A Layered Certifying Compiler Architecture

Jacco Krijnen
PhD advisors: Wouter Swierstra, Manuel Chakravarty, Gabriele Keller
IFIP WG 2.8 (24 April 2024)

Utrecht University

How do you structure the verification of a compiler?

Motivation

- Classic approach to compiler verification (CompCert, CakeML)

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
  - Compiler is implemented in a proof assistant

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
  - Compiler is implemented in a proof assistant
  - Target fixed, well-understood languages

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
  - Compiler is implemented in a proof assistant
  - Target fixed, well-understood languages
- Many compilers and languages are:

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
  - Compiler is implemented in a proof assistant
  - Target fixed, well-understood languages
- Many compilers and languages are:
  - Implementated in language not suitable for verification

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
  - Compiler is implemented in a proof assistant
  - Target fixed, well-understood languages
- Many compilers and languages are:
  - Implementated in language not suitable for verification
  - Community-developed, open-source: in constant flux

Motivation

- Classic approach to compiler verification (CompCert, CakeML)
    - Compiler is implemented in a proof assistant
    - Target fixed, well-understood languages
- Many compilers and languages are:
    - Implementated in language not suitable for verification
    - Community-developed, open-source: in constant flux
- Instead of proving the compiler code correct, prove individual runs correct (translation validation)

Approach

Approach

We take a "gray-box" approach

- Tool that runs simultaneously with the compiler

Approach

We take a "gray-box" approach

- Tool that runs simultaneously with the compiler
- Compiler dumps the program between every pass

Approach

We take a “gray-box” approach

- Tool that runs simultaneously with the compiler
- Compiler dumps the program between every pass
- Tool checks if each pass was performed correctly

Approach

We take a “gray-box” approach

- Tool that runs simultaneously with the compiler
- Compiler dumps the program between every pass
- Tool checks if each pass was performed correctly
- The tool should produce some checkable evidence

Layered Architecture

- For each compiler pass, develop four layers:
    1. Specification layer
    2. Interface layer
    3. Automation layer
    4. Verification layer
- Implemented in a proof assistant (here Coq)
- Each layer gradually improves the trustworthiness of the overall system

Language: STLC

$$
\begin{array}{rcl}
t & ::= & x \\
  & | & \lambda x.\ t \\
  & | & t\ t \\
  & | & \texttt{let}\ x = t\ \texttt{in}\ t
\end{array}
$$

Inlining

Inliner behaviour

“pre-term”

```
let x = 3
in ... (2 * x) ... x ...
```

Inlining

Inliner behaviour

“pre-term” $\mapsto$ “post-term”

```
let x = 3
in ... (2 * x) ... x ...
```

```
let x = 3
in ... (2 * 3) ... x ...
```

Inliner: specification layer

Specification layer (translation relation)

$$\boxed{\Gamma \vdash s \triangleright t}$$

Specification layer (translation relation)

$$\boxed{\Gamma \vdash s \triangleright t}$$

$$\Gamma ::= \bullet \mid x \mapsto t, \Gamma \mid x, \Gamma$$

Specification layer (translation relation)

$$\boxed{\Gamma \vdash s \triangleright t}$$

$$\Gamma ::= \bullet \mid x \mapsto t, \Gamma \mid x, \Gamma$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \triangleright t}\ \textsc{Inline-Var}_1 \qquad \frac{}{\Gamma \vdash x \triangleright x}\ \textsc{Inline-Var}_2$$

Specification layer (translation relation)

$\boxed{\Gamma \vdash s \triangleright t}$

$$\Gamma ::= \bullet \mid x \mapsto t, \Gamma \mid x, \Gamma$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \triangleright t}\ \textsc{Inline-Var}_1 \qquad \frac{}{\Gamma \vdash x \triangleright x}\ \textsc{Inline-Var}_2$$

$$\frac{\Gamma \vdash s \triangleright s' \qquad x \mapsto s, \Gamma \vdash t \triangleright t'}{\Gamma \vdash \texttt{let}\ x = s\ \texttt{in}\ t \triangleright \texttt{let}\ x = s'\ \texttt{in}\ t'}\ \textsc{Inline-Let}$$

Specification layer (translation relation)

