al. [13]. The two dumped ASTs were given as arguments to the decision procedure derived over the `inline` translation relation, and the procedure successfully computed that the `inline` relation is inhabited for the two ASTs. Using the soundness proof derived, we can then indeed get our hands on a Coq proof that `inline` holds over the two ASTs.

The `unique` relation differs from the `inline` relation in that it describes a property over one program, and not a relation over two programs. Uniqueness is a property that is required to hold over a program after the dead-code pass of Plutus, thus, we can apply the decision procedure of `unique` to a program outputted by this pass to verify whether uniqueness holds. To do manual verification using the decision procedure of the `unique` relation, first, it was extracted to Haskell using Coq's extraction mechanisms. The procedure was then applied to several test cases resulting from the dead-code pass of Plutus[1], and was successfully able to verify that the `unique` relation holds over the test cases.

By manually using the derived decision procedures, we can already do two primitive ways of verification for the translation of the Plutus compiler. First, if the decision procedure always, or frequently fails to establish that a relation should hold over some two ASTs dumped by the Plutus compiler, then this implies that the translation relation defined in Coq does not correctly describe the behaviour of the translation that occurs in the Plutus compiler. This can either mean that the translation in the Plutus compiler does not behave as intended, and thus needs to be fixed. Or it could mean we incorrectly defined the translation relation in Coq, and thus need to inspect what is wrong in the Coq definition of the relation.

The second type of verification happens after the decision procedures do manage to compute successfully that a relation holds over some two ASTs dumped by the compiler. In this case, we can establish with some confidence that the translation relation defined in Coq accurately reflects the behaviour of the respective relation of the Plutus compiler. The next step in the verification framework would be to prove that for any two ASTs before and after the translation, the two ASTs are semantically equivalent. Possibly, we are unable to prove this semantic equivalence over the two ASTs before and after the translation. Then, we can determine that the translation in the Plutus compiler does not preserve semantics, from which we can again establish that either the compiler or the specification is incorrect.

[1]`https://github.com/input-output-hk/plutus/tree/master/plutus-core/plutus-ir/test/transform/deadCode`

Chapter 6

Discussion

Verifying the correctness of compilers, and the integrity of the software that they produce, is increasingly important in a world where the use of software has become ubiquitous. Krijnen et al [13] described a certification architecture for the Plutus Tx compiler, that aims to verify semantic preservation over a compilation from PIR (Plutus Intermediate Representation) to PLC (Plutus Core). Their approach is to use a certification engine implemented in the Coq theorem prover, that can generate translation certificates over the translations of a compilation. The translation certificates assert the validity of a compiled Plutus Core program, and in the future, can be used as a verifiable link back to the program's source code.

The certificates are modelled as inductive relations in Coq, and describe the admissible behaviour of the compiler passes of the Plutus Tx compiler. To generate the certificates over the compilation trace, manual decision procedures were implemented, which is a complex and laborious task. Then, semantic preservation proofs can be finalized over the translation certificates. This guarantees that for any compilation over which the certificates hold, it preserved semantics in the compiled program. However, finalizing these proofs was out of the scope of Krijnen et al., and was thus left to be done in the future.

Paraskevopoulou et al. implemented a framework in the QuickChick library, for deriving computation over inductive relations defined in Coq[19]. Tactics were also included in the framework, to derive correctness proofs over the computation in terms of the relation. One computation that can be derived is a checker, which computes over some arguments of a relation whether the relation holds. The purpose of such a checker aligns exactly with that of the manual decision procedures in the certification engine of Krijnen et al.

The already existing complexity of using manual checkers, and the notion that such checkers can be derived automatically, lead me to ask the following research question in my thesis:

Research question: Can we automatically derive checkers used for compiler verification?

6.1 Related work

Compiler correctness as a field of study has seen a rise in interest, and aims to make formal guarantees on the correctness of compilers. It is critical to verify that a compilation does not alter the intended behaviour of a compiler program, and that it does not silently introduce bugs or security vulnerabilities. The ever-increasing complexity of modern compilers makes proving them correct difficult, but many efforts have been successful in proving the correctness of realistic compilers. Fortunately, developments in compiler correctness, but also other fields, can continuously be leveraged to aid us in this endeavour.

Leroy et al. used a 'verified compilers' approach for the CompCert compiler, to verify its correctness with the theorem prover Coq[16]. The CompCert compiler was entirely implemented as a computation in Coq, which allows proofs to be defined and proven directly over this computation. Instead of verifying individual compilations as is done for the Plutus Tx compiler, for CompCert semantic preservation can be proven to hold over all compilations over any possible program. This approach, however, requires the entire compiler to be implemented in Coq, which limits its use in practice.

The certification engine of the Plutus Tx compiler, instead uses decision procedures of translation relations, which are used to generate proof-objects on the validity of compilations of the compiler. This methodology is also used by the SSReflect library, which uses sophisticated decision procedures as a means to compute over symbolic representations[9].

Joris Dral established in his master thesis, that for the translation relations used for certification of the Plutus Tx compiler, semantic preservation can be formally verified[8]. Formal static and dynamic semantics were defined for the PIR and Plutus Core languages, and step-indexed logical relations were defined which can be reused throughout the semantic preservation proofs for the translation relations of the certification engine. Then, for the translation relation that describes non-recursive let-binding desugaring, semantic preservation was proven. Due to the modular approach used to finalize the proof, semantic preservation can expectedly be proven similarly for the other translation relations.

Anand et al. are working on the verified compiler CertiCoq, which is a compiler for the Language Coq, that has been implemented and verified in Coq itself[2]. Coq is often used to write programs, mechanically verify the desired properties over them, and then use one of several extraction methods to acquire a verified program. Unfortunately, these extraction methods are not verified, and thus there are no strong guarantees on whether the resulting low-level code they produce is correct. The goal of CertiCoq is very relevant for other projects such as QuickChick, or the certification engine for Plutus Tx, since these are implemented in Coq. Any confidence we can have in these Coq projects is diminished knowing the extraction methods of Coq itself are not end-to-end verified.

Finally, the intuition I used to rewrite mutually inductive relations into non-mutually inductive proxies using a tag is not new. Yakushev et al. describe a similar method in Haskell for encoding mutually recursive types into a regular type using a GADT as tag[25]. They can then obtain isomorphic instances between the two types using conversion functions. I instead had to define correctness proofs between the mutually and non-mutually inductive relations to relate them, since Coq does not allow computation over relations.

6.2 Discussion and future work

An important motivation for this thesis was to replace manual checkers with derived checkers, to reduce the complexity of the certification engine of the Plutus Tx compiler. Therefore, we would still like to derive decision procedures over all the remaining translation relations of Plutus as well. As was the case for the relations considered so far, certain problems will have to be solved to achieve this.

The root of some of these problems is bugs in the implementation of the QuickChick derivation framework. When these bugs get fixed, it will get rid of these problems, and will make derivations in the future an easier task. Furthermore, the current changes made as a solution to circumvent these problems, are arguably not very elegant. Thus, after the bugs are fixed, any such changes can be reverted if desired.

Another issue that is not foreseen, but possibly could happen, is that our MetaCoq derivation will fail to derive a non-mutually inductive proxy over a mutually inductive translation relation. The algorithm currently used is by no means an algorithm verified to always work, it just so happens to have worked for all our purposes so far. It would be relevant to our cause, to formally define and verify a method of deriving non-mutually inductive proxies. This would overall increase the confidence we can have in our solution of dealing with mutually inductive translation relations.

A secondary goal in this thesis was to quantify the performance of the derived decision procedures of Plutus, similarly as was done in the toy compiler. Unfortunately, I did not manage to acquire any credible data on performance, due to a lack of time. It would have been interesting to see how long the execution of some derived procedure takes, when it is applied to a realistic Plutus program. Furthermore, benchmarking a derived and manual checker for some translation relation could have given me more insight into the feasibility of using the derived checkers over a manual checker.

For the toy compiler, gathering the data on its performance required implementing a generator of the toy compiler programs. I refrained from implementing such a generator for Plutus Tx programs, since they include more complex constructs and types, which makes the generation of realistic programs a non-trivial task.

In the future, a generator for realistic Plutus programs could still be implemented, if a benchmark over a substantial amount of programs is desirable. However, for satisfactory performance results, it would be sufficient to derive all the decision procedures of the necessary translation relations, and implement them in the verification engine of the Plutus Tx compiler. Then, we can simply compile large realistic Plutus Tx programs, and record the time it takes to verify the compilations, as a means to reason about the performance of the derived checkers.