$\boxed{\Gamma \vdash s \triangleright t}$

$$\Gamma ::= \bullet \mid x \mapsto t, \Gamma \mid x, \Gamma$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \triangleright t}\ \textsc{Inline-Var}_1 \qquad \frac{}{\Gamma \vdash x \triangleright x}\ \textsc{Inline-Var}_2$$

$$\frac{\Gamma \vdash s \triangleright s' \qquad x \mapsto s, \Gamma \vdash t \triangleright t'}{\Gamma \vdash \texttt{let}\ x = s\ \texttt{in}\ t \triangleright \texttt{let}\ x = s'\ \texttt{in}\ t'}\ \textsc{Inline-Let}$$

$$\frac{\Gamma \vdash s \triangleright s' \qquad \Gamma \vdash t \triangleright t'}{\Gamma \vdash s\ t \triangleright s'\ t'}\ \textsc{Inline-App}$$

Specification layer (translation relation)

$\boxed{\Gamma \vdash s \rhd t}$

$$\Gamma ::= \bullet \mid x \mapsto t, \Gamma \mid x, \Gamma$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \rhd t}\ \textsc{Inline-Var}_1 \qquad \frac{}{\Gamma \vdash x \rhd x}\ \textsc{Inline-Var}_2$$

$$\frac{\Gamma \vdash s \rhd s' \qquad x \mapsto s, \Gamma \vdash t \rhd t'}{\Gamma \vdash \texttt{let}\ \ x = s\ \ \texttt{in}\ \ t \rhd \texttt{let}\ \ x = s'\ \ \texttt{in}\ \ t'}\ \textsc{Inline-Let}$$

$$\frac{\Gamma \vdash s \rhd s' \qquad \Gamma \vdash t \rhd t'}{\Gamma \vdash s\ t \rhd s'\ t'}\ \textsc{Inline-App}$$

$$\frac{x, \Gamma \vdash t \rhd t'}{\Gamma \vdash \lambda x.\ t \rhd \lambda x.\ t'}\ \textsc{Inline-Lam}$$

Specification layer

$$\Gamma \vdash s \triangleright t$$

- Serves as a *formal reference* for compiler implementors
- This relation does not prescribe *what* inlining strategy to use
- It captures all possible inlining strategies

Formalize in proof assistant

```
Inductive inline : list (string * binder_info) -> term -> term -> Prop
  | inline_Var_1 : forall G x t t',
    lookup x G = let_bound t ->
    inline G t t' ->
    inline G (Var x) t
  | inline_Var_2 : forall G x,
    inline G (Var x) (Var x)
  | inline_Let : forall G x s t s' t',
    inline G s s' ->
    inline (let_bound x s :: G) t t' ->
    inline G (Let x s t) (Let x s' t')
  | inline_Lam : forall G x t t',
    inline (lam_bound x :: G) t t' ->
    inline G (Lam x t) (Lam x t')
  | inline_App : forall G s t s' t',
    inline G s s' ->
    inline G t t' ->
    inline G (App s t) (App s' t').
```

Inliner: interface layer

2. Interface Layer

- Bridges the gap between the compiler and the proof assistant
- Modify the compiler: dump intermediate ASTs (gray-box)
- Parse in the proof assistant

```
Definition p1 := Let "x" (App (Lambda ...
Definition p2 := Let "x" (App (Lambda ...

Theorem valid : inline [] p1 p2.
```

Infrastructure to formulate and prove that a pass was performed *according to specification*.

Inliner: automation layer

3. Automation Layer

- Writing proofs by hand gets tedious: large proof objects, proof search becomes slower
- Instead: prove specification decidable (reflection)

```
Fixpoint dec_inline :
  list binder_info -> AST -> AST -> bool.
...

Theorem iff_dec_inline : forall G p q,
  dec_inline G p q = true <-> inline G p q.
```

Inliner: verification layer

4. Verification Layer

- Prove that a specification is correct: related programs should be semantically equivalent
- Choose appropriate semantics, notion of program equivalence

```
Theorem correct : forall p q,
  inline [] p q -> p ≃ q.
```

Scaling up: the Plutus compiler

Plutus Compiler

Plutus is an optimizing compiler, used for compiling smart contracts on the Cardano blockchain.

Plutus Compiler

Plutus is an optimizing compiler, used for compiling smart contracts on the Cardano blockchain.

- Smart contract: code that enforces a financial agreement, manages the flow of financial assets on a blockchain

Plutus Compiler

Plutus is an optimizing compiler, used for compiling smart contracts on the Cardano blockchain.

- Smart contract: code that enforces a financial agreement, manages the flow of financial assets on a blockchain
- Compiled code is deployed on the the blockchain (once committed, immutable)

Plutus Compiler

Plutus is an optimizing compiler, used for compiling smart contracts on the Cardano blockchain.

- Smart contract: code that enforces a financial agreement, manages the flow of financial assets on a blockchain
- Compiled code is deployed on the the blockchain (once committed, immutable)
- These programs have to be extremely well-behaved (DAO vulnerability: $\sim$ \$50 million)

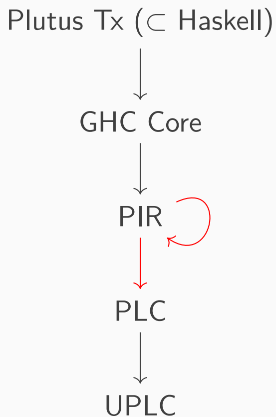
Plutus Compiler

Plutus is an optimizing compiler, used for compiling smart contracts on the Cardano blockchain.

- Smart contract: code that enforces a financial agreement, manages the flow of financial assets on a blockchain
- Compiled code is deployed on the the blockchain (once committed, immutable)
- These programs have to be extremely well-behaved (DAO vulnerability: $\sim$ \$50 million)
- Is the compiler correct?

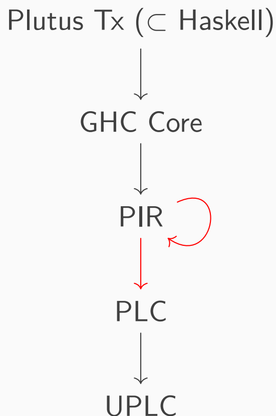
The Plutus compiler pipeline

The Plutus compiler



The Plutus compiler pipeline

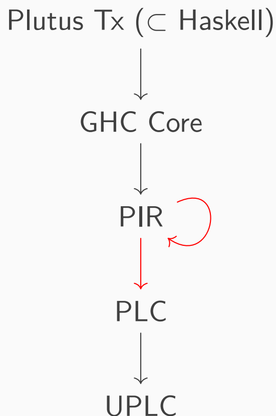
The Plutus compiler



- PIR = System F^{μ}_{ω} + `let(rec)` + ADTs

The Plutus compiler pipeline

The Plutus compiler



- PIR = System F^{μ}_{ω} + `let(rec)` + ADTs
- PLC = System F^{μ}_{ω}

Plutus pass: dead code elimination

1. Specification Layer (Dead Code elimination)

$$\frac{}{\texttt{let } x = t; bs \texttt{ in } s \rhd} \text{DCE-ELIM}$$

$$\frac{}{\texttt{let } x = t; bs \texttt{ in } s \rhd} \text{DCE-KEEP}$$

1. Specification Layer (Dead Code elimination)

$$\frac{}{\texttt{let } x = t; bs \texttt{ in } s \rhd \texttt{let } bs' \texttt{ in } s'} \textsc{DCE-Elim}$$

$$\frac{}{\texttt{let } x = t; bs \texttt{ in } s \rhd \texttt{let } x = t'; bs' \texttt{ in } s'} \textsc{DCE-Keep}$$

1. Specification Layer (Dead Code elimination)

$$\frac{x \notin \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ s')}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}\ \textsc{DCE-Elim}$$

$$\frac{}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ x = t'; bs'\ \texttt{in}\ s'}\ \textsc{DCE-Keep}$$

1. Specification Layer (Dead Code elimination)

$$\frac{x \notin \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ s') \qquad t \in value}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}\ \textsc{DCE-Elim}$$

$$\frac{}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ x = t'; bs'\ \texttt{in}\ s'}\ \textsc{DCE-Keep}$$

1. Specification Layer (Dead Code elimination)

$$\frac{t \in \mathit{value} \qquad x \notin \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ s') \qquad \texttt{let}\ bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}\ \textsc{DCE-Elim}$$

$$\frac{}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ x = t'; bs'\ \texttt{in}\ s'}\ \textsc{DCE-Keep}$$

1. Specification Layer (Dead Code elimination)

$$\frac{t \in \mathit{value} \qquad x \notin \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ s') \qquad \texttt{let}\ bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}\ \textsc{DCE-Elim}$$

$$\frac{t \rhd t' \qquad \texttt{let}\ bs\ \texttt{in}\ s \rhd \texttt{let}\ bs\ \texttt{in}\ s'}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ x = t'; bs'\ \texttt{in}\ s'}\ \textsc{DCE-Keep}$$

1. Specification Layer (Dead Code elimination)

$$\frac{x \notin \mathrm{FV}(\texttt{let}\ bs'\ \texttt{in}\ s') \quad t \in \mathit{value} \qquad \texttt{let}\ bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ bs'\ \texttt{in}\ s'}\ \textsc{DCE-Elim}$$

$$\frac{t \rhd t' \qquad \texttt{let}\ bs\ \texttt{in}\ s \rhd \texttt{let}\ bs\ \texttt{in}\ s'}{\texttt{let}\ x = t; bs\ \texttt{in}\ s \rhd \texttt{let}\ x = t'; bs'\ \texttt{in}\ s'}\ \textsc{DCE-Keep}$$

$$\frac{s \rhd t \qquad t \rhd t'}{s\ t \rhd s'\ t'}\ \textsc{DCE-Apply} \qquad \frac{t \ \rhd\ t'}{\lambda x : \tau.\ t \rhd \lambda x : \tau.\ t'}\ \textsc{DCE-Lambda} \qquad \ldots$$

Specification Layer: Scaling up

- More syntax $\Rightarrow$ More rules

Specification Layer: Scaling up

- More syntax ⇒ More rules
- Scoping of letrec

$$\frac{x \notin \mathrm{FV}(\texttt{let rec } bs'_0 \texttt{ in } s') \qquad t \in value \qquad \texttt{let rec } bs \texttt{ in } s \rhd_{rec} \texttt{let rec } bs' \texttt{ in } s' \mid bs'_0}{\texttt{let rec } x = t; bs \texttt{ in } s \ \rhd_{rec} \ \texttt{let rec } bs' \texttt{ in } s' \mid bs'_0} \ \textsc{DCE-Elim-Rec}$$

Specification Layer: Scaling up

- More syntax ⇒ More rules
- Scoping of letrec

$$\frac{x \notin \mathrm{FV}(\texttt{let rec } {\color{red} bs'_0} \texttt{ in } s') \qquad t \in \mathit{value} \qquad \texttt{let rec } bs \texttt{ in } s \rhd_{rec} \texttt{let rec } bs' \texttt{ in } s' \mid {\color{red} bs'_0}}{\texttt{let rec } x = t; bs \texttt{ in } s \ \rhd_{rec} \ \texttt{let rec } bs' \texttt{ in } s' \mid {\color{red} bs'_0}} \textsc{ DCE-Elim-Rec}$$

- Composite pass: inliner includes dead code elimination

$$t \rhd t' := \exists t_1.\ t \rhd_{inl} t_1 \ \wedge \ t_1 \rhd_{dce} t'$$

2. Interface Layer

- Plutus implementation and process are open-source: fork + pull request on Github
- Dump the intermediate ASTs (and sometimes pass-specific information)

Decision procedure dead-code elimination:

```
Fixpoint dec_dce : AST -> AST -> bool.

Theorem iff_dec_dce : forall p q,
  dec_dce p q = true <-> dce p q.
```

3. Automation Layer

Composite pass: Inlining + Dead Code Elimination

$$t \triangleright t' := \exists t_1.\ t \triangleright_{inl} t_1 \ \wedge\ t_1 \triangleright_{dce} t'$$

Intermediate term t_1 is not dumped by the compiler...

...But we can reconstruct it!

```
Fixpoint make_intermediate :
    list Var -> AST -> AST -> option AST.
...
```

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation
- Static semantics preservation:

$$s \triangleright t \ \land \ \Delta; \Gamma \vdash s : \tau \ \Rightarrow \ \Delta; \Gamma \vdash t : \tau$$

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation
- Static semantics preservation:

$$s \triangleright t \;\wedge\; \Delta; \Gamma \vdash s : \tau \;\Rightarrow\; \Delta; \Gamma \vdash t : \tau$$

- Big-step semantics with substitution: $t \Downarrow v$

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation
- Static semantics preservation:

$$s \triangleright t \ \wedge \ \Delta; \Gamma \vdash s : \tau \ \Rightarrow \ \Delta; \Gamma \vdash t : \tau$$

- Big-step semantics with substitution: $t \Downarrow v$
- Contextual equivalence: $\Delta; \Gamma \vdash s \simeq_{ctx} t : \tau$

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation
- Static semantics preservation:

$$s \triangleright t \;\wedge\; \Delta; \Gamma \vdash s : \tau \;\Rightarrow\; \Delta; \Gamma \vdash t : \tau$$

- Big-step semantics with substitution: $t \Downarrow v$
- Contextual equivalence: $\Delta; \Gamma \vdash s \simeq_{ctx} t : \tau$
- Step-indexed logical relation that implies contextual equivalence

4. Verification Layer

We formalised PIR (PLC) semantics in Coq

- Type system $\Delta; \Gamma \vdash t : \tau$ with type normalisation
- Static semantics preservation:

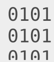
$$s \triangleright t \ \wedge \ \Delta; \Gamma \vdash s : \tau \ \Rightarrow \ \Delta; \Gamma \vdash t : \tau$$

- Big-step semantics with substitution: $t \Downarrow v$
- Contextual equivalence: $\Delta; \Gamma \vdash s \simeq_{ctx} t : \tau$
- Step-indexed logical relation that implies contextual equivalence
- Correctness of translation relation

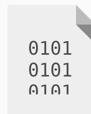
$$s \triangleright t \ \wedge \ \Delta; \Gamma \vdash s : \tau \ \Rightarrow \ \Delta; \Gamma \vdash s \simeq_{ctx} t : \tau$$

Certificates

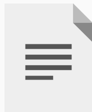
A Certificate



A Certificate



A Certificate



We want a *verifiable link* (certificate) that

A Certificate

We want a *verifiable link* (certificate) that

1. Relates the source code to the compiled code

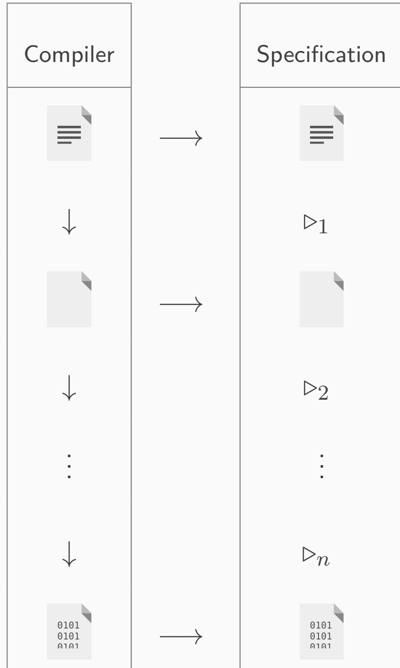
A Certificate

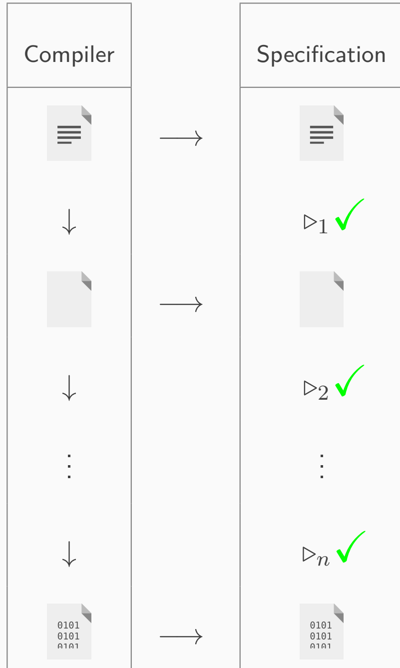
We want a *verifiable link* (certificate) that

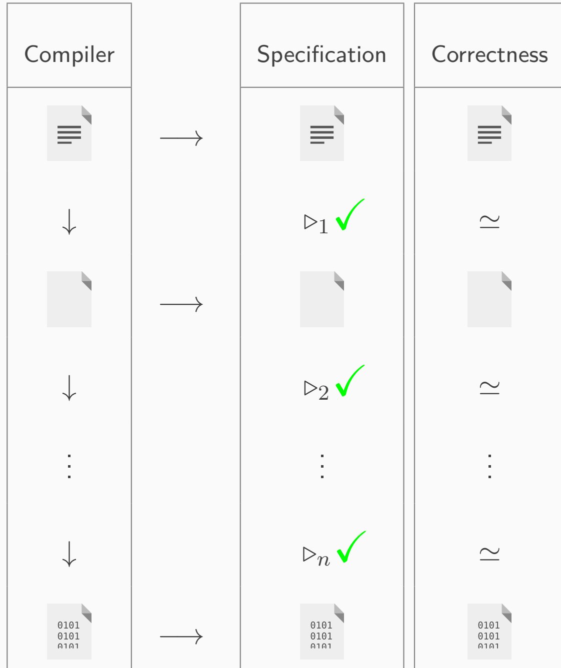
1. Relates the source code to the compiled code
2. The compiler did not introduce any bugs

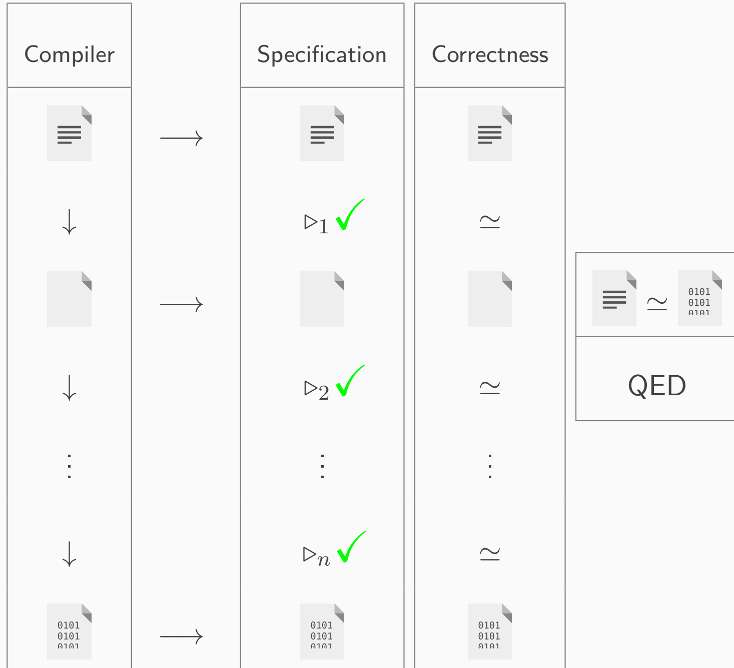
Compiler

Compiler

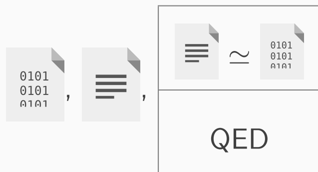

Compiler
Specification
$\triangleright_1$
$\triangleright_2$
$\triangleright_n$
0101
0101
0101
0101


Compiler
Specification
$\triangleright_1$ ✓
$\triangleright_2$ ✓
$\triangleright_n$ ✓
0101
0101

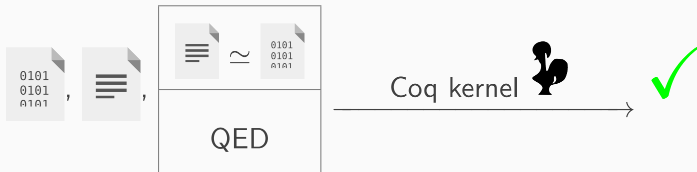

Compiler
Specification
Correctness
0101
0101
0101
0101
0101
0101


Compiler
Specification
Correctness
$\triangleright_1$
$\triangleright_2$
$\triangleright_n$
0101
0101
QED