6.3 Conclusion

We successfully derived decision procedures with QuickChick, over several translation relations of the certification engine of the Plutus Tx compiler. For problematic mutually inductive translation relations, we implemented a MetaCoq algorithm for deriving non-mutually inductive proxies. Any other problems that occurred with the translation relations, could be solved by rewriting parts of their definition. Then, for a compilation of a realistic Plutus Tx program, we were able to verify the inlining pass using a derived checker. Furthermore, one of the derived checkers was extracted to Haskell, and was able to verify the uniqueness property over a dozen test cases. We believe these positive results are sufficient evidence to proclaim the following conclusion for our research question:

Conclusion: We can automatically derive checkers used for compiler verification

Appendix A

Appendix

```
Inductive well_scoped (Δ Γ: ctx) : Term → Prop :=
  | WS_Var : forall x,
      In x Γ →
      well_scoped Δ Γ (Var x)
  | WS_LamAbs : forall x T t,
      well_scoped_ty Δ T →
      well_scoped Δ (x :: Γ) t →
      well_scoped Δ Γ (LamAbs x T t)
  | WS_Apply : forall t1 t2,
      well_scoped Δ Γ t1 →
      well_scoped Δ Γ t2 →
      well_scoped Δ Γ (Apply t1 t2)
  | WS_TyAbs : forall X K t,
      well_scoped (X :: Δ) Γ t →
      well_scoped Δ Γ (TyAbs X K t)
  | WS_TyInst : forall t T,
      well_scoped Δ Γ t →
      well_scoped_ty Δ T →
      well_scoped Δ Γ (TyInst t T)
  | WS_IWrap : forall F T M,
      well_scoped_ty Δ F →
      well_scoped_ty Δ T →
      well_scoped Δ Γ M →
      well_scoped Δ Γ (IWrap F T M)
  | WS_Unwrap : forall M,
      well_scoped Δ Γ M →
      well_scoped Δ Γ (Unwrap M)
  | WS_Constant : forall u a,
      well_scoped Δ Γ (Constant (Some (ValueOf u a)))
  | WS_Builtin : forall f,
      well_scoped Δ Γ (Builtin f)
  | WS_Error : forall S,
      well_scoped_ty Δ S →
      well_scoped Δ Γ (Error S)
  | WS_Let : forall bs t Δ' Γ',
      Δ' = rev (btvbs bs) ++ Δ →
      Γ' = rev (bvbs bs) ++ Γ →
      well_formed_nonrec Δ Γ bs →
      well_scoped Δ' Γ' t →
      well_scoped Δ Γ (Let NonRec bs t)
  | WS_LetRec : forall bs t Δ' Γ',
      Δ' = rev (btvbs bs) ++ Δ →
      Γ' = rev (bvbs bs) ++ Γ →
      well_formed_rec Δ' Γ' bs →
      well_scoped Δ' Γ' t →
      well_scoped Δ Γ (Let Rec bs t)
with well_formed_nonrec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_NonRec :
      well_formed_nonrec Δ Γ nil
  | W_ConsB_NonRec : forall b bs,
      well_formed Δ Γ b →
      well_formed_nonrec (btvb b ++ Δ) (bvb b ++ Γ) bs →
      well_formed_nonrec Δ Γ (b :: bs)
with well_formed_rec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_Rec :
      well_formed_rec Δ Γ nil
  | W_ConsB_Rec : forall b bs,
      well_formed Δ Γ b →
      well_formed_rec Δ Γ bs →
      well_formed_rec Δ Γ (b :: bs)
with well_formed (Δ Γ : ctx) : Binding → Prop :=
  | W_Term : forall s x T t,
      well_scoped_ty Δ T →
      well_scoped Δ Γ t →
      well_formed Δ Γ (TermBind s (VarDecl x T) t)
  | W_Type : forall X K T,
      well_scoped_ty Δ T →
      well_formed Δ Γ (TypeBind (TyVarDecl X K) T)
  | W_Data : forall X YKs cs matchFunc Δ',
      Δ' = rev (map tvd_name YKs) ++ Δ  →
      (forall c, In c cs →  Δ' |−ok_c c) →
      well_formed Δ Γ (DatatypeBind (Datatype X YKs matchFunc cs)).
```

Listing A.1: The initial problematic definition of the well_scoped relation

```
Definition btvb_app (b : Binding) Δ := btvb b ++ Δ.
Definition bvb_app (b : Binding) Δ := bvb b ++ Δ.
Definition map_tvd_name_rev_app x c := rev (map tvd_name x) ++ c.
Definition rev_btvbs_app (x : list Binding) c := rev (btvbs x) ++ c.
Definition rev_bvbs_app (x : list Binding) c := rev (bvbs x) ++ c.

Inductive well_scoped (Δ Γ: ctx) : Term → Prop :=
  | WS_Var : forall x,
      NameIn x Γ →
      well_scoped Δ Γ (Var x)
  | WS_LamAbs : forall x T t,
      well_scoped_ty Δ T →
      well_scoped Δ (x :: Γ) t →
      well_scoped Δ Γ (LamAbs x T t)
  | WS_Apply : forall t1 t2,
      well_scoped Δ Γ t1 →
      well_scoped Δ Γ t2 →
      well_scoped Δ Γ (Apply t1 t2)
  | WS_TyAbs : forall X K t,
      well_scoped (X :: Δ) Γ t →
      well_scoped Δ Γ (TyAbs X K t)
  | WS_TyInst : forall t T,
      well_scoped Δ Γ t →
      well_scoped_ty Δ T →
      well_scoped Δ Γ (TyInst t T)
  | WS_IWrap : forall F T M,
      well_scoped_ty Δ F →
      well_scoped_ty Δ T →
      well_scoped Δ Γ M →
      well_scoped Δ Γ (IWrap F T M)
  | WS_Unwrap : forall M,
      well_scoped Δ Γ M →
      well_scoped Δ Γ (Unwrap M)
  | WS_Constant : forall u (x : valueOf u),
      well_scoped Δ Γ (Constant (Some' x))
  | WS_Builtin : forall f,
      well_scoped Δ Γ (Builtin f)
  | WS_Error : forall s,
      well_scoped_ty Δ s →
      well_scoped Δ Γ (Error s)
  | WS_Let : forall bs t,
      well_formed_nonrec Δ Γ bs →
      well_scoped (rev_btvbs_app bs Δ) (rev_bvbs_app bs Γ) t →
      well_scoped Δ Γ (Let NonRec bs t)
  | WS_LetRec : forall bs t,
      well_formed_rec (rev_btvbs_app bs Δ) (rev_bvbs_app bs Γ) bs →
      well_scoped (rev_btvbs_app bs Δ) (rev_bvbs_app bs Γ) t →
      well_scoped Δ Γ (Let Rec bs t)
with well_formed_nonrec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_NonRec :
      well_formed_nonrec Δ Γ nil
  | W_ConsB_NonRec : forall b bs,
      well_formed Δ Γ b →
      well_formed_nonrec (btvb_app b Δ) (bvb_app b Γ) bs →
      well_formed_nonrec Δ Γ (b :: bs)
with well_formed_rec (Δ Γ : ctx) : list Binding → Prop :=
  | W_NilB_Rec :
      well_formed_rec Δ Γ nil
  | W_ConsB_Rec : forall b bs,
      well_formed Δ Γ b →
      well_formed_rec Δ Γ bs →
      well_formed_rec Δ Γ (b :: bs)
with well_formed (Δ Γ : ctx) : Binding → Prop :=
  | W_Term : forall s x T t,
      well_scoped_ty Δ T →
      well_scoped Δ Γ t →
      well_formed Δ Γ (TermBind s (VarDecl x T) t)
  | W_Type : forall X K T,
      well_scoped_ty Δ T →
      well_formed Δ Γ (TypeBind (TyVarDecl X K) T)
  | W_Data : forall X YKs cs matchFunc,
      constructors_well_formed (map_tvd_name_rev_app YKs Δ) cs →
      well_formed Δ Γ (DatatypeBind (Datatype X YKs matchFunc cs)).
```

Listing A.2: The rewritten definition of the `well_scoped` relation, together with the necessary auxiliary functions

Bibliography

[1] Carmine Abate et al. 'Trace-relating compiler correctness and secure compilation'. In: Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings 29. Springer. 2020, pp. 1–28.

[2] Abhishek Anand et al. 'CertiCoq: A verified compiler for Coq'. In: The third international workshop on Coq for programming languages (CoqPL). 2017.

[3] Nick Benton and Chung-Kil Hur. 'Biorthogonality, Step-Indexing and Compiler Correctness'. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. ICFP '09. Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 97–108. isbn: 9781605583327. doi: `10.1145/1596550.1596567`. url: `https://doi.org/10.1145/1596550.1596567`.

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

[5] Koen Claessen. QuickCheck. Version 2.14.2. url: `https://hackage.haskell.org/package/QuickCheck`.