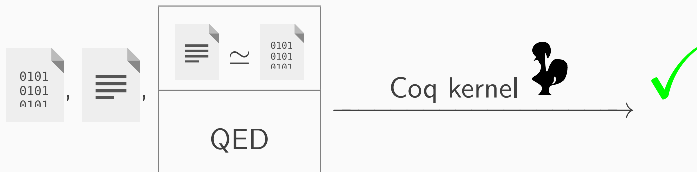
Certifier



Certifier

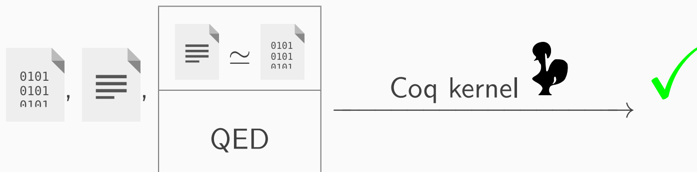


Certifier



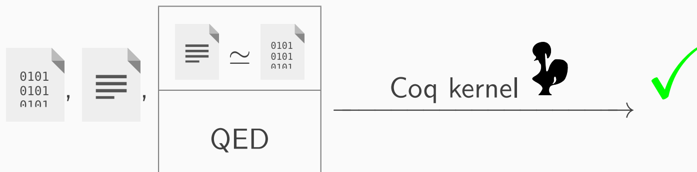
- Trusted computing base

Certifier



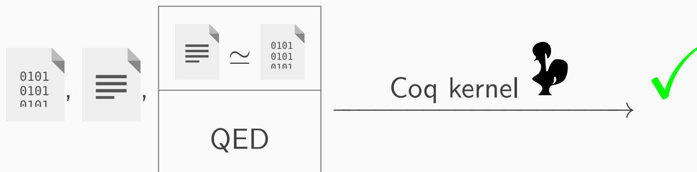
- Trusted computing base
    - Definition of $\simeq$

Certifier



- Trusted computing base
    - Definition of $\simeq$
    - Coq Kernel

Certifier



- Trusted computing base
  - Definition of $\simeq$
  - Coq Kernel
- Coq kernel: relatively small and well-studied code base

Additionally

- Formal pedantic mode

Additionally

- Formal pedantic mode
  - Integrate decision procedures directly in the compiler

Additionally

- Formal pedantic mode
  - Integrate decision procedures directly in the compiler
- Improved compiler testing

Additionally

- Formal pedantic mode
  - Integrate decision procedures directly in the compiler
- Improved compiler testing
  - Plutus: existing test suite (example input/outputs)

Additionally

- Formal pedantic mode
  - Integrate decision procedures directly in the compiler
- Improved compiler testing
  - Plutus: existing test suite (example input/outputs)
  - Property-based testing

Additionally

- Formal pedantic mode
  - Integrate decision procedures directly in the compiler
- Improved compiler testing
  - Plutus: existing test suite (example input/outputs)
  - Property-based testing
- Generate LaTeX from specification (Coq code)

Further work

- End-to-end verification
  - Plutus: Haskell $\rightarrow$ PIR $\rightarrow$ PLC
  - Alternative front-end: Gallina + CertiCoq
- Proof transportation?
  - Use equivalence proof
- Certifier implementation: Coq vs Agda?
- Hints for constructing typing derivation, instead of re-implementing type-checker

Conclusion

- Gradual form of verification: each layer improves the trustworthiness of the system.

Conclusion

- Gradual form of verification: each layer improves the trustworthiness of the system.
- We don't have to reason about an implementation, only a high-level specification

Conclusion

- Gradual form of verification: each layer improves the trustworthiness of the system.
- We don't have to reason about an implementation, only a high-level specification
- Robustness: certain changes in the compiler do not affect the verification at all.

Conclusion

- Gradual form of verification: each layer improves the trustworthiness of the system.
- We don't have to reason about an implementation, only a high-level specification
- Robustness: certain changes in the compiler do not affect the verification at all.
- Separation of concerns: compiler writers do not have to be proof engineers