[6] Koen Claessen and John Hughes. 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs'. In: SIGPLAN Not. 46.4 (May 2011), pp. 53–64. issn: 0362-1340. doi: `10.1145/1988042.1988046`. url: `https://doi.org/10.1145/1988042.1988046`.

[7] Vijay D'Silva, Mathias Payer and Dawn Song. 'The correctness-security gap in compiler optimization'. In: 2015 IEEE Security and Privacy Workshops. IEEE. 2015, pp. 73–87.

[8] Joris Dral. 'Verified Compiler Optimisations'. MA thesis. Utrecht University, 2022. url: `https://studenttheses.uu.nl/handle/20.500.12932/446`.

[9] Georges Gonthier, Assia Mahboubi and Enrico Tassi. 'A small scale reflection extension for the Coq system'. PhD thesis. Inria Saclay Ile de France, 2016.

[10] Marco Guarnieri et al. 'Spectector: Principled Detection of Speculative Information Flows'. In: 2020 IEEE Symposium on Security and Privacy (SP). 2020, pp. 1–19. doi: `10.1109/SP40000.2020.00011`.

[11] Jacco Krijnen, Joris Dral and Bart Remmers. Certifying Compiler for a Simply Typed Lambda Calculus. Version 1.0.0. Oct. 2022. url: `https://github.com/jaccokrijnen/ulc-cert`.

[12] Jacco Krijnen, Joris Dral and Bart Remmers. Certifying Compiler for Plutus Tx. Version 1.0.0. Apr. 2023. url: `https://github.com/jaccokrijnen/plutus-cert/tree/dec-deriving`.

[13] Jacco O. G. Krijnen et al. Translation Certification for Smart Contracts. Feb. 2022. arXiv: `2201.04919v2 [cs.PL]`. url: `http://arxiv.org/abs/2201.04919v2;%20http://arxiv.org/pdf/2201.04919v2`.

[14] Leonidas Lampropoulos and Bart Remmers. QuickChick fork. Version 1.0.0. Apr. 2023. url: `https://github.com/Xiving/QuickChick/tree/dec-derivining-mutind-bugfix`.

[15] Sorin Lerner, Todd Millstein and Craig Chambers. 'Automatically Proving the Correctness of Compiler Optimizations'. In: SIGPLAN Not. 38.5 (May 2003), pp. 220–231. issn: 0362-1340. doi: `10.1145/780822.781156`. url: `https://doi.org/10.1145/780822.781156`.

[16] Xavier Leroy. 'Formal Verification of a Realistic Compiler'. In: Commun. ACM 52.7 (July 2009), pp. 107–115. issn: 0001-0782. doi: `10.1145/1538788.1538814`. url: `https://doi.org/10.1145/1538788.1538814`.

[17] Bohdan Liesnikov, Marcel Ullrich and Yannick Forster. 'Generating induction principles and subterm relations for inductive types using MetaCoq'. In: arXiv preprint arXiv:2006.15135 (2020).

[18] Robin Milner and Richard Weyhrauch. 'Proving compiler correctness in a mechanized logic'. In: Machine intelligence 7.3 (1972), pp. 51–70.

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

[20] Lawrence C Paulson. ‘The foundation of a generic theorem prover’. In: Journal of Automated Reasoning 5 (1989), pp. 363–397.

[21] Bart Remmers. Collapsed Tagged Proxy Derivation Algorithm. Version 1.0.0. Apr. 2023. url: `https://github.com/Xiving/coq-ctproxy`.

[22] Matthieu Sozeau et al. ‘The metacoq project’. In: Journal of automated reasoning 64.5 (2020), pp. 947–999.

[23] Yong Kiam Tan et al. ‘A New Verified Compiler Backend for CakeML’. In: SIGPLAN Not. 51.9 (Sept. 2016), pp. 60–73. issn: 0362-1340. doi: `10.1145/3022670.2951924`. url: `https://doi.org/10.1145/3022670.2951924`.

[24] Mitchell Wand. ‘Compiler correctness for parallel languages’. In: Proceedings of the seventh international conference on Functional programming languages and computer architecture. 1995, pp. 120–134.

[25] Alexey Rodriguez Yakushev et al. ‘Generic Programming with Fixed Points for Mutually Recursive Datatypes’. In: SIGPLAN Not. 44.9 (Aug. 2009), pp. 233–244. issn: 0362-1340. doi: `10.1145/1631687.1596585`. url: `https://doi.org/10.1145/1631687.1596585`